



PostgreSQL 提供了丰富的高级特性，本章将介绍 PostgreSQL 在 SQL 方面的高级特性，例如 WITH 查询、批量插入、RETURNING 返回修改的数据、UPSERT、数据抽样、聚合函数、窗口函数。

第 4 章

Chapter 4

SQL 高级特性

本章将介绍 PostgreSQL 在 SQL 方面的高级特性，例如 WITH 查询、批量插入、RETURNING 返回修改的数据、UPSERT、数据抽样、聚合函数、窗口函数。

4.1 WITH 查询

WITH 查询是 PostgreSQL 支持的高级 SQL 特性之一，这一特性常称为 CTE(Common Table Expressions)，WITH 查询在复杂查询中定义一个辅助语句（可理解成在一个查询中定义的临时表），这一特性常用于复杂查询或递归查询应用场景。

4.1.1 复杂查询使用 CTE

先通过一个简单的 CTE 示例了解 WITH 查询，如下所示：

```
WITH t as (
    SELECT generate_series(1,3)
)
SELECT * FROM t;
```

执行结果如下：

```
generate_series
-----
 1
 2
 3
(3 rows)
```

这个简单的 CTE 示例中，一开始定义了一条辅助语句 t 取数，之后在主查询语句中查



询 t，定义的辅助语句就像是定义了一张临时表，对于复杂查询如果不使用 CTE，可以通过创建视图方式简化 SQL。

CTE 可以简化 SQL 并且减少嵌套，因为可以预先定义辅助语句，之后在主查询中多次调用。接着看一个稍复杂 CTE 例子，这个例子来自手册，如下所示：

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

这个例子首先定义了 regional_sales 和 top_regions 两个辅助语句，regional_sales 算出每个区域的总销售量，top_regions 算出销售量占总销售量 10% 以上的所有区域，主查询语句通过辅助语句与 orders 表关联，算出了顶级区域每件商品的销售量和销售额。

4.1.2 递归查询使用 CTE

WITH 查询的一个重要属性是 RECURSIVE，使用 RECURSIVE 属性可以引用自己的输出，从而实现递归，一般用于层次结构或树状结构的应用场景，一个简单的 RECURSIVE 例子如下所示：

```
WITH recursive t (x) as (
    SELECT 1
    UNION
    SELECT x + 1
    FROM t
    WHERE x < 5
)
SELECT sum(x) FROM t;
```

输出结果为：

```
sum
-----
15
(1 row)
```

上述例子中 x 从 1 开始，union 加 1 后的值，循环直到 x 小于 5 结束，之后计算 x 值的总和。



接着分享一个递归查询的案例，这个案例来自 PostgreSQL 社区论坛一位朋友的问题，他的问题是这样的，存在一张包含如下数据的表：

```
id name fatherid
1 中国 0
2 辽宁 1
3 山东 1
4 沈阳 2
5 大连 2
6 济南 3
7 和平区 4
8 沈河区 4
```

当给定一个 id 时能得到它完整的地名，例如当 id=7 时，地名是：中国辽宁沈阳和平区，当 id=5 时，地名是：中国辽宁大连，这是一个典型的层次数据递归应用场景，恰好可以通过 PostgreSQL 的 WITH 查询实现，首先创建测试表并插入数据，如下所示：

```
CREATE TABLE test_area(id int4, name varchar(32), fatherid int4);

INSERT INTO test_area VALUES (1, '中国', 0);
INSERT INTO test_area VALUES (2, '辽宁', 1);
INSERT INTO test_area VALUES (3, '山东', 1);
INSERT INTO test_area VALUES (4, '沈阳', 2);
INSERT INTO test_area VALUES (5, '大连', 2);
INSERT INTO test_area VALUES (6, '济南', 3);
INSERT INTO test_area VALUES (7, '和平区', 4);
INSERT INTO test_area VALUES (8, '沈河区', 4);
```

使用 PostgreSQL 的 WITH 查询检索 ID 为 7 以及以上的所有父节点，如下所示：

```
WITH RECURSIVE r AS (
    SELECT * FROM test_area WHERE id = 7
    UNION ALL
    SELECT test_area.* FROM test_area, r WHERE test_area.id = r.fatherid
)
SELECT * FROM r ORDER BY id;
```

查询结果如下：

```
id | name | fatherid
----+-----+
 1 | 中国 |      0
 2 | 辽宁 |      1
 4 | 沈阳 |      2
 7 | 和平区 |     4
(4 rows)
```

查询结果正好是 ID=7 节点以及它的所有父节点，接下来将输出结果的 name 字段合并成“中国辽宁沈阳和平区”，方法很多，这里通过 string_agg 函数实现，如下所示：

```
mydb=> WITH RECURSIVE r AS (
```



```
mydb=> SELECT * FROM test_area WHERE id = 7
      UNION ALL
      SELECT test_area.* FROM test_area, r WHERE test_area.id = r.fatherid
    )
SELECT string_agg(name,'') FROM ( SELECT name FROM r ORDER BY id) n;
   string_agg
-----
中国辽宁沈阳和平区
```

以上是查找当前节点以及当前节点的所有父节点，也可以查找当前节点以及其下的所有子节点，需更改 where 条件，如查找沈阳市及管辖的区，代码如下所示。

```
mydb=> WITH RECURSIVE r AS (
      SELECT * FROM test_area WHERE id = 4
      UNION ALL
      SELECT test_area.* FROM test_area, r WHERE test_area.fatherid = r.id
    )
SELECT * FROM r ORDER BY id;
id | name | fatherid
----+-----+
 4 | 沈阳 |      2
 7 | 和平区 |     4
 8 | 沈河区 |     4
(3 rows)
```

以上给出了 CTE 的两个应用场景：复杂查询中的应用和递归查询中的应用，通过示例很容易知道 CTE 具有以下优点：

- CTE 可以简化 SQL 代码，减少 SQL 嵌套层数，提高 SQL 代码的可读性。
- CTE 的辅助语句只需要计算一次，在主查询中可以多次使用。
- 当不需要共享查询结果时，相比视图更轻量。

4.2 批量插入

批量插入是指一次性插入多条数据，主要用于提升数据插入效率，PostgreSQL 有多种方法实现批量插入。

4.2.1 方式一：INSERT INTO...SELECT...

通过表数据或函数批量插入，语法如下：

```
INSERT INