

北京郵電大學



数据库实验报告

openGauss

上课时间	:	周五3~5节
授课教师	:	梁美玉
姓名	:	臧泽元 蔡景旭 庄卜荣
学号	:	2022210085 2022210053 2022210324
班级	:	2022217804 2022217803 2022217801
日期	:	2024.11.30

成员分工：

- 臧泽元：环境搭建、任务1、任务4、任务5
- 蔡景旭：任务3、任务7
- 庄卜荣：任务2、任务6

0 环境搭建

实验 1-3 所用数据库部署于本地 docker 容器中，基于 [enmotech/enmotech-docker-opengauss: Ennotech openGauss Docker Image](#) 中给出的 openGauss 1.1.0 x86 镜像进行配置，容器内系统为 CentOS 7.6。拉取镜像创建容器的指令如下：

```
1 $ docker run --name opengauss --privileged=true -d -e  
GS_PASSWORD=Enmo@123 enmotech/opengauss:1.1.0
```

后续进入容器（`docker exec`）时，将容器的 22 端口映射到主机的 22222 端口以便 ssh 远程连接。

1 数据库建表及数据导入

1.1 创建关系表

1.1.1 订单表 ORDERS

终端显示：

```
1 omm=# CREATE TABLE ORDERS ( O_ORDERKEY      INTEGER NOT NULL,  
2 omm(# O_CUSTKEY      INTEGER NOT NULL,  
3 omm(# O_ORDERSTATUS    CHAR(1) NOT NULL,  
4 omm(# O_TOTALPRICE    DECIMAL(15,2) NOT NULL,  
5 omm(# O_ORDERDATE     DATE NOT NULL,  
6 omm(# O_ORDERPRIORITY   CHAR(15) NOT NULL,  
7 omm(# O_CLERK        CHAR(15) NOT NULL,  
8 omm(# O_SHIPPRIORITY   INTEGER NOT NULL,  
9 omm(# O_COMMENT       VARCHAR(79) NOT NULL);  
10 CREATE TABLE
```

末尾显示 `CREATE TABLE` 表示建表成功。

1.1.2 区域表 REGION

```
1 omm=# CREATE TABLE REGION ( R_REGIONKEY      INTEGER NOT NULL,
2 omm(# R_NAME      CHAR(25) NOT NULL,
3 omm(# R_COMMENT   VARCHAR(152));
4 CREATE TABLE
```

1.1.3 国家表 NATION

```
1 omm=# CREATE TABLE NATION ( N_NATIONKEY      INTEGER NOT NULL,
2 omm(# N_NAME      CHAR(25) NOT NULL,
3 omm(# N_REGIONKEY  INTEGER NOT NULL,
4 omm(# N_COMMENT   VARCHAR(152));
5 CREATE TABLE
```

1.1.4 供应商表 SUPPLIER

```
1 omm=# CREATE TABLE SUPPLIER ( S_SUPPKEY      INTEGER NOT NULL,
2 omm(# S_NAME      CHAR(25) NOT NULL,
3 omm(# S_ADDRESS   VARCHAR(40) NOT NULL,
4 omm(# S_NATIONKEY  INTEGER NOT NULL,
5 omm(# S_PHONE     CHAR(15) NOT NULL,
6 omm(# S_ACCTBAL   DECIMAL(15,2) NOT NULL,
7 omm(# S_COMMENT   VARCHAR(101) NOT NULL);
8 CREATE TABLE
```

1.1.5 零部件表 PART

```
1 omm=# CREATE TABLE PART ( P_PARTKEY      INTEGER NOT NULL,
2 omm(# P_NAME      VARCHAR(55) NOT NULL,
3 omm(# P_MFGR      CHAR(25) NOT NULL,
4 omm(# P_BRAND     CHAR(10) NOT NULL,
5 omm(# P_TYPE      VARCHAR(25) NOT NULL,
6 omm(# P_SIZE      INTEGER NOT NULL,
7 omm(# P_CONTAINER  CHAR(10) NOT NULL,
8 omm(# P_RETAILPRICE DECIMAL(15,2) NOT NULL,
9 omm(# P_COMMENT    VARCHAR(23) NOT NULL);
10 CREATE TABLE
```

1.3.6 零部件供应表 PARTSUPP

```
1 omm=# CREATE TABLE PARTSUPP ( PS_PARTKEY      INTEGER NOT NULL,
2 omm(# PS_SUPPKEY    INTEGER NOT NULL,
3 omm(# PS_AVAILQTY   INTEGER NOT NULL,
4 omm(# PS_SUPPLYCOST DECIMAL(15,2) NOT NULL,
5 omm(# PS_COMMENT    VARCHAR(199) NOT NULL);
6 CREATE TABLE
```

1.3.7 客户表 CUSTOMER

```
1 omm=# CREATE TABLE CUSTOMER ( C_CUSTKEY      INTEGER NOT NULL,
2 omm(# C_NAME        VARCHAR(25) NOT NULL,
3 omm(# C_ADDRESS     VARCHAR(40) NOT NULL,
4 omm(# C_NATIONKEY   INTEGER NOT NULL,
5 omm(# C_PHONE       CHAR(15) NOT NULL,
6 omm(# C_ACCTBAL    DECIMAL(15,2) NOT NULL,
7 omm(# C_MKTSEGMENT  CHAR(10) NOT NULL,
8 omm(# C_COMMENT    VARCHAR(117) NOT NULL);
9 CREATE TABLE
```

1.3.8 订单明细表 LINEITEM

```
1 omm=# CREATE TABLE LINEITEM ( L_ORDERKEY      INTEGER NOT NULL,
2 omm(# L_PARTKEY     INTEGER NOT NULL,
3 omm(# L_SUPPKEY     INTEGER NOT NULL,
4 omm(# L_LINENUMBER  INTEGER NOT NULL,
5 omm(# L_QUANTITY    DECIMAL(15,2) NOT NULL,
6 omm(# L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL,
7 omm(# L_DISCOUNT    DECIMAL(15,2) NOT NULL,
8 omm(# L_TAX         DECIMAL(15,2) NOT NULL,
9 omm(# L_RETURNFLAG   CHAR(1) NOT NULL,
10 omm(# L_LINESSTATUS CHAR(1) NOT NULL,
11 omm(# L_SHIPDATE    DATE NOT NULL,
12 omm(# L_COMMITDATE   DATE NOT NULL,
13 omm(# L_RECEIPTDATE  DATE NOT NULL,
14 omm(# L_SHIPINSTRUCT CHAR(25) NOT NULL,
15 omm(# L_SHIPMODE    CHAR(10) NOT NULL,
16 omm(# L_COMMENT     VARCHAR(44) NOT NULL);
17 CREATE TABLE
```

通过 `\d+` 命令可看到创建完的 8 个表：

```
lomm=# \d+
                                         List of relations
 Schema |   Name    | Type  | Owner |  Size   |           Storage           | Description
-----+-----+-----+-----+-----+-----+-----+
 public | customer | table | omm  | 0 bytes | {orientation=row,compression=no} |
 public | lineitem | table | omm  | 0 bytes | {orientation=row,compression=no} |
 public | nation   | table | omm  | 0 bytes | {orientation=row,compression=no} |
 public | orders   | table | omm  | 0 bytes | {orientation=row,compression=no} |
 public | part     | table | omm  | 0 bytes | {orientation=row,compression=no} |
 public | partsupp | table | omm  | 0 bytes | {orientation=row,compression=no} |
 public | region   | table | omm  | 0 bytes | {orientation=row,compression=no} |
 public | supplier | table | omm  | 0 bytes | {orientation=row,compression=no} |
(8 rows)
```

1.2 数据导入

1.2.1 上传文件

将 tpc-h 的 txt 格式数据文件用 SCP 上传到 openGauss 所在的容器上，首先在容器 omm 用户下创建目录 `tpc-h/data` 并授予全部权限：

```
1 [omm@cfde74516988 ~]$ pwd
2 /home/omm
3 [omm@cfde74516988 ~]$ mkdir tpc-h
4 [omm@cfde74516988 ~]$ cd tpc-h
5 [omm@cfde74516988 tpc-h]$ mkdir data
```

```
1 [root@cfde74516988 ~]# cd /home/omm
2 [root@cfde74516988 omm]# chmod 777 tpc-h
3 [root@cfde74516988 omm]# cd tpc-h
4 [root@cfde74516988 tpc-h]# chmod 777 data
```

然后退出 omm 用户回到 root，将 txt 格式的数据文件移到该文件夹里（第一次远程连接需要设置密钥）：

```
[root@cfde74516988 ~]# scp king@10.29.206.110:~/Documents/GitHub/Database-lab-openGauss/tpc-h/* /home/omm/tpc-h/data
[Password:
customer.txt                                100% 4722KB  15.1MB/s  00:00
lineitem.txt                                 100% 148MB   16.1MB/s  00:09
nation.txt                                  100% 2356    278.7KB/s 00:00
orders.txt                                 100% 24MB    17.5MB/s  00:01
part.txt                                    100% 4865KB  18.4MB/s  00:00
partsupp.txt                               100% 22MB    17.1MB/s  00:01
region.txt                                 100% 199     25.8KB/s  00:00
supplier.txt                               100% 245KB   9.4MB/s  00:00]
```

最后授予文件全部权限：

```
1 [root@cfde74516988 ~]# cd /home/omm/tpc-h/data
2 [root@cfde74516988 data]# chmod 777 *
3 [root@cfde74516988 data]# ls -l
4 total 208992
5 -rwxrwxrwx 1 root root 4835158 Nov 25 10:19 customer.txt
6 -rwxrwxrwx 1 root root 155651209 Nov 25 10:19 lineitem.txt
7 -rwxrwxrwx 1 root root 2356 Nov 25 10:19 nation.txt
8 -rwxrwxrwx 1 root root 25118002 Nov 25 10:19 orders.txt
9 -rwxrwxrwx 1 root root 23143960 Nov 25 10:19 partsupp.txt
10 -rwxrwxrwx 1 root root 4982186 Nov 25 10:19 part.txt
11 -rwxrwxrwx 1 root root 199 Nov 25 10:19 region.txt
12 -rwxrwxrwx 1 root root 250820 Nov 25 10:19 supplier.txt
```

1.2.2 导入数据

以 omm 用户登录数据库主节点，并连接数据库，然后使用 `\copy` 命令导入数据：

```
1 omm=# copy region FROM '/home/omm/tpc-h/data/region.txt' with
2 delimiter as '|';
3 COPY 5
4 omm=# copy nation FROM 'home/omm/tpc-h/data/nation.txt' with
5 delimiter as '|';
```

```

4  ERROR:  could not open file "home/omm/tpc-h/data/nation.txt" for
      reading: No such file or directory
5  omm=# copy nation FROM '/home/omm/tpc-h/data/nation.txt' with
      delimiter as '|';
6  COPY 25
7  omm=# copy part FROM '/home/omm/tpc-h/data/part.txt' with delimiter
      as '|';
8  COPY 40000
9  omm=# copy supplier FROM '/home/omm/tpc-h/data/supplier.txt' with
      delimiter as '|';
10 COPY 2000
11 omm=# copy customer FROM '/home/omm/tpc-h/data/customer.txt' with
      delimiter as '|';
12 COPY 30000
13 omm=# copy lineitem FROM '/home/omm/tpc-h/data/lineitem.txt' with
      delimiter as '|';
14 COPY 1199969
15 omm=# copy partsupp FROM '/home/omm/tpc-h/data/partsupp.txt' with
      delimiter as '|';
16 COPY 160000
17 omm=# copy orders FROM '/home/omm/tpc-h/data/orders.txt' with
      delimiter as '|';
18 COPY 300000

```

通过 `\d+` 命令可看到表的大小有更新：

List of relations						
Schema	Name	Type	Owner	Size	Storage	Description
public	customer	table	omm	5776 kB	{orientation=row,compression=no}	
public	lineitem	table	omm	195 MB	{orientation=row,compression=no}	
public	nation	table	omm	8192 bytes	{orientation=row,compression=no}	
public	orders	table	omm	34 MB	{orientation=row,compression=no}	
public	part	table	omm	6840 kB	{orientation=row,compression=no}	
public	partsupp	table	omm	27 MB	{orientation=row,compression=no}	
public	region	table	omm	8192 bytes	{orientation=row,compression=no}	
public	supplier	table	omm	360 kB	{orientation=row,compression=no}	

(8 rows)

导入后为关系表添加约束：

```

1  omm=# ALTER TABLE REGION
2  omm-# ADD PRIMARY KEY (R_REGIONKEY);
3  NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
          "region_pkey" for table "region"
4  ALTER TABLE
5
6  omm=# ALTER TABLE NATION
7  omm-# ADD PRIMARY KEY (N_NATIONKEY);
8  NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
          "nation_pkey" for table "nation"
9  ALTER TABLE
10
11 omm=# ALTER TABLE NATION

```

```

12 omm-># ADD FOREIGN KEY (N_REGIONKEY) references REGION;
13 ALTER TABLE
14
15 omm=># ALTER TABLE PART
16 omm-># ADD PRIMARY KEY (P_PARTKEY);
17 NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
18 "part_pkey" for table "part"
19 ALTER TABLE
20
21 omm=># ALTER TABLE SUPPLIER
22 omm-># ADD PRIMARY KEY (S_SUPPKEY);
23 NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
24 "supplier_pkey" for table "supplier"
25 ALTER TABLE
26
27 omm=># ALTER TABLE SUPPLIER
28 omm-># ADD FOREIGN KEY (S_NATIONKEY) references NATION;
29 ALTER TABLE
30
31 omm=># ALTER TABLE PARTSUPP
32 omm-># ADD PRIMARY KEY (PS_PARTKEY,PS_SUPPKEY);
33 NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
34 "partsupp_pkey" for table "partsupp"
35 ALTER TABLE
36
37 omm=># ALTER TABLE CUSTOMER
38 omm-># ADD PRIMARY KEY (C_CUSTKEY);
39 NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
40 "customer_pkey" for table "customer"
41 ALTER TABLE
42
43 omm=># ALTER TABLE CUSTOMER
44 omm-># ADD FOREIGN KEY (C_NATIONKEY) references NATION;
45 ALTER TABLE
46
47 omm=># ALTER TABLE LINEITEM
48 omm-># ADD PRIMARY KEY (L_ORDERKEY,L_LINENUMBER);
49 NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
50 "lineitem_pkey" for table "lineitem"
51 ALTER TABLE
52
53 omm=># ALTER TABLE ORDERS
54 omm-># ADD PRIMARY KEY (O_ORDERKEY);
55 NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index
      "orders_pkey" for table "orders"
56 ALTER TABLE
57
58 omm=># ALTER TABLE PARTSUPP
59 omm-># ADD FOREIGN KEY (PS_SUPPKEY) references SUPPLIER;
60 ALTER TABLE

```

```

56
57 omm=# ALTER TABLE PARTSUPP
58 omm-# ADD FOREIGN KEY (PS_PARTKEY) references PART;
59 ALTER TABLE
60
61 omm=# ALTER TABLE ORDERS
62 omm-# ADD FOREIGN KEY (O_CUSTKEY) references CUSTOMER;
63 ALTER TABLE
64
65 omm=# ALTER TABLE LINEITEM
66 omm-# ADD FOREIGN KEY (L_ORDERKEY) references ORDERS;
67 ALTER TABLE
68
69 omm=# ALTER TABLE LINEITEM
70 omm-# ADD FOREIGN KEY (L_PARTKEY,L_SUPPKEY) references PARTSUPP;
71 ALTER TABLE

```

用 `select` 语句查看关系表可以显示刚刚导入进来的数据：

```

omm# select * from part
omm-#
omm-# ;
p_partkey | p_name | p_mfgr | p_brand | p_type | p_size | p_container | p_retailprice | p_comment
-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#13 | STANDARD ANODIZED TIN | 7 | JUMBO PKG | 901.00 | ly, slyly ironi
2 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#13 | STANDARD ANODIZED TIN | 1 | LG CASE | 902.00 | lar accounts amo
3 | almond antique aquamarine azure beige | Manufacturer#4 | Brand#42 | STANDARD ANODIZED TIN | 21 | WRAP CASE | 903.00 | regular deposits hag
4 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#34 | STANDARD ANODIZED TIN | 14 | MED DRUM | 904.00 | p furiously r
5 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#32 | STANDARD ANODIZED TIN | 15 | SM PKG | 905.00 | wake carefully
6 | almond antique aquamarine azure beige | Manufacturer#2 | Brand#24 | STANDARD ANODIZED TIN | 4 | MED BAG | 906.00 | sual a
7 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#11 | STANDARD ANODIZED TIN | 45 | SM BAG | 907.00 | lyly, ex
8 | almond antique aquamarine azure beige | Manufacturer#4 | Brand#44 | STANDARD ANODIZED TIN | 41 | LG DRUM | 908.00 | eposi
9 | almond antique aquamarine azure beige | Manufacturer#4 | Brand#33 | STANDARD ANODIZED TIN | 12 | WRAP CASE | 909.00 | neric foxe
10 | almond antique aquamarine azure beige | Manufacturer#5 | Brand#54 | STANDARD ANODIZED TIN | 44 | LG CAN | 910.01 | itinely final deposit
11 | almond antique aquamarine azure beige | Manufacturer#2 | Brand#25 | STANDARD ANODIZED TIN | 43 | WRAP BOX | 911.01 | ng gr
12 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#33 | STANDARD ANODIZED TIN | 25 | JUMBO CASE | 912.01 | quickly
13 | almond antique aquamarine azure beige | Manufacturer#5 | Brand#55 | STANDARD ANODIZED TIN | 1 | JUMBO PACK | 913.01 | osits.
14 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#13 | STANDARD ANODIZED TIN | 28 | JUMBO BOX | 914.01 | kages c
15 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#15 | STANDARD ANODIZED TIN | 45 | LG CASE | 915.01 | usual ac
16 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#32 | STANDARD ANODIZED TIN | 2 | MED PACK | 916.01 | unts a
17 | almond antique aquamarine azure beige | Manufacturer#4 | Brand#43 | STANDARD ANODIZED TIN | 16 | LG BOX | 917.01 | regular accounts
18 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#11 | STANDARD ANODIZED TIN | 42 | JUMBO PACK | 918.01 | s cajole slyly a
19 | almond antique aquamarine azure beige | Manufacturer#2 | Brand#23 | STANDARD ANODIZED TIN | 33 | WRAP BOX | 919.01 | pending acc
20 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#12 | STANDARD ANODIZED TIN | 48 | MED BAG | 920.02 | are across the asympt
21 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#33 | STANDARD ANODIZED TIN | 31 | MED BAG | 921.02 | ss packages. pendin
22 | almond antique aquamarine azure beige | Manufacturer#4 | Brand#43 | STANDARD ANODIZED TIN | 19 | LG DRUM | 922.02 | even p
23 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#35 | STANDARD ANODIZED TIN | 42 | JUMBO JAR | 923.02 | nice fina
24 | almond antique aquamarine azure beige | Manufacturer#5 | Brand#52 | STANDARD ANODIZED TIN | 20 | MED CASE | 924.02 | final the
25 | almond antique aquamarine azure beige | Manufacturer#5 | Brand#55 | STANDARD ANODIZED TIN | 3 | JUMBO BAG | 925.02 | requests wake
26 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#32 | STANDARD ANODIZED TIN | 32 | SM CASE | 926.02 | instructions i
27 | almond antique aquamarine azure beige | Manufacturer#1 | Brand#14 | STANDARD ANODIZED TIN | 28 | MED PKG | 927.02 | s wake, ir
28 | almond antique aquamarine azure beige | Manufacturer#4 | Brand#44 | STANDARD ANODIZED TIN | 19 | JUMBO PKG | 928.02 | x-ray pending, iron
29 | almond antique aquamarine azure beige | Manufacturer#3 | Brand#33 | STANDARD ANODIZED TIN | 7 | LG DRUM | 929.02 | carefully fluffi
--More-1

```

2 数据查询与修改

2.1 单表查询

查询1：

从订单表 `ORDERS` 中，找出由收银员 `Clerk#000000951` 处理的满足下列条件的所有订单

`O_ORDERKEY` :

(1) 订单总价位于 [起始价格 5000, 结束价格 100000]

(2) 下单日期在 `开始日期 2019-01-02 00:00:00` 至 `结束日期 2020-08-31 00:00:00` 之间，

(3) 订单状态 `O_ORDERSTATUS` 不为空

列出这些订单的订单 `key` (`O_ORDERKEY`)、客户 `key`、订单状态、订单总价、下单日期 (重命名为 `O_DATE`)、订单优先级和发货优先级；

要求：对查询结果，按照订单优先级从高到低、发货优先级从高到低排序。

```
1 -- 查询订单信息
2 SELECT
3     O_ORDERKEY,
4     O_CUSTKEY,
5     O_ORDERSTATUS,
6     O_TOTALPRICE,
7     O_ORDERDATE AS O_DATE,
8     O_ORDERPRIORITY,
9     O_SHIPPRIORITY
10 FROM
11     ORDERS
12 WHERE
13     O_CLERK = 'Clerk#000000951'
14     AND O_TOTALPRICE BETWEEN 5000 AND 100000
15     AND O_ORDERDATE BETWEEN '2019-01-02' AND '2020-08-31'
16     AND O_ORDERSTATUS IS NOT NULL
17 ORDER BY
18     O_ORDERPRIORITY DESC,
19     O_SHIPPRIORITY DESC;
```

```
1 omm=# SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS, O_TOTALPRICE,
2     O_ORDERDATE AS O_DATE, O_ORDERPRIORITY, O_SHIPPRIORITY
3 FROM ORDERS
4 WHERE O_CLERK = 'Clerk#000000951';omm-# omm-#
5     o_orderkey | o_custkey | o_orderstatus | o_totalprice |
6     o_date      | o_orderpriority | o_shippriority
7
8
9
10
11
12
13
14
```

6	1	7381	O	181585.13	2019-01-02	
7	00:00:00	5-LOW		0		
8	839	5578	O	104005.14	2018-08-08	
9	00:00:00	1-URGENT		0		
10	2338	27874	O	22264.72	2020-09-15	
11	00:00:00	2-HIGH		0		
12	4579	20828	O	147919.32	2018-12-01	
13	00:00:00	2-HIGH		0		
14	8452	27832	F	147102.45	2015-07-31	
15	00:00:00	4-NOT SPECIFIED		0		
16	9185	2893	F	92840.21	2017-06-16	
17	00:00:00	2-HIGH		0		
18	12163	1733	O	183726.95	2020-07-20	
19	00:00:00	5-LOW		0		
20	13508	16033	O	42756.76	2020-04-17	
21	00:00:00	4-NOT SPECIFIED		0		
22	14277	18920	O	133599.91	2021-02-14	
23	00:00:00	4-NOT SPECIFIED		0		

15	15073	1468	F		138584.35	2015-01-26
	00:00:00	3-MEDIUM		0		
16	17636	15205	F		137295.03	2017-02-05
	00:00:00	5-LOW		0		
17	19200	854	O		144151.90	2020-07-27
	00:00:00	4-NOT SPECIFIED		0		
18	19205	11662	F		327627.84	2016-07-27
	00:00:00	3-MEDIUM		0		
19	20547	16147	F		38376.91	2016-08-27
	00:00:00	2-HIGH		0		
20	21312	5638	O		55741.75	2019-02-01
	00:00:00	1-URGENT		0		
21	25639	25135	F		46746.31	2017-10-12
	00:00:00	3-MEDIUM		0		
22	26885	25732	O		226010.66	2020-05-11
	00:00:00	3-MEDIUM		0		
23	27364	4489	O		201233.85	2018-05-21
	00:00:00	3-MEDIUM		0		
24	40932	23078	O		176808.89	2021-05-12
	00:00:00	3-MEDIUM		0		
25	42817	21661	O		44438.99	2020-08-31
	00:00:00	5-LOW		0		
26	47142	19433	O		73866.30	2019-03-24
	00:00:00	1-URGENT		0		
27	60419	21305	F		114536.40	2015-06-20
	00:00:00	4-NOT SPECIFIED		0		
28	60867	22738	O		92114.35	2020-07-27
	00:00:00	3-MEDIUM		0		
29	64612	20027	F		307272.91	2016-12-26
	00:00:00	4-NOT SPECIFIED		0		
30	66470	1373	F		68381.45	2016-05-10
	00:00:00	3-MEDIUM		0		
31	66531	23941	F		101155.27	2015-10-20
	00:00:00	3-MEDIUM		0		
32	84197	28901	O		79241.84	2019-05-22
	00:00:00	5-LOW		0		
33	95623	20209	O		99889.91	2020-12-10
	00:00:00	2-HIGH		0		
34	96870	13928	F		117595.25	2015-01-07
	00:00:00	1-URGENT		0		
35	100611	6268	O		119354.28	2019-02-14
	00:00:00	5-LOW		0		
36	105920	4939	F		113306.11	2016-06-20
	00:00:00	2-HIGH		0		
37	112965	472	F		15036.22	2016-08-11
	00:00:00	3-MEDIUM		0		
38	114599	28057	F		61960.06	2015-04-24
	00:00:00	4-NOT SPECIFIED		0		

查询2:

从订单明细表 `LINEITEM` 中，找出满足下列条件的所有订单 `L_ORDERKEY`：

- (1) 数量位于 [起始数量 30, 结束数量 50]，
- (2) 退货标志为 '`N`' 的订单中，价格不小于 最低价格 20000

列出这些订单的 `L_ORDERKEY`、`L_SUPPKEY`、`L_EXTENDEDPRICE`；要求：对查询结果，按照价格从高到低排序，并且对查询结果使用 `DISTINCT` 去重。

比较对查询结果去重和不去重，在查询时间和查询结果上的差异。

```
1 -- 去重查询
2 EXPLAIN ANALYZE
3 SELECT DISTINCT L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE
4 FROM LINEITEM
5 WHERE L_QUANTITY BETWEEN 30 AND 50
6     AND L_RETURNFLAG = 'N'
7     AND L_EXTENDEDPRICE >= 96000
8 ORDER BY L_EXTENDEDPRICE DESC;
9
10 -- 不去重查询
11 EXPLAIN ANALYZE
12 SELECT L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE
13 FROM LINEITEM
14 WHERE L_QUANTITY BETWEEN 30 AND 50
15     AND L_RETURNFLAG = 'N'
16     AND L_EXTENDEDPRICE >= 96000
17 ORDER BY L_EXTENDEDPRICE DESC;
```

去重

查询结果

```
1 omm=# SELECT DISTINCT L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE
2 FROM LINEITEM
3 WHERE L_QUANTITY BETWEEN 30 AND 50
4     AND L_RETURNFLAG = 'N'
5     AND L_EXTENDEDPRICE >= 96000
6 ORDER BY L_EXTENDEDPRICE DESC;omm-# omm-# omm-# omm-# omm-#
7 l_orderkey | l_suppkey | l_extendedprice
8 -----+-----+-----
9      549057 |      2000 |    96949.50
10     272229 |       557 |    96899.50
11     705441 |      1999 |    96899.50
12     719041 |      1519 |    96899.50
13    1028352 |      1036 |    96849.50
14     814209 |      1999 |    96799.50
15    1114084 |       517 |    96799.50
16     111329 |      1553 |    96749.50
17     456194 |        35 |    96749.50
```

18	747776	1552	96749.50
19	163712	516	96699.50
20	323143	1516	96699.50
21	10308	999	96649.50
22	330503	1032	96649.50
23	916000	1998	96649.50
24		
25	244676	506	96149.00
26	428514	1984	96149.00
27	868550	1538	96149.00
28	1101380	23	96149.00
29	293607	537	96099.50
30	400421	1509	96099.50
31	787011	537	96099.50
32	1096550	533	96099.50
33	265669	1538	96099.00
34	368230	540	96099.00
35	787937	505	96099.00
36	892805	984	96099.00
37	933921	1021	96099.00
38	603969	991	96049.50
39	177411	505	96049.00
40	334182	1535	96049.00
41	591331	1020	96049.00
42	591716	1537	96049.00
43	610368	502	96049.00
44	(70 rows)		
45			

查询时间: 223.757 ms

```

1 omm=# EXPLAIN ANALYZE
2 SELECT DISTINCT L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE
3 FROM LINEITEM
4 WHERE L_QUANTITY BETWEEN 30 AND 50
5     AND L_RETURNFLAG = 'N'
6     AND L_EXTENDEDPRICE >= 96000
7 ORDER BY L_EXTENDEDPRICE DESC;omm-# omm-# omm-# omm-# omm-#
8
9         QUERY PLAN
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45

```

```

14      -> Seq Scan on lineitem (cost=0.00..48915.38 rows=26
15 width=16) (actual time=5.454..223.591 rows=70 loops=1)
16             Filter: ((l_quantity >= 30::numeric) AND (l_quantity
17 <= 50::numeric) AND (l_extendedprice >= 96000::numeric) AND
18 (l_returnflag = 'N'::bpchar))
19             Rows Removed by Filter: 1199899
20
21 Total runtime: 223.757 ms

```

不去重

查询结果

```

1 omm=# SELECT L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE
2 FROM LINEITEM
3 WHERE L_QUANTITY BETWEEN 30 AND 50
4     AND L_RETURNFLAG = 'N'
5     AND L_EXTENDEDPRICE >= 96000
6 ORDER BY L_EXTENDEDPRICE DESC;omm-# omm-# omm-# omm-#
7 l_orderkey | l_suppkey | l_extendedprice
8
9 549057 | 2000 | 96949.50
10 272229 | 557 | 96899.50
11 719041 | 1519 | 96899.50
12 705441 | 1999 | 96899.50
13 1028352 | 1036 | 96849.50
14 814209 | 1999 | 96799.50
15 1114084 | 517 | 96799.50
16 456194 | 35 | 96749.50
17 747776 | 1552 | 96749.50
18 111329 | 1553 | 96749.50
19 323143 | 1516 | 96699.50
20 .....
21 578660 | 1509 | 96199.50
22 8134 | 988 | 96199.00
23 809378 | 1022 | 96149.50
24 965285 | 1535 | 96149.50
25 1075681 | 1510 | 96149.50
26 1101380 | 23 | 96149.00
27 868550 | 1538 | 96149.00
28 244676 | 506 | 96149.00
29 428514 | 1984 | 96149.00
30 400421 | 1509 | 96099.50
31 787011 | 537 | 96099.50
32 293607 | 537 | 96099.50
33 1096550 | 533 | 96099.50
34 933921 | 1021 | 96099.00
35 892805 | 984 | 96099.00
36 787937 | 505 | 96099.00
37 265669 | 1538 | 96099.00
38 368230 | 540 | 96099.00

```

```

39      603969 |      991 |    96049.50
40      610368 |      502 |    96049.00
41      591716 |     1537 |    96049.00
42      591331 |     1020 |    96049.00
43      334182 |     1535 |    96049.00
44      177411 |      505 |    96049.00
45 (70 rows)

```

查询时间: 223.187 ms

```

1 omm=# EXPLAIN ANALYZE
2 SELECT L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE
3 FROM LINEITEM
4 WHERE L_QUANTITY BETWEEN 30 AND 50
5   AND L_RETURNFLAG = 'N'
6   AND L_EXTENDEDPRICE >= 96000
7 ORDER BY L_EXTENDEDPRICE DESC;omm-# omm-# omm-# omm-# omm-#
8
9       QUERY PLAN
10 -----
11
12 -----
13
14
15
16
17

```

```

Sort  (cost=48915.99..48916.06 rows=26 width=16) (actual
time=223.107..223.110 rows=70 loops=1)
  Sort Key: l_extendedprice DESC
  Sort Method: quicksort  Memory: 29kB
  -> Seq Scan on lineitem  (cost=0.00..48915.38 rows=26 width=16)
    (actual time=5.467..223.043 rows=70 loops=1)
        Filter: ((l_quantity >= 30::numeric) AND (l_quantity <=
50::numeric) AND (l_extendedprice >= 96000::numeric) AND
(l_returnflag = 'N'::bpchar))
        Rows Removed by Filter: 1199899
  Total runtime: 223.187 ms
  (7 rows)

```

2.2 字符串操作

查询3:

从客户表 CUSTOMER 中, 找出满足下列条件的客户:

- (1) 客户电话开头部分包含 '10', 或者客户市场领域中包含 'BUILDING', 并且
- (2) 客户电话结尾不为 '8'

```

1 SELECT C_CUSTKEY, C_NAME
2 FROM CUSTOMER
3 WHERE (C_PHONE LIKE '10%' OR C_MKTSEGMENT LIKE '%BUILDING%')
4   AND C_PHONE NOT LIKE '%8';

```

实验结果

```
1 | c_custkey | c_name
2 |-----+-----|
3 | 2 | Customer#000000002
4 | 3 | Customer#000000003
5 | 4 | Customer#000000004
6 | 5 | Customer#000000005
7 | 6 | Customer#000000006
8 | 7 | Customer#000000007
9 | 8 | Customer#000000008
10 | 9 | Customer#000000009
11 | 10 | Customer#000000010
12 | .....
13 | 1955 | Customer#000001955
14 | 1956 | Customer#000001956
15 | 1958 | Customer#000001958
16 | 1959 | Customer#000001959
17 | 1960 | Customer#000001960
18 | 1962 | Customer#000001962
19 | 1963 | Customer#000001963
20 | 1964 | Customer#000001964
21 | 1965 | Customer#000001965
22 | 1966 | Customer#000001966
23 | 1967 | Customer#000001967
24 | .....
```

查询4：

从客户表 CUSTOMER 中，找出满足下列条件的客户姓名：

- (1) 客户 key 由 2 个字符组成
- (2) 客户地址至少包括 18 个字符，即地址字符串的长度不小于 18。

```
1 | SELECT C_NAME
2 | FROM CUSTOMER
3 | WHERE C_CUSTKEY::TEXT LIKE '__' -- 假设C_CUSTKEY为整数，需要转换为文本比较
4 | AND LENGTH(C_ADDRESS) >= 18;
```

实验结果

```
1 | c_name
2 |-----|
3 | Customer#000000010
4 | Customer#000000012
5 | Customer#000000013
6 | Customer#000000014
```

```
7 Customer#000000015
8 Customer#000000017
9 Customer#000000018
10 Customer#000000019
11 Customer#000000020
12 Customer#000000022
13 Customer#000000023
14 Customer#000000024
15 .....
16 Customer#000000084
17 Customer#000000085
18 Customer#000000087
19 Customer#000000088
20 Customer#000000089
21 Customer#000000091
22 Customer#000000092
23 Customer#000000093
24 Customer#000000094
25 Customer#000000095
26 Customer#000000096
27 Customer#000000097
28 Customer#000000098
29 Customer#000000099
30 ( 64 rows)
```

2.3 集合操作

查询5：

使用集合并操作 `UNION`、`UNION ALL`，从订单明细表 `LINEITEM` 查询满足下列条件的订单 `L_ORDERKEY`：

- (1) 订单发货日期早于 `'2016-01-01'`，或者
- (2) 订单数量大于 `100`

对比 `UNION ALL`、`UNION` 操作在查询结果、执行时间上的差异。

```
1 -- 使用 UNION ALL
2 EXPLAIN ANALYZE
3 SELECT L_ORDERKEY
4 FROM LINEITEM
5 WHERE L_SHIPDATE < '2016-01-01' :: DATE
6
7 UNION ALL
8
9 SELECT L_ORDERKEY
10 FROM LINEITEM
11 WHERE L_QUANTITY > 100;
12
```

```
13 -- 使用 UNION
14 EXPLAIN ANALYZE
15 SELECT L_ORDERKEY
16 FROM LINEITEM
17 WHERE L_SHIPDATE < '2016-01-01'::DATE
18
19 UNION
20
21 SELECT L_ORDERKEY
22 FROM LINEITEM
23 WHERE L_QUANTITY > 100;
```

使用 UNION ALL

实验结果

```
1 l_orderkey
2 -----
3      6
4      37
5      37
6      37
7     128
8     129
9     129
10 .....
11    1504
12    1504
13    1505
14    1505
15    1506
16    1506
17    1506
18    1506
19    1506
20    1537
21    1537
22 .....
23    5218
24    5220
25    5254
26    5254
27    5254
28    5254
29    5254
30    5254
31 .....
```

执行时间: 365.901 ms

```
1          QUERY
2
3 PLAN
4 -----
5
6
7
8
9
10
11
12
```

Result (cost=0.00..81345.32 rows=151409 width=4) (actual time=0.057..361.636 rows=151587 loops=1)
-> Append (cost=0.00..81345.32 rows=151409 width=4) (actual time=0.056..352.949 rows=151587 loops=1)
 -> Seq Scan on lineitem (cost=0.00..39915.61 rows=151408 width=4) (actual time=0.055..167.003 rows=151587 loops=1)
 Filter: (l_shipdate < '2016-01-01 00:00:00'::timestamp(0) without time zone)
 Rows Removed by Filter: 1048382
 -> Seq Scan on lineitem (cost=0.00..39915.61 rows=1 width=4) (actual time=178.241..178.241 rows=0 loops=1)
 Filter: (l_quantity > 100::numeric)
 Rows Removed by Filter: 1199969
Total runtime: 365.901 ms
(9 rows)

使用 UNION

实验结果

```
1 l_orderkey
2 -----
3     865029
4     370596
5     310753
6     363111
7     1096454
8     728802
9     .....
10    834023
11    733601
12    344993
13    970624
14    104033
15    738276
16    821126
17    1147808
18    551749
19    1083878
20    361732
21    .....
```

执行时间: 364.971 ms

```

1 omm=# EXPLAIN ANALYZE
2 SELECT L_ORDERKEY
3 FROM LINEITEM
4 WHERE L_SHIPDATE < '2016-01-01'::DATE
5
6 UNION
7
8 SELECT L_ORDERKEY
9 FROM LINEITEM
10 WHERE L_QUANTITY > 100;omm-# omm-# omm-# omm-# omm-# omm-# omm-#
11 omm-# omm-#                                     QUERY
12 PLAN
13 -----
14 HashAggregate  (cost=81723.84..83237.93 rows=151409 width=4)
15 (actual time=359.014..363.225 rows=41621 loops=1)
16     Group By Key: public.lineitem.l_orderkey
17     -> Append  (cost=0.00..81345.32 rows=151409 width=4) (actual
18         time=0.069..337.867 rows=151587 loops=1)
19             -> Seq Scan on lineitem  (cost=0.00..39915.61 rows=151408
20                 width=4) (actual time=0.066..155.120 rows=151587 loops=1)
21                 Filter: (l_shipdate < '2016-01-01
22 00:00:00'::timestamp(0) without time zone)
23                 Rows Removed by Filter: 1048382
24             -> Seq Scan on lineitem  (cost=0.00..39915.61 rows=1
25                 width=4) (actual time=174.747..174.747 rows=0 loops=1)
26                 Filter: (l_quantity > 100::numeric)
27                 Rows Removed by Filter: 1199969
28
29 Total runtime: 364.971 ms
30 (10 rows)

```

查询6：

结合教材 3.4.1 节元组变量样例，使用集合操作 `EXCEPT`、`EXCEPT ALL`，从供应商表 `SUPPLIER` 中，查询账户余额最大的供应商。

对比使用 `EXCEPT`、`EXCEPT ALL`、聚集函数 `MAX`，完成此查询在执行时间、查询结果上的异同。

```

1 -- 更新统计信息
2 ANALYZE supplier;
3
4 -- 使用 EXCEPT
5 EXPLAIN ANALYZE
6 SELECT s_suppkey, s_name
7 FROM supplier
8 EXCEPT
9 (
10     SELECT t1.s_suppkey, t1.s_name

```

```

11      FROM supplier t1
12      JOIN supplier t2 ON t1.s_acctbal < t2.s_acctbal
13  );
14
15 -- 使用 EXCEPT ALL
16 EXPLAIN ANALYZE
17 SELECT s_suppkey, s_name
18 FROM supplier
19 EXCEPT ALL
20 (
21     SELECT t1.s_suppkey, t1.s_name
22     FROM supplier t1
23     JOIN supplier t2 ON t1.s_acctbal < t2.s_acctbal
24 );
25
26 -- 使用 MAX 聚集函数
27 EXPLAIN ANALYZE
28 SELECT s_suppkey, s_name
29 FROM supplier
30 WHERE s_acctbal = (
31     SELECT MAX(s_acctbal)
32     FROM supplier
33 );

```

使用 EXCEPT

实验结果

s_suppkey	s_name
892	Supplier#000000892

(1 row)

执行时间: 1001.204 ms

```

1
2   QUERY PLAN
3
4
5   HashSetOp Except  (cost=0.00..80220.99 rows=2000 width=30) (actual
6   time=1001.026..1001.039 rows=1 loops=1)
7       -> Append  (cost=0.00..73544.33 rows=1335333 width=30) (actual
8       time=0.018..788.066 rows=2000997 loops=1)
9           -> Subquery Scan on "*SELECT* 1"  (cost=0.00..82.00
10          rows=2000 width=30) (actual time=0.017..1.341 rows=2000 loops=1)
11              -> Seq Scan on supplier  (cost=0.00..62.00
12              rows=2000 width=30) (actual time=0.014..0.802 rows=2000 loops=1)

```

```
7      -> Subquery Scan on "*SELECT* 2" (cost=0.00..73462.33
rows=1333333 width=30) (actual time=0.034..700.892 rows=1998997
loops=1)
8          -> Nested Loop (cost=0.00..60129.00 rows=1333333
width=30) (actual time=0.033..586.081 rows=1998997 loops=1)
9              Join Filter: (t1.s_acctbal < t2.s_acctbal)
10             Rows Removed by Join Filter: 2001003
11             -> Seq Scan on supplier t1 (cost=0.00..62.00
rows=2000 width=36) (actual time=0.004..0.151 rows=2000 loops=1)
12                 -> Materialize (cost=0.00..72.00 rows=2000
width=6) (actual time=0.065..111.885 rows=4000000 loops=2000)
13                     -> Seq Scan on supplier t2
14                         (cost=0.00..62.00 rows=2000 width=6) (actual time=0.004..0.636
rows=2000 loops=1)
15 Total runtime: 1001.204 ms
(12 rows)
```

使用 EXCEPT ALL

实验结果

```
1   s_suppkey |          s_name
2   -----+-----+
3       892 | Supplier#000000892
4 (1 row)
```

执行时间: 995.772 ms

```
1 QUERY PLAN
2 -----
3 HashSetOp Except All  (cost=0.00..80220.99 rows=2000 width=30)
4   (actual time=995.580..995.594 rows=1 loops=1)
5     -> Append  (cost=0.00..73544.33 rows=1335333 width=30) (actual
6       time=0.020..783.643 rows=2000997 loops=1)
7       -> Subquery Scan on "*SELECT* 1"  (cost=0.00..82.00
8         rows=2000 width=30) (actual time=0.020..1.236 rows=2000 loops=1)
9         -> Seq Scan on supplier  (cost=0.00..62.00
10        rows=2000 width=30) (actual time=0.017..0.711 rows=2000 loops=1)
11         -> Subquery Scan on "*SELECT* 2"  (cost=0.00..73462.33
12        rows=1333333 width=30) (actual time=0.032..697.765 rows=1998997
13        loops=1)
14           -> Nested Loop  (cost=0.00..60129.00 rows=1333333
15             width=30) (actual time=0.032..583.712 rows=1998997 loops=1)
16               Join Filter: (t1.s_acctbal < t2.s_acctbal)
17               Rows Removed by Join Filter: 2001003
```

```

11          -> Seq Scan on supplier t1 (cost=0.00..62.00
12 rows=2000 width=36) (actual time=0.004..0.189 rows=2000 loops=1)
13             -> Materialize (cost=0.00..72.00 rows=2000
14 width=6) (actual time=0.061..110.777 rows=4000000 loops=2000)
15                 -> Seq Scan on supplier t2
16 (cost=0.00..62.00 rows=2000 width=6) (actual time=0.004..0.594
17 rows=2000 loops=1)
18 Total runtime: 995.772 ms
19 (12 rows)

```

使用 MAX 聚集函数

实验结果

	s_suppkey	s_name
3	892	Supplier#000000892
4	(1 row)	

执行时间: 2.550 ms

	QUERY PLAN
3	Seq Scan on supplier (cost=67.01..134.01 rows=1 width=30) (actual time=1.894..2.413 rows=1 loops=1) Filter: (s_acctbal = \$0) Rows Removed by Filter: 1999 InitPlan 1 (returns \$0) -> Aggregate (cost=67.00..67.01 rows=1 width=38) (actual time=1.416..1.416 rows=1 loops=1) -> Seq Scan on supplier (cost=0.00..62.00 rows=2000 width=6) (actual time=0.005..0.556 rows=2000 loops=1) Total runtime: 2.550 ms (7 rows)

2.4 多表查询

查询7:

选取两张数据量比较小的表 T1 和 T2 (如 REGION、NATION、SUPPLIER)，执行如下无连接条件的笛卡尔积操作，观察数据库系统的反应和查询结果：

```

1 | SELECT *
2 | FROM REGION, NATION;

```

实验结果

	r_regionkey		r_name		r_comment
	n_nationkey		n_name		n_regionkey
					n_comment
2	-----+-----+		-----+-----+		-----+-----+
3	0 AFRICA			furiously special foxes	
	hagg 0 ALGERIA				0
	posits use carefully pending accounts. special deposits haggle.				
	ironic, silent accounts are furio				
4	1 AMERICA			furiously special foxes	
	hagg 0 ALGERIA				0
	posits use carefully pending accounts. special deposits haggle.				
	ironic, silent accounts are furio				
5	2 ASIA			furiously special foxes	
	hagg 0 ALGERIA				0
	posits use carefully pending accounts. special deposits haggle.				
	ironic, silent accounts are furio				
6	3 EUROPE			furiously special foxes	
	hagg 0 ALGERIA				0
	posits use carefully pending accounts. special deposits haggle.				
	ironic, silent accounts are furio				
7	4 MIDDLE EAST			furiously special foxes	
	hagg 0 ALGERIA				0
	posits use carefully pending accounts. special deposits haggle.				
	ironic, silent accounts are furio				
8	0 AFRICA			furiously special foxes	
	hagg 1 ARGENTINA				1 ly
	bold instructions haggle quickly across the blithely close dep				
9	1 AMERICA			furiously special foxes	
	hagg 1 ARGENTINA				1 ly
	bold instructions haggle quickly across the blithely close dep				
10	2 ASIA			furiously special foxes	
	hagg 1 ARGENTINA				1 ly
	bold instructions haggle quickly across the blithely close dep				
11	3 EUROPE			furiously special foxes	
	hagg 1 ARGENTINA				1 ly
	bold instructions haggle quickly across the blithely close dep				
12	4 MIDDLE EAST			furiously special foxes	
	hagg 1 ARGENTINA				1 ly
	bold instructions haggle quickly across the blithely close dep				
13				
14	3 EUROPE			furiously special foxes	
	hagg 10 IRAN				4
	equests. packages are ironic, regular theodolites. carefully				
	regular ideas sleep slyly final, ex				

```

15      4 | MIDDLE EAST          | furiously special foxes
hagg |       10 | IRAN           | 4 |
equests. packages are ironic, regular theodolites. carefully
regular ideas sleep slyly final, ex
16      0 | AFRICA              | furiously special foxes
hagg |       11 | IRAQ            | 4 |
cording to the quickly regular platelets. carefully ironic pinto
beans against the slyly unusual theodolites d
17      1 | AMERICA              | furiously special foxes
hagg |       11 | IRAQ            | 4 |
cording to the quickly regular platelets. carefully ironic pinto
beans against the slyly unusual theodolites d
18      2 | ASIA                 | furiously special foxes
hagg |       11 | IRAQ            | 4 |
cording to the quickly regular platelets. carefully ironic pinto
beans against the slyly unusual theodolites d
19      3 | EUROPE                | furiously special foxes
hagg |       11 | IRAQ            | 4 |
cording to the quickly regular platelets. carefully ironic pinto
beans against the slyly unusual theodolites d
20      .....
21 (125 row)

```

查询8：

使用多表连接操作，从订单表 `ORDERS`、供应商表 `SUPPLIER`、订单明细表 `LINEITEM` 中，
查询实际到达日期小于预计到达日期的订单，列出这些订单的订单 `key`、订单总价、下单日期以及该供应商的姓名、地址和手机号。

```

1 SELECT O.O_ORDERKEY, O.O_TOTALPRICE, O.O_ORDERDATE, S.S_NAME,
S.S_ADDRESS, S.S_PHONE
2 FROM ORDERS O
3 JOIN LINEITEM L ON O.O_ORDERKEY = L.L_ORDERKEY
4 JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
5 WHERE L.L_RECEIPTDATE < L.L_COMMITDATE;

```

实验结果

```

1   o_orderkey | o_totalprice |      o_orderdate      |           s_name
2   | s_address  |      s_phone
3   -----
3   59108 | 268538.27 | 2018-10-08 00:00:00 |
4   Supplier#000001022 | 0000000000 | 24-859-889-7512
4   66787 | 233043.61 | 2015-12-12 00:00:00 |
5   Supplier#000001512 | 0000000000 | 33-670-389-3311
5   85475 | 217043.30 | 2017-05-24 00:00:00 |
6   Supplier#000000994 | 0000000000 | 14-183-331-6019
6   96772 | 255936.60 | 2017-03-04 00:00:00 |
7   Supplier#000001303 | 0000000000 | 22-688-457-2776
7   98022 | 331558.89 | 2017-06-07 00:00:00 |
8   Supplier#000001321 | 0000000000 | 32-708-579-1992
8   .....
9   ( 56424 row)

```

查询9：

使用多表连接操作，从供应商表 `SUPPLIER`、零部件表 `PART`、零部件供应表 `PARTSUPP` 中，查询供应零件品牌为 `'Brand#13'` 的供应商信息，列出零件供应数量与成本，以及供应商的姓名与手机号。

```

1   SELECT PS.PS_AVAILQTY, PS.PS_SUPPLYCOST, S.S_NAME, S.S_PHONE
2   FROM PARTSUPP PS
3   JOIN SUPPLIER S ON PS.PS_SUPPKEY = S.S_SUPPKEY
4   JOIN PART P ON PS.PS_PARTKEY = P.P_PARTKEY
5   WHERE P.P_BRAND = 'Brand#13';

```

实验结果

```

1 | ps_availqty | ps_supplycost |           s_name          |
2 | s_phone      |
2 |-----+-----+-----+
3 |     1 |    771.64 | Supplier#00000002 | 15-679-
4 | 861-2259   |
4 |     1 |    993.49 | Supplier#000000502 | 14-678-
5 | 262-5636   |
5 |     1 |    337.09 | Supplier#000001002 | 32-102-
6 | 374-6308   |
6 |     1 |    357.84 | Supplier#000001502 | 12-226-
7 | 454-8297   |
7 |     1 |    378.49 | Supplier#000000003 | 11-383-
8 | 516-1199   |
8 |     1 |    915.27 | Supplier#000000503 | 30-263-
9 | 152-1630   |
9 |     1 |    438.37 | Supplier#000001003 | 20-763-
10| 167-9528  |
10| .....
```

11 (6424 row)

查询10：

利用订单明细表 LINEITEM，使用元组变量方式，查询所有比流水号为 '1'，订单号为 '1' 的折扣高的订单 key 和流水号，列出这些订单的零件、折扣，结果按照折扣的降序排列。

```

1 | SELECT L1.L_ORDERKEY, L1.L_LINENUMBER, L1.L_PARTKEY, L1.L_DISCOUNT
2 | FROM LINEITEM L1
3 | WHERE L1.L_DISCOUNT > (
4 |   SELECT L2.L_DISCOUNT
5 |   FROM LINEITEM L2
6 |   WHERE L2.L_ORDERKEY = '1' AND L2.L_LINENUMBER = '1'
7 | )
8 | ORDER BY L1.L_DISCOUNT DESC;
```

实验结果

l_orderkey	l_linenumber	l_partkey	l_discount
940324	1	29535	.10
1114981	1	914	.10
831842	3	34597	.10
172641	3	9552	.10
411648	2	8455	.10
179014	4	18310	.10
683425	7	15672	.10
1109477	4	31501	.10
1109476	4	1137	.10
.....			
(493904 row)			

2.5 聚集函数

查询11：

从订单明细表 LINEITEM、订单表 ORDERS、客户表 CUSTOMER、国家表 NATION，查询客户来自 ALGERIA，下单日期为 '2015-01-01' 到 '2015-02-02' 的订单下列信息：

- (1) 满足条件订单的最大数量、最小数量和平均数量。
- (2) 具有最大数量且满足上述条件的订单，列出该订单的发货日期、下单日期。

```

1  -- (1)
2  SELECT MAX(L.L_QUANTITY) AS MAX_QTY, MIN(L.L_QUANTITY) AS MIN_QTY,
3    AVG(L.L_QUANTITY) AS AVG_QTY
4  FROM LINEITEM L
5  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
6  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
7  JOIN NATION N ON C.C_NATIONKEY = N.N_NATIONKEY
8  WHERE N.N_NAME = 'ALGERIA'
9    AND O.O_ORDERDATE BETWEEN '2015-01-01'::DATE AND '2015-02-
10   02'::DATE;
11
12  -- (2)
13  SELECT L.L_QUANTITY, L.L_SHIPDATE, O.O_ORDERDATE
14  FROM LINEITEM L
15  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
16  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
17  JOIN NATION N ON C.C_NATIONKEY = N.N_NATIONKEY
18  WHERE N.N_NAME = 'ALGERIA'
19    AND O.O_ORDERDATE BETWEEN '2015-01-01'::DATE AND '2015-02-
      02'::DATE
20  ORDER BY L.L_QUANTITY DESC
21  LIMIT 1;

```

实验结果-- (1)

	max_qty	min_qty	avg_qty
1			
2			
3	50.00	1.00	25.5653962492437992
4	(1 row)		

实验结果--(2)

	l_quantity	l_shipdate	o_orderdate
1			
2			
3	50.00	2015-02-02 00:00:00	2015-01-05 00:00:00
4	(1 row)		

查询12:

根据零部件表 PART 和零部件供应表 PARTSUPP 及供应商表 SUPPLIER，查询有多少零件厂商提供了品牌为 Brand#13 的零件，给出这些零件的类型、零售价和供应商数量，并将查询结果按照零售价降序排列。

```

1  SELECT P.P_TYPE, P.P_RETAILPRICE, COUNT(DISTINCT S.S_SUPPKEY) AS
2    SUPPLIER_COUNT
3  FROM PART P
4  JOIN PARTSUPP PS ON P.P_PARTKEY = PS.PS_PARTKEY
5  JOIN SUPPLIER S ON PS.PS_SUPPKEY = S.S_SUPPKEY
6  WHERE P.P_BRAND = 'Brand#13'
7  GROUP BY P.P_TYPE, P.P_RETAILPRICE
8  ORDER BY P.P_RETAILPRICE DESC

```

实验结果

	p_type	p_retailprice	supplier_count
1			
2			
3	STANDARD ANODIZED TIN	1932.99	8
4	STANDARD ANODIZED TIN	1930.99	4
5	STANDARD ANODIZED TIN	1929.99	4
6	STANDARD ANODIZED TIN	1923.99	8
7		
8	(1349 row)		

查询13:

从零部件表 PART 和零部件供应表 PARTSUPP 中，查询所有零件大小在 [7,14] 之间的零件的平均零售价，给出零件 key，供应成本，平均零售价，结果按照零售价降序排列。

```

1 | SELECT P.P_PARTKEY, PS.PS_SUPPLYCOST, AVG(P.P_RETAILPRICE) AS
2 | AVG_RETAILPRICE
3 | FROM PART P
4 | JOIN PARTSUPP PS ON P.P_PARTKEY = PS.PS_PARTKEY
5 | WHERE P.P_SIZE BETWEEN 7 AND 14
6 | GROUP BY P.P_PARTKEY, PS.PS_SUPPLYCOST
7 | ORDER BY AVG_RETAILPRICE DESC;

```

实验结果

	p_partkey	ps_supplycost	avg_retailprice
3	39998	254.68	1937.99000000000000000000
4	39998	711.23	1937.99000000000000000000
5	39998	755.19	1937.99000000000000000000
6	39998	880.04	1937.99000000000000000000
7	35999	215.76	1934.99000000000000000000
8	35999	154.96	1934.99000000000000000000
9		
10	22998	455.36	1920.99000000000000000000
11	24996	698.30	1920.99000000000000000000
12	21999	71.50	1920.99000000000000000000
13	22998	916.71	1920.99000000000000000000
14	22998	11.62	1920.99000000000000000000
15	24996	152.65	1920.99000000000000000000
16		
17	(26084 row)		

2.6 嵌套查询

查询14：

从订单明细表 `LINEITEM`、订单表 `ORDERS`、客户表 `CUSTOMER` 中，使用 `IN` 运算符，查询明细折扣小于 `0.01` 的订单，列出这些订单的 `key` 和采购订单的客户姓名。

对比使用多表连接、非嵌套的查询在执行时间、查询结果上的异同。

```

1 | -- 使用嵌套查询
2 | EXPLAIN ANALYZE
3 | SELECT O.O_ORDERKEY, C.C_NAME
4 | FROM ORDERS O
5 | JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
6 | WHERE O.O_ORDERKEY IN (
7 |     SELECT L.L_ORDERKEY
8 |     FROM LINEITEM L
9 |     WHERE L.L_DISCOUNT < 0.01
10| );
11|

```

```
12 -- 使用多表连接
13 EXPLAIN ANALYZE
14 SELECT DISTINCT O.O_ORDERKEY, C.C_NAME
15 FROM ORDERS O
16 JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
17 JOIN LINEITEM L ON O.O_ORDERKEY = L.L_ORDERKEY
18 WHERE L.L_DISCOUNT < 0.01;
```

使用嵌套查询

实验结果

o_orderkey	c_name
2	Customer#000015601
34	Customer#000012202
65	Customer#000003251
71	Customer#000000676
98	Customer#000020896
100	Customer#000029401
101	Customer#000005600
133	Customer#000008800
.....	
2215	Customer#000007738
2241	Customer#000020257
2273	Customer#000026851
2304	Customer#000008986
2305	Customer#000008395
2306	Customer#000005342
.....	
4994	Customer#000008488
4995	Customer#000007742
4996	Customer#000026572
4997	Customer#000009260
5025	Customer#000023927
5027	Customer#000029248
5029	Customer#000002077
.....	

执行时间: 403.720 ms

```
1 QUERY PLAN
2 -----
3 Hash Join  (cost=99251.34..108238.79 rows=62217 width=23) (actual
   time=328.351..401.525 rows=70400 loops=1)
4   Hash Cond: (o.o_custkey = c.c_custkey)
```

```

5      -> Hash Join  (cost=97857.34..105989.30 rows=62217 width=8)
6        (actual time=316.412..376.821 rows=70400 loops=1)
7          Hash Cond: (o.o_orderkey = l.l_orderkey)
8            -> Seq Scan on orders o  (cost=0.00..7286.00 rows=300000
9              width=8) (actual time=0.004..25.497 rows=300001 loops=1)
10             -> Hash  (cost=97737.39..97737.39 rows=9596 width=4)
11               (actual time=316.189..316.189 rows=70400 loops=1)
12                 Buckets: 32768  Batches: 1  Memory Usage: 2475kB
13               -> HashAggregate  (cost=97641.43..97737.39
14                 rows=9596 width=4) (actual time=303.742..310.659 rows=70400
15                 loops=1)
16                   Group By Key: l.l_orderkey
17                   -> Seq Scan on lineitem l
18                     (cost=0.00..97408.30 rows=93252 width=4) (actual
19                     time=0.063..287.357 rows=81498 loops=1)
20                         Filter: (l_discount < .01)
21                         Rows Removed by Filter: 822355
22               -> Hash  (cost=1019.00..1019.00 rows=30000 width=23) (actual
23                 time=11.747..11.747 rows=30000 loops=1)
24                   Buckets: 32768  Batches: 1  Memory Usage: 1641kB
25                   -> Seq Scan on customer c  (cost=0.00..1019.00 rows=30000
26                     width=23) (actual time=0.010..6.179 rows=30000 loops=1)
27                     Total runtime: 403.720 ms
28
29  (16 rows)

```

使用多表连接

实验结果

	o_orderkey c_name
1	o_orderkey c_name
2	-----+-----
3	674469 Customer#00002546
4	835143 Customer#00003364
5	964549 Customer#000026950
6	1049190 Customer#000003877
7	223141 Customer#000025421
8
9	49056 Customer#000013006
10	526529 Customer#000022004
11	599013 Customer#000022811
12	255523 Customer#000009337
13	703137 Customer#000007178
14	822976 Customer#000026330
15
16	393604 Customer#000012704
17	289568 Customer#000015125
18	231591 Customer#000023413
19	1067269 Customer#000004435
20	680103 Customer#000025159

执行时间: 452.057 ms

```
1
2     QUERY PLAN
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
-- HashAggregate (cost=112868.99..113801.51 rows=93252 width=23)
-- (actual time=440.946..448.941 rows=70400 loops=1)
  Group By Key: o.o_orderkey, c.c_name
    -> Hash Join (cost=12430.00..112402.73 rows=93252 width=23)
      (actual time=80.918..421.119 rows=81498 loops=1)
        Hash Cond: (o.o_custkey = c.c_custkey)
          -> Hash Join (cost=11036.00..109726.52 rows=93252
            width=8) (actual time=69.279..390.006 rows=81498 loops=1)
            Hash Cond: (l.l_orderkey = o.o_orderkey)
              -> Seq Scan on lineitem l (cost=0.00..97408.30
                rows=93252 width=4) (actual time=0.037..294.453 rows=81498 loops=1)
                Filter: (l_discount < .01)
                Rows Removed by Filter: 822355
              -> Hash (cost=7286.00..7286.00 rows=300000
                width=8) (actual time=67.846..67.846 rows=300001 loops=1)
                Buckets: 524288 Batches: 1 Memory Usage:
                11719kB
                -> Seq Scan on orders o (cost=0.00..7286.00
                  rows=300000 width=8) (actual time=0.008..37.125 rows=300001
                  loops=1)
                  -> Hash (cost=1019.00..1019.00 rows=30000 width=23)
                  (actual time=11.364..11.364 rows=30000 loops=1)
                  Buckets: 32768 Batches: 1 Memory Usage: 1641kB
                  -> Seq Scan on customer c (cost=0.00..1019.00
                    rows=30000 width=23) (actual time=0.011..6.277 rows=30000 loops=1)
                  Total runtime: 452.057 ms
                (16 rows)
```

查询15-1:

从订单明细表 LINEITEM，使用 SOME 运算符，查询满足下列条件的订单：该订单的数量大于发货日期在 [开始日期 2018-10-10，结束日期 2021-10-10] 之间的部分（至少一个）订单的数量，列出这些订单的流水号、key 和税。

```
1
2
3
4
5
6
7
```

```
SELECT L_ORDERKEY, L_LINENUMBER, L_TAX
FROM LINEITEM
WHERE L_QUANTITY > SOME (
  SELECT L_QUANTITY
  FROM LINEITEM
  WHERE L_SHIPDATE BETWEEN '2021-1-9'::DATE AND '2021-1-10'::DATE
);
```

实验结果

```
1 l_orderkey | l_linenumber | l_tax
2 -----
3 126017 | 3 | .03
4 126017 | 4 | .06
5 126017 | 5 | 0.00
6 126017 | 6 | .05
7 126018 | 1 | .07
8 126049 | 1 | 0.00
9 126080 | 1 | .02
10 .....
11 126884 | 4 | .07
12 126884 | 5 | .05
13 126885 | 1 | .08
14 126914 | 1 | 0.00
15 126914 | 2 | .01
16 .....
17 127524 | 1 | .04
18 127524 | 2 | .02
19 127524 | 3 | .08
20 127524 | 4 | .03
21 127524 | 5 | .03
22 .....
```

查询15-2：

从订单表 `ORDERS`，使用 `SOME` 运算符，查询满足下列条件的订单：订单状态为 `'O'`，订单总价大于部分在 `2020` 年之后下单的订单。列出这些订单的 `key`、客户 `key`、收银员。

```
1 SELECT O_ORDERKEY, O_CUSTKEY, O_CLERK
2 FROM ORDERS
3 WHERE O_ORDERSTATUS = 'O'
4 AND O_TOTALPRICE > SOME (
5     SELECT O_TOTALPRICE
6     FROM ORDERS
7     WHERE O_ORDERDATE >= '2020-01-01'::DATE
8 );
9 SELECT count(*)
10 FROM ORDERS
11 WHERE O_ORDERSTATUS = 'O'
12 AND O_TOTALPRICE > SOME (
13     SELECT O_TOTALPRICE
14     FROM ORDERS
15     WHERE O_ORDERDATE >= '2020-01-01'::DATE
16 );
```

实验结果

	<code>o_orderkey</code>	<code>o_custkey</code>	<code>o_clerk</code>
3	1	7381	Clerk#000000951
4	2	15601	Clerk#000000880
5	4	27356	Clerk#000000124
6	7	7828	Clerk#000000470
7	32	26012	Clerk#000000616
8	34	12202	Clerk#000000223
9		
10	(146319 row)		

查询16-1：

从订单明细表 `LINEITEM` 中，使用 `>= ALL` 运算符，查询满足下列条件的供应商：该供应商在 `2019` 年出货量大于等于同时段其他供应商的出货量，即 `2019` 年该供应商的出货量最高。

```

1  SELECT L.L_SUPPKEY
2  FROM LINEITEM L
3  WHERE L.L_SHIPDATE BETWEEN '2019-01-01'::DATE AND '2019-12-
31'::DATE
4  GROUP BY L.L_SUPPKEY
5  HAVING SUM(L.L_QUANTITY) >= ALL (
6      SELECT SUM(L2.L_QUANTITY)
7      FROM LINEITEM L2
8      WHERE L2.L_SHIPDATE BETWEEN '2019-01-01'::DATE AND '2019-12-
31'::DATE
9      GROUP BY L2.L_SUPPKEY
10 );

```

实验结果

	<code>l_suppkey</code>
3	370
4	(1 row)

查询16-2：

供应商表 `SUPPLIER`，使用 `ALL` 运算符，查询账户余额大于等于其他供应商的供应商。列出该供应商的姓名、`key`、手机号。

```

1 | SELECT S_SUPPKEY, S_NAME, S_PHONE
2 | FROM SUPPLIER
3 | WHERE S_ACCTBAL >= ALL (
4 |     SELECT S_ACCTBAL
5 |     FROM SUPPLIER
6 | );

```

实验结果

	s_suppkey	s_name	s_phone
1	892	Supplier#000000892	18-893-665-3629
2			
3			
4	(1 row)		

查询17-1：

从供应商表 `SUPPLIER`、国家表 `NATION`，使用 `EXISTS` 运算符，查询国家为日本，账户余额大于 `5000` 的供应商。

```

1 | SELECT S.S_SUPPKEY, S.S_NAME, S.S_ACCTBAL
2 | FROM SUPPLIER S
3 | WHERE S.S_NATIONKEY = (
4 |     SELECT N.N_NATIONKEY
5 |     FROM NATION N
6 |     WHERE N.N_NAME = 'JAPAN'
7 | )
8 | AND S.S_ACCTBAL > 5000;

```

实验结果

	s_suppkey	s_name	s_acctbal
1	43	Supplier#000000043	7773.41
2	143	Supplier#000000143	9658.99
3	163	Supplier#000000163	7999.27
4	173	Supplier#000000173	9583.11
5	175	Supplier#000000175	9845.98
6	215	Supplier#000000215	6125.89
7		
8	1568	Supplier#000001568	7834.92
9	1570	Supplier#000001570	7963.33
10	1614	Supplier#000001614	9896.02
11	1631	Supplier#000001631	7687.91
12	1638	Supplier#000001638	8611.17
13	1661	Supplier#000001661	6817.13
14	1681	Supplier#000001681	6144.37
15	1741	Supplier#000001741	5050.43
16	1862	Supplier#000001862	6697.54

```

19 |      1875 | Supplier#000001875      |    9358.58
20 |      1886 | Supplier#000001886      |   6449.94
21 (42 rows)
22

```

查询17-2:

从客户表 CUSTOMER、国家表 NATION、订单表 ORDERS、订单明细表 LINEITEM、供应商表 SUPPLIER 中，使用 NOT EXISTS EXCEPT 运算符，查询满足下列条件的供应商：该供应商不能供应所有的零件。

```

1 | SELECT S.S_SUPPKEY, S.S_NAME
2 | FROM SUPPLIER S
3 | WHERE NOT EXISTS (
4 |     SELECT P.P_PARTKEY
5 |     FROM PART P
6 |     EXCEPT
7 |     SELECT PS.PS_PARTKEY
8 |     FROM PARTSUPP PS
9 |     WHERE PS.PS_SUPPKEY = S.S_SUPPKEY
10 );

```

实验结果

```

1 | n_nationkey |          n_name
2 | -----+-----+
3 |      0 | ALGERIA
4 | (1 row)

```

查询18:

从国家表 NATION、客户表 CUSTOMER 中，使用 COUNT，查询满足下列条件的国家：至少有 3 个客户来自这个国家，并列出该国家的国家 key 和国家名。

```

1 | SELECT N.N_NATIONKEY, N.N_NAME
2 | FROM NATION N
3 | JOIN CUSTOMER C ON N.N_NATIONKEY = C.C_NATIONKEY
4 | GROUP BY N.N_NATIONKEY, N.N_NAME
5 | HAVING COUNT(C.C_CUSTKEY) >= 3;

```

实验结果

```

1 | n_nationkey |          n_name
2 | -----+-----+
3 |      0 | ALGERIA
4 | (1 row)

```

查询19:

从零部件表 PART 和零部件供应表 PARTSUPP 中，使用 FROM 子句中的子查询，查询满足下列条件的零件：零件由 2 个以上的供应商供应，且零件大小在 20 以上。

```
1 | SELECT T.PS_PARTKEY
2 | FROM (
3 |     SELECT PS.PS_PARTKEY, P.P_SIZE, COUNT(DISTINCT PS.PS_SUPPKEY) AS
4 |     SUPP_COUNT
5 |     FROM PART P
6 |     JOIN PARTSUPP PS ON P.P_PARTKEY = PS.PS_PARTKEY
7 |     GROUP BY PS.PS_PARTKEY, P.P_SIZE
8 |     HAVING COUNT(DISTINCT PS.PS_SUPPKEY) > 2
9 | ) T
10| WHERE T.P_SIZE >= 20;
```

实验结果

```
1 | ps_partkey
2 | -----
3 |      3
4 |      7
5 |      8
6 |      10
7 |      11
8 |      12
9 |      .....
10|      24593
```

2.7 WITH 临时视图查询

查询20:

用 WITH 临时视图方式，实现查询19中的查询要求。

```
1 | WITH TEMP AS (
2 |     SELECT PS.PS_PARTKEY, P.P_SIZE, COUNT(DISTINCT PS.PS_SUPPKEY)
3 |     AS SUPP_COUNT
4 |     FROM PART P
5 |     JOIN PARTSUPP PS ON P.P_PARTKEY = PS.PS_PARTKEY
6 |     GROUP BY PS.PS_PARTKEY, P.P_SIZE
7 |     HAVING COUNT(DISTINCT PS.PS_SUPPKEY) > 2
8 | )
9 | SELECT T.PS_PARTKEY
10| FROM TEMP T
11| WHERE T.P_SIZE >= 20;
```

实验结果

```

1 ps_partkey
2 -----
3      3
4      7
5      8
6      10
7      11
8      12
9      14
10     .....
11     211
12     212
13     214
14     216
15     217
16     218
17     219
18     .....
19     24593

```

查询21：

从零部件供应表 PARTSUPP 中，用 WITH 临时视图方式，查询零件供应数量最多的供应商 key 和其供应的数量。

```

1 WITH SUP_MAX AS (
2     SELECT PS.PS_SUPPKEY, SUM(PS.PS_AVAILQTY) AS TOTAL_QTY
3     FROM PARTSUPP PS
4     GROUP BY PS.PS_SUPPKEY
5 )
6     SELECT S.PS_SUPPKEY, S.TOTAL_QTY
7     FROM SUP_MAX S
8     WHERE S.TOTAL_QTY = (
9         SELECT MAX(TOTAL_QTY)
10        FROM SUP_MAX
11    );

```

实验结果

ps_suppkey	total_qty
33	81
1033	81
533	81
1533	81

(4 rows)

2.8 键/函数依赖分析

查询22:

在订单明细表 `LINEITEM` 中，检查订单 `key`、零件 `key`、供应商 `key`、流水号是否组成超键。

```
1 | SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, COUNT(*) AS  
2 | COUNT  
3 | FROM LINEITEM  
4 | GROUP BY L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER  
5 | HAVING COUNT(*) > 1;
```

如果查询结果为空，说明上述字段组合能唯一标识一条记录，组成超键。

实验结果

```
1 | l_orderkey | l_partkey | l_suppkey | l_linenumber | count  
2 | -----+-----+-----+-----+-----+  
3 | (0 rows)
```

查询23:

在订单明细表 `LINEITEM` 中，利用 SQL 语句检查函数依赖 `L_PARTKEY → L_EXTENDEDPRICE` 是否成立；如果不成立，利用 SQL 语句找出导致函数依赖不成立的元组。

```
1 | -- 检查函数依赖是否成立  
2 | SELECT L_PARTKEY  
3 | FROM LINEITEM  
4 | GROUP BY L_PARTKEY  
5 | HAVING COUNT(DISTINCT L_EXTENDEDPRICE) > 1;  
6 |  
7 | -- 找出导致函数依赖不成立的元组(由于元组过长，使用三个属性代表元组)  
8 | SELECT ALL l_orderkey, l_partkey, l_suppkey  
9 | FROM LINEITEM  
10 | WHERE L_PARTKEY IN (  
11 |     SELECT L_PARTKEY  
12 |     FROM LINEITEM  
13 |     GROUP BY L_PARTKEY  
14 |     HAVING COUNT(DISTINCT L_EXTENDEDPRICE) > 1  
15 | );
```

实验结果-- 检查函数依赖是否成立

```
1 | l_partkey  
2 | -----  
3 | 1  
4 | 2  
5 | 3
```

6	4
7	5
8	6
9	7
10	8
11	9
12	10
13
14	21
15	22
16	23
17	24
18	25
19	26
20	27
21	28
22

实验结果-- 找出导致函数依赖不成立的元组

1	l_orderkey	l_partkey	l_suppkey
2			
3	896	15262	784
4	967	14400	908
5	1413	5129	636
6	2628	535	36
7	2693	11550	1551
8	4419	31124	1640
9	5282	529	30
10	10279	12702	703
11	10532	5928	933
12		
13	114339	17364	381
14	118784	17674	1199
15	121123	32420	421
16	121221	31825	1826
17	122021	3647	650
18	123781	6795	796
19	124804	5180	1683
20	124839	11660	1661
21		
22	251875	19522	1523
23	252865	25721	746
24	254117	5266	1769
25	254177	28791	334
26	255299	34388	389
27	255552	20069	1090
28		

2.9 关系表的插入/删除/更新

查询24:

向订单表 `ORDERS` 中插入一条订单数据。

```
1 -- 插入新订单
2 INSERT INTO ORDERS (O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
3 O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY, O_CLERK,
4 O_SHIPPRIORITY, O_COMMENT)
5 VALUES
6     ('1200001',      -- 订单号
7      '20045',        -- 客户号
8      'F',            -- 订单状态
9      61365.24,       -- 订单总价
10     '2017-03-19'::DATE, -- 下单日期
11     '2-HIGH',       -- 订单优先级
12     'Clerk#000000098', -- 收银员
13     0,              -- 发货优先级
14     'furiously special f'); -- 订单备注
```

实验结果

```
1 | INSERT 0 1
```

查询25:

将零件 32 的全部供应商，作为零件 20 的供应商，加入到零部件供应表 `PARTSUPP` 中。

```
1 INSERT INTO PARTSUPP (PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY,
2 PS_SUPPLYCOST, PS_COMMENT)
3 SELECT 20, PS_SUPPKEY, PS_AVAILQTY, PS_SUPPLYCOST, PS_COMMENT
4 FROM PARTSUPP
5 WHERE PS_PARTKEY = 32
6 AND PS_SUPPKEY NOT IN (
7     SELECT PS_SUPPKEY
8     FROM PARTSUPP
9     WHERE PS_PARTKEY = 20
9 );
```

实验结果

```
1 | INSERT 0 0
```

查询26:

在订单明细表 `LINEITEM` 中，删除已退货的订单记录 (`L_RETURNFLAG = 'R'`) 。

```
1 | DELETE FROM LINEITEM  
2 | WHERE L_RETURNFLAG = 'R';
```

实验结果

```
1 | DELETE 296116
```

查询27：

用订单明细表 `LINEITEM` 中在 `2019` 年之后交易中的预计到达日期，替换表中的实际到达日期。

```
1 | UPDATE LINEITEM L  
2 | SET L_RECEIPTDATE = L_COMMITDATE  
3 | FROM ORDERS O  
4 | WHERE L.L_ORDERKEY = O.O_ORDERKEY  
5 | AND O.O_ORDERDATE >= '2019-01-01'::DATE;
```

实验结果

```
1 | UPDATE 470931
```

查询28：

针对订单明细表 `LINEITEM`、订单表 `ORDERS`，使用 `UPDATE / CASE` 语句做出如下修改：如果订单的订单优先级低于 `MEDIUM`，则其在订单明细表中的预计到达日期推后 `2` 天，否则推迟一天。

```
1 | UPDATE LINEITEM L  
2 | SET L_COMMITDATE = L_COMMITDATE + INTERVAL '1 day' * (  
3 | CASE  
4 |     WHEN O.O_ORDERPRIORITY < '3-MEDIUM' THEN 2  
5 |     ELSE 1  
6 | END  
7 | )  
8 | FROM ORDERS O  
9 | WHERE L.L_ORDERKEY = O.O_ORDERKEY;
```

实验结果

```
1 | UPDATE 1199969
```

查询29：

在订单表 `ORDERS` 中，利用 `RANK` 函数，按照订单总价对订单进行降序排序，并输出订单 `key` 和排名。

```
1 | SELECT O_ORDERKEY, O_TOTALPRICE, RANK() OVER (ORDER BY O_TOTALPRICE
2 | DESC) AS "Rank"
2 | FROM ORDERS;
```

实验结果

	o_orderkey	o_totalprice	Rank
3	209028	505770.15	1
4	528388	497758.84	2
5	993697	487758.42	3
6	1111238	485577.76	4
7	489319	484671.66	5
8	366692	483521.14	6
9	546785	481047.81	7
10	326117	473020.26	8
11	149509	471154.02	9
12	185124	460604.60	10
13		
14	1024160	408809.93	193
15	89859	408472.42	194
16	135046	408452.16	195
17	294343	408342.63	196
18	885252	408223.13	197
19	651718	408173.93	198
20	189509	408153.63	199
21	809125	408116.32	200
22		

3 完整性约束实验

3.1 完整性约束的建立

完整性约束可以在建表的同时建立，此时使用如下的 `create table` 中定义完整性约束的语句；建表后运行 `select *` 语句，可查看新建的空表：

```
1 | CREATE TABLE LINEITEMcopy1(
2 | L_ORDERKEY integer NOT NULL,
3 | L_PARTKEY integer NOT NULL,
4 | L_SUPPKEY integer NOT NULL,
5 | L_LINENUMBER integer NOT NULL,
6 | L_QUANTITY DECIMAL(15,2) NOT NULL,
7 | L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL,
8 | L_DISCOUNT DECIMAL(15,2) NOT NULL,
9 | L_TAX DECIMAL(15,2) NOT NULL,
10 | L_RETURNFLAG CHAR(1) NOT NULL,
11 | L_LINESTATUS CHAR(1) NOT NULL,
```

```

12    L_SHIPDATE DATE NOT NULL,
13    L_COMMITDATE DATE NOT NULL,
14    L_RECEIPTDATE DATE NOT NULL,
15    L_SHIPINSTRUCT CHAR(25) NOT NULL,
16    L_SHIPMODE CHAR(10) NOT NULL,
17    L_COMMENT VARCHAR(44) NOT NULL,
18    PRIMARY KEY (L_ORDERKEY, L_LINENUMBER),
19    FOREIGN KEY (L_PARTKEY) REFERENCES PART(P_PARTKEY),
20    FOREIGN KEY (L_SUPPKEY) REFERENCES SUPPLIER(S_SUPPKEY)
21 );

```

```

omm=# select * from lineitemcopy1;
l_orderkey | l_partkey | l_suppkey | l_linenumber | l_quantity | l_extendedprice | l_discount
t | l_tax | l_returnflag | l_linenstatus | l_shipdate | l_commitdate | l_receiptdate | l_shipi
nstruct | l_shipmode | l_comment
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+
(0 rows)

```

SSH: database

对于已经建立的表，可以使用 `alter table` 语句为其添加新的完整性约束；如下，先建立一个不含完整性约束的表 `LINEITEMcopy2`：

```

1 CREATE TABLE LINEITEMcopy2(
2     L_ORDERKEY integer NOT NULL,
3     L_PARTKEY integer NOT NULL,
4     L_SUPPKEY integer NOT NULL,
5     L_LINENUMBER integer NOT NULL,
6     L_QUANTITY DECIMAL(15,2) NOT NULL,
7     L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL,
8     L_DISCOUNT DECIMAL(15,2) NOT NULL,
9     L_TAX DECIMAL(15,2) NOT NULL,
10    L_RETURNFLAG CHAR(1) NOT NULL,
11    L_LINESTATUS CHAR(1) NOT NULL,
12    L_SHIPDATE DATE NOT NULL,
13    L_COMMITDATE DATE NOT NULL,
14    L_RECEIPTDATE DATE NOT NULL,
15    L_SHIPINSTRUCT CHAR(25) NOT NULL,
16    L_SHIPMODE CHAR(10) NOT NULL,
17    L_COMMENT VARCHAR(44) NOT NULL
18 );

```

```

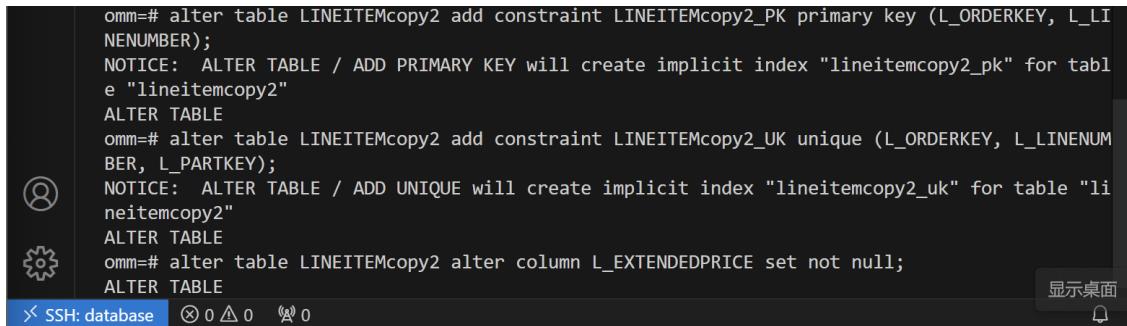
omm=# select * from lineitemcopy2;
l_orderkey | l_partkey | l_suppkey | l_linenumber | l_quantity | l_extendedprice | l_discount
t | l_tax | l_returnflag | l_linenstatus | l_shipdate | l_commitdate | l_receiptdate | l_shipi
nstruct | l_shipmode | l_comment
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+
(0 rows)

```

SSH: database

再添加完整性约束，终端返回添加成功的信息：

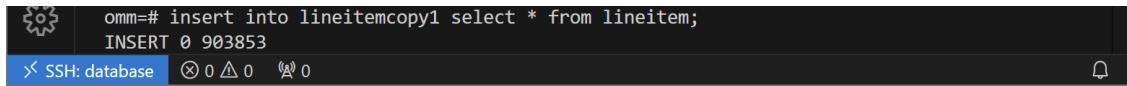
```
1 alter table LINEITEMcopy2 add constraint LINEITEMcopy2_PK primary
key (L_ORDERKEY, L_LINENUMBER);
2 alter table LINEITEMcopy2 add constraint LINEITEMcopy2_UK unique
(L_ORDERKEY, L_LINENUMBER, L_PARTKEY);
3 alter table LINEITEMcopy2 alter column L_EXTENDEDPRICE set not null;
```



```
omm=# alter table LINEITEMcopy2 add constraint LINEITEMcopy2_PK primary key (L_ORDERKEY, L_LINENUMBER);
NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index "lineitemcopy2_pk" for table "lineitemcopy2"
ALTER TABLE
omm=# alter table LINEITEMcopy2 add constraint LINEITEMcopy2_UK unique (L_ORDERKEY, L_LINENUMBER, L_PARTKEY);
NOTICE: ALTER TABLE / ADD UNIQUE will create implicit index "lineitemcopy2_uk" for table "lineitemcopy2"
ALTER TABLE
omm=# alter table LINEITEMcopy2 alter column L_EXTENDEDPRICE set not null;
ALTER TABLE
```

向表内加入内容，用于后续实验：

```
1 | insert into lineitemcopy1 select * from lineitem;
```

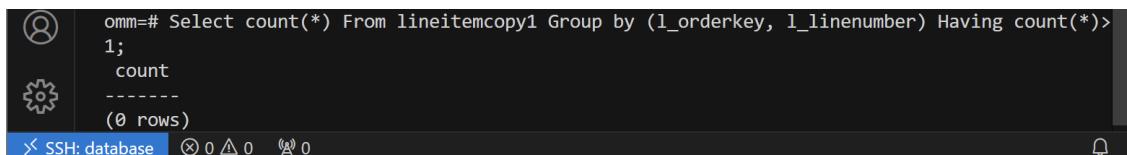


```
omm=# insert into lineitemcopy1 select * from lineitem;
INSERT 0 903853
```

3.2 主键/候选键/空值/check/默认值约束验证

下面使用表 `lineitemcopy1` 验证主键、空值、默认值和 `check` 约束；先验证对于已经存在的数据，`(l_orderkey, l_linenumber)` 是主键，`l_quantity` 非空且大于等于 `0`，之后分别使用它们验证主键、空值和 `check` 约束；默认值约束也使用 `l_quantity` 验证，为其设置默认值 `0`。

```
1 | Select count(*)
2 | From lineitemcopy1
3 | Group by (l_orderkey, l_linenumber)
4 | Having count(*)>1;
```



```
omm=# Select count(*) From lineitemcopy1 Group by (l_orderkey, l_linenumber) Having count(*)>1;
count
-----
(0 rows)
```

这验证了 `(l_orderkey, l_linenumber)` 是主键。

```
1 | select l_orderkey, l_linenumber
2 | from lineitemcopy1
3 | where l_quantity is null;
```

```
omm=# select l_orderkey, l_linenumber from lineitemcopy1 where l_quantity is null;
l_orderkey | l_linenumber
-----+-----
(0 rows)
× SSH: database ⊗ 0 △ 0 ⌂ 0
```

这验证了 `l_quantity` 非空。

```
1 select l_orderkey, l_linenumber
2 from lineitemcopy1
3 where l_quantity<0;
```

```
omm=# select l_orderkey, l_linenumber from lineitemcopy1 where l_quantity<0;
l_orderkey | l_linenumber
-----+-----
(0 rows)
× SSH: database ⊗ 0 △ 0 ⌂ 0
```

这验证了 `l_quantity` 大于等于 0。

3.2.1 主键约束的验证：

主键约束保证主键唯一和主键非空，先验证主键非空，使用 `update` 将某一存在的主键值设为空值，出现报错：

```
1 UPDATE lineitemcopy1
2 SET l_orderkey=null, l_linenumber=null
3 WHERE l_orderkey=1 and l_linenumber=5;
```

```
omm=# UPDATE lineitemcopy1 SET l_orderkey=null, l_linenumber=null WHERE l_orderkey =1 and l_linenumber=5;
ERROR: null value in column "l_orderkey" violates not-null constraint
DETAIL: Failing row contains (null, 4806, 313, null, 24.00, 41059.20, .10, .04, N, 0, 2019-03-31 00:00:00, 2019-02-03 00:00:00, 2019-02-03 00:00:00, DELIVER IN PERSON , FOB , pending foxes. slyly re).
× SSH: database ⊗ 0 △ 0 ⌂ 0
```

再验证主键唯一，使用 `update` 更改某一主键的值，使其与另一主键的值相等，出现报错：

```
1 UPDATE lineitemcopy1
2 SET l_linenumber=2
3 WHERE l_orderkey=1 and l_linenumber=1;
```

```
omm=# UPDATE lineitemcopy1 SET l_linenumber=2 WHERE l_orderkey=1 and l_linenumber=1;
ERROR: duplicate key value violates unique constraint "lineitemcopy1_pkey"
DETAIL: Key (l_orderkey, l_linenumber)=(1, 2) already exists.
× SSH: database ⊗ 0 △ 0 ⌂ 0
```

3.2.2 空值约束的验证

插入符合主键约束但 `l_quantity` 为空的元组，出现报错：

```
1 INSERT INTO lineitemcopy1 values(999, 0, 0, 999, null, 0, 0, 0, 0, 'a',
'b', '2020-01-01'::date, '2020-01-12'::date, '2020-01-15'::date,
'name3', 'name4', 'name5');
```

```
⑧ omm=# INSERT INTO lineitemcopy1 values(999, 0, 0, 999, null, 0, 0, 0, 'a', 'b', '2020-01-01':  
:date, '2020-01-12'::date, '2020-01-15'::date, 'name3', 'name4', 'name5');  
ERROR: null value in column "l_quantity" violates not-null constraint  
DETAIL: Failing row contains (999, 0, 0, 999, null, 0.00, 0.00, 0.00, a, b, 2020-01-01 00:00:  
:00, 2020-01-12 00:00:00, 2020-01-15 00:00:00, name3 , name4 , name5).  
✗ SSH: database ⑧ 0 △ 0 ⌂
```

3.2.3 check 约束的验证

为 `l_quantity` 添加 `check` 约束：

```
1 alter table lineitemcopy1  
2 add constraint larger_than_zero check(l_quantity>=0);
```

之后插入满足主键约束和空值约束，但不满足 `check` 约束的元组，出现报错：

```
1 INSERT INTO lineitemcopy1 values(999, 0, 0, 999, -1, 0, 0, 0, 'a',  
'b', '2020-01-01'::date, '2020-01-12'::date, '2020-01-15'::date,  
'name3', 'name4', 'name5');
```

```
⑧ omm=# INSERT INTO lineitemcopy1 values(999, 0, 0, 999, -1, 0, 0, 'a', 'b', '2020-01-01'::d  
ate, '2020-01-12'::date, '2020-01-15'::date, 'name3', 'name4', 'name5');  
ERROR: new row for relation "lineitemcopy1" violates check constraint "larger_than_zero"  
DETAIL: Failing row contains (999, 0, 0, 999, -1.00, 0.00, 0.00, 0.00, a, b, 2020-01-01 00:0  
0:00, 2020-01-12 00:00:00, 2020-01-15 00:00:00, name3 , name4 , name5).  
✗ SSH: database ⑧ 0 △ 0 ⌂
```

3.2.4 默认值约束的验证

为 `l_quantity` 添加默认值约束，默认取值为 `0`：

```
1 alter table lineitemcopy1  
2 alter l_quantity set default 0;
```

插入满足主键约束，但未说明 `l_quantity` 取值的元组（此处需指明属性顺序），从查询结果中可见，`l_quantity` 取值为 `0`（第五个属性）：

```
1 INSERT INTO lineitemcopy1(l_orderkey, l_partkey, l_suppkey,  
l_linenumber, l_extendedprice, l_discount, l_tax, l_returnflag,  
l_linenstatus, l_shipdate, l_commitdate, l_receiptdate,  
l_shipinstruct, l_shipmode, l_comment) values(999, 31038, 1554,  
1000, 0, 0, 'a', 'b', '2020-01-01'::date, '2020-01-12'::date,  
'2020-01-15'::date, 'name3', 'name4', 'name5');
```

```

omm=# INSERT INTO lineitemcopy1(l_orderkey, l_partkey, l_suppkey, l_linenumber, l_extendedpri
ce, l_discount, l_tax, l_returnflag, l_linenstatus, l_shipdate, l_commitdate, l_receiptdat
e, l_shipinstruct, l_shipmode, l_comment) values(999, 31038, 1554, 1000, 0, 0, 0, 'a', 'b', '2020-01-01'::date, '2020-01-12'::date, '2020-01-15'::date, 'name3', 'name4', 'name5');
INSERT 0 1
omm=# select * from lineitemcopy1 where l_orderkey=999 and l_linenumber=1000;
l_orderkey | l_partkey | l_suppkey | l_linenumber | l_quantity | l_extendedprice | l_discoun
t | l_tax | l_returnflag | l_linenstatus | l_shipdate | l_commitdate | l_re
ceiptdate | l_shipinstruct | l_shipmode | l_comment
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
999 | 31038 | 1554 | 1000 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00
0 | 0.00 | a | b | 2020-01-01 00:00:00 | 2020-01-12 00:00:00 | 2020-01-15 00:00:00 | name3 | name4 | name5
(1 row)

```

3.3 外键/参照完整性约束验证

使用表 `customercopy1` 和 `orderscopy1` 进行这部分实验，如下是建表过程：

```

1 CREATE TABLE customercopy1(
2   c_custkey integer,
3   c_name varchar(25),
4   c_address varchar(40),
5   c_nationkey integer,
6   c_phone char(15),
7   c_acctbal decimal(15,2),
8   c_mktsegment char(10),
9   c_comment varchar(117),
10  PRIMARY KEY (c_custkey),
11  FOREIGN KEY (c_nationkey) REFERENCES nation(n_nationkey)
12 );
13
14  INSERT INTO customercopy1
15  SELECT *
16  FROM customer;

```

```

omm=# CREATE TABLE customercopy1(c_custkey integer,c_name varchar(25),c_address varchar(40),c_
_nationkey integer,c_phone char(15),c_acctbal decimal(15,2),c_mktsegment char(10),c_comment v
archar(117),PRIMARY KEY (c_custkey),FOREIGN KEY (c_nationkey) REFERENCES nation(n_nationkey))
;
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "customercopy1_pkey" for table
"customercopy1"
CREATE TABLE
omm=# INSERT INTO customercopy1 SELECT * FROM customer;
INSERT 0 30000

```

```

1 CREATE TABLE orderscopy1(
2   o_orderkey integer,
3   o_custkey integer,
4   o_orderstatus char(1),
5   o_totalprice decimal(15,2),
6   o_orderdate date,
7   o_orderpriority char(15),
8   o_clerk char(15),

```

```

9   o_shipppriority integer,
10  o_comment varchar(79),
11  PRIMARY KEY (o_orderkey),
12 );
13
14 INSERT INTO orderscopy1
15 SELECT *
16 FROM orders;

```

```

omm=# CREATE TABLE orderscopy1(o_orderkey integer,o_custkey integer,o_orderstatus char(1),o_totalprice decimal(15,2),o_orderdate date,o_orderpriority char(15),o_clerk char(15),o_shipppriority integer,o_comment varchar(79),PRIMARY KEY (o_orderkey),FOREIGN KEY (o_custkey) REFERENCES customercopy1(c_custkey));
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "orderscopy1_pkey" for table "orderscopy1"
CREATE TABLE
omm=# INSERT INTO orderscopy1 SELECT * FROM orders;
INSERT 0 300001

```

SSH: database

3.3.1 参照完整性约束验证

先验证已存在的元组满足参照完整性约束，即证明表 `orderscopy1` 在属性 `o_custkey` 上的取值一定存在于表 `customercopy1` 的 `c_custkey` 列中：

```

1 select count(O_CUSTKEY)
2 from orderscopy1
3 where O_CUSTKEY not in (
4 select C_CUSTKEY
5 from customercopy1
6 );

```

```

omm=# select count(O_CUSTKEY) from orderscopy1 where O_CUSTKEY not in (select C_CUSTKEY from customercopy1);
count
-----
0
(1 row)

```

SSH: database

这说明没有取值不满足上述条件，即验证了参照完整性。

3.3.2 级联外键关联下数据访问

使用如下语句定义级联外键关联：

```

1 alter table orderscopy1
2 add constraint FK_O_CUSTKEY
3 foreign key(O_CUSTKEY) references customercopy1(C_CUSTKEY)
4 on delete cascade
5 on update cascade;

```

级联外键关联会在被参照关系表被改动时改动参照关系表，而改动参照关系表在两种外键关联下产生的作用相同，当参照关系表的改动满足外键关联时：

```
1 | insert into orderscopy1 values(1200002, 0, 'O', 181580, '2019-01-02'::date, '5-LOW', 'Clerk#000000406', 0, 'special f');
```

```
⑧ omm=# insert into orderscopy1 values(1200002, 0, 'O', 181580, '2019-01-02'::date, '5-LOW', 'Clerk#000000406', 0, 'special f');
ERROR: insert or update on table "orderscopy1" violates foreign key constraint "orderscopy1_o_custkey_fkey"
DETAIL: Key (o_custkey)=(0) is not present in table "customercopy1".
```

当改动不满足外键关联时：

```
1 | insert into orderscopy1 values(1200002, 25519, 'O', 181580, '2019-01-02'::date, '5-LOW', 'Clerk#000000406', 0, 'special f');
```

```
⚙️ omm=# insert into orderscopy1 values(1200002, 25519, 'O', 181580, '2019-01-02'::date, '5-LOW', 'Clerk#000000406', 0, 'special f');
INSERT 0 1
```

两种外键关联都只关心关系表中产生外键关联的属性值，当属性值没有产生外键关联（不是关联属性值，或是被参照关系表中的关联属性值，但没有被参照），它相关的改动在两种外键关联下产生的作用相同；如下是被参照关系表中新增关联属性值，因为是新增的，所以肯定没有被参照：

```
1 | insert into customercopy1 values(30001, 'Customer#000030001', 'a', 0, '10-396-325-3144', 100, 'b', 'x');
```

```
⚙️ omm=# INSERT INTO customercopy1 values(30001, 'Customer#000030001', 'a', 0, '10-396-325-3144', 100, 'b', 'x');
INSERT 0 1
```

如下是改动被参照关系表中没有被参照的关联属性值：

```
1 | UPDATE customercopy1
2 | SET C_CUSTKEY=31001
3 | WHERE C_CUSTKEY=12;
```

```
⚙️ omm=# UPDATE customercopy1 SET C_CUSTKEY=31001 WHERE C_CUSTKEY=12;
UPDATE 1
```

如下是删除被参照关系表中没有被参照的关联属性值：

```
1 | delete from customercopy1
2 | where C_CUSTKEY=31001;
```

```
⚙️ omm=# delete from customercopy1 where C_CUSTKEY=31001;
DELETE 1
```

对被参照关系表中被参照的关联属性值进行改动在两种外键关联下产生的效果有所不同，先记录改动前参照关系表 `orderscopy1` 的情况：

```

1 | select *
2 | from orderscopy1
3 | where O_CUSTKEY=8890;

```

o_orderkey	o_custkey	o_orderstatus	o_totalprice	o_orderdate	o_orderpriority	o_clerk	o_shippriority	o_comment
6148	8898	O	91917.31	2018-12-25 00:00:00	3-MEDIUM	Clerk#0000000201	0	furiously special f
11685	8898	O	275083.28	2020-03-19 00:00:00	3-MEDIUM	Clerk#0000000641	0	furiously special f
26311	8898	O	167406.53	2018-10-31 00:00:00	3-MEDIUM	Clerk#0000000458	0	furiously special f
119288	8898	O	63506.01	2019-08-13 00:00:00	1-URGENT	Clerk#0000000113	0	furiously special f
149999	8898	P	375129.58	2018-03-26 00:00:00	2-HIGH	Clerk#0000000581	0	furiously special f
195402	8898	F	200863.5	2018-03-19 00:00:00	4-NOT SPECIFIED	Clerk#0000000350	0	furiously special f
322678	8898	O	70371.36	2018-05-21 00:00:00	1-URGENT	Clerk#000000021	0	furiously special f
448448	8898	F	55873.61	2016-07-18 00:00:00	5-LOW	Clerk#0000000964	0	furiously special f
534188	8898	O	240580.03	2019-09-04 00:00:00	5-LOW	Clerk#0000000107	0	furiously special f
588067	8898	F	59589.47	2016-05-12 00:00:00	2-HIGH	Clerk#0000000777	0	furiously special f
633488	8898	F	285446.11	2017-02-04 00:00:00	5-LOW	Clerk#0000000000	0	furiously special f
665248	8898	O	99663.66	2020-04-18 00:00:00	1-URGENT	Clerk#0000000216	0	furiously special f
796579	8898	O	230800.14	2018-08-07 00:00:00	1-URGENT	Clerk#0000000974	0	furiously special f
817768	8898	F	185100.0	2018-03-19 00:00:00	1-URGENT	Clerk#0000000925	0	furiously special f
833960	8898	O	270231.88	2019-07-27 00:00:00	1-URGENT	Clerk#0000000955	0	furiously special f
860322	8898	O	98723.67	2019-01-25 00:00:00	5-LOW	Clerk#0000000951	0	furiously special f
887169	8898	O	177876.01	2016-06-15 00:00:00	2-HIGH	Clerk#0000000735	0	furiously special f
899873	8898	F	238527.09	2016-05-05 00:00:00	2-HIGH	Clerk#00000000221	0	furiously special f
979429	8898	O	67776.69	2021-02-16 00:00:00	5-LOW	Clerk#0000000172	0	furiously special f
1002950	8898	O	271364.01	2021-03-05 00:00:00	5-LOW	Clerk#0000000590	0	furiously special f
1016291	8898	O	236136.68	2018-10-01 00:00:00	4-NOT SPECIFIED	Clerk#0000000331	0	furiously special f
1153857	8898	O	88564.94	2019-09-22 00:00:00	3-MEDIUM	Clerk#0000000214	0	furiously special f
1189921	8898	F	192893.41	2018-02-19 00:00:00	4-NOT SPECIFIED	Clerk#0000000210	0	furiously special f
1190794	8898	O	18353.27	2021-01-28 00:00:00	5-LOW	Clerk#0000000456	0	furiously special f

(24 rows)

改动 `customercopy1` 中被参照的关联属性值 `C_CUSTKEY=8890`:

```

1 | UPDATE customercopy1
2 | SET C_CUSTKEY=30005
3 | WHERE C_CUSTKEY=8890;

```

这在非级联外键关联下出现报错:

ERROR: update or delete on table "customercopy1" violates foreign key constraint "orderscopy1_1_o_custkey_fkey" on table "orderscopy1"
DETAIL: Key (c_custkey)=(8890) is still referenced from table "orderscopy1".

在级联外键关联下, 该操作合法, 并造成表 `orderscopy1` 的级联改动:

```

1 | select count(*)
2 | from orderscopy1
3 | where O_CUSTKEY=8890;
4 |
5 | select count(*)
6 | from orderscopy1
7 | where O_CUSTKEY=30005;

```

omm=# select count(*) from orderscopy1 where O_CUSTKEY=8890;
count

0
(1 row)

omm=# select count(*) from orderscopy1 where O_CUSTKEY=30005;
count

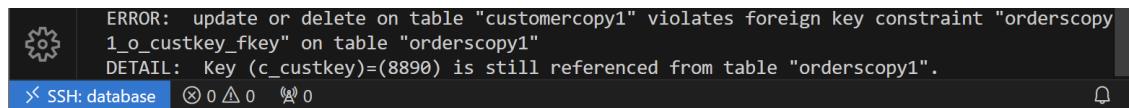
24
(1 row)

可见, 表 `orderscopy1` 中外键属性值 `O_CUSTKEY=8890` 被改动为 `O_CUSTKEY=30005`.

重置两表, 删除 `customercopy1` 中被参照的关联属性值 `C_CUSTKEY=8890`:

```
1 | delete from customercopy1  
2 | where C_CUSTKEY=8890;
```

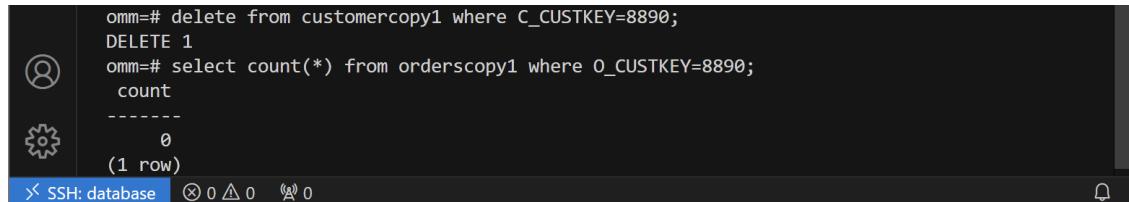
非级联外键关联下出现报错：



```
ERROR: update or delete on table "customercopy1" violates foreign key constraint "orderscopy1_o_custkey_fkey" on table "orderscopy1"  
DETAIL: Key (c_custkey)=(8890) is still referenced from table "orderscopy1".
```

级联外键关联下不报错，运行以下查询：

```
1 | select count(*)  
2 | from orderscopy1  
3 | where O_CUSTKEY=8890;
```



```
omm=# delete from customercopy1 where C_CUSTKEY=8890;  
DELETE 1  
omm=# select count(*) from orderscopy1 where O_CUSTKEY=8890;  
count  
-----  
 0  
(1 row)
```

可见 `O_CUSTKEY=8890` 的元组被级联删除.

3.4 函数依赖分析验证

验证函数依赖可使用映射基数 `l..m`，使用如下语句计算 `l` 和 `m` 的值（分别计算出 `P_BRAND` 和 `P_MFGR` 的映射基数）：

```
1 | select min(a), max(a)  
2 | from(  
3 | select P_BRAND, count(DISTINCT P_MFGR) as a  
4 | from part  
5 | group by P_BRAND  
6 | );  
7 |  
8 | select min(a), max(a)  
9 | from(  
10 | select P_MFGR, count(DISTINCT P_BRAND) as a  
11 | from part  
12 | group by P_MFGR  
13 | );
```

```
omm=# select min(a), max(a) from( select P_BRAND, count(DISTINCT P_MFGR) as a from part group by P_BRAND );
min | max
-----+-----
 1 |   1
(1 row)

omm=# select min(a), max(a) from( select P_MFGR, count(DISTINCT P_BRAND) as a from part group by P_MFGR );
min | max
-----+-----
 5 |   5
(1 row)
```

这说明存在函数依赖 `P_BRAND → P_MFGR`.

3.5 触发器约束

选择实验3：开发一个触发器，实现：当客户账户余额小于50时，不允许向订单表中插入来自该客户的新订单。

使用“判断+插入”替换原有的直接插入，使用“`instead of`”触发器；OpenGauss上的触发器会调用触发函数实现功能；先定义触发函数：

```
1 CREATE OR REPLACE FUNCTION judge_insert()
2 RETURNS TRIGGER AS $$ 
3 BEGIN
4     IF (SELECT C_ACCTBAL FROM customercopy1 WHERE
5         C_CUSTKEY=new.O_CUSTKEY) < 50 THEN
6         -- 如果余额小于50，则抛出错误，后续不会继续插入
7         RAISE EXCEPTION 'Customer balance is below 50. Cannot place
8         order.';
9     END IF;
10    -- return后会继续插入
11    RETURN NEW;
12 END;
13 $$ LANGUAGE plpgsql;
```

再定义触发器：

```
1 CREATE TRIGGER trig_judge_insert
2 BEFORE INSERT ON orderscopy1
3 FOR EACH ROW
4 EXECUTE FUNCTION judge_insert();
```

下面构造测试用例，搜索发现，`o_custkey=11`的客户欠了不少：

```

omm=# select * from customercopy1 where c_custkey=11;
   c_custkey |      c_name      | c_address | c_nationkey |    c_phone     | c_acctbal | c_
mktsegment |
-----+-----+-----+-----+-----+-----+
-----+
-----+
11 | Customer#000000011 | BCP1yGJ8xc |          0 | 10-464-151-3439 | -272.60 | BU
ILDING | ckages. requests sleep slyly. quickly even pinto beans promise above the slyly r
egular pinto beans.
(1 row)

```

SSH: database ⊗ 0 △ 0 ⌂ 0

于是构造如下的测试用例，按照预期，这一行应无法插入：

```

1 | insert into orderscopy1 values (10, 11, 'F', 198603.73, '2017-07-
30', '5-LOW', 'Clerk#000000925', 0, 'furiously special f');

omm=# insert into orderscopy1 values (10, 11, 'F', 198603.73, '2017-07-30', '5-LOW', 'Clerk#0
00000925', 0, 'furiously special f');
ERROR: Customer balance is below 50. Cannot place order.

```

SSH: database ⊗ 0 △ 0 ⌂ 0

与预期相符。

如果 `o_custkey=10`，应该可以插入：

```

omm=# select * from customercopy1 where c_custkey=10;
   c_custkey |      c_name      | c_address | c_nationkey |    c_phone     | c_
acctbal | c_mktsegment |
-----+-----+-----+-----+-----+-----+
-----+
-----+
10 | Customer#00000010 | Lbrg3EIDieI0B10bB0Aymm |          0 | 10-741-346-9870 | 2753.54
| HOUSEHOLD | es regular deposits haggle. fur
(1 row)

omm=# insert into orderscopy1 values (10, 10, 'F', 198603.73, '2017-07-30', '5-LOW', 'Clerk#0
00000925', 0, 'furiously special f');
INSERT 0 1
omm=# select * from orderscopy1 where o_orderkey=10;
   o_orderkey | o_custkey | o_orderstatus | o_totalprice |    o_orderdate     | o_orderpriorit
y |    o_clerk    | o_shipppriority |    o_comment
-----+-----+-----+-----+-----+-----+
-----+
-----+
10 |        10 | F | 198603.73 | 2017-07-30 00:00:00 | 5-LOW
| Clerk#000000925 | 0 | furiously special f
(1 row)

omm=#

```

SSH: database ⊗ 0 △ 0 ⌂ 0

这也与预期相符，因此该设计可行。

4 数据库接口实验

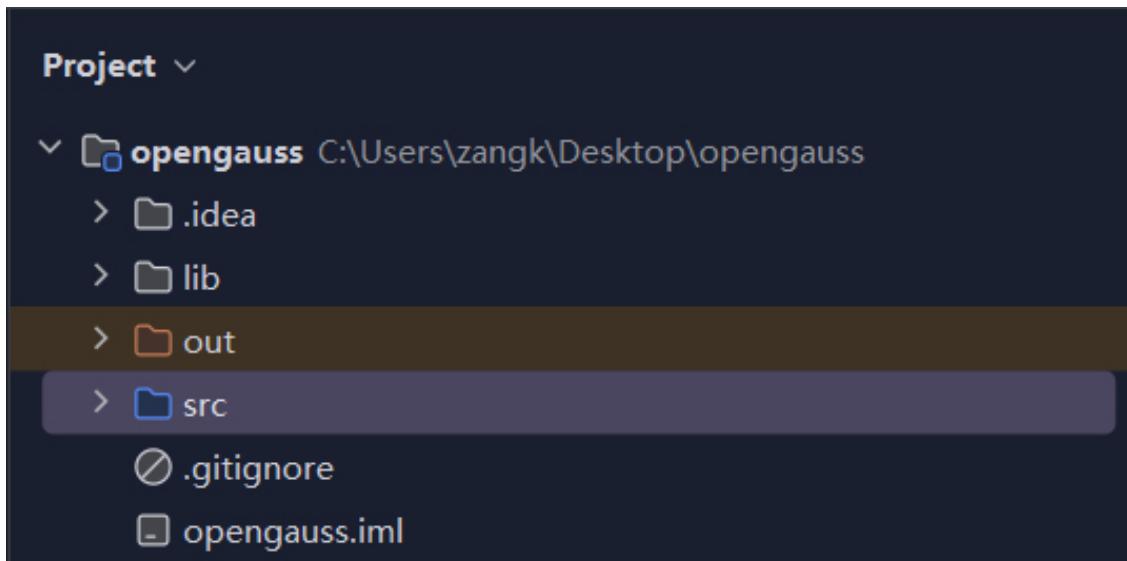
由于本地容器涉及较多端口映射操作，访问不便，故任务 4 基于华为云数据库和 JDBC 接口进行实验。

DBeaver 配置及建表在此处不再赘述，数据导入时所用的 `txt` 转 `csv` `python` 程序如下：

```
1 import csv
2
3 tables = ['customer', 'lineitem', 'nation', 'orders', 'part',
4           'partsupp', 'region', 'supplier']
5
6 for t in tables:
7     with open(t+'.csv', 'w', newline='') as f:
8         f_csv = csv.writer(f)
9         for line in open(t+'.txt', 'r'):
10             new_line = line.strip().split('|')
11             f_csv.writerow(new_line)
12     print(t+' finished!')
```

4.1 JDBC 配置

采用 JDK 1.8 环境以及 IDEA 作为 IDE，创建项目文件夹，层次如下：



打开[此链接](#)获取与 DBeaver 相同的 openGauss JDBC 驱动，解压后将 `postgresql.jar` 文件移动至上图所示 `lib` 文件夹后，右键点击选择 `Add as Library`。

openauss Version control

Project openauss C:\Users\zangk\Desktop\openauss

.idea lib out src .gitignore openauss.iml External Libraries Scratches and Console

New >

Cut Ctrl+X

Copy Ctrl+C

Copy Path/Reference...

Paste Ctrl+V

Find Usages Alt+F7

Find in Files... Ctrl+Shift+F

Replace in Files... Ctrl+Shift+R

Analyze >

Rename... Shift+F6

Refactor >

Bookmarks >

Reformat Code Ctrl+Alt+L

Optimize Imports Ctrl+Alt+O

Delete... Delete

Open In >

Local History >

Repair IDE on File

Reload from Disk

Compare With... Ctrl+D

Mark Directory as >

Add as Library...

	1	ARGENTINA
↑	2	BRAZIL
↓	3	CANADA
≡	4	EGYPT
≡	5	ETHIOPIA
↻	6	FRANCE
↶	7	GERMANY
↶	8	INDIA
↶	9	INDONESIA
(*)	10	IRAN
(*)	11	IRAQ
(*)	12	JAPAN
(*)	13	JORDAN
(!)	14	KENYA
(!)	15	MOROCCO
(!)	16	MOZAMBIQUE

openauss > src

Gau

4.2 执行 java 程序访问数据库

从 `src` 文件夹将默认生成的 `Main.java` 删除，新建文件名为 `Database.java`：

```
1 import java.sql.*;
2 public class Database{
3     static final String JDBC_DRIVER = "org.postgresql.Driver";
4     static final String DB_URL =
"jdbc:postgresql://110.41.123.152:8000/finance";
5     // 数据库的用户名与密码，需要根据自己的设置
6     static final String USER = "fuser202405";
7     static final String PASS = "fuser202405@bupt";
8     public static void main(String[] args) {
9         System.out.println("java版本号: "+
System.getProperty("java.version"));
10        Connection conn = null;
11        Statement stmt = null;
12        try{
13            // 注册 JDBC 驱动
14            Class.forName(JDBC_DRIVER);
15            // 打开链接
16            System.out.println("连接数据库...");
17            conn = DriverManager.getConnection(DB_URL,USER,PASS);
18            // 执行查询
19            System.out.println(" 实例化Statement对象...");
20            stmt = conn.createStatement();
21            String sql;
22            sql = "select * from nation";
23            ResultSet rs = stmt.executeQuery(sql);
24            System.out.print("nationkey name regionkey comment\n"
);
25            // 展开结果集数据库
26            while(rs.next()){
27                // 通过字段检索
28                    int nationkey = rs.getInt("n_nationkey");
29                    String name = rs.getString("n_name");
30                    int regionkey = rs.getInt("n_regionkey");
31                    String comment = rs.getString("n_comment");
32                    System.out.printf(" %-5d %-15s %-5d %s\n",nationkey
,name.trim(), regionkey, comment);
33            }
34            // 完成后关闭
35            rs.close();
36            stmt.close();
37            conn.close();
38        }catch(SQLException se){
39            // 处理 JDBC 错误
40            se.printStackTrace();
41        }catch(Exception e){
```

```

42 // 处理 Class.forName 错误
43         e.printStackTrace();
44     }finally{
45 // 关闭资源
46     try{
47         if(stmt!=null) stmt.close();
48     }catch(SQLException se){
49     }
50     try{
51         if(conn!=null) conn.close();
52     }catch(SQLException se){
53         se.printStackTrace();
54     }
55 }
56 }
57 }

```

需要注意数据库名称为 `finance`。

运行程序，返回如下结果：

```

'C:\Program Files\Java\jdk-1.8\bin\java.exe' ...
java 版本号: 1.8.0_421
连接数据库...
十一月 30, 2024 8:55:54 下午 org.postgresql.core.v3.ConnectionFactoryImpl openConnectionImpl
信息: [6c41aa4e-1935-4054-9024-0dc73dc99e67] Try to connect. IP: 110.41.123.152:8000
十一月 30, 2024 8:55:55 下午 org.postgresql.core.v3.ConnectionFactoryImpl openConnectionImpl
信息: [10.29.81.131:10368/110.41.123.152:8000] Connection is established. ID: 6c41aa4e-1935-4054-9024-0dc73dc99e67
十一月 30, 2024 8:55:55 下午 org.postgresql.core.v3.ConnectionFactoryImpl openConnectionImpl
信息: Connect complete. ID: 6c41aa4e-1935-4054-9024-0dc73dc99e67
实例化Statement对象...
nationkey name regionkey comment
0 ALGERIA 0 posits use carefully pending accounts. special deposits haggle. ironic, silent accounts are furio
1 ARGENTINA 1 ly bold instructions haggle quickly across the blithely close dep
2 BRAZIL 1 carefully regular dependencies are quickly. stealthily ironic platelets sleep
3 CANADA 1 packages must are. quickly regular requests among the slyly pending theodolites cajole quickly foxes; fluffily fin
4 EGYPT 4 e furiously silent packages. furiously ironic accounts af
5 ETHIOPIA 0 . ironic foxes haggle slyly. slyly special accounts nod among the furiously express de
6 FRANCE 3 encies. carefully even accounts after the asymptotes are ironic, ironic accoun
7 GERMANY 3 sual hockey players against the unusual, ironic packages nag carefully against the carefully ironic packages
8 INDIA 2 the even, regular accounts. slyly special pinto bea
9 INDONESIA 2 y across the final foxes. requests are fluffily near
10 IRAN 4 equests. packages are ironic, regular theodolites. carefully regular ideas sleep slyly final, ex
11 IRAQ 4 cording to the quickly regular platelets. carefully ironic pinto beans against the slyly unusual theodolites d
12 JAPAN 2 ites integrate across the requests. slyly pending depths n
13 JORDAN 4 nag furiously. carefully unusual pinto beans against the fluffily bold req
14 KENYA 0 foxes wake fluffily around the fluffily unusual grouchies. carefully unusual theodolites are slyly a
15 MOROCCO 0 uilar accounts wake carefully carefully close frays. furiously express dugouts above the furiously pending platele
16 MOZAMBIQUE 0 blithely regular instructions haggle qui
17 PERU 1 the requests. regular foxes sleep furiously. final requests integrate carefully about the slyly regular foxes.

18 CHINA 2 he blithely express theodolites. carefully final deposits after the blithely stealthy instructions gr
19 ROMANIA 3 express, even deposits. unusual, final ideas along
20 SAUDI ARABIA 4 ole against the slyly unusual platelets. regular accounts after the blithe
21 VIETNAM 2 r the carefully special pinto b
22 RUSSIA 3 lly silent excuses. accounts are quickly
23 UNITED KINGDOM 3 riously after the even, express ideas. slyly final theodolites are slyly deposits. blithely fina
24 UNITED STATES 1 e bold requests. carefully unusual packages cajole blithely regular Tiresias. ironic requests nag blithely

Process finished with exit code 0

```

4.3 对数据库进行其他操作

(1) 增加。添加一条 `n_nationkey` 为 25, `n_name` 为 South Korea, `n_regionkey` 为 3 的数据到数据库中。并根据 `n_nationkey` 查询是否插入成功。

所对应 SQL 语句为：

```

1 | insert into nation (n_nationkey, n_name, n_regionkey) values (25,
|   'South Korea', 3);

```

执行查询部分代码修改为：

```
1 // 执行查询
2     System.out.println(" 实例化statement对象... ");
3     stmt = conn.createStatement();
4     String sql;
5     sql = "insert into nation (n_nationkey, n_name,
6         n_regionkey) values (25, 'South Korea', 3)";
7     stmt.execute(sql);
8     sql = "select * from nation where n_nationkey = 25";
9     ResultSet rs = stmt.executeQuery(sql);
System.out.print("nationkey name regionkey comment\n" );
```

得到输出：

```
1 nationkey name regionkey comment
2 25      South Korea      3      null
```

表明插入成功。

(2) 更新，将(1)中插入的数据中的 `n_regionkey` 更新为 2。并根据 `n_nationkey` 查询是否更新成功。

```
1 // 执行查询
2     System.out.println(" 实例化statement对象... ");
3     stmt = conn.createStatement();
4     String sql;
5     sql = "update nation set n_regionkey = 2 where
6         n_nationkey = 25";
7     stmt.execute(sql);
8     sql = "select * from nation where n_nationkey = 25";
9     ResultSet rs = stmt.executeQuery(sql);
System.out.print("nationkey name regionkey comment\n" );
```

得到输出：

```
1 nationkey name regionkey comment
2 25      South Korea      2      null
```

表明更新成功。

(3) 删除，删除(1)插入的信息。根据 `n_nationkey` 查询是否删除成功。

```
1 // 执行查询
2         System.out.println(" 实例化statement对象... ");
3         stmt = conn.createStatement();
4         String sql;
5         sql = "delete from nation where n_nationkey = 25";
6         stmt.execute(sql);
7         sql = "select * from nation where n_nationkey = 25";
8         ResultSet rs = stmt.executeQuery(sql);
9         System.out.print("nationkey name regionkey comment\n" );
```

得到输出：

```
1 nationkey name regionkey comment
2
```

表明已不存在编号为 25 的项目，删除成功。

5 数据库物理设计

5.1 表空间

通过使用表空间，管理员可以控制一个数据库安装的磁盘布局。这样有以下特点：

1. 如果初始化数据库所在的分区或者卷空间已满，又不能逻辑上扩展更多空间，可以在不同的分区上创建和使用表空间，直到系统重新配置空间。
2. 表空间允许管理员根据数据库对象的使用模式安排数据位置，从而提高性能。
3. 一个频繁使用的索引可以放在性能稳定且运算速度较快的磁盘上，比如一种固态设备。
4. 一个存储归档的数据，很少使用的或者对性能要求不高的表可以存储在一个运算速度较慢的磁盘上。
5. 管理员通过表空间可以设置占用的磁盘空间。用以在和其他数据共用分区的时候，防止表空间占用相同分区上的其他空间。
6. 表空间可以控制数据库数据占用的磁盘空间，当表空间所在磁盘的使用率达到 90% 时，数据库将被设置为只读模式，当磁盘使用率降到 90% 以下时，数据库将恢复到读写模式。
7. 建议用户使用数据库时，通过后台监控程序或者 Database Manager 进行磁盘空间使用率监控，以免出现数据库只读情况。
8. 表空间对应于一个文件系统目录，比如：数据库节点数据目录/
pg_location/tablespace/tablespace_1 是用户拥有读写权限的空目录。
9. 使用表空间配额管理会使性能有 30% 左右的影响，MAXSIZE 指定每个数据库节点的配额大小，误差范围在 500MB 以内。请根据实际的情况确认是否需要设置表空间的最大值。

5.1.1 创建表空间

创建用户 jack:

```
1 omm=# CREATE USER jack IDENTIFIED BY 'openeuler12345!';
2 NOTICE: The encrypted password contains MD5 ciphertext, which is
not secure.
3 CREATE ROLE
```

创建表空间:

```
1 omm=# CREATE TABLESPACE fastspace RELATIVE LOCATION
'tablespace/tablespace_1';
2 CREATE TABLESPACE
```

其中“fastspace”为新创建的表空

间，`/var/lib/opengauss/data/pg_location/tablespace/tablespace_1`是用户拥有读写权限的空目录。

数据库系统管理员（本例中，为 omm 用户）执行如下命令将“fastspace”表空间的访问权限赋予数据用户 jack:

```
1 omm=# GRANT CREATE ON TABLESPACE fastspace TO jack;
2 GRANT
```

以此类推，创建多个表空间:

```
1 omm=# CREATE TABLESPACE example2 RELATIVE LOCATION
'tablespace/tablespace_2';
2 CREATE TABLESPACE
3 omm=# CREATE TABLESPACE example3 RELATIVE LOCATION
'tablespace/tablespace_3';
4 CREATE TABLESPACE
5 omm=# CREATE TABLESPACE example4 RELATIVE LOCATION
'tablespace/tablespace_4';
6 CREATE TABLESPACE
```

5.1.2 在表空间上创建对象

如果用户拥有表空间的 CREATE 权限，就可以在表空间上创建数据库对象。操作系统管理员（omm 用户）具有以上表空间的 CREATE 权限，并且 jack 用户拥有 fastspace 表空间的 CREATE 权限。在指定的表空间上创建表（创建其他的对象方法类似）：

```
1 | omm=# CREATE TABLE table_1(i int) TABLESPACE fastspace;
2 | CREATE TABLE
3 | omm=# CREATE TABLE table_2(i int) TABLESPACE example2;
4 | CREATE TABLE
```

在默认表空间上创建表：

首先设置默认表空间：

```
1 | omm=# SET default_tablespace = 'example3';
2 | SET
```

再创建表，这样无需指定表空间，表创建在默认表空间：

```
1 | omm=# CREATE TABLE table_3(i int);
2 | CREATE TABLE
```

5.1.3 管理表空间

5.1.3.1 查询表空间

方式 1：检查 pg_tablespace 系统表。如下命令可查到系统和用户定义的全部表空间。

```
1 | SELECT spcname FROM pg_tablespace;
```

```
1 | omm=# SELECT spcname FROM pg_tablespace;
2 | spcname
3 | -----
4 | pg_default
5 | pg_global
6 | fastspace
7 | example2
8 | example3
9 | example4
10 | (6 rows)
```

方式2：使用 gsql 程序的元命令查询表空间：

```
1 omm=# \db
2
3     Name   | Owner | Location
4
5 example2 | omm  | tablespace/tablespace_2
6 example3 | omm  | tablespace/tablespace_3
7 example4 | omm  | tablespace/tablespace_4
8 fastspace | omm  | tablespace/tablespace_1
9 pg_default | omm |
10 pg_global | omm |
11 (6 rows)
```

5.1.3.2 查询表空间当前使用情况

```
1 omm=# SELECT PG_TABLESPACE_SIZE('fastspace');
2 pg_tablespace_size
3
4 8192
5 (1 row)
```

其中 8192 表示表空间的大小，单位为字节。

5.1.3.3 重命名表空间

执行如下命令对表空间 fastspace 重命名为 example：

```
1 omm=# ALTER TABLESPACE fastspace RENAME TO example;
2 ALTER TABLESPACE
3 omm=# \db
4
5     Name   | Owner | Location
6
7 example | omm  | tablespace/tablespace_1
8 example2 | omm  | tablespace/tablespace_2
9 example3 | omm  | tablespace/tablespace_3
10 example4 | omm  | tablespace/tablespace_4
11 pg_default | omm |
12 pg_global | omm |
13 (6 rows)
```

5.1.3.4 删除表空间

用户必须是表空间的 owner 或者系统管理员才能删除表空间。

```
1 omm=# DROP TABLESPACE example;
2 ERROR:  tablespace "example" is not empty
```

删除失败，表空间不为空的情况下无法删除表空间 (避免误删里面的重要数据)。

先清空表空间：

```
1 | omm=# DROP TABLE table_1;
2 | DROP TABLE
```

然后再删除表空间：

```
1 | omm=# DROP TABLESPACE example;
2 | DROP TABLESPACE
```

删除成功：

```
1 | omm=# \db
2 |           List of tablespaces
3 |   Name    | Owner     |          Location
4 | +-----+-----+
5 | example2 | omm      | tablespace/tablespace_2
6 | example3 | omm      | tablespace/tablespace_3
7 | example4 | omm      | tablespace/tablespace_4
8 | pg_default | omm    |
9 | pg_global | omm    |
10| (5 rows)
```

5.2 分区表

分区表和普通表相比具有以下优点：

- 改善查询性能：对分区对象的查询可以仅搜索自己关心的分区，提高检索效率。
- 增强可用性：如果分区表的某个分区出现故障，表在其他分区的数据仍然可用。
- 方便维护：如果分区表的某个分区出现故障，需要修复数据，只修复该分区即可。
- 均衡 I/O：可以把不同的分区映射到不同的磁盘以平衡 I/O，改善整个系统性能。
- 普通表若要转成分区表，需要新建分区表，然后把普通表中的数据导入到新建的分区表中。因此在初始设计表时，请根据业务提前规划是否使用分区表。

openGauss分区表限制和特点：

- 主键约束或唯一约束必须要包含分区字段
- 分区表表名只能在 `pg_partition` 视图中查看，在 `pg_tables` 和 `pg_stat_all_tables` 中无法查到
- 分区表索引在 opengauss 里分 local 和 global，默认是 global
- 分区个数不能超过 327675
- 选择分区使用 `PARTITION FOR()`，括号里指定值个数应该与定义分区时使用的列个数相同，并且一一对应。

6. Value 分区表不支持相应的 `Alter Partition` 操作
7. 列存分区表不支持切割分区
8. 间隔分区表不支持添加分区

5.2.1 创建分区表

5.2.1.1 方法一： `VALUES LESS THAN`

语法：`PARTITION BY RANGE(partition_key)` 从句是 `VALUES LESS THAN` 的语法格式，范围分区策略的分区键最多支持 4 列。

```
1 | PARTITION partition_name VALUES LESS THAN ( { partition_value |
    MAXVALUE } )
```

- 每个分区都需要指定一个上边界。
- 分区上边界的类型应当和分区键的类型一致。
- 分区列表是按照分区上边界升序排列的，值较小的分区位于值较大的分区之前。

实例：`task1.sql`

```
1 | create table partition_orders_1(
2 |   o_orderkey integer,
3 |   o_custkey integer,
4 |   o_orderstatus char(1),
5 |   o_totalprice decimal(15,2),
6 |   o_orderdate date,
7 |   o_orderpriority char(15),
8 |   o_clerk char(15),
9 |   o_shippriority integer,
10 |   o_comment varchar(79),
11 |   PRIMARY KEY (o_orderkey)
12 | )
13 | partition by range(o_orderkey)
14 | (
15 |   partition p1 values less than(100),
16 |   partition p2 values less than(200),
17 |   partition p3 values less than(300),
18 |   partition p4 values less than(maxvalue)
19 | );
```

此分区表分区键为 `id`，分了 4 个区，分别是 $p1 < 100$, $100 \leq p2 < 200$, $200 \leq p3 < 300$, $300 \leq p4$ 。

文件保存在 `/home/omm` 下，执行 `gsql -f task1.sql`：

```

1 [omm@cfde74516988 ~]$ gsql -f task1.sql
2 gsql:task1.sql:19: NOTICE: CREATE TABLE / PRIMARY KEY will create
implicit index "partition_orders_1_pkey" for table
"partition_orders_1"
3 CREATE TABLE
4 total time: 24 ms

```

数据库中执行 `\d+` 命令显示该分区表的详细信息：

```

omm=# \d+ partition_orders_1
                                         Table "public.partition_orders_1"
   Column    |          Type          | Modifiers | Storage | Stats target | Description
---+-----+-----+-----+-----+-----+-----+
 o_orderkey | integer            | not null | plain   | plain        |
 o_custkey   | integer            |           | plain   | extended     |
 o_orderstatus | character(1)      |           |          | extended     |
 o_totalprice | numeric(15,2)      |           |          | main         |
 o_orderdate | timestamp(0) without time zone |           |          | plain        |
 o_orderpriority | character(15)    |           |          | extended     |
 o_clerk     | character(15)      |           |          | extended     |
 o_shipppriority | integer          |           |          | plain        |
 o_comment   | character varying(79) |           |          | extended     |
Indexes:
"partition_orders_1_pkey" PRIMARY KEY, btree (o_orderkey) LOCAL(PARTITION p1_o_orderkey_idx, PARTITION p2_o_orderkey_idx, PARTITION p3_o_orderkey_idx, PARTITION p4_o_orderkey_idx) TABLESPACE pg_default
Range partition by(o_orderkey)
Number of partition: 4 (View pg_partition to check each partition range.)
Has OIDs: no
Options: orientation=row, compression=no

```

表基本信息

- **Table:** 表名是 public.partition_orders_1，属于 public 模式。

索引

- 索引定义部分：
 - **PRIMARY KEY:** 主键是 o_orderkey，使用的是 B-tree 索引。
 - **LOCAL 索引:**
 - 每个分区都有独立的索引：
 - p1_o_orderkey_idx 对应分区 p1。
 - p2_o_orderkey_idx 对应分区 p2。
 - p3_o_orderkey_idx 对应分区 p3。
 - p4_o_orderkey_idx 对应分区 p4。
 - 索引的表空间是 pg_default (数据库默认表空间)。

分区定义

- **Range partition by(o_orderkey):**
 - 表按照 o_orderkey 进行范围分区。
 - 范围分区将数据根据范围划分到不同的子表中。
- **Number of partition: 4:** 表明分区表有 4 个分区。
 - 可以通过 pg_partition 表查看每个分区的范围和元数据。

其他信息

- **Has OIDs:** 表中没有启用 OIDs (对象标识符)。
- **Options:**
 - **orientation=row:** 表以行存储 (Row-Oriented)。
 - **compression=no:** 表未启用压缩。

通过 select 命令列出各分区情况：

```

1 omm=# select relname,parttype,parentid,boundaries from pg_partition
2   where parentid in(select oid from pg_class where
3     relname='partition_orders_1');
4   relname      | parttype | parentid | boundaries
5   +-----+-----+-----+
6   partition_orders_1 | r       | 16728  |
7   p1             | p       | 16728  | {100}
8   p2             | p       | 16728  | {200}
9   p3             | p       | 16728  | {300}
10  p4             | p       | 16728  | {NULL}
11  (5 rows)

```

其中 parttype 为 r (Root) 表示根分区，即分区表本身 (对应主表)；为 p (Partition) 表示子分区或具体的分区表。

5.2.1.2 方法二： START END

语法：`PARTITION BY RANGE(partition_key)` 从句是 `START END` 的语法规格式，范围分区策略的分区键仅支持 1 列。

```

1 PARTITION partition_name {START(partition_value)
2   END(partition_value) EVERY(interval_value)} |
3   {START(partition_value) END(partition_value | MAXVALUE)} |
4   {START(partition_value)} | {END(partition_value | MAXVALUE)}

```

- 在创建分区表若第一个分区定义含 START 值，则范围 (MINVALUE, START) 将自动作为实际的第一个分区。
- 每个 partition_start_end_item 中的 START 值（如果有的话，下同）必须小于其 END 值；
- 相邻的两个 partition_start_end_item，第一个的 END 值必须等于第二个的 START 值；
- 每个 partition_start_end_item 中的 EVERY 值必须是正向递增的，且必须小于 (END-START) 值；
- 每个分区包含起始值，不包含终点值，即形如：[起始值, 终点值)，起始值是 MINVALUE 时则不包含；
- 一个 partition_start_end_item 创建的每个分区所属的 TABLESPACE 一样；
- partition_name 作为分区名称前缀时，其长度不要超过 57 字节，超过时自动截断；

- 在创建、修改分区表时请注意分区表的分区总数不可超过最大限制（32767）；
- 在创建分区表时 START END 与 LESS THAN 语法不可混合使用。
- 即使创建分区表时使用 START END 语法，备份（gs_dump）出的 SQL 语句也是 VALUES LESS THAN · 单一 start 分区不能紧挨着单一 end 分区，否则会报错

实例： task2.sql

```

1  create table partition_orders_2(
2    o_orderkey integer,
3    o_custkey integer,
4    o_orderstatus char(1),
5    o_totalprice decimal(15,2),
6    o_orderdate date,
7    o_orderpriority char(15),
8    o_clerk char(15),
9    o_shipppriority integer,
10   o_comment varchar(79),
11   PRIMARY KEY (o_orderkey)
12 )
13 partition by range(o_orderkey)
14 (
15   partition p1 start(2) end(100) every(10),
16   partition p2 end(200),
17   partition p3 end(300),
18   partition p4 start(300),
19   partition p5 start(400),
20   partition p6 start(500) end(600)
21 );

```

此实例第一个分区定义含 start，则范围 (minvalue, 2) 自动作为第一个分区 p1_0, p1_0<2

由于 every (10)，则 p1 分区进行间隔分区，间隔为 10

即 p1_1 [2,12), p1_2 [12,22), p1_3 [22,32), p1_4 [32,42), p1_5 [42,52), p1_6 [52,62),
p1_7 [62,72), p1_8 [72,82), p1_9 [82,92), p1_10 [92,100)

之后 5 个分区，p2 [100,200), p3 [200,300), p4 [300,400), p5 [400,500), p6 [500,600)

文件保存在 /home/omm 下，执行 gsql -f task2.sql：

```

1  gsql:task2.sql:21: NOTICE:  CREATE TABLE / PRIMARY KEY will create
2    implicit index "partition_orders_2_pkey" for table
3    "partition_orders_2"
4  CREATE TABLE
5  total time: 24 ms

```

数据库中执行 \d+ 命令显示该分区表的详细信息：

```

omm=# \d+ partition_orders_2
      Table "public.partition_orders_2"
 Column | Type | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+
 o_orderkey | integer | not null | plain | |
 o_custkey | integer | | plain | |
 o_orderstatus | character(1) | | extended | |
 o_totalprice | numeric(15,2) | | main | |
 o_orderdate | timestamp(0) without time zone | | plain | |
 o_orderpriority | character(15) | | extended | |
 o_clerk | character(15) | | extended | |
 o_shipppriority | integer | | plain | |
 o_comment | character varying(79) | | extended | |
Indexes:
    "partition_orders_2_pkey" PRIMARY KEY, btree (o_orderkey) LOCAL(PARTITION p1_0_o_orderkey_idx, PARTITION p1_1_o_orderkey_idx, PARTITION p1_2_o_orderkey_idx, PARTITION p1_3_o_orderkey_idx, PARTITION p1_4_o_orderkey_idx, PARTITION p1_5_o_orderkey_idx, PARTITION p1_6_o_orderkey_idx, PARTITION p1_7_o_orderkey_idx, PARTITION p1_8_o_orderkey_idx, PARTITION p1_9_o_orderkey_idx, PARTITION p1_10_o_orderkey_idx, PARTITION p2_o_orderkey_idx, PARTITION p3_o_orderkey_idx, PARTITION p4_o_orderkey_idx, PARTITION p5_o_orderkey_idx, PARTITION p6_o_orderkey_idx) TABLESPACE pg_default
Range partition by(o_orderkey)
Number of partition: 16 (View pg_partition to check each partition range.)
Has OIDs: no
Options: orientation=row, compression=no

```

表基本信息

- **Table:** 表名是 public.partition_orders_2, 属于 public 模式。

索引

- 索引定义部分：
 - **PRIMARY KEY:** 主键是 o_orderkey, 使用的是 B-tree 索引。
 - **LOCAL 索引:**
 - 每个分区都有独立的索引, 每个索引的名称以分区名称为前缀:
 - p1_0_o_orderkey_idx 到 p1_10_o_orderkey_idx: 这些索引对应分区 p1 的子分区。
 - p2_o_orderkey_idx: 对应分区 p2。
 - p3_o_orderkey_idx 到 p6_o_orderkey_idx: 对应分区 p3 到 p6。
 - 每个分区及其子分区都有各自的局部索引 (LOCAL), 便于查询优化。
 - 索引的表空间是 pg_default (数据库默认表空间) 。

分区定义

- **Range partition by(o_orderkey):**
 - 表按照 o_orderkey 进行范围分区。
 - 范围分区将数据根据范围划分到不同的子表中。
- **Number of partition: 16:** 表明分区表有 16 个分区。
 - 可以通过 pg_partition 表查看每个分区的范围和元数据。

其他信息

- **Has OIDs:** 表中没有启用 OIDs (对象标识符) 。
- **Options:**
 - **orientation=row:** 表以行存储 (Row-Oriented) 。
 - **compression=no:** 表未启用压缩。

通过 select 命令列出各分区情况：

```
1 omm=# SELECT relname, parttype, boundaries
2 FROM pg_partition
3 WHERE parentid = (SELECT oid FROM pg_class WHERE relname =
4   'partition_orders_2');omm-# omm-#
5   relname      | parttype | boundaries
6   +-----+-----+
7   partition_orders_2 | r       |
8   p1_0           | p       | {2}
9   p1_1           | p       | {12}
10  p1_2           | p       | {22}
11  p1_3           | p       | {32}
12  p1_4           | p       | {42}
13  p1_5           | p       | {52}
14  p1_6           | p       | {62}
15  p1_7           | p       | {72}
16  p1_8           | p       | {82}
17  p1_9           | p       | {92}
18  p1_10          | p       | {100}
19  p2              | p       | {200}
20  p3              | p       | {300}
21  p4              | p       | {400}
22  p5              | p       | {500}
23  p6              | p       | {600}
23 (17 rows)
```

5.2.1.3 方法三： INTERVAL

语法：从句指定了 `INTERVAL` 子句的语句格式，范围分区策略的分区键仅支持 1 列。

```
1 | INTERVAL ('interval_expr') [ STORE IN (tablespace_name [, ...] ) ]
```

- 列存表不支持间隔分区
- `interval_expr`: 自动创建分区的间隔，例如：1 day、1 month。
- `STORE IN (tablespace_name [, ...])`: 指定存放自动创建分区的表空间列表，如果有指定，则自动创建的分区从表空间列表中循环选择使用，否则使用分区表默认的表空间。

实例：`task3.sql`

```
1 | create table partition_orders_3(
2 |   o_orderkey integer,
3 |   o_custkey integer,
4 |   o_orderstatus char(1),
5 |   o_totalprice decimal(15,2),
6 |   o_orderdate date,
```

```
7   o_orderpriority char(15),
8   o_clerk char(15),
9   o_shipppriority integer,
10  o_comment varchar(79),
11  PRIMARY KEY (o_orderkey)
12  )
13  partition by range(o_orderdate)
14  interval('1 day')
15  (
16  partition p1 values less than('2021-03-08 00:00:00'),
17  partition p2 values less than('2021-03-09 00:00:00')
18 );
```

直接执行此段代码，出现如下报错：

```
1 [omm@cfde74516988 ~]$ gsql -f task3.sql
2 gsql:task3.sql:18: ERROR: Invalid PRIMARY KEY/UNIQUE constraint for
partitioned table
3 DETAIL: Columns of PRIMARY KEY/UNIQUE constraint Must contain
PARTITION KEY
4 total time: 0 ms
```

错误原因为分区键非主键，原因为组成约束的单个索引只能直接在它们自己的分区内强制执行唯一性，因此分区结构本身必须保证在不同分区中不存在重复项（分区表唯一约束），解决方法为设置联合主键：

```
1 create table partition_orders_3(
2   o_orderkey integer,
3   o_custkey integer,
4   o_orderstatus char(1),
5   o_totalprice decimal(15,2),
6   o_orderdate date,
7   o_orderpriority char(15),
8   o_clerk char(15),
9   o_shipppriority integer,
10  o_comment varchar(79),
11  PRIMARY KEY (o_orderkey, o_orderdate)
12  )
13  partition by range(o_orderdate)
14  interval('1 day')
15  (
16  partition p1 values less than('2021-03-08 00:00:00'),
17  partition p2 values less than('2021-03-09 00:00:00')
18 );
```

执行结果：

```

1 [omm@cfde74516988 ~]$ gsql -f task3.sql
2 gsql:task3.sql:18: NOTICE: CREATE TABLE / PRIMARY KEY will create
implicit index "partition_orders_3_pkey" for table
"partition_orders_3"
3 CREATE TABLE
4 total time: 8 ms

```

```

omm=# select relname,parttype,parentid,boundaries from pg_partition where parentid in(select oid from pg_class where re
lname='partition_orders_3');
   relname    | parttype | parentid |      boundaries
-----+-----+-----+-----+
partition_orders_3 | r       | 16780 | {"2021-03-08 00:00:00"}
p1             | p       | 16780 | {"2021-03-09 00:00:00"}
p2             | p       | 16780 | {"2021-03-09 00:00:00"}
(3 rows)

```

此分区表，一开始创建的分区只有 p1,p2。但如果向表中插入键值不在已有分区范围内的元组，比如 (1, '2021-03-11 00:00:00')，则会自动创建一个分区，其上界与插入的元组间隔 1 day，该元组存入该分区。

插入元组 (1, '2021-03-11 00:00:00')，再次通过 select 命令列出各分区情况（由于版本变化，插入语句略有不同）：

```

1 omm=# insert into partition_orders_3 (o_orderkey, o_orderdate)
2   values (1, '2021-03-11 00:00:00');
2 INSERT 0 1

```

```

omm=# select relname,parttype,parentid,boundaries from pg_partition where parentid in(select oid from pg_class where re
lname='partition_orders_3');
   relname    | parttype | parentid |      boundaries
-----+-----+-----+-----+
partition_orders_3 | r       | 16780 | {"2021-03-08 00:00:00"}
p1             | p       | 16780 | {"2021-03-09 00:00:00"}
p2             | p       | 16780 | {"2021-03-09 00:00:00"}
sys_p1         | p       | 16780 | {"2021-03-12 00:00:00"}
(4 rows)

```

可见，自动生成了 1 个新的分区，上界为'2021-03-12 00:00:00'，与插入元组间隔为 1 day。

5.2.1.4 设置分区所在的表空间

在创建分区表时，可以把分区表的不同分区设置在不同的表空间，从而提升整个系统的性能通过在 partition 语句后面加上 tablespace 来指定创建的分区所在的表空间。

实例： task4.sql

```

1 create table partition_orders_4(
2   o_orderkey integer,
3   o_custkey integer,
4   o_orderstatus char(1),
5   o_totalprice decimal(15,2),
6   o_orderdate date,
7   o_orderpriority char(15),
8   o_clerk char(15),
9   o_shipppriority integer,
10  o_comment varchar(79),
11  PRIMARY KEY (o_orderkey)

```

```
12  )
13  tablespace example2
14  partition by range(o_orderkey)
15  (
16  partition p1 values less than(100),
17  partition p2 values less than(200) tablespace example4,
18  partition p3 values less than(300),
19  partition p4 values less than(maxvalue)
20 );
```

创建成功：

```
1 [omm@cfde74516988 ~]$ gsql -f task4.sql
2 gsql:task4.sql:20: NOTICE: CREATE TABLE / PRIMARY KEY will create
implicit index "partition_orders_4_pkey" for table
"partition_orders_4"
3 CREATE TABLE
4 total time: 8 ms
```

此分区表，分区 p1, p3, p4 都在表空间 example2 中，而分区 p2 在表空间 example4 中。

5.2.2 管理分区表

5.2.2.1 删除、添加、重命名分区

删除分区 p4

```
1 omm=# ALTER TABLE partition_orders_4 DROP PARTITION p4;
2 ALTER TABLE
```

添加分区 p_4

```
1 omm=# ALTER TABLE partition_orders_4 ADD PARTITION p_4 VALUES LESS
THAN(MAXVALUE);
2 ALTER TABLE
```

重命名分区 p3

```
1 omm=# ALTER TABLE partition_orders_4 RENAME PARTITION p3 TO p_3;
2 ALTER TABLE
```

修改后分区显示如下：

```

1 omm=# select relname,parttype,parentid,boundaries from pg_partition
2   where parentid in(select oid from pg_class where
3     relname='partition_orders_4');
4
5      relname      | parttype | parentid | boundaries
6      +-----+-----+
7 partition_orders_4 | r       | 16793    |
8 p1                | p       | 16793    | {100}
9 p2                | p       | 16793    | {200}
10 p_4               | p       | 16793    | {NULL}
11 p_3               | p       | 16793    | {300}
12
13 (5 rows)

```

5.2.2.2 修改分区的表空间

将分区 p1 由原来所在的表空间 example2 移动到 example4:

```

1 omm=# ALTER TABLE partition_orders_4 MOVE PARTITION p1 TABLESPACE
2   example4;
3 ALTER TABLE
4 omm=# select relname,parttype,parentid,boundaries from pg_partition
5   where parentid in(select oid from pg_class where
6     relname='partition_orders_4');
7
8      relname      | parttype | parentid | boundaries
9      +-----+-----+
10 partition_orders_4 | r       | 16793    |
11 p2                | p       | 16793    | {200}
12 p_4               | p       | 16793    | {NULL}
13 p_3               | p       | 16793    | {300}
14 p1                | p       | 16793    | {100}
15
16 (5 rows)

```

为显示分区所在表空间，可采用如下方法：

```

1 omm=# SELECT
2   p.relname AS partition_name,
3   p.parttype AS partition_type,
4   t.spcname AS tablespace_name,
5   p.boundaries
6 FROM
7   pg_partition p
8 LEFT JOIN
9   pg_tablespace t
10 ON
11   p.reltblespace = t.oid
12 WHERE
13   p.parentid IN (SELECT oid FROM pg_class WHERE relname =
14     'partition_orders_4');omm-# omm-# omm-# omm-# omm-# omm-#
15   omm-# omm-# omm-# omm-# omm-# omm-#

```

	partition_name	partition_type	tablespace_name	boundaries
14				
15				
16	partition_orders_4	r	example2	
17	p2	p	example4	{200}
18	p_4	p	example2	{NULL}
19	p_3	p	example2	{300}
20	p1	p	example4	{100}
21	(5 rows)			

5.2.2.3 查询分区

查询分区表的分区情况：

```
1 | select relname,parttype,parentid,boundaries from pg_partition where
  | parentid in(select oid from pg_class where relname=$分区表名称);
```

查询单独分区内的数据：

```
1 | SELECT * FROM partition_orders_4 PARTITION(p2);
2 | SELECT * FROM partition_orders_4 PARTITION FOR(150);
3 | ...
```

选择分区有两种方法，一是 partition (分区名称) ， 二是 partition for (数值) ， 此括号内的数值为所选分区范围内的任意值，如果定义分区时分区键不只一个，那么此括号内的数值个数应该与定义分区时使用的分区键个数相同，并且一一对应。

5.2.2.4 数据转移

进行交换的普通表和分区必须满足如下条件：

- 普通表和分区的列数目相同，对应列的信息严格一致，包括：列名、列的数据类型、列约束、列的 Collation 信息、列的存储参数、列的压缩信息等。
- 普通表和分区的表压缩信息严格一致。
- 普通表和分区的分布列信息严格一致。
- 普通表和分区的索引个数相同，且对应索引的信息严格一致。
- 普通表和分区的表约束个数相同，且对应表约束的信息严格一致。
- 普通表不可以是临时表。

实例：

查看 orders 表中的数据量：

```

1 omm=# select count(*) from orders
2 omm-# ;
3 count
4 -----
5 300001
6 (1 row)

```

将 orders 表中的数据转移到分区表 partition_orders_4 中：

```

1 omm=# insert into partition_orders_4 select * from orders;
2 INSERT 0 300001

```

查看整个分区表情况，以及各分区情况：

```

1 omm=# select count(*) from partition_orders_4;
2 count
3 -----
4 300001
5 (1 row)

```

```
1 | select * from partition_orders_4;
```

	<u>o_orderkey</u>	<u>o_custkey</u>	<u>o_orderstatus</u>	<u>o_totalprice</u>	<u>o_orderdate</u>	<u>o_orderpriority</u>	<u>o_clerk</u>	<u>o_shipppriority</u>	<u>o_comment</u>
1	7381	0	Git Merge Status	181585.13	2019-01-02 00:00:00	5-LOW	Clerk#000000951	0	furiously special f
2	15601	0	40736.64	2019-12-02 00:00:00	1-URGENT	Clerk#000000880	0	furiously special f	
3	24664	F	221256.05	2016-10-14 00:00:00	5-LOW	Clerk#000000955	0	furiously special f	
4	27356	0	4天前	5953.65	2018-10-11 00:00:00	5-LOW	Clerk#000000124	0	furiously special f
5	8897	F	198693.73	2017-07-30 00:00:00	5-LOW	Clerk#000000925	0	furiously special f	
6	11125	F	Git Commit Message	3475.35	2015-02-21 00:00:00	4-NOT SPECIFIED	Clerk#000000958	0	furiously special f
7	7828	0	201050.68	2019-01-10 00:00:00	2-HIGH	Clerk#000000470	0	furiously special f	
32	26912	0	8天前	222247.61	2018-07-16 00:00:00	2-HIGH	Clerk#000000616	0	furiously special f
33	13393	F	83484.38	2016-10-27 00:00:00	3-MEDIUM	Clerk#000000409	0	furiously special f	
34	12202	0	CNN GAN Optimiz	163243.89	2021-07-21 00:00:00	3-MEDIUM	Clerk#000000223	0	furiously special f
35	25819	0	174658.48	2018-10-23 00:00:00	4-NOT SPECIFIED	Clerk#000000259	0	furiously special f	
36	23051	0	83979.37	2018-11-03 00:00:00	1-URGENT	Clerk#000000358	0	furiously special f	
37	17224	F	149261.54	2015-06-01 00:00:00	3-MEDIUM	Clerk#000000456	0	furiously special f	
38	24967	0	6753.73	2019-08-22 00:00:00	4-NOT SPECIFIED	Clerk#000000684	0	furiously special f	
39	16354	0	GAN Loss Behavior	150879.93	2019-09-21 00:00:00	3-MEDIUM	Clerk#000000659	0	furiously special f
64	6424	F	41098.12	2017-07-16 00:00:00	3-MEDIUM	Clerk#000000661	0	furiously special f	
65	3251	F	77712.78	2018-03-18 00:00:00	1-URGENT	Clerk#000000632	0	furiously special f	
66	25840	F	17B Benefit Mu	80001.93	2017-01-20 00:00:00	5-LOW	Clerk#000000743	0	furiously special f
67	11323	0	208931.31	2019-12-20 00:00:00	4-NOT SPECIFIED	Clerk#000000547	0	furiously special f	
68	5710	0	176912.19	2021-04-18 00:00:00	3-MEDIUM	Clerk#000000440	0	furiously special f	
69	16598	F	252570.46	2017-06-04 00:00:00	4-NOT SPECIFIED	Clerk#000000330	0	furiously special f	
78	12868	F	358593.39	2016-12-18 00:00:00	5-LOW	Clerk#000000322	0	furiously special f	
71	676	0	377135.44	2021-01-24 00:00:00	4-NOT SPECIFIED	Clerk#000000271	0	furiously special f	
96	21556	F	92038.75	2017-04-17 00:00:00	2-HIGH	Clerk#000000395	0	furiously special f	
97	4213	F	95734.73	2016-01-30 00:00:00	3-MEDIUM	Clerk#000000547	0	furiously special f	
98	20896	F	214181.33	2017-09-23 00:00:00	1-URGENT	Clerk#000000448	0	furiously special f	
99	17782	F	214996.74	2017-03-13 00:00:00	4-NOT SPECIFIED	Clerk#000000973	0	furiously special f	
100	29401	0	198797.26	2021-02-23 00:00:00	4-NOT SPECIFIED	Clerk#000000677	0	furiously special f	
101	5600	0	176464.57	2019-03-18 00:00:00	3-MEDIUM	Clerk#000000419	0	furiously special f	
102	145	0	79442.23	2020-05-01 00:00:00	2-HIGH	Clerk#000000596	0	furiously special f	
103	5821	0	134647.84	2019-06-21 00:00:00	4-NOT SPECIFIED	Clerk#000000890	0	furiously special f	
128	14792	F	67724.39	2015-06-16 00:00:00	1-URGENT	Clerk#000000385	0	furiously special f	
129	14227	F	151238.08	2015-11-20 00:00:00	5-LOW	Clerk#000000859	0	furiously special f	
138	7393	F	176613.75	2015-05-09 00:00:00	2-HIGH	Clerk#000000836	0	furiously special f	
131	18550	F	168451.68	2017-06-08 00:00:00	3-MEDIUM	Clerk#000000625	0	furiously special f	
132	5279	F	207341.91	2018-06-11 00:00:00	3-MEDIUM	Clerk#000000488	0	furiously special f	
133	8806	0	141313.38	2020-11-29 00:00:00	1-URGENT	Clerk#000000738	0	furiously special f	
134	1240	F	85072.11	2015-05-02 00:00:00	4-NOT SPECIFIED	Clerk#000000711	0	furiously special f	
135	12897	0	201635.24	2018-10-22 00:00:00	4-NOT SPECIFIED	Clerk#000000804	0	furiously special f	
168	16499	0	139910.49	2019-12-20 00:00:00	4-NOT SPECIFIED	Clerk#000000832	0	furiously special f	
161	3325	F	7317.45	2017-08-31 00:00:00	2-HIGH	Clerk#000000322	0	furiously special f	
162	2824	0	36808.86	2018-05-08 00:00:00	3-MEDIUM	Clerk#000000278	0	furiously special f	

```

1 omm=# select count(*) from partition_orders_4 partition(p2);
2 count
3 -----
4 28
5 (1 row)

```

```
1 | select * from partition_orders_4 partition(p2);
```

o_orderkey	o_custkey	o_orderstatus	o_totalprice	o_orderdate	o_orderpriority	o_clerk	o_shippriority	o_comment
100	29401	O	100.00	198797.26	2021-02-28 00:00:00	4-NOT SPECIFIED	Clerk#000000577	0 furiously special f
101	5600	O	100.00	176464.57	2019-03-18 00:00:00	3-MEDIUM	Clerk#000000419	0 furiously special f
102	145	O	100.00	79442.23	2020-05-09 00:00:00	2-HIGH	Clerk#000000596	0 furiously special f
103	5821	O	100.00	134647.84	2019-06-21 00:00:00	4-NOT SPECIFIED	Clerk#000000090	0 furiously special f
104	14792	F	100.00	67724.39	2015-06-16 00:00:00	1-URGENT	Clerk#000000385	0 furiously special f
105	14227	F	100.00	151238.00	2015-11-28 00:00:00	5-LOW	Clerk#000000859	0 furiously special f
106	7393	F	100.00	176613.75	2015-05-09 00:00:00	2-HIGH	Clerk#000000036	0 furiously special f
107	18550	F	100.00	168451.68	2017-06-08 00:00:00	3-MEDIUM	Clerk#000000625	0 furiously special f
108	5279	F	100.00	207341.91	2016-06-11 00:00:00	3-MEDIUM	Clerk#000000488	0 furiously special f
109	8800	O	100.00	141313.30	2020-11-29 00:00:00	1-URGENT	Clerk#000000738	0 furiously special f
110	1240	F	100.00	85072.11	2015-05-02 00:00:00	4-NOT SPECIFIED	Clerk#000000711	0 furiously special f
111	12897	O	100.00	201635.24	2018-10-21 00:00:00	4-NOT SPECIFIED	Clerk#000000884	0 furiously special f
112	16449	O	100.00	139918.49	2019-12-28 00:00:00	4-NOT SPECIFIED	Clerk#000000342	0 furiously special f
113	3325	F	100.00	7317.45	2017-08-31 00:00:00	2-HIGH	Clerk#000000322	0 furiously special f
114	2824	O	100.00	36898.86	2018-05-08 00:00:00	3-MEDIUM	Clerk#000000378	0 furiously special f
115	17552	O	100.00	154353.54	2020-09-05 00:00:00	3-MEDIUM	Clerk#000000379	0 furiously special f
116	157	F	100.00	216564.91	2015-10-22 00:00:00	5-LOW	Clerk#000000289	0 furiously special f
117	5449	F	100.00	138418.61	2016-01-31 00:00:00	4-NOT SPECIFIED	Clerk#000000292	0 furiously special f
118	21563	O	100.00	92321.94	2018-09-12 00:00:00	2-HIGH	Clerk#000000440	0 furiously special f
119	23881	F	100.00	123341.32	2016-01-05 00:00:00	4-NOT SPECIFIED	Clerk#000000731	0 furiously special f
120	16514	O	100.00	151679.90	2020-11-25 00:00:00	5-LOW	Clerk#000000483	0 furiously special f
121	15814	F	100.00	114859.26	2016-08-08 00:00:00	1-URGENT	Clerk#000000025	0 furiously special f
122	12346	F	100.00	295610.18	2015-04-06 00:00:00	3-MEDIUM	Clerk#000000352	0 furiously special f
123	27985	F	100.00	176587.32	2016-12-28 00:00:00	3-MEDIUM	Clerk#000000216	0 furiously special f
124	12965	F	100.00	72222.51	2016-03-17 00:00:00	2-HIGH	Clerk#000000988	0 furiously special f
125	6583	P	100.00	187336.46	2018-04-07 00:00:00	2-HIGH	Clerk#000000949	0 furiously special f
126	22845	O	100.00	207295.82	2021-01-02 00:00:00	4-NOT SPECIFIED	Clerk#000000331	0 furiously special f
127	10594	O	100.00	116756.56	2019-03-08 00:00:00	2-HIGH	Clerk#000000489	0 furiously special f

5.3 索引

索引可以提高数据的访问速度，但同时也增加了插入、更新和删除操作的处理时间。所以是否要为表增加索引，索引建立在哪些字段上，是创建索引前必须要考虑的问题。需要分析应用程序的业务处理、数据使用、经常被用作查询的条件或者被要求排序的字段来确定是否建立索引。

索引经常建立在数据库表中的以下列上：

- 在经常需要搜索查询的列上创建索引，可以加快搜索的速度。
- 在作为主键的列上创建索引，强制该列的唯一性和组织表中数据的排列结构。
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的。
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间。
- 在经常使用 WHERE 子句的列上创建索引，加快条件的判断速度。
- 为经常出现在关键字 ORDER BY、GROUP BY、DISTINCT 后面的字段建立索引。

索引方式：

- 唯一索引：可用于约束索引属性值的唯一性，或者属性组合值的唯一性。如果一个表声明了唯一约束或者主键，则 openGauss 自动在组成主键或唯一约束的字段上创建唯一索引（可能是多字段索引），以实现这些约束。目前，openGauss 只有 B-Tree 可以创建唯一索引。
- 多字段索引：一个索引可以定义在表中的多个属性上。目前，openGauss 中的 B-Tree 支持多字段索引，且最多可在 32 个字段上创建索引（全局分区索引最多支持 31 个字段）。
- 部分索引：建立在一个表的子集上的索引，这种索引方式只包含满足条件表达式的元组。

4. 表达式索引：索引建立在一个函数或者从表中一个或多个属性计算出来的表达式上。表达式索引只有在查询时使用与创建时相同的表达式才会起作用。

注意：

- 索引创建成功后，系统会自动判断何时引用索引。当系统认为使用索引比顺序扫描更快时，就会使用索引。
- 索引创建成功后，必须和表保持同步以保证能够准确地找到新数据，这样就增加了数据操作的负荷。因此需定期删除无用的索引。

5.3.1 普通表上创建管理索引

5.3.1.1 创建索引

步骤1：创建 orders 表的备份 orderscopy2，并将数据导入

```
1  create table orderscopy2(
2    o_orderkey integer,
3    o_custkey integer,
4    o_orderstatus char(1),
5    o_totalprice decimal(15,2),
6    o_orderdate date,
7    o_orderpriority char(15),
8    o_clerk char(15),
9    o_shipppriority integer,
10   o_comment varchar(79),
11   PRIMARY KEY (o_orderkey)
12 )
13   tablespace example2;
14
15   INSERT INTO orderscopy2
16   SELECT * FROM orders;
```

```
1 [omm@cfde74516988 ~]$ gsql -f task5.sql
2 gsql:task5.sql:13: NOTICE: CREATE TABLE / PRIMARY KEY will create
3 implicit index "orderscopy2_pkey" for table "orderscopy2"
4 CREATE TABLE
5 INSERT 0 300001
5 total time: 501 ms
```

步骤2：创建普通索引

如果对于 orderscopy2 表，需要经常进行以下查询：

```
1  SELECT * FROM orderscopy2 WHERE o_totalprice>10000;
```

使用以下命令创建索引：

```
1 | omm=# CREATE INDEX orderscopy2_index_totalprice ON  
2 |   orderscopy2(o_totalprice);  
2 | CREATE INDEX
```

步骤3：创建多字段索引

假如用户需要经常查询表 orderscopy2 中 `o_orderstatus` 是 'O'，且 `o_totalprice` 小于 10000 的记录，使用以下命令进行查询：

```
1 | SELECT * FROM orderscopy2 WHERE o_orderstatus='O' AND  
2 |   o_totalprice<10000;
```

使用以下命令在字段 `o_orderstatus` 和 `o_totalprice` 上定义一个多字段索引：

```
1 | omm=# CREATE INDEX orderscopy2_index_more_column ON  
2 |   orderscopy2(o_orderstatus, o_totalprice);  
2 | CREATE INDEX
```

步骤4：创建部分索引

如果只需要查询 `o_orderstatus` 为 'O' 的记录：

```
1 | SELECT * FROM orderscopy2 WHERE o_orderstatus='O';
```

可以创建部分索引来提升查询效率：

```
1 | omm=# CREATE INDEX orderscopy2_part_index ON  
2 |   orderscopy2(o_orderstatus) WHERE o_orderstatus='O';  
2 | CREATE INDEX
```

步骤5：创建表达式索引

假如经常需要查询代号以 8 结尾的收银员的信息，执行如下命令进行查询：

```
1 | SELECT * FROM orderscopy2 WHERE o_clerk like '%8';
```

可以为上面的查询创建表达式索引：

```
1 | omm=# CREATE INDEX orderscopy2_para_index ON  
2 |   orderscopy2(reverse(o_clerk) varchar_pattern_ops);  
2 | CREATE INDEX
```

5.3.1.2 管理索引

步骤1：查询索引

执行如下命令查询系统和用户定义的所有索引：

```
1 omm=# SELECT RELNAME FROM PG_CLASS WHERE RELKIND='i';
2
3          relname
4
5 -----
6
7 region_pkey
8 nation_pkey
9 part_pkey
10 supplier_pkey
11 partsupp_pkey
12 customer_pkey
13 lineitem_pkey
14 orders_pkey
15 lineitemcopy1_pkey
16 pg_toast_2619_index
17 pg_toast_3220_index
18 pg_toast_1255_index
19 lineitemcopy2_pk
20 lineitemcopy2_uk
21 customercopy1_pkey
22 orderscopy2_pkey
23 orderscopy2_index_totalprice
24 orderscopy2_index_more_column
25 orderscopy1_pkey
26 orderscopy2_part_index
27 orderscopy2_para_index
28 ...
29
```

可看到上述创建的索引。

\di+ 可以查询用户定义的所有索引：

List of relations							
Schema	Name	Type	Owner	Table	Size	Storage	Description
public	customer_pkey	index	omm	customer	680 kB		
public	customercopy1_pkey	index	omm	customercopy1	672 kB		
public	lineitem_pkey	index	omm	lineitem	103 MB		
public	lineitemcopy1_pkey	index	omm	lineitemcopy1	25 MB		
public	lineitemcopy2_pk	index	omm	lineitemcopy2	8192 bytes		
public	lineitemcopy2_uk	index	omm	lineitemcopy2	8192 bytes		
public	nation_pkey	index	omm	nation	16 kB		
public	orders_pkey	index	omm	orders	6616 kB		
public	orderscopy1_pkey	index	omm	orderscopy1	6600 kB		
public	orderscopy2_index_more_column	index	omm	orderscopy2	9264 kB		
public	orderscopy2_index_totalprice	index	omm	orderscopy2	9192 kB		
public	orderscopy2_para_index	index	omm	orderscopy2	9264 kB		
public	orderscopy2_part_index	index	omm	orderscopy2	3240 kB		
public	orderscopy2_pkey	index	omm	orderscopy2	6600 kB		
public	part_pkey	index	omm	part	896 kB		
public	partition_orders_1_pkey	index	omm	partition_orders_1	32 kB		
public	partition_orders_2_pkey	index	omm	partition_orders_2	128 kB		
public	partition_orders_3_pkey	index	omm	partition_orders_3	32 kB		
public	partition_orders_4_pkey	index	omm	partition_orders_4	6648 kB		
public	partsupp_pkey	index	omm	partsupp	3544 kB		
public	region_pkey	index	omm	region	16 kB		
public	supplier_pkey	index	omm	supplier	64 kB		
(22 rows)							

查询指定索引的信息：

List of relations							
Schema	Name	Type	Owner	Table	Size	Storage	Description
public	orderscopy2_index_totalprice	index	omm	orderscopy2	9192 kB		

步骤2：重命名索引

执行如下命令对索引 orderscopy2_index_more_column 重命名为 orderscopy2_index_orderstatus_totalprice：

```
1 omm=# ALTER INDEX orderscopy2_index_more_column RENAME TO
2   orderscopy2_index_orderstatus_totalprice;
3 ALTER INDEX
```

用 \di+ 查看，索引重命名成功：

List of relations							
Schema	Name	Type	Owner	Table	Size	Storage	Description
public	customer_pkey	INDEX	omm	customer	680 kB		
public	customercopy1_pkey	INDEX	omm	customercopy1	672 kB		
public	lineitem_pkey	INDEX	omm	lineitem	103 kB		
public	lineitemcopy1_pkey	INDEX	omm	lineitemcopy1	25 kB		
public	lineitemcopy2_pk	INDEX	omm	lineitemcopy2	8192 bytes		
public	lineitemcopy2_uk	INDEX	omm	lineitemcopy2	8192 bytes		
public	nation_pkey	INDEX	omm	nation	16 kB		
public	orders_pkey	INDEX	omm	orders	6616 kB		
public	orderscopy1_pkey	INDEX	omm	orderscopy1	6600 kB		
public	orderscopy2_index_orderstatus_totalprice	INDEX	omm	orderscopy2	9264 kB		
public	orderscopy2_index_totalprice	INDEX	omm	orderscopy2	9192 kB		
public	orderscopy2_para_index	INDEX	omm	orderscopy2	9264 kB		
public	orderscopy2_part_index	INDEX	omm	orderscopy2	3240 kB		
public	orderscopy2_pkey	INDEX	omm	orderscopy2_index_orderstatus_totalprice	6600 kB		
public	part_pkey	INDEX	omm	part	896 kB		
public	partition_orders_1_pkey	INDEX	omm	partition_orders_1	32 kB		
public	partition_orders_2_pkey	INDEX	omm	partition_orders_2	128 kB		
public	partition_orders_3_pkey	INDEX	omm	partition_orders_3	32 kB		
public	partition_orders_4_pkey	INDEX	omm	partition_orders_4	6648 kB		
public	partsupp_pkey	INDEX	omm	partsupp	3544 kB		
public	region_pkey	INDEX	omm	region	16 kB		
public	supplier_pkey	INDEX	omm	supplier	64 kB		

步骤3：删除索引

删除 orderscopy2_para_index 索引：

```
1 omm=# DROP INDEX orderscopy2_para_index;
2 DROP INDEX
```

可看到删除成功：

List of relations							
Schema	Name	Type	Owner	Table	Size	Storage	Description
public	customer_pkey	index	omm	customer	680 kB		
public	customercopy1_pkey	index	omm	customercopy1	672 kB		
public	lineitem_pkey	index	omm	lineitem	103 MB		
public	lineitemcopy1_pkey	index	omm	lineitemcopy1	25 MB		
public	lineitemcopy2_pk	index	omm	lineitemcopy2	8192 bytes		
public	lineitemcopy2_uk	index	omm	lineitemcopy2	8192 bytes		
public	nation_pkey	index	omm	nation	16 kB		
public	orders_pkey	index	omm	orders	6616 kB		
public	orderscopy1_pkey	index	omm	orderscopy1	6600 kB		
public	orderscopy2_index_orderstatus_totalprice	index	omm	orderscopy2	9264 kB		
public	orderscopy2_index_totalprice	index	omm	orderscopy2	3240 kB		
public	orderscopy2_part_index	index	omm	orderscopy2	6600 kB		
public	part_pkey	index	omm	part	896 kB		
public	partition_orders_1_pkey	index	omm	partition_orders_1	32 kB		
public	partition_orders_2_pkey	index	omm	partition_orders_2	128 kB		
public	partition_orders_3_pkey	index	omm	partition_orders_3	32 kB		
public	partition_orders_4_pkey	index	omm	partition_orders_4	6648 kB		
public	partsupp_pkey	index	omm	partsupp	3544 kB		
public	region_pkey	index	omm	region	16 kB		
public	supplier_pkey	index	omm	supplier	64 kB		
(21 rows)							

5.3.2 分区表上创建管理索引

5.3.2.1 创建索引

分区表索引分为 LOCAL 索引与 GLOBAL 索引，一个 LOCAL 索引对应一个具体分区(有多少个分区就有多少个索引文件)，而 GLOBAL 索引则对应整个分区表（只有一个索引文件）。

步骤1：以 orders 表为例创建一个分区表

```

1  create table partition_orders_0(
2    o_orderkey integer,
3    o_custkey integer,
4    o_orderstatus char(1),
5    o_totalprice decimal(15,2),
6    o_orderdate date,
7    o_orderpriority char(15),
8    o_clerk char(15),
9    o_shipppriority integer,
10   o_comment varchar(79),
11   PRIMARY KEY (o_orderkey)
12 )
13  tablespace example2
14  partition by range(o_orderkey)
15  (
16    partition p1 values less than(100),
17    partition p2 values less than(200),
18    partition p3 values less than(300),
19    partition p4 values less than(maxvalue)
20  );

```

```

1 [omm@cfde74516988 ~]$ gsql -f task6.sql
2 gsql:task6.sql:20: NOTICE: CREATE TABLE / PRIMARY KEY will create
implicit index "partition_orders_0_pkey" for table
"partition_orders_0"
3 CREATE TABLE
4 total time: 13 ms

```

步骤2：创建 GLOBAL 索引

在 o_custkey 上创建分区表 GLOBAL 索引，存储在表空间 example2 上：

```

1 omm=# create index global_index_custkey on
partition_orders_0(o_custkey) global tablespace example2;
2 CREATE INDEX

```

List of relations								
Schema	Name	Type	Owner	Table	Size	Storage	Description	
步骤2：创建 GLOBAL 索引								
-	customer_pkey	index	omm	customer	680 kB		will create implicit index	
public	customercopy1_pkey	index	omm	customercopy1	672 kB			
public	global_index_custkey	global partition index	omm	partition_orders_0	8192 bytes			
public	lineitem_pkey	index	omm	lineitem	103 MB			
public	lineitemcopy1_pkey	index	omm	lineitemcopy1	25 MB			
public	lineitemcopy2_pk	index	omm	lineitemcopy2	8192 bytes			
public	lineitemcopy2_uk	index	omm	lineitemcopy2	8192 bytes			
public	nation_pkey	index	omm	nation	16 kB			
public	orders_pkey	index	omm	orders	6616 kB			
public	orderscopy1_pkey	index	omm	orderscopy1	6600 kB			
public	orderscopy2_index_orderstatus_totalprice	index	omm	orderscopy2	9264 kB			
public	orderscopy2_index_orderstatus_totalprice	index	omm	orderscopy2	9192 kB			
public	orderscopy2_part_index	index	omm	orderscopy2	3240 kB			
public	orderscopy2_pkey	index	omm	orderscopy2	6600 kB			
public	part_pkey	index	omm	part	896 kB			
public	partition_orders_0_pkey	index	omm	partition_orders_0	32 kB			
public	partition_orders_1_pkey	index	omm	partition_orders_1	32 kB			
public	partition_orders_2_pkey	index	omm	partition_orders_2	128 kB			
public	partition_orders_3_pkey	index	omm	partition_orders_3	32 kB			
public	partition_orders_4_pkey	index	omm	partition_orders_4	6648 kB			
public	partsupp_pkey	index	omm	partsupp	3544 kB			
public	region_pkey	index	omm	region	16 kB			
public	supplier_pkey	index	omm	supplier	64 kB			

在容器内 /var/lib/opengauss/data/pg_location/tablespace/tablespace_2/16384 路径下可看到：

```

[omm@cfde74516988 16384]$ ls -l
total 68648
-rw----- 1 omm omm Scripts      0 Dec 22 09:47 16722
-rw----- 1 omm omm 8192 Dec 22 14:58 16799
-rw----- 1 omm omm 35102720 Dec 22 14:58 16807
-rw----- 1 omm omm 32768 Dec 22 14:58 16807_fsm
-rw----- 1 omm omm 35110912 Dec 23 10:22 16810
-rw----- 1 omm omm 32768 Dec 23 10:20 16810_fsm
-rw----- 1 omm omm 0 Dec 23 12:42 16823
-rw----- 1 omm omm 0 Dec 23 12:42 16824
-rw----- 1 omm omm 0 Dec 23 12:42 16825
-rw----- 1 omm omm 0 Dec 23 12:42 16826
-rw----- 1 omm omm 8192 Dec 23 12:51 16833
[omm@cfde74516988 16384]$ (不同的表空间上)

```

只有一个索引文件（前面 4 个是上面创建的 partition_orders_0 分区表，4 个分区对应 4 个文件）。

步骤3：创建不指定索引分区名称的 LOCAL 索引

在 o_totalprice 上创建分区表 LOCAL 索引，不指定索引分区的名称，存储在 example3 上
(原表与索引可以分别存储在不同的表空间上)

```
1 omm=# create index local_index_totalprice on
2 partition_orders_0(o_totalprice) local tablespace example3;
3 CREATE INDEX
4 omm=# \q
5 [omm@cfde74516988 16384]$ cd ../../../../../../tablespace_3
6 [omm@cfde74516988 tablespace_3]$ cd PG_9.2_201611171_gaussdb/
7 [omm@cfde74516988 PG_9.2_201611171_gaussdb]$ cd 16384/
8 [omm@cfde74516988 16384]$ ls -l
total 32
9 -rw----- 1 omm omm 0 Dec 22 09:50 16725
10 -rw----- 1 omm omm 8192 Dec 23 13:23 16841
11 -rw----- 1 omm omm 8192 Dec 23 13:23 16842
12 -rw----- 1 omm omm 8192 Dec 23 13:23 16843
13 -rw----- 1 omm omm 8192 Dec 23 13:23 16844
```

有 4 个索引文件（因为有 4 个分区）。

步骤4：创建指定索引分区名称的 LOCAL 索引

在 o_shippriority 上创建分区表 LOCAL 索引，指定索引分区的名称，p1, p2 分区索引存储在 example3 上，p3, p4f 分区索引存储在 example4 上（不同分区的索引可以分别存储在不同的表空间上）：

创建成功后查看：

		List of relations					
Schema	Name	Type	Owner	Table	Size	Storage	Description
	创建成功后						
public	customer_pkey	index	omm	customer	680 kB		
public	customercopy1_pkey	index	omm	customercopy1	672 kB		
public	global_index_custkey	global partition index	omm	partition_orders_0	8192 bytes		
public	lineitem_pkey	index	omm	lineitem	103 MB		
public	lineitemcopy1_pkey	index	omm	lineitemcopy1	25 MB		
public	lineitemcopy2_pk	index	omm	lineitemcopy2	8192 bytes		
public	lineitemcopy2_uk	index	omm	lineitemcopy2	8192 bytes		
public	local_index_shippriority	index	omm	partition_orders_0	32 kB		
public	nation_pkey	index	omm	nation	16 kB		
public	orders_pkey	index	omm	orders	6616 kB		
public	orderscopy1_pkey	index	omm	orderscopy1	6600 kB		

```
1 [omm@cfde74516988 tablespace]$ ls
2 tablespace_1  tablespace_2  tablespace_3  tablespace_4
3 [omm@cfde74516988 tablespace]$ cd tablespace_3
4 [omm@cfde74516988 tablespace_3]$ cd PG_9.2_201611171_gaussdb/
5 [omm@cfde74516988 PG_9.2_201611171_gaussdb]$ cd 16384/
6 [omm@cfde74516988 16384]$ ls -l
7 total 16
8 -rw----- 1 omm omm 0 Dec 22 09:50 16725
9 -rw----- 1 omm omm 8192 Dec 23 13:07 16835
10 -rw----- 1 omm omm 8192 Dec 23 13:07 16836
11 [omm@cfde74516988 16384]$ cd ../../../../../../tablespace_4
12 [omm@cfde74516988 tablespace_4]$ cd PG_9.2_201611171_gaussdb/
13 [omm@cfde74516988 PG_9.2_201611171_gaussdb]$ cd 16384/
14 [omm@cfde74516988 16384]$ ls -l
15 total 32
16 -rw----- 1 omm omm 8192 Dec 22 14:58 16798
17 -rw----- 1 omm omm 8192 Dec 22 14:58 16809
18 -rw----- 1 omm omm 8192 Dec 23 13:07 16837
19 -rw----- 1 omm omm 8192 Dec 23 13:07 16838
```

分别在 example3 有 2 个索引文件和 example4 有 2 个索引文件。

5.3.2.2 管理索引

步骤1：修改索引分区所在的表空间

将分区索引 p1_index 从 example3 移到 example2：

```
1 omm=# ALTER INDEX local_index_shippriority MOVE PARTITION p1_index
2 TABLESPACE example2;
2 ALTER INDEX
```

步骤2：重命名索引分区

将分区索引 p2_index 重命名为 p2_index_new：

```
1 omm=# ALTER INDEX local_index_shippriority RENAME PARTITION p2_index
2 TO p2_index_new;
2 ALTER INDEX
```

步骤3：删除索引

要删除索引只能删除整个索引，不能删除单独的分区索引：

```
1 omm=# drop index global_index_custkey;
2 DROP INDEX
3 omm=# drop index local_index_totalprice;
4 DROP INDEX
5 omm=# drop index local_index_shippriority;
6 DROP INDEX
```

5.4 OpenGauss 段页式特性

由于段页式特性在 OpenGauss 3.1.0 及以上才能使用，故本部分实验在某其他组的 OpenGauss 5.0.3 容器进行。

openGauss 通用的普通表，每个数据表对应一个逻辑逻辑上的大文件（最大 32T），该逻辑文件又按照固定的大小划分多个实际文件存在对应的数据库目录下面。所以，每张数据表随着数据量的增多，底层的数据存储所需文件数量会逐渐增多。同时，openGauss 对外提供 hashbucket 表、大分区表等特性，每张数据表会被拆分为若干个子表，底层所需文件数量更是成倍增长。由此，这种存储管理模式存在以下问题：

1. 对文件系统依赖大，无法进行细粒度的控制提升可维护性；
2. 大数据量下文件句柄过多，目前只能依赖虚拟句柄来解决，影响系统性能；
3. 小文件数量过多会导致全量 build、全量备份等场景下的随机 IO 问题，影响性能；

为了解决以上问题，openGauss 引入段页式存储管理机制，类似于操作系统的段页式内存管理，但是在实现机制上区别很大。

本实验通过在 TPC-H 测试基准的 orders 表的备份表上应用段页式存储的相关函数操作，观察应用段页式存储后的效果。

5.4.1 使用指导

用户在用 SQL 语句 create table 建表时可以通过指定参数 segment=on，使得行存表可以使用段页式的方式存储数据。目前段页式存储不支持列存表。段页式表空间是自动创建的，不需要用户有额外的命令。

1. 以订单表 **orders** 为例，指定参数 **segment=on**，创建段页式普通表

zzy.sql

```
1 CREATE TABLE orderszzy(
2   o_orderkey integer,
3   o_custkey integer,
4   o_orderstatus char(1),
5   o_totalprice decimal(15,2),
6   o_orderdate date,
7   o_orderpriority char(15),
8   o_clerk char(15),
9   o_shipppriority integer,
10  o_comment varchar(79),
11  PRIMARY KEY (o_orderkey)
12 )
13 with(segment=on);
```

```
1 omm@openauss:~/openGauss$ gsql -d homework -f zzy.sql
2 gsql:zzy.sql:13: NOTICE:  CREATE TABLE / PRIMARY KEY will create
3 implicit index "orderszzy_pkey" for table "orderszzy"
4 CREATE TABLE
4 total time: 12 ms
```

创建成功后，导入 **orders** 表数据。

```
1 homework=# insert into orderszzy select * from orders;
2 INSERT 0 300001
```

为了让用户更好使用段页式功能，openGauss 提供了两个 built in 的系统函数，显示 extent 的使用情况。用户可以使用这两个视图，决定是否回收和回收哪一部分的数据。

```
•  
1 local_segment_space_info(tablespacename TEXT, databasename TEXT)
```

描述：输出为该表空间下所有 ExtentGroup 的使用信息。

名称	描述
node_name	节点名称
extent_size	该 ExtentGroup 的 extent 规格，单位是 block 数
forknum	Fork 号
total_blocks	物理文件总 extent 数目
meta_data_blocks	表空间管理的 metadata 占用的 block 数，只包括 space header、map page 等，不包括 segment head
used_data_blocks	存数据占用的 extent 数目。包括 segment head
utilization	使用的 block 数占总 block 数的百分比，即 $(\text{used_data_blocks} + \text{meta_data_block}) / \text{total_blocks}$
high_water_mark	高水位线，被分配出去的 extent，最大的物理页号。超过高水位线的 block 都没有被使用，可以被直接回收

以新建 orderszzy 表所在表空间和数据库为例，查看 local_segment_space_info() 函数的输出结果：

```
homework=# select * from local_segment_space_info('pg_default', 'homework')
homework-# ;
node_name | extent_size | forknum | total_blocks | meta_data_blocks | used_data_blocks | utilization | high_water_mark
-----+-----+-----+-----+-----+-----+-----+-----+
gaussdb | 16384 | 1 | 16384 | 4157 | 8 | .254211 | 4165
gaussdb | 16384 | 8 | 16384 | 4157 | 520 | .285461 | 4677
gaussdb | 16384 | 8 | 16384 | 4157 | 16 | .2547 | 4173
gaussdb | 16384 | 128 | 16384 | 4157 | 9472 | .831848 | 13629
(4 rows)
```

```
1 pg_stat_segment_extent_usage(int4 tablespace oid, int4 database oid,
int4 extent_type, int4 forknum)
```

描述：每次返回一个 ExtentGroup 中，每个被分配出去的 extent 的使用情况。**extent_type** 表示 ExtentGroup 的类型，合理取值为 [1,5] 的 int 值。在此范围外的会报 error。**forknum** 表示 fork 号，合法取值为 [0,4] 的 int 值，目前只有三种值有效，数据文件为 0，FSM 文件为 1，visibility map 文件为 2。

名称	描述
start_block	Extent 的起始物理页号
extent_size	Extent 的大小
usage_type	Extent 的使用类型，比如 segment head、data extent 等
owner_location	有指针指向该 extent 的对象的位置。比如 data extent 的 owner 就是它所属的 segment 的 head 位置
special_data	该 extent 在它 owner 中的位置。该字段的数据与使用类型有关。比如 data extent 的 special data 就是它在所属 segment 中的 extent id

查找 orderszzy 表所在表空间和数据库的 oid，查看 pg_stat_segment_extent_usage() 函数的输出结果：

```
homework=# select * from pg_stat_segment_extent_usage((select oid::int4 from pg_tablespace where spcname='pg_default'), (select oid::int4 from pg_database where datname='homework'), 1, 0);
 start_block | extent_size | usage_type | owner_location | special_data
-----+-----+-----+-----+-----+
 4157 | 1 | Non-bucket table segment head | 4294967295 | 0
 4158 | 1 | Non-bucket table segment head | 4294967295 | 0
 4159 | 1 | Non-bucket table fork head | 4157 | 1
 4160 | 1 | Non-bucket table segment head | 4294967295 | 0
 4161 | 1 | Non-bucket table segment head | 4294967295 | 0
 4162 | 1 | Non-bucket table segment head | 4294967295 | 0
 4163 | 1 | Non-bucket table segment head | 4294967295 | 0
 4164 | 1 | Non-bucket table fork head | 4162 | 1
(8 rows)
```

```
select * from pg_stat_segment_extent_usage((select oid::int4 from pg_tablespace where spcname='pg_default'), (select oid::int4 from pg_database where datname='homework'), 2, 0);
```

```
homework=# select * from pg_stat_segment_extent_usage((select oid::int4 from pg_tablespace where spcname='pg_default'), (select oid::int4 from pg_database where datname='homework'), 2, 0);
 start_block | extent_size | usage_type | owner_location | special_data
-----+-----+-----+-----+-----+
 4157 | 8 | Data extent | 4158 | 0
 4165 | 8 | Data extent | 4157 | 0
 4173 | 8 | Data extent | 4157 | 1
 4181 | 102406 | Data extent | 4157 | 2
 4189 | 102407 | Data extent | 4157 | 3
 4197 | 102408 | Data extent | 4158 | 1
 4205 | 102409 | Data extent | 4157 | 4
 4213 | 102410 | Data extent | 4157 | 5
 4221 | 102411 | Data extent | 4157 | 6
 4229 | 102412 | Data extent | 4157 | 7
 4237 | 102413 | Data extent | 4157 | 8
 4245 | 102414 | Data extent | 4158 | 2
 4253 | 102415 | Data extent | 4157 | 9
 4261 | 102416 | Data extent | 4157 | 10
 4269 | 102417 | Data extent | 4157 | 11
 4277 | 8 | Data extent | 4157 | 12
 4285 | 8 | Data extent | 4157 | 13
 4293 | 8 | Data extent | 4158 | 3
 4301 | 8 | Data extent | 4157 | 14
 4309 | 8 | Data extent | 4157 | 15
(15 rows)
```

```
1 | gs_space_shrink(int4 tablespace, int4 database, int4 extent_type,  
int4 forknum)
```

描述：当前节点上对指定段页式空间做物理空间收缩。注意，目前只支持对当前连接的 database 做 shrink。（对指定段页式空间做物理空间收缩），传入的参数是 tablespace 和 database 的 oid, extent_type 为 [2,5] 的 int 值。注意：extent_type = 1 表示段页式元数据，目前不支持对元数据所在的物理文件做收缩。该函数仅限工具使用，不建议用户直接使用。

6 数据库查询优化

一、openGauss 执行计划的查看与分析

实验步骤

1. **查询1：** 查询零部件表中零售价小于920，且供应商key为5的零件key。
2. **查询2：** 查询客户表中账户余额小于1000且客户国家为ALGERIA的客户名称和电话。

实验代码

```
1 -- 查询1
2 EXPLAIN
3 SELECT p_partkey
4 FROM part
5 WHERE p_retailprice < 920 AND p_partkey IN (SELECT ps_partkey FROM
partsupp WHERE ps_suppkey = 5);
6
7 -- 查询2
8 EXPLAIN
9 SELECT c_name, c_phone
10 FROM customer
11 WHERE c_acctbal < 1000 AND c_nationkey = (SELECT n_nationkey FROM
nation WHERE n_name = 'ALGERIA');
```

执行计划

查询1

执行计划

```
1 QUERY PLAN
2 -----
3 Nested Loop Semi Join (cost=0.00..1999.07 rows=78 width=4)
4   -> Seq Scan on part (cost=0.00..1352.00 rows=281 width=4)
5     Filter: (p_retailprice < 920::numeric)
6   -> Index Only Scan using partsupp_pkey on partsupp
7     (cost=0.00..7.19 rows=21 width=4)
8       Index Cond: ((ps_partkey = part.p_partkey) AND (ps_suppkey
= 5))
9       (5 rows)
```

关系代数的实现方式

1. 使用了半连接（Semi Join），结合了顺序扫描和索引扫描，适合在数据量较大的情况下优化性能。通过索引扫描 partsupp 表，减少了需要扫描的数据行数。

执行成本

1. 外层扫描 `part` 表的成本: `0.00..1352.00`
 2. 内层索引扫描 `partsupp` 表的成本: `0.00..7.19`
 3. 总体成本: `0.00..1999.07`, 表示该查询的执行成本相对较低, 主要是因为 `partsupp` 表上的索引扫描很高效。
-

查询2

执行计划

```
1 | QUERY PLAN
2 |
3 |-----|
3 | Seq Scan on customer (cost=1.31..1170.31 rows=5495 width=35)
4 |   Filter: ((c_acctbal < 1000::numeric) AND (c_nationkey = $0))
5 |   InitPlan 1 (returns $0)
6 |     -> Seq Scan on nation (cost=0.00..1.31 rows=1 width=4)
7 |       Filter: (n_name = 'ALGERIA'::bpchar)
8 | (5 rows)
```

关系代数的实现方式

1. 使用了两次顺序扫描: 一次扫描 `nation` 表查找 `ALGERIA`, 然后对 `customer` 表进行扫描, 利用内层查询返回的 `nationkey` 过滤外层结果。由于没有合适的索引, 这个查询的执行效率较低。

执行成本

1. 外层扫描 `customer` 表的成本: `cost=1.31..1170.31`
 2. 内层扫描 `nation` 表的成本: `cost=0.00..1.31`
 3. 总体成本: `cost=1.31..1170.31`, 这个查询的总体成本较高, 主要是由于 `customer` 表的大量数据需要扫描。
-

二、观察视图查询和 WITH 临时视图查询的执行计划

实验步骤

1. 在 `nation` 表上创建视图 `nation_view`。
2. 使用 `WITH` 临时视图 `nation_tempview`。
3. 比较视图、`WITH` 临时视图和直接查询的执行计划和执行时间。

实验代码

```
1 | -- 创建视图
```

```
2 CREATE VIEW nation_view AS
3   SELECT * FROM nation;
4
5   -- 通过视图查询
6 EXPLAIN ANALYZE
7   SELECT n_name FROM nation_view;
8
9   -- 使用 WITH 临时视图查询
10 EXPLAIN ANALYZE
11 WITH nation_tempview AS (
12   SELECT * FROM nation
13 )
14   SELECT * FROM nation_tempview;
15
16   -- 直接查询
17 EXPLAIN ANALYZE
18   SELECT n_name
19     FROM nation;
```

执行计划

通过视图查询

执行计划

```
1 QUERY PLAN
2 -----
3   Seq Scan on nation  (cost=0.00..1.25 rows=25 width=104) (actual
4   time=0.009..0.015 rows=25 loops=1)
   Total runtime: 0.070 ms
```

分析

1. 执行方式：顺序扫描 `nation` 表。
2. 成本：`cost=0.00..1.25`，表扫描效率较高。
3. 总运行时间：`0.070 ms`，性能较优。
4. 结论：对于 `nation` 表这种数据量较小的表，顺序扫描是非常高效的，因此即使是通过视图查询，执行时间也非常快。

使用 WITH 临时视图查询

执行计划

```
1 | QUERY PLAN
2 |
3 | -----
4 | CTE Scan on nation_tempview (cost=1.25..1.75 rows=25 width=434)
5 |   (actual time=0.024..0.047 rows=25 loops=1)
6 |     CTE nation_tempview
7 |       -> Seq Scan on nation (cost=0.00..1.25 rows=25 width=434)
8 |         (actual time=0.017..0.024 rows=25 loops=1)
9 | Total runtime: 0.178 ms
```

分析

1. 执行方式：引入 CTE（临时视图）。
2. 成本：`cost=1.25..1.75`，引入额外的扫描和数据存储开销。
3. 总运行时间：`0.178 ms`，高于视图查询和直接查询。
4. 结论：使用 `WITH` 临时视图查询时，相比直接查询和视图查询，会引入一些额外的开销。尽管 CTE 提供了查询结构上的便利（尤其是在复杂查询中），但在这种简单查询下，开销较大。

直接查询

执行计划

```
1 | QUERY PLAN
2 |
3 | -----
4 | Seq Scan on nation (cost=0.00..1.25 rows=25 width=104) (actual
5 |   time=0.009..0.015 rows=25 loops=1)
6 | Total runtime: 0.065 ms
```

分析

1. 执行方式：顺序扫描。
 2. 成本：`cost=0.00..1.25`，与视图查询一致。
 3. 总运行时间：`0.065 ms`，为最优性能。
- 结论：直接查询的执行时间和视图查询非常相似，几乎没有额外的性能损耗，特别是在查询非常简单时。

三、优化 SQL 语句

3.1. 复合索引左前缀

实验步骤

1. 在 `lineitem_new` 表上创建复合索引。
2. 比较有索引和无索引的查询性能。

实验代码

```
1  -- 创建备份表
2  CREATE TABLE lineitem_new AS TABLE lineitem;
3
4  -- 创建复合索引
5  CREATE INDEX lineitem_index ON lineitem_new(l_quantity, l_tax,
6  l_extendedprice);
7
8  -- 无索引查询l_quantity=24的全部数据
9  EXPLAIN ANALYZE
10 SELECT *
11 FROM lineitem
12 WHERE l_quantity = 24;
13
14 -- 使用最左前缀索引查询l_quantity=24的全部数据
15 EXPLAIN ANALYZE
16 SELECT *
17 FROM lineitem_new
18 WHERE l_quantity = 24;
19
20 -- 无索引查询l_tax=0.02的全部数据
21 EXPLAIN ANALYZE
22 SELECT *
23 FROM lineitem
24 WHERE l_tax=0.02;
25
26 -- 使用最左前缀索引查询l_tax=0.02的全部数据
27 EXPLAIN ANALYZE
28 SELECT *
29 FROM lineitem_new
30 WHERE l_tax=0.02;
```

执行计划

无索引查询 `**l_quantity = 24**` 的全部数据**

执行计划：

```
1 Seq Scan on lineitem (cost=0.00..97408.30 rows=20950 width=129)
   (actual time=58.944..275.978 rows=18229 loops=1)
2   Filter: (l_quantity = 24::numeric)
3     Rows Removed by Filter: 885624
4 Total runtime: 276.975 ms
5 (4 rows)
```

分析：

- 执行方式：无索引查询对 `lineitem` 表进行了顺序扫描（`Seq Scan`）。
- 过滤条件：`l_quantity = 24`。
- 行数：查询返回的行数为 18229 行，过滤掉了 885624 行不符合条件的记录。
- 实际执行时间：执行时间为 `276.975 ms`。这是因为顺序扫描需要遍历整个表，过滤掉不满足条件的行，过程较为耗时。

** 使用最左前缀索引查询 `l_quantity = 24` 的全部数据 **

执行计划：

```
1 Bitmap Heap Scan on lineitem_new (cost=512.73..20123.63 rows=18619
   width=129) (actual time=6.430..40.468 rows=18229 loops=1)
2   Recheck Cond: (l_quantity = 24::numeric)
3     -> Bitmap Index Scan on lineitem_index (cost=0.00..508.07
   rows=18619 width=0) (actual time=5.275..5.275 rows=18229 loops=1)
4       Index Cond: (l_quantity = 24::numeric)
5 Total runtime: 41.252 ms
6 (5 rows)
```

分析：

- 执行方式：使用了 `Bitmap Heap Scan` 与 `Bitmap Index Scan`，首先通过索引扫描来找出满足条件的行，然后通过堆扫描获取数据。
- 索引条件：`l_quantity = 24`。
- 实际执行时间：执行时间为 `41.252 ms`，显著低于无索引查询的执行时间。

** 无索引查询 `l_tax = 0.02` 的全部数据 **

执行计划：

```
1 Seq Scan on lineitem (cost=0.00..97408.30 rows=117803 width=129)
   (actual time=56.628..287.455 rows=100598 loops=1)
2   Filter: (l_tax = .02)
3     Rows Removed by Filter: 803255
4 Total runtime: 291.463 ms
5 (4 rows)
```

分析：

- 执行方式：无索引查询对 `lineitem` 表进行了顺序扫描（`Seq Scan`）。
- 过滤条件：`l_tax = 0.02`。
- 行数：查询返回的行数为 100598 行，过滤掉了 803255 行不符合条件的记录。
- 实际执行时间：执行时间为 `291.463 ms`，也比索引查询的时间长，原因与 `l_quantity = 24` 查询相同，都是因为需要扫描整个表。

使用最左前缀索引查询 `l_tax = 0.02` 的全部数据**

执行计划：

```
1 Seq Scan on lineitem_new (cost=0.00..30066.16 rows=97616 width=129)
  (actual time=0.044..144.631 rows=100598 loops=1)
  Filter: (l_tax = .02)
  Rows Removed by Filter: 803255
  Total runtime: 148.464 ms
  (4 rows)
```

分析：

- 执行方式：使用了最左前缀索引（`lineitem_index`）进行扫描。
- 过滤条件：`l_tax = 0.02`。
- 实际执行时间：执行时间为 `148.464 ms`，比无索引查询快，但没有比最左前缀索引的 `l_quantity = 24` 查询更快。

1. 无索引查询

- 执行时间较长，顺序扫描导致高开销。

2. 使用复合索引

- 查询性能显著提升，索引有效降低了扫描行数。

结果分析

** 执行结果是否一样？**

- `l_quantity = 24` 查询的执行结果一致，无论是使用最左前缀索引，还是无索引查询，返回的结果行数都是一致的，都是 18229 行。**
- `l_tax = 0.02` 查询的执行结果也一致，无论是使用最左前缀索引，还是无索引查询，返回的结果行数都是一致的，都是 100598 行。**

对比执行效果和执行速度

- 无索引查询 vs 使用最左前缀索引查询

- `**l_quantity = 24**` 执行时间差异：无索引查询的执行时间为 `276.975 ms`，而使用索引查询的执行时间为 `41.252 ms`，索引查询快得多。
- `**l_tax = 0.02**` 执行时间差异：无索引查询的执行时间为 `291.463 ms`，使用索引查询的执行时间为 `148.464 ms`，同样索引查询较快，但差异没有 `l_quantity = 24` 的查询那么明显。

解释执行时间差异的原因

- 索引的使用：最左前缀索引对于查询条件中第一个列的过滤效果最为显著。对于 `l_quantity = 24` 和 `l_tax = 0.02` 这两个查询，索引帮助减少了需要扫描的行数，尤其在有大量不符合条件的行时，索引的作用尤为明显。
- 查询条件的不同：
 - 对于 `**l_quantity = 24**`，查询的结果集行数相对较少（18229行），使用索引可以显著提高查询性能。
 - 对于 `**l_tax = 0.02**`，虽然查询条件相对更复杂，涉及到浮动的数值，但是由于没有索引列的顺序，扫描的行数相对较多（100598行）。因此，索引的作用虽然存在，但在这种情况下，索引的效率提升比 `**l_quantity = 24**` 时要小一些。
- Bitmap索引的使用：在最左前缀索引查询中，`Bitmap Index Scan` 会首先在索引中查找符合条件的行，并生成一个位图（`Bitmap`），然后通过 `Bitmap Heap Scan` 在数据表中读取相关数据。这种方法减少了对数据表的重复扫描，使得查询速度更快。无索引查询则需要顺序扫描整个表，比较慢。

结论

- 执行结果**在两种查询中是相同的，无论是否使用索引。**
- 执行时间**上，使用最左前缀索引的查询显著优于无索引查询，尤其在涉及大量数据时，索引查询的性能更为突出。**
- 差异原因：索引通过减少扫描的数据量，加速了查询过程。顺序扫描则需要扫描整个表，导致执行时间较长。`**Bitmap Index Scan**` 能显著提升查询效率，尤其是在匹配条件较多的情况下。

3.2 多表连接操作，在连接属性上建立索引

实验步骤

1. 创建 `lineitem` 表的备份表 `lineitem_new`。
2. 在 `lineitem_new` 上删除实验 3.1 创建的组合索引，并在 `l_suppkey` 和 `l_partkey` 上分别创建两个索引：
 - 索引 1： `lineitem_index1`，针对 `l_suppkey`。
 - 索引 2： `lineitem_index2`，针对 `l_partkey`。

3. 编写并执行两组 SQL 语句：

- 使用索引和不使用索引分别查询 `supplier` 表中供应商手机号。
- 使用索引和不使用索引分别查询 `part` 表中零件名称。

4. 对比两组查询的执行计划和查询执行的耗时。

实验代码

索引创建

```
1 DROP INDEX IF EXISTS lineitem_index;
2 CREATE INDEX lineitem_index1 ON lineitem_new(l_suppkey);
3 CREATE INDEX lineitem_index2 ON lineitem_new(l_partkey);
```

无索引查询供应商手机号

```
1 EXPLAIN ANALYZE
2 SELECT DISTINCT l_suppkey, s_phone
3 FROM lineitem, supplier
4 WHERE l_suppkey = 10 AND l_suppkey = supplier.s_suppkey;
```

使用索引查询供应商手机号

```
1 EXPLAIN ANALYZE
2 SELECT DISTINCT l_suppkey, s_phone
3 FROM lineitem_new, supplier
4 WHERE l_suppkey = 10 AND l_suppkey = supplier.s_suppkey;
```

无索引查询零件名称

```
1 EXPLAIN ANALYZE
2 SELECT DISTINCT l_partkey, p_name
3 FROM lineitem, part
4 WHERE l_partkey = 928 AND l_partkey = part.p_partkey;
```

使用索引查询零件名称

```
1 EXPLAIN ANALYZE
2 SELECT DISTINCT l_partkey, p_name
3 FROM lineitem_new, part
4 WHERE l_partkey = 928 AND l_partkey = part.p_partkey;
```

执行计划

无索引查询供应商手机号

```

1                                     QUERY
2 PLAN
3 -----
4
5 HashAggregate (cost=97424.11..97424.12 rows=1 width=20) (actual
6   time=235.376..235.376 rows=1 loops=1)
7     Group By Key: lineitem.l_suppkey, supplier.s_phone
8       -> Nested Loop (cost=0.00..97421.60 rows=503 width=20) (actual
9         time=63.298..235.224 rows=450 loops=1)
10           -> Index Scan using supplier_pkey on supplier
11             (cost=0.00..8.27 rows=1 width=20) (actual time=0.037..0.039 rows=1
12               loops=1)
13                 Index Cond: (s_suppkey = 10)
14               -> Seq Scan on lineitem (cost=0.00..97408.30 rows=503
15                 width=4) (actual time=63.255..235.093 rows=450 loops=1)
16                   Filter: (l_suppkey = 10)
17                   Rows Removed by Filter: 903403
18
19 Total runtime: 235.540 ms
20
21 (9 rows)

```

使用索引查询供应商手机号

```

1
2 QUERY PLAN
3 -----
4
5 HashAggregate (cost=1570.10..1570.11 rows=1 width=20) (actual
6   time=4.265..4.265 rows=1 loops=1)
7   Group By Key: lineitem_new.l_suppkey, supplier.s_phone
8     -> Nested Loop (cost=11.75..1567.91 rows=439 width=20) (actual
9       time=0.400..4.044 rows=450 loops=1)
10         -> Index Scan using supplier_pkey on supplier
11           (cost=0.00..8.27 rows=1 width=20) (actual time=0.013..0.015 rows=1
12             loops=1)
13               Index Cond: (s_suppkey = 10)
14             -> Bitmap Heap Scan on lineitem_new (cost=11.75..1555.25
15               rows=439 width=4) (actual time=0.379..3.880 rows=450 loops=1)
16                 Recheck Cond: (l_suppkey = 10)
17                 -> Bitmap Index Scan on lineitem_index1
18                   (cost=0.00..11.64 rows=439 width=0) (actual time=0.218..0.218
19                     rows=450 loops=1)
20                     Index Cond: (l_suppkey = 10)
21
22 Total runtime: 4.401 ms
23
24 (10 rows)

```

无索引查询零件名称

```

1                                     QUERY PLAN
2 -----
3 HashAggregate  (cost=97416.96..97416.97 rows=1 width=42) (actual
4   time=236.517..236.518 rows=1 loops=1)
5     Group By Key: lineitem.l_partkey, part.p_name
6       -> Nested Loop  (cost=0.00..97416.83 rows=26 width=42) (actual
7         time=116.263..236.486 rows=14 loops=1)
8           -> Index Scan using part_pkey on part  (cost=0.00..8.27
9             rows=1 width=42) (actual time=0.761..0.763 rows=1 loops=1)
10            Index Cond: (p_partkey = 928)
11           -> Seq Scan on lineitem  (cost=0.00..97408.30 rows=26
12             width=4) (actual time=115.489..235.704 rows=14 loops=1)
13             Filter: (l_partkey = 928)
14             Rows Removed by Filter: 903839
15
16 Total runtime: 236.661 ms
17
18 (9 rows)

```

使用索引查询零件名称

```

1                                     QUERY
2 PLAN
3 -----
4 HashAggregate  (cost=103.02..103.03 rows=1 width=42) (actual
5   time=0.413..0.414 rows=1 loops=1)
6   Group By Key: lineitem_new.l_partkey, part.p_name
7     -> Nested Loop  (cost=4.53..102.90 rows=23 width=42) (actual
8       time=0.188..0.370 rows=14 loops=1)
9       -> Index Scan using part_pkey on part  (cost=0.00..8.27
10      rows=1 width=42) (actual time=0.018..0.020 rows=1 loops=1)
11      Index Cond: (p_partkey = 928)
12      -> Bitmap Heap Scan on lineitem_new  (cost=4.53..94.40
13      rows=23 width=4) (actual time=0.160..0.328 rows=14 loops=1)
14      Recheck Cond: (l_partkey = 928)
15      -> Bitmap Index Scan on lineitem_index2
16      (cost=0.00..4.52 rows=23 width=0) (actual time=0.131..0.131
17      rows=14 loops=1)
18      Index Cond: (l_partkey = 928)
19
20 Total runtime: 0.595 ms
21
22 (10 rows)

```

结果分析

无索引与有索引查询供应商手机号

查询目标: 列出在 `lineitem` 表中某个 `l_suppkey` (供应商 ID) 对应的所有供应商的手机号。

无索引查询:

- **查询计划:**
 - 使用 `Nested Loop` 连接 `supplier` 和 `lineitem`。
 - 在 `supplier` 表上使用主键索引进行扫描 (`supplier_pkey`)，并通过 `s_suppkey = 10` 查找供应商。
 - 对于 `lineitem` 表，执行顺序扫描 (`seq Scan`)，然后根据 `l_suppkey = 10` 筛选记录。
 - 运行时间: 235.540 ms。

有索引查询:

- **查询计划:**
 - 使用 `Nested Loop` 连接 `supplier` 和 `lineitem_new`。
 - 在 `supplier` 表上同样使用主键索引进行扫描 (`supplier_pkey`)，通过 `s_suppkey = 10` 查找供应商。
 - 对于 `lineitem_new` 表，使用 `Bitmap Heap Scan`，并通过 `l_suppkey = 10` 条件进行索引扫描 (`Bitmap Index Scan`)，因为已经在 `l_suppkey` 上建立了索引。
 - 运行时间: 4.401 ms。

执行时间差异的原因:

- **无索引查询** 使用了全表扫描 (`seq Scan`)，因此需要检查整个 `lineitem` 表中是否有符合条件的记录。这导致了查询时间较长 (235.540 ms)。
- **有索引查询** 使用了 `Bitmap Index Scan`，这大大减少了扫描的范围，避免了对整个 `lineitem` 表的顺序扫描，从而提高了查询效率，显著缩短了执行时间 (4.401 ms)。
- 主要的性能提升来自索引的使用，它使得查询能够直接通过索引定位到符合条件的行，而无需逐行扫描整个表。

无索引与有索引查询零件名称

查询目标: 列出在 `lineitem` 表中某个 `l_partkey` (零件 ID) 对应的所有零件名称。

无索引查询:

- **查询计划:**
 - 使用 `Nested Loop` 连接 `part` 和 `lineitem`。
 - 在 `part` 表上使用主键索引进行扫描 (`part_pkey`)，通过 `p_partkey = 928` 查找零件。
 - 对于 `lineitem` 表，执行顺序扫描 (`seq Scan`)，然后根据 `l_partkey = 928` 筛选记录。
 - 运行时间: 236.661 ms。

有索引查询：

- **查询计划：**
 - 使用 `Nested Loop` 连接 `part` 和 `lineitem_new`。
 - 在 `part` 表上同样使用主键索引进行扫描 (`part_pkey`)，通过 `p_partkey = 928` 查找零件。
 - 对于 `lineitem_new` 表，使用 `Bitmap Heap Scan`，并通过 `l_partkey = 928` 条件进行索引扫描 (`Bitmap Index Scan`)，因为已经在 `l_partkey` 上建立了索引。
 - 运行时间：0.595 ms。

执行时间差异的原因：

- **无索引查询**同样使用了顺序扫描 (`Seq Scan`)，需要扫描整个 `lineitem` 表，查找与给定 `l_partkey` 匹配的记录。由于没有索引支持，这导致了较高的执行时间 (236.661 ms)。
- **有索引查询**使用了 `Bitmap Index Scan`，通过索引快速定位到符合条件的行，并使用 `Bitmap Heap Scan` 提取实际数据，显著提升了查询效率。由于索引加速了数据访问，查询时间减少到了 0.595 ms。

实验结论：

- **无索引查询**由于必须执行全表扫描 (`Seq Scan`)，查询时间较长。
- **有索引查询**使用了索引扫描 (`Bitmap Index Scan`) 来加速数据访问，减少了需要扫描的行数，从而显著缩短了执行时间。
- 在这两个查询中，执行时间的差异主要是由是否存在合适的索引来决定的，索引能有效减少数据扫描的范围，提高查询效率。

结论

在多表连接查询中，针对连接属性建立索引能够有效减少查询的时间开销，显著提升性能。

3.3 索引对小表查询的作用

实验步骤

1. 创建 `supplier` 表的不带主键索引的备份表 `supplier_new`。
2. 在 `supplier` 表的 `s_suppkey` 上创建索引。
3. 执行查询：
 - 不强制使用索引：查询是否会自动使用索引。
 - 强制使用索引：通过设置 `enable_seqscan=false` 禁用顺序扫描，强制使用索引进行查询。
4. 比较两种查询的执行计划和执行时间。

实验代码

创建备份表

```
1 CREATE TABLE supplier_new AS TABLE supplier;
```

不强制使用索引查询

```
1 EXPLAIN  
2 SELECT s_suppkey  
3 FROM supplier;
```

强制使用索引查询

```
1 SET enable_seqscan = OFF;  
2 EXPLAIN  
3 SELECT s_suppkey  
4 FROM supplier;
```

执行计划

不强制使用索引查询

```
1                                     QUERY PLAN  
2 -----  
3   Seq Scan on supplier  (cost=0.00..62.00 rows=2000 width=4)  
4   (1 row)
```

强制使用索引查询

```
1                                     QUERY PLAN  
2 -----  
3   Bitmap Heap Scan on supplier  (cost=42.75..104.75 rows=2000  
4   width=4)  
5     ->  Bitmap Index Scan on supplier_pkey  (cost=0.00..42.25  
6       rows=2000 width=0)  
7     (2 rows)
```

结果分析

不强制使用索引时：

- 查询优化器决定不使用索引，而是执行全表扫描（`seq scan`）。
- 这种情况通常发生在查询的结果集较大，或者索引无法有效过滤掉大量数据时。在这种情况下，执行全表扫描可能比使用索引更加高效。

强制使用索引时：

- 强制索引扫描后，查询计划显示使用了 `Bitmap Index Scan` 和 `Bitmap Heap Scan`。
- 通过强制索引扫描，你迫使查询优化器使用索引，虽然在某些情况下这可能不是最优选择，因为索引扫描会涉及额外的成本（例如，在执行 `Bitmap Heap Scan` 时需要访问实际数据）。

对比和分析：

- 不强制使用索引时，由于没有强制要求使用索引，优化器选择了全表扫描（`seq scan`），这通常发生在查询范围较大或索引不够有效时。
- 强制使用索引时，索引扫描虽然能提高定位的精确度，但在某些情况下，它的成本反而较高，因为查询需要额外的步骤（`Bitmap Heap Scan`）来访问数据。

结论

- 索引不一定总是优于全表扫描。查询优化器通常会根据查询的实际情况（如数据量、过滤条件等）选择最优的执行路径。
- 强制使用索引虽然能确保索引被使用，但在某些情况下，这可能反而导致较高的执行成本。因此，在实际使用中，应根据查询的特点来决定是否强制使用索引

3.4 查询条件中函数对索引的影响

实验步骤

1. 创建 `lineitem_new` 表，并在 `l_discount` 和 `l_quantity` 上分别创建索引。
2. 编写两条等价查询语句：
 - 一条在查询条件中使用函数。
 - 一条通过改写条件避免使用函数。
3. 执行两条语句，比较其执行计划和执行时间。

实验代码

创建索引

```
1 CREATE INDEX lineitem_index1 ON lineitem_new(l_discount);
2 CREATE INDEX lineitem_index2 ON lineitem_new(l_quantity);
```

使用函数的查询

```
1 EXPLAIN
2 SELECT DISTINCT B.l_orderkey
3 FROM lineitem_new AS A, lineitem_new AS B
4 WHERE A.l_extendedprice = 16473.51
5   AND A.l_discount = 0.04
6   AND ABS(A.l_quantity - B.l_quantity) < 20;
```

不使用函数的查询

```
1 EXPLAIN
2 SELECT DISTINCT B.l_orderkey
3 FROM lineitem_new AS A, lineitem_new AS B
4 WHERE A.l_extendedprice = 16473.51
5   AND A.l_discount = 0.04
6   AND B.l_quantity > A.l_quantity - 20
7   AND B.l_quantity < A.l_quantity + 20;
```

执行计划

使用函数的查询

```
1                                     QUERY PLAN
2 -----
3 HashAggregate  (cost=1000002818715.57..1000002820813.88
4   rows=209831 width=4)
5     Group By Key: b.l_orderkey
6       -> Nested Loop  (cost=10000001505.54..1000002817962.36
7         rows=301284 width=4)
8           Join Filter: (abs((a.l_quantity - b.l_quantity)) <
9             20::numeric)
10            -> Bitmap Heap Scan on lineitem_new a
11              (cost=1505.54..21491.93 rows=1 width=5)
12                Recheck Cond: (l_discount = .04)
13                Filter: (l_extendedprice = 16473.51)
14                  -> Bitmap Index Scan on lineitem_index1
15                    (cost=0.00..1505.54 rows=81226 width=0)
16                      Index Cond: (l_discount = .04)
17                        -> Seq Scan on lineitem_new b
18                          (cost=10000000000.00..1000002780653.00 rows=903853 width=9)
19                            (10 rows)
```

不使用函数的查询

```
1                                     QUERY
2 PLAN
3 -----
4 HashAggregate  (cost=45661.59..46665.87 rows=100428 width=4)
5   Group By Key: b.l_orderkey
6     -> Nested Loop  (cost=3643.29..45410.52 rows=100428 width=4)
7       -> Bitmap Heap Scan on lineitem_new a
8         (cost=1505.54..21491.93 rows=1 width=5)
9           Recheck Cond: (l_discount = .04)
10          Filter: (l_extendedprice = 16473.51)
```

```

9          -> Bitmap Index Scan on lineitem_index1
10         (cost=0.00..1505.54 rows=81226 width=0)
11             Index Cond: (l_discount = .04)
12             -> Bitmap Heap Scan on lineitem_new b
13             (cost=2137.74..22914.30 rows=100428 width=9)
14                 Recheck Cond: ((l_quantity > (a.l_quantity -
15                   20::numeric)) AND (l_quantity < (a.l_quantity + 20::numeric)))
16                     -> Bitmap Index Scan on lineitem_index2
17                     (cost=0.00..2112.63 rows=100428 width=0)
18                         Index Cond: ((l_quantity > (a.l_quantity -
19                           20::numeric)) AND (l_quantity < (a.l_quantity + 20::numeric)))
20     (12 rows)

```

结果分析

使用函数时的执行计划

- 执行方式：

- 在 WHERE 子句中使用了 `abs(a.l_quantity - b.l_quantity) < 20` 函数。由于该函数的存在，查询优化器无法利用 `l_quantity` 上的索引来进行高效的查找。
- 在执行计划中，查询优化器选择了 `Seq Scan`，即全表扫描，针对 `lineitem_new` 表的 `b` 表进行扫描，而没有使用索引。这导致了查询性能的显著降低。
- 虽然在 `a` 表上能有效使用索引，但由于 `b` 表的查询条件包含了函数，导致不能有效利用索引，进而导致了全表扫描（`Seq Scan`）的使用。

不使用函数时的执行计划

- 执行方式：

- 使用了 `BETWEEN a.l_quantity - 20 AND a.l_quantity + 20` 条件，而没有使用函数。这时，查询优化器能够利用 `l_quantity` 列上的索引 (`lineitem_index2`)，因为索引能够直接处理范围查询。
- 在执行计划中，查询优化器选择了 `Bitmap Index Scan`，并使用了 `Bitmap Heap Scan` 来根据索引定位数据，显著提升了查询性能。

性能差异的原因

- **使用函数的查询：**由于 `abs()` 函数的存在，查询优化器无法利用索引对 `l_quantity` 列进行高效的扫描，因此选择了全表扫描（`seq scan`）。
- **不使用函数的查询：**查询条件可以直接通过索引扫描处理（`Bitmap Index Scan`），因此查询性能较好。

结论

- 使用函数（如 `abs()`）会导致查询条件失效，优化器无法使用索引，最终可能选择全表扫描，从而导致性能下降。

- 通过优化查询语句，避免在 WHERE 子句中使用函数，可以提高查询的性能，并使索引得以充分利用。
-

3.5 多表嵌入式 SQL 查询

实验步骤

- 编写两条等价查询语句：
 - 一条使用嵌套查询。
 - 一条使用连接查询。
- 执行两条语句，比较其执行计划和执行时间。
- 对实验结果进行分析，确定两种方法在性能上的差异。

实验代码

嵌套查询

```
1 EXPLAIN ANALYZE
2 SELECT A.p_name
3 FROM part AS A
4 WHERE A.p_partkey IN (
5   SELECT B.ps_partkey
6   FROM partsupp AS B
7   WHERE B.ps_suppkey = 1002
8 );
```

连接查询

```
1 EXPLAIN ANALYZE
2 SELECT A.p_name
3 FROM part AS A, partsupp AS B
4 WHERE B.ps_suppkey = 1002
5 AND A.p_partkey = B.ps_partkey;
```

执行计划

嵌套查询

```

1
2 QUERY PLAN
3 -----
4
5
6
7
8
9

```

Nested Loop Semi Join (cost=0.00..3273.74 rows=78 width=38)
(actual time=0.042..41.381 rows=80 loops=1)
-> Index Scan using part_pkey on part a (cost=0.00..1903.25
rows=40000 width=42) (actual time=0.012..5.820 rows=40000 loops=1)
-> Index Only Scan using partsupp_pkey on partsupp b
(cost=0.00..0.68 rows=21 width=4) (actual time=30.008..30.008
rows=80 loops=40000)
Index Cond: ((ps_partkey = a.p_partkey) AND (ps_suppkey =
1002))
Heap Fetches: 80
Total runtime: 41.493 ms
(6 rows)

连接查询

```

1
2 QUERY PLAN
3 -----
4
5
6
7
8
9
10

```

Nested Loop (cost=0.00..3666.56 rows=78 width=38) (actual
time=0.050..5.205 rows=80 loops=1)
-> Index Only Scan using partsupp_pkey on partsupp b
(cost=0.00..3112.37 rows=78 width=4) (actual time=0.022..4.722
rows=80 loops=1)
Index Cond: (ps_suppkey = 1002)
Heap Fetches: 80
-> Index Scan using part_pkey on part a (cost=0.00..7.09
rows=1 width=42) (actual time=0.373..0.401 rows=80 loops=80)
Index Cond: (p_partkey = b.ps_partkey)
Total runtime: 5.314 ms
(7 rows)

结果分析

查询执行计划对比

嵌套查询执行计划

- 执行计划分析:
- 在嵌套查询中，查询优化器执行了一个 **Nested Loop Semi Join**，即首先通过 `part` 表扫描 (`Index Scan`)，然后通过 `partsupp` 表对每个零件查询是否存在供应商 `ps_suppkey = 1002`。

- `partsupp` 表的扫描是通过 **Index Only Scan** 来实现的，它直接利用了 `ps_partkey` 和 `ps_suppkey` 列的索引。每个 `part` 表中的条目都需要对 `partsupp` 表执行一次扫描，导致执行计划中有多次 `Heap Fetches`。
- **性能瓶颈：**
- 执行时间为 **41.493 ms**，嵌套查询使用了 **Index Only Scan** 和 **Heap Fetches**，导致对于每个 `part` 表的记录都需要查找对应的 `partsupp` 表记录，可能会导致较高的 I/O 开销。

连接查询执行计划

- **执行计划分析：**
- 在连接查询中，查询优化器首先通过 `partsupp` 表上的索引（**Index Only Scan**）扫描所有供应商 `ps_suppkey = 1002` 的记录。
- 然后，它通过 **Index Scan** 在 `part` 表中查找相应的零件 `p_partkey`。
- 查询优化器使用了有效的索引扫描，没有出现额外的 `Heap Fetches`，因此查询速度较快。
- **性能瓶颈：**
- 执行时间为 **5.314 ms**，连接查询直接利用了索引，减少了额外的 I/O 开销，因此查询时间比嵌套查询要低得多。

性能差异的原因

嵌套查询 (Subquery) :

- **效率较低：**嵌套查询通常会对外层查询的每一行执行一次内层查询，因此在涉及到多个记录时，查询可能会变得非常低效。
- **多次索引查找：**在执行嵌套查询时，`part` 表的每一条记录都需要访问 `partsupp` 表来查找是否存在符合条件的记录，这可能导致较高的 I/O 操作（即每次对 `partsupp` 表执行查找）。

连接查询 (Join) :

- **更高效的索引使用：**连接查询通过索引扫描在 `partsupp` 和 `part` 表之间建立直接的连接，避免了对每一条记录都执行子查询。直接连接使得查询能够更有效地利用索引，减少了不必要的 I/O 操作。
- **索引优化：**在连接查询中，使用了对 `ps_suppkey` 和 `p_partkey` 的索引，这样能够快速找到匹配的记录。

结论

- 尽量避免使用嵌套查询，优先选择连接查询以提升查询性能。

3.6 where 查询条件中复合查询条件 OR 对索引的影响

实验步骤

1. 在 `lineitem_new` 表的 `l_quantity` 上创建索引。
2. 编写查询语句：
 - 一条使用 `OR` 条件。
 - 另一条将 `OR` 条件改写为两条查询通过 `UNION` 连接。
3. 比较两种查询方式的执行计划和查询性能。

实验代码

创建索引

```
1 | CREATE INDEX lineitem_index ON lineitem_new(l_quantity);
```

含有 `OR` 的查询

```
1 | EXPLAIN
2 | SELECT *
3 | FROM lineitem_new
4 | WHERE l_tax = 0.06 AND (l_quantity = 36 OR l_discount = 0.09);
```

改写为 `UNION` 的查询

```
1 | EXPLAIN
2 | (SELECT *
3 |   FROM lineitem_new
4 |   WHERE l_tax = 0.06 AND l_quantity = 36)
5 | UNION
6 | (SELECT *
7 |   FROM lineitem_new
8 |   WHERE l_tax = 0.06 AND l_discount = 0.09);
```

执行计划

含有 `OR` 的查询

QUERY PLAN	
1	
2	-----
3	Seq Scan on lineitem_new (cost=10000000000.00..1000003458542.75 rows=10825 width=129)
4	Filter: ((l_tax = .06) AND ((l_quantity = 36::numeric) OR (l_discount = .09)))
5	(2 rows)

改写为 `UNION` 的查询

```

1
2
3
4 -----
5 -----
6 -----
7 -----
8 HashAggregate  (cost=1000003253050.43..1000003253160.49 rows=11006
width=129)
9   Group By Key: public.lineitem_new.l_orderkey,
10  public.lineitem_new.l_partkey, public.lineitem_new.l_suppkey,
11  public.lineitem_new.l_linenumber,
12  public.lineitem_new.l_quantity,
13  public.lineitem_new.l_extendedprice,
14  public.lineitem_new.l_discount, public.lineitem_new.l_tax,
15  public.lineitem
16  _new.l_returnflag, public.lineitem_new.l_linestatus,
17  public.lineitem_new.l_shipdate, public.lineitem_new.l_commitdate,
18  public.lineitem_new.l_rec
19  eiptdate, public.lineitem_new.l_shipinstruct,
20  public.lineitem_new.l_shipmode, public.lineitem_new.l_comment
21  -> Append (cost=341.10..1000003252610.19 rows=11006 width=129)
22      -> Bitmap Heap Scan on lineitem_new
23      (cost=341.10..19920.63 rows=2010 width=129)
24          Recheck Cond: (l_quantity = 36::numeric)
25          Filter: (l_tax = .06)
26          -> Bitmap Index Scan on lineitem_index
27      (cost=0.00..340.60 rows=18167 width=0)
28          Index Cond: (l_quantity = 36::numeric)
29          -> Seq Scan on lineitem_new
30      (cost=10000000000.00..1000003232579.50 rows=8996 width=129)
31          Filter: ((l_tax = .06) AND (l_discount = .09))
32      (10 rows)

```

结果分析

含有 OR 的查询

- 执行计划分析:

- **Seq Scan**: 查询计划显示，尽管有条件 `l_quantity = 36` 和 `l_discount = 0.09`，但由于包含了 `OR`，查询执行计划选择了顺序扫描 (`seq scan`)，而没有利用任何索引。
- 原因：由于 `OR` 条件可能导致查询无法利用现有的单列索引。如果某个条件（如 `l_quantity = 36`）可以使用索引，而另一个条件（如 `l_discount = 0.09`）不能，则数据库通常会选择顺序扫描 (`seq scan`)，因为 `OR` 会使得查询的两部分条件被独立地处理，从而无法利用一个单一的索引。

使用 UNION 的查询

- 执行计划分析：
 - 第一个子查询利用了 `Bitmap Heap Scan` 和 `Bitmap Index Scan`，能够有效地利用索引 (`l_quantity = 36`)。
 - 第二个子查询则是对 `l_discount = 0.09` 进行了顺序扫描 (`seq scan`)，因为 `l_discount` 可能没有索引或者索引的使用不如 `l_quantity` 高效。

性能差异：

- 原始查询（含 `OR` 条件）：
- 由于 `OR` 条件可能导致查询无法有效使用索引，查询会执行顺序扫描 (`seq scan`)，这会导致性能较差，尤其是在数据量较大时。
- 改写为 `UNION` 的查询：
- 将查询拆成两个独立的查询，可以分别利用不同的条件 (`l_quantity` 和 `l_discount`) 来使用索引，从而提高查询效率。尤其是对于 `l_quantity` 使用了索引，查询的执行会显著加速。
- 通过 `UNION` 合并结果，可以避免对 `OR` 条件的处理，使得每个查询都能利用其特定的索引。

结论

- 含 `OR` 的查询：由于包含 `OR` 条件，查询计划可能会选择顺序扫描 (`seq scan`)，不能有效利用索引，从而导致性能较差。
- 使用 `UNION` 的查询：将查询拆分为两个子查询并通过 `UNION` 合并，每个查询都能分别利用适当的索引，从而提高了查询效率。

3.7 聚集运算中的索引设计

实验步骤

1. 在 `lineitem_new` 表上创建适合分组和聚集操作的索引：
 - 针对 `GROUP BY` 的分组属性建立索引。
 - 针对聚集运算的属性建立索引。
2. 分别在无索引表和有索引表上执行相同的查询。

3. 强制使用索引后再次执行查询，观察性能变化。

实验代码

创建索引

```
1 CREATE INDEX lineitem_index1 ON lineitem_new(l_extendedprice);
2 CREATE INDEX lineitem_index2 ON lineitem_new(l_suppkey);
```

无索引查询

```
1 EXPLAIN
2 SELECT l_suppkey, AVG(l_extendedprice) AS avg_price
3 FROM lineitem
4 GROUP BY l_suppkey;
```

有索引查询

```
1 EXPLAIN
2 SELECT l_suppkey, AVG(l_extendedprice) AS avg_price
3 FROM lineitem_new
4 GROUP BY l_suppkey;
```

强制使用索引查询

```
1 SET enable_seqscan = OFF;
2 EXPLAIN
3 SELECT l_suppkey, AVG(l_extendedprice) AS avg_price
4 FROM lineitem_new
5 GROUP BY l_suppkey;
```

执行计划

无索引查询

```
1                                     QUERY PLAN
2 -----
3 HashAggregate  (cost=100005.36..100030.36 rows=2000 width=44)
4   Group By Key: l_suppkey
5     -> Seq Scan on lineitem  (cost=0.00..94811.24 rows=1038824
6       width=12)
7   (3 rows)
```

有索引查询

```
1                                     QUERY PLAN
2 -----
3 HashAggregate  (cost=32325.79..32350.79  rows=2000  width=44)
4   Group By Key: l_suppkey
5     -> Seq Scan on lineitem_new  (cost=0.00..27806.53  rows=903853
6       width=12)
(3 rows)
```

强制使用索引查询

```
1                                     QUERY PLAN
2 -----
3 GroupAggregate  (cost=0.00..892878.29  rows=2000  width=44)
4   Group By Key: l_suppkey
5     -> Index Scan using lineitem_index2 on lineitem_new
6       (cost=0.00..888334.02  rows=903853 width=12)
(3 rows)
```

结果分析

无索引查询

- 执行计划分析：
 - **Seq Scan**: 查询计划显示，数据库执行顺序扫描（`seq scan`）来处理`lineitem`表，因为没有合适的索引来优化分组操作。扫描表中的所有数据，计算每个供应商的平均价格。
 - **HashAggregate**: 由于没有索引，数据库使用了哈希聚合（`HashAggregate`）来按`l_suppkey`进行分组并计算平均值。
 - 执行时间：由于没有索引，扫描整个表需要较长的时间，尤其是表数据量较大时。

有索引查询

- 执行计划分析：
 - **Seq Scan**: 即使我们创建了索引，查询仍然使用了顺序扫描（`seq scan`）。虽然`l_suppkey`有索引，但是在这种情况下数据库可能认为顺序扫描更高效，因为它可能需要读取整个表的数据来进行聚合操作，特别是当表中的数据量较大时，索引可能没有明显的优势。
 - **HashAggregate**: 数据库仍然使用了哈希聚合（`HashAggregate`）来按`l_suppkey`分组并计算平均值。即使有索引，数据库并没有选择使用索引扫描（`Index Scan`）来代替顺序扫描。

强制使用索引查询

- 执行计划分析：

- **Index Scan**: 由于强制启用索引扫描，数据库选择了使用 `lineitem_index2` 索引来扫描 `lineitem_new` 表。通过索引扫描，可以有效地按 `l_suppkey` 进行分组，避免了顺序扫描（`Seq Scan`）带来的性能损失。
- **GroupAggregate**: 与无索引查询类似，数据库使用哈希聚合来按 `l_suppkey` 进行分组并计算平均值，但索引扫描提供了更高效的数据访问方式。
- 执行时间：强制使用索引通常会提升查询性能，尤其是当表中有大量数据时。

对比执行计划：

查询类型	执行计划	说明
无索引查询	<code>Seq Scan</code> + <code>HashAggregate</code>	执行顺序扫描，按 <code>l_suppkey</code> 进行哈希聚合。数据量大时性能差。
有索引查询	<code>Seq Scan</code> + <code>HashAggregate</code>	即使有索引，数据库选择了顺序扫描，可能因为需要对整个表进行聚合。
强制使用索引查询	<code>Index Scan</code> + <code>GroupAggregate</code>	强制使用索引扫描，避免顺序扫描，能够更高效地利用索引。

- 执行速度对比：

- **无索引查询**：由于没有索引，查询性能较差，尤其是当表数据量较大时，顺序扫描会导致性能瓶颈。
- **有索引查询**：虽然创建了索引，数据库仍然选择顺序扫描，导致性能没有显著提升。
- **强制使用索引查询**：通过强制使用索引扫描，查询性能得到明显提升。索引帮助避免了顺序扫描，从而加速了查询。

结论

- **无索引查询**：由于没有索引，数据库需要扫描整个表来执行聚合操作，性能较差，特别是在数据量大时。
- **有索引查询**：即使创建了索引，如果查询条件没有完全匹配索引或数据库认为顺序扫描更高效，查询可能仍然会执行顺序扫描，导致性能没有提升。
- **强制使用索引查询**：强制启用索引扫描通常会提高查询效率，尤其是在大表聚合时，通过索引扫描能够避免顺序扫描带来的性能损失。

3.8 Select 子句中有无 distinct 的区别

实验步骤

1. 编写两条查询语句：

- 一条不使用 `DISTINCT`。
- 一条使用 `DISTINCT`。

2. 比较两种方式的执行计划和查询效率。

实验代码

无 `DISTINCT` 查询

```
1 EXPLAIN ANALYZE
2 SELECT o_orderkey
3 FROM orders
4 WHERE o_orderstatus = 'O'
5 AND o_orderpriority = '4-NOT SPECIFIED';
```

有 `DISTINCT` 查询

```
1 EXPLAIN ANALYZE
2 SELECT DISTINCT o_orderkey
3 FROM orders
4 WHERE o_orderstatus = 'O'
5 AND o_orderpriority = '4-NOT SPECIFIED';
```

执行计划

无 `DISTINCT` 查询

	QUERY PLAN
1	
2	-----
3	Seq Scan on orders (cost=1000000000.00..100000878600.00 rows=29224 width=4) (actual time=0.072..46.682 rows=29224 loops=1) Filter: ((o_orderstatus = 'O'::bpchar) AND (o_orderpriority = '4- NOT SPECIFIED'::bpchar)) Rows Removed by Filter: 270777 Total runtime: 48.135 ms (4 rows)

有 `DISTINCT` 查询

	QUERY
1	PLAN
2	-----
3	-----
4	-----
5	-----
6	-----
7	-----
8	-----
	Unique (cost=0.00..13670.34 rows=29224 width=4) (actual time=47.678..112.071 rows=29224 loops=1)
	-> Index Scan using orders_pkey on orders (cost=0.00..13597.28 rows=29224 width=4) (actual time=47.673..108.759 rows=29224 loops=1)
	Filter: ((o_orderstatus = 'O'::bpchar) AND (o_orderpriority = '4-NOT SPECIFIED'::bpchar))
	Rows Removed by Filter: 270777
	Total runtime: 113.043 ms
	(5 rows)

结果分析

无 DISTINCT 查询

- 执行计划分析：
 - **Seq Scan**: 查询执行计划显示，数据库对 `orders` 表进行了顺序扫描 (`seq scan`)。这是因为没有索引可以有效地用于 `o_orderstatus` 和 `o_orderpriority` 的过滤条件，或者数据库评估认为顺序扫描在此情况下更为高效。
 - **Filter**: 使用了过滤条件 `o_orderstatus = 'O'` 和 `o_orderpriority = '4-NOT SPECIFIED'` 来筛选符合条件的记录。
 - **Rows Removed by Filter**: 从原始数据集中，过滤掉了 270,777 条不符合条件的记录。
- 执行时间：查询扫描整个表并筛选符合条件的记录，执行时间为 48.135 毫秒。

有 DISTINCT 查询

- 执行计划分析：
 - **Unique**: 查询计划中使用了 `Unique` 操作符，这是因为 `DISTINCT` 需要确保结果中没有重复的记录，因此会进行额外的去重操作。
 - **Index Scan**: 数据库选择了使用 `orders_pkey` 索引进行扫描。与无 `DISTINCT` 查询相比，查询计划使用了索引扫描 (`Index Scan`)，这通常比顺序扫描更高效，尤其是对于带有条件过滤的查询。
 - **Filter**: 使用了和之前相同的过滤条件 `o_orderstatus = 'O'` 和 `o_orderpriority = '4-NOT SPECIFIED'`。
 - **Rows Removed by Filter**: 过滤掉了 270,777 条不符合条件的记录。
- 执行时间：尽管使用了索引扫描，但 `DISTINCT` 操作导致查询的执行时间较长，达到了 113.043 毫秒。去重操作消耗了更多时间。

对比执行计划

查询类型	执行计划	说明
无 DISTINCT 查询	Seq Scan + Filter	使用顺序扫描处理表，过滤符合条件的记录。没有去重操作。
有 DISTINCT 查询	Index Scan + Unique + Filter	使用索引扫描，额外进行去重操作（Unique）。执行时间更长。

• 执行时间对比

- 无 DISTINCT 查询：使用顺序扫描，执行时间为 48.135 毫秒。没有去重操作，所以相对较快。
- 有 DISTINCT 查询：使用索引扫描，但增加了去重操作（DISTINCT），执行时间为 113.043 毫秒。去重操作导致查询时间大大增加，尽管索引扫描本身通常较快，但 DISTINCT 操作增加了额外的开销。

结论

- 无 DISTINCT 查询：由于没有去重操作，查询执行时间较短，适合在无需去重的场合使用。
- 有 DISTINCT 查询：使用 DISTINCT 会导致查询执行时间显著增加，尤其是当去重操作需要额外的排序或去重步骤时。在某些情况下，尽管有索引扫描，但去重操作的额外开销使得整体查询效率较低。

3.9 union、union all 的区别

实验步骤

1. 编写两条查询语句：
 - 一条使用 UNION。
 - 一条使用 UNION ALL。
2. 比较两种方式的查询结果和执行性能。

实验代码

使用 UNION 查询

```
1 EXPLAIN ANALYZE
2 SELECT p_partkey FROM part
3 UNION
4 SELECT ps_partkey FROM partsupp;
```

使用 UNION ALL 查询

```
1 | EXPLAIN ANALYZE
2 | SELECT p_partkey FROM part
3 | UNION ALL
4 | SELECT ps_partkey FROM partsupp;
```

执行计划

使用 UNION 查询

```
1 |
2 | QUERY PLAN
3 |
4 | -----
5 | -----
6 | -----
7 | HashAggregate  (cost=12112.52..14112.52 rows=200000 width=4)
8 |   (actual time=191.195..194.706 rows=40000 loops=1)
9 |     Group By Key: part.p_partkey
10 |       -> Append  (cost=0.00..11612.52 rows=200000 width=4) (actual
11 |           time=0.150..170.226 rows=200004 loops=1)
12 |             -> Index Only Scan using part_pkey on part
13 |               (cost=0.00..1903.25 rows=40000 width=4) (actual time=0.148..8.650
14 |               rows=40000 loops=1)
15 |                 Heap Fetches: 40000
16 |               -> Bitmap Heap Scan on partsupp  (cost=2612.27..7709.27
17 |                   rows=160000 width=4) (actual time=4.239..152.128 rows=160004
18 |                   loops=1)
19 |                     -> Bitmap Index Scan on partsupp_pkey
20 |                       (cost=0.00..2572.27 rows=160000 width=0) (actual time=3.947..3.947
21 |                       rows=160004 loops=1)
22 |             Total runtime: 196.416 ms
23 |             (8 rows)
```

使用 UNION ALL 查询

```

1
2   QUERY PLAN
3
4
5
6
7
8
9
10

```

```

3   Result  (cost=0.00..9612.52 rows=200000 width=4) (actual
4     time=0.041..45.067 rows=200004 loops=1)
5       -> Append  (cost=0.00..9612.52 rows=200000 width=4) (actual
6         time=0.039..34.400 rows=200004 loops=1)
7           -> Index Only Scan using part_pkey on part
8             (cost=0.00..1903.25 rows=40000 width=4) (actual time=0.038..5.956
9               rows=40000 loops=1)
10              Heap Fetches: 40000
11            -> Bitmap Heap Scan on partsupp  (cost=2612.27..7709.27
12              rows=160000 width=4) (actual time=3.232..19.204 rows=160004
13                loops=1)
14                  -> Bitmap Index Scan on partsupp_pkey
15                    (cost=0.00..2572.27 rows=160000 width=0) (actual time=2.945..2.945
16                      rows=160004 loops=1)
17
18      Total runtime: 50.268 ms
19
20  (7 rows)

```

结果分析

使用 UNION 查询

- 执行计划分析：
 - **HashAggregate**: `UNION` 会去除重复的记录，因此执行计划中使用了 `HashAggregate` 来进行去重操作。
 - **Append**: 执行计划中使用了 `Append`，将两个子查询的结果合并。
 - **Index Only Scan**: `part` 表使用了索引扫描。
 - **Bitmap Heap Scan**: `partsupp` 表使用了位图堆扫描，首先通过索引扫描找到符合条件的记录。
 - **Total runtime**: 执行时间为 196.416 毫秒。
- 执行计划和效率分析：
 - `UNION` 操作会去除重复的记录，因此需要额外的去重步骤 (`HashAggregate`)。去重操作增加了额外的开销，因此查询执行时间较长。

使用 UNION ALL 查询

- 执行计划分析：
 - **Result**: `UNION ALL` 没有进行去重，因此只使用了 `Result` 操作来合并两个子查询的结果。
 - **Append**: 同样使用了 `Append` 操作来合并两个子查询的结果。

- **Index Only Scan**: `part` 表使用了索引扫描。
- **Bitmap Heap Scan**: `partsupp` 表使用了位图堆扫描，首先通过索引扫描找到符合条件的记录。
- **Total runtime**: 执行时间为 50.268 毫秒。
- 执行计划和效率分析:
 - `UNION ALL` 不进行去重操作，因此没有 `HashAggregate` 步骤，执行效率较高。只需要合并两个查询的结果，减少了去重带来的开销，因此执行时间显著低于 `UNION`。

执行计划对比

查询类型	执行计划	执行时间	说明
UNION 查询	<code>HashAggregate</code> + <code>Append</code> + <code>Index Only Scan</code>	196.416 ms	需要去重操作，增加了额外的开销，执行时间较长。
UNION ALL 查询	<code>Result</code> + <code>Append</code> + <code>Index Only Scan</code>	50.268 ms	不进行去重操作，执行时间较短。

- 执行时间对比:
 - **UNION 查询**: 执行时间较长，主要因为它需要进行去重 (`HashAggregate`)。虽然在两个表的查询结果中可能有重复的 `p_partkey`，但去重操作的增加了额外的计算开销。
 - **UNION ALL 查询**: 执行时间较短，因为它不进行去重操作，直接将两个查询的结果合并，减少了计算量。

结论

- **UNION**: 适用于需要去重的场景，但由于去重操作带来的额外计算开销，会导致执行时间更长。
- **UNION ALL**: 适用于无需去重的场景，因为它跳过了去重步骤，通常执行速度更快。

3.10 from 中存在多余的关系表，即查询非最简化

实验步骤

1. 编写两条等价查询语句：

- 一条在 `FROM` 子句中加入多余的关系表，并在 `WHERE` 子句中增加对应的连接条件。
- 一条移除多余的关系表。

2. 执行查询并比较两种方式的执行计划和性能。

实验代码

无多余关系表

```

1 | EXPLAIN ANALYZE
2 | SELECT DISTINCT orders.o_custkey
3 | FROM lineitem, orders
4 | WHERE lineitem.l_orderkey = orders.o_orderkey;

```

有多余关系表

```

1 | EXPLAIN ANALYZE
2 | SELECT DISTINCT orders.o_custkey
3 | FROM lineitem, orders, part
4 | WHERE lineitem.l_orderkey = orders.o_orderkey
5 |   AND lineitem.l_partkey = part.p_partkey;

```

执行计划

无多余关系表

```

1 |
2 |      QUERY PLAN
3 |
4 |
5 |
6 |
7 |
8 |
9 |
10 |
11 |

```

```

1 |      QUERY PLAN
2 | -----
3 | -----
4 | HashAggregate  (cost=96807.84..96991.50 rows=18366 width=4)
5 | (actual time=358.020..359.230 rows=19998 loops=1)
6 |     Group By Key: orders.o_custkey
7 |     ->  Merge Join  (cost=1.16..94210.78 rows=1038824 width=4)
8 |       (actual time=0.050..271.497 rows=903853 loops=1)
9 |         Merge Cond: (orders.o_orderkey = lineitem.l_orderkey)
10 |           ->  Index Scan using orders_pkey on orders
11 |             (cost=0.00..12097.28 rows=300000 width=8) (actual

```

```

1 |             time=0.010..53.182 rows=300001 loops=1)
2 |               ->  Index Only Scan using lineitem_pkey on lineitem
3 |                 (cost=0.00..68379.14 rows=1038824 width=4) (actual
4 |                   time=0.034..114.542 rows=903853 loops=1)
5 |                     Heap Fetches: 0
6 |
7 | Total runtime: 360.127 ms
8 |
9 | (8 rows)
10 |
11 |

```

有多余关系表

1	QUERY
PLAN	
2	<hr/>
3	HashAggregate (cost=138763.96..138947.62 rows=18366 width=4) (actual time=684.124..685.281 rows=19998 loops=1)
4	Group By Key: orders.o_custkey
5	-> Hash Join (cost=12788.00..136166.90 rows=1038824 width=4) (actual time=129.458..595.131 rows=903853 loops=1)
6	Hash Cond: (lineitem.l_partkey = part.p_partkey)
7	-> Hash Join (cost=11036.00..120131.07 rows=1038824 width=8) (actual time=117.737..455.655 rows=903853 loops=1)
8	Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
9	-> Seq Scan on lineitem (cost=0.00..94811.24 rows=1038824 width=8) (actual time=53.132..234.957 rows=903853 loops=1)
10	-> Hash (cost=7286.00..7286.00 rows=300000 width=8) (actual time=63.706..63.706 rows=300001 loops=1)
11	Buckets: 524288 Batches: 1 Memory Usage: 11719kB
12	-> Seq Scan on orders (cost=0.00..7286.00 rows=300000 width=8) (actual time=0.003..33.202 rows=300001 loops=1)
13	-> Hash (cost=1252.00..1252.00 rows=40000 width=4) (actual time=11.320..11.320 rows=40000 loops=1)
14	Buckets: 65536 Batches: 1 Memory Usage: 1407kB
15	-> Seq Scan on part (cost=0.00..1252.00 rows=40000 width=4) (actual time=0.012..6.847 rows=40000 loops=1)
16	Total runtime: 686.363 ms
17	(14 rows)

结果分析

无多余关系表

- 执行计划分析：
 - **Merge Join**: 使用了 `Merge Join` 对 `orders` 和 `lineitem` 表进行连接。
 - **Index Scan 和 Index Only Scan**: 对 `orders` 表和 `lineitem` 表都使用了索引扫描。
 - **Total runtime**: 执行时间为 360.127 毫秒。

有多余关系表

- 执行计划分析：
 - **Hash Join**: 两个 `Hash Join` 操作。首先是将 `lineitem` 和 `orders` 表进行哈希连接，然后再将 `lineitem` 和 `part` 表进行哈希连接。
 - **Seq Scan**: 对 `orders`、`lineitem` 和 `part` 表进行顺序扫描。

- **Total runtime:** 执行时间为 686.363 毫秒。

执行结果和时间对比

查询类型	执行计划	执行时间	说明
查询 1 (多余关系表)	使用了多个 Hash Join 和 Seq Scan	686.363 ms	增加了多余的 part 表，导致查询变慢。
查询 2 (最简化查询)	使用了 Merge Join 和 Index Scan	360.127 ms	仅连接了 orders 和 lineitem 表，查询更快。

- **执行时间对比:**
 - **查询 1 (多余关系表) :** 由于增加了 part 表，查询中执行了更多的连接操作（两个 Hash Join），导致查询时间显著增加。总执行时间为 686.363 毫秒。
 - **查询 2 (最简化查询) :** 仅连接了 orders 和 lineitem 表，查询执行更高效。总执行时间为 360.127 毫秒。
- **执行计划对比:**
 - **查询 1** 使用了多余的 part 表，导致执行计划中增加了两个 Hash Join 操作。这不仅增加了计算开销，还导致了更多的数据处理。
 - **查询 2** 使用了 Merge Join，这是连接两个表时更高效的方式，特别是当两个表的连接条件有索引时，Merge Join 通常比 Hash Join 更快速。

结论

- **多余关系表的影响:**
 - 增加不必要的表会导致更多的连接操作，这增加了执行计划的复杂性，特别是当这些表需要通过 Hash Join 或 Merge Join 进行连接时，计算量显著增加。
 - 在执行计划中，使用更多的表和更复杂的连接操作会增加查询的执行时间。
- **最简化查询的优势:**
 - 最简化的查询只连接了必要的表，执行计划简单，使用了高效的 Merge Join，因此查询速度较快。

7 数据库事务管理

单事务/串行事务执行原子性保障机制验证

建立用于实验的表的拷贝（对于前两句语句，建立完表 partsupp_1 后，将 partsupp_1 改为 partsupp_2 建立另一个表）：

```

1 CREATE TABLE partsupp_1(
2     PS_PARTKEY integer NOT NULL,
3     PS_SUPPKEY integer NOT NULL,
4     PS_AVAILQTY integer NOT NULL,
5     PS_SUPPLYCOST decimal(15, 2) NOT NULL,
6     PS_COMMENT varchar(199) NOT NULL
7 );
8
9 ## 插入数据
10 insert into partsupp_1
11 select *
12 from partsupp;
13
14 ## 添加约束(用于实验)
15 alter table partsupp_1 add constraint partsupp_chk_1
16 check(PS_SUPPKEY >= 0);
17 alter table partsupp_2 add constraint partsupp_chk_2
18 check(PS_SUPPKEY >= 0);

```

查看实验操作前表中的数据：

```

1 select count(*)
2 from partsupp_1
3 where PS_SUPPKEY < 10;

```

```

omm=# select count(*) from partsupp_1 where PS_SUPPKEY < 10;
count
-----
 720
(1 row)

× SSH: database ⊗ 0 △ 0  ⟲ 0

```

```

1 select count(*)
2 from partsupp_2
3 where PS_SUPPKEY < 10;

```

```

omm=# select count(*) from partsupp_2 where PS_SUPPKEY < 10;
count
-----
 720
(1 row)

× SSH: database ⊗ 0 △ 0  ⟲ 0

```

单一事务执行 update 操作

在单一事务里运行一条语句：

```

1 START TRANSACTION;
2
3 update partsupp_2
4 set PS_SUPPKEY = PS_SUPPKEY - 10;
5
6 END;

```

该更新将导致部分元组属性 PS_SUPPKEY 的值小于 0，不满足约束，运行后事务回滚：

```
1 omm=# START TRANSACTION;
2 START TRANSACTION
3 omm=# update partsupp_2 set PS_SUPPKEY = PS_SUPPKEY - 10;
4 ERROR: new row for relation "partsupp_2" violates check constraint "partsupp_chk_2"
5 DETAIL: Failing row contains (1, -8, 1, 771.64, , even theodolites. regular, final theodolites eat after
6 the car...).
7 omm=# end;
8 ROLLBACK
```

经过尝试，即使将最后一句 END 更换为 COMMIT，最终事务也会回滚。

若更新满足约束，则事务可以正常提交：

```
1 START TRANSACTION;
2
3 update partsupp_2
4 set PS_SUPPKEY = PS_SUPPKEY + 5;
5
6 COMMIT;
```

```
omm="# START TRANSACTION;
START TRANSACTION
omm="# update partsupp_2 set PS_SUPPKEY = PS_SUPPKEY + 5;
UPDATE 160004
omm="# COMMIT;
COMMIT
```

事务内部单条语句失败对事务的整体影响

不将指令放在一个事务里顺序执行：这样其实是串行地执行了两个事务。

指令1：

```
1 update partsupp_1
2 set PS_SUPPKEY = PS_SUPPKEY - 10;
```

不满足约束，报错：

```
1 omm="# update partsupp_1 set PS_SUPPKEY = PS_SUPPKEY - 10;
2 ERROR: new row for relation "partsupp_1" violates check constraint "partsupp_chk_1"
3 DETAIL: Failing row contains (1, -8, 1, 771.64, , even theodolites. regular, final theodolites eat after
4 the car...).
5 omm="#
```

指令2：

```
1 update partsupp_1
2 set PS_SUPPKEY = PS_SUPPKEY + 5;
```

执行成功。

运行后查询：

```
1 select count(*)
2 from partsupp_1
3 where PS_SUPPKEY < 10;
```

```
① omm=# select count(*) from partsupp_1 where PS_SUPPKEY < 10;
   count
   -----
      320
(1 row)

omm=#
× SSH: database ⊗ 0 △ 0 ॥ 0
```

满足查询条件的元组数量发生变化，说明在上面的语句运行失败的情况下，下面的语句正常执行（两者相互独立）。

将指令放在同一个事务里执行：

```
1 START TRANSACTION;
2
3 update partsupp_2
4 set PS_SUPPKEY = PS_SUPPKEY - 10;
5
6 update partsupp_2
7 set PS_SUPPKEY = PS_SUPPKEY + 5;
8
9 END;
```

第二句语句不满足约束，事务结束后自动回滚：

```
① omm=# START TRANSACTION;
START TRANSACTION
omm=# update partsupp_2 set PS_SUPPKEY = PS_SUPPKEY - 10;
ERROR: new row for relation "partsupp_2" violates check constraint "partsupp_chk_2"
DETAIL: Failing row contains (1, -8, 1, 771.64, , even theodolites. regular, final theodolites eat after
the car...).
omm=# update partsupp_2 set PS_SUPPKEY = PS_SUPPKEY + 5;
ERROR: current transaction is aborted, commands ignored until end of transaction block, firstChar[Q]
omm=# END;
ROLLBACK
× SSH: database ⊗ 0 △ 0 ॥ 0
```

运行后查询：

```
1 select count(*)
2 from partsupp_2
3 where PS_SUPPKEY < 10;
```

```
① omm=# select count(*) from partsupp_2 where PS_SUPPKEY < 10;
   count
   -----
      720
(1 row)

omm=#
× SSH: database ⊗ 0 △ 0 ॥ 0
```

满足查询条件的元组数量没有变化，说明事务中的两句话都没有执行，这保证了事务的原子性，即事务内的操作要么完全运行，要么完全不运行。

利用保存点回滚事务

openGauss 中的保存点类似在事务内的检查点，使用保存点可以实现事务回滚到保存点位置，而非直接回滚整个事务。

以下代码在保存点前插入一条元组，又在保存点后将这条元组删除，运行回滚到保存点的语句后提交事务：

```
1 START TRANSACTION;
2
3 INSERT INTO partsupp_1
4 values(2022, 2022, 7, 0, 'comment');
5
6 savepoint sp;
7
8 delete from partsupp_1
9 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
10
11 rollback to sp;
12
13 COMMIT;
```

运行前查询：

```
1 select count(*)
2 from partsupp_1;
```

```
o�m# select count(*) from partsupp_1;
count
-----
160004
(1 row)
```

在将要运行的事务代码中插入查询语句，观察表的变化：

```
1 START TRANSACTION;
2
3 INSERT INTO partsupp_1
4 values(2022, 2022, 7, 0, 'comment');
5
6 savepoint sp;
7
8 select count(*)
9 from partsupp_1;
```

```
o�m# START TRANSACTION;
START TRANSACTION
o�m# INSERT INTO partsupp_1
o�m# values(2022, 2022, 7, 0, 'comment');
INSERT 0 1
o�m# savepoint sp;
SAVEPOINT
o�m# select count(*)
o�m# from partsupp_1;
count
-----
160005
(1 row)
```

这说明新的元组插入成功。

```

1 delete from partsupp_1
2 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
3
4 select count(*)
5 from partsupp_1;

```

```

omm=# delete from partsupp_1
omm-# where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
DELETE 1
omm-# select count(*)
omm-# from partsupp_1;
count
-----
160004
(1 row)

```

The screenshot shows the command being run in an Oracle terminal. The output indicates that one row was deleted from the partsupp_1 table where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022'. The count command shows there are now 160004 rows left.

这说明在事务内将新插入的元组删除成功.

```

1 rollback to sp;
2
3 COMMIT;
4
5 select count(*)
6 from partsupp_1;

```

```

omm=# rollback to sp;
ROLLBACK
omm-# COMMIT;
COMMIT
omm-# select count(*)
omm-# from partsupp_1;
count
-----
160005
(1 row)

```

The screenshot shows the command being run in an Oracle terminal. After a partial rollback to save point 'sp', the count command shows there are 160005 rows in the partsupp_1 table, indicating that the DELETE operation was rolled back.

这说明使用保存点实现了事务的部分回滚，撤销了删除操作，没有撤销插入操作.

事务并发执行时的独立性保障机制验证

打开两个终端，连接相同的数据库.

实验前查询用于实验的元组的值：

```

1 select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY
2 from partsupp_1
3 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';

```

Terminal 1 (Left)	Terminal 2 (Right)
<code>omm=# select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY f rom partsupp_1 where PS_PARTKEY = '2022' and PS_SU PPKEY = '2022'; ps_partkey ps_suppkey ps_availqty -----+-----+----- 2022 2022 7 (1 row)</code>	<code>omm=# select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY f rom partsupp_1 where PS_PARTKEY = '2022' and PS_SU PPKEY = '2022'; ps_partkey ps_suppkey ps_availqty -----+-----+----- 2022 2022 7 (1 row)</code>
<code>omm=# </code>	<code>omm=# </code>

The screenshot shows two Oracle terminals running the same query simultaneously. Both terminals return the same result set, showing a single row with PS_PARTKEY = 2022, PS_SUPPKEY = 2022, and PS_AVAILQTY = 7. This demonstrates that the reads are independent even though they are happening at the same time.

两终端显示相同查询结果.

read committed 隔离级别下的脏读、不可重复读、幻读

分别在两个窗口创建 read committed 隔离级别下的事务 T1 和 T2：

```
1 | START TRANSACTION ISOLATION LEVEL read committed;
```

在事务 T1 中将上述元组的 PS_AVAILQTY 值修改为 1，但不提交，在事务 T2 中查询上述元组的值：

```
1 | # T1
2 | update partsupp_1
3 | set PS_AVAILQTY = 1
4 | where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
```

```
1 | # T2
2 | select PS_AVAILQTY
3 | from partsupp_1
4 | where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
```

```
-----+-----+
2022 | 2022 | 7
(1 row)

omm=# START TRANSACTION ISOLATION LEVEL read committed;
START TRANSACTION
omm=# update partsupp_1 set PS_AVAILQTY = 1 where
PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
UPDATE 1
omm=#
-----+-----+
7
(1 row)

omm=#
-----+-----+
omm=#
-----+-----+
```

事务 T2 读取到的不是事务 T1 未提交的值，这说明在 read committed 隔离级别下，T2 没有读取到脏数据。

将事务 T1 提交，再在事务 T2 中查询上述元组的值：

```
1 | # T1
2 | COMMIT;
```

```
1 | # T2
2 | select PS_AVAILQTY
3 | from partsupp_1
4 | where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
```

```
-----+-----+
2022 | 2022 | 7
(1 row)

omm=# select PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY f
rom partsupp_1 where PS_PARTKEY = '2022' and PS_SU
PPKEY = '2022';
ps_partkey | ps_suppkey | ps_availqty
-----+-----+
2022 | 2022 | 7
(1 row)

omm=# START TRANSACTION ISOLATION LEVEL read commi
tted;
START TRANSACTION
omm=# update partsupp_1 set PS_AVAILQTY = 1 where
PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
UPDATE 1
omm=#
-----+-----+
1
(1 row)

omm=#
-----+-----+
omm=#
-----+-----+
```

事务 T2 中出现了连续两次相同查询结果不同的现象，这说明在 read committed 隔离级别下可能出现不可重复读。

在第一个窗口再次创建 read committed 隔离级别的事务 T3，在 T3 中删除上述元组并提交，在事务 T2 中查询上述元组的值：

```
1 # T3
2 delete from partsupp_1
3 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
```

```
1 # T2
2 select PS_AVAILQTY
3 from partsupp_1
4 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
```

The screenshot shows two sessions in Oracle SQL*Plus. Session 1 (T3) starts with 'START TRANSACTION ISOLATION LEVEL read committed'. It then performs a delete operation: 'delete from partsupp_1 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';'. This is followed by a commit: 'COMMIT;'. Session 2 (T2) starts with 'select PS_AVAILQTY from partsupp_1 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';'. It returns one row (1 row). Then it runs the same select statement again, which returns zero rows (0 rows). This demonstrates that the data has been deleted and is no longer visible, characteristic of non-repeatable reads.

事务 T2 此时查不到原有的元组了，这说明在 read committed 隔离级别下可能出现幻读。

repeatable read 隔离级别下的脏读、不可重复读、幻读

为方便对比，将上面删去的元组重新插入表中，这次设置隔离级别为 repeatable read：

```
1 # T1, T2
2 START TRANSACTION ISOLATION LEVEL repeatable read;
```

之后重复相同的步骤：

```
1 # T1
2 update partsupp_1
3 set PS_AVAILQTY = 1
4 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
```

```
1 # T2
2 select PS_AVAILQTY
3 from partsupp_1
4 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
```

```

    COMMIT
omm="# update partsupp_1 set PS_AVAILQTY = 7 where
PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
UPDATE 1
omm="# START TRANSACTION ISOLATION LEVEL repeatable
read;
START TRANSACTION
omm="# select PS_AVAILQTY from partsupp_1 where PS_
PARTKEY = '2022' and PS_SUPPKEY = '2022';
ps_availqty
-----
7
(1 row)
omm="# "

```

SSH: database

仍然不会出现脏读.

```

1 # T1
2 COMMIT;

```

```

1 # T2
2 select PS_AVAILQTY
3 from partsupp_1
4 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';

```

```

    read;
START TRANSACTION
omm="# update partsupp_1 set PS_AVAILQTY = 1 where
PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';
UPDATE 1
omm="# COMMIT;
COMMIT
omm="# "

```

SSH: database

这次没有出现不可重复读.

```

1 # T3
2 delete from partsupp_1
3 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';

```

```

1 # T2
2 select PS_AVAILQTY
3 from partsupp_1
4 where PS_PARTKEY = '2022' and PS_SUPPKEY = '2022';

```

```

    omm="# START TRANSACTION ISOLATION LEVEL repeatable
read;
START TRANSACTION
omm="# delete from partsupp_1 where PS_PARTKEY = '2
022' and PS_SUPPKEY = '2022';
DELETE 1
omm="# COMMIT;
COMMIT
omm="# "

```

SSH: database

也没有出现幻读.

在 `repeatable read` 隔离级别下，事务内的查询操作看到的是它开始时的表的快照，这样，事务查询到的数据只受到它开始前执行的操作的影响；而在 `read committed` 隔离级别下，事务查询到的数据受到它开始后执行的操作的影响，如果两次查询之间有其它事务对查询的数据进行了更改，两次查询的结果就会不一致，造成不可重复读和幻读。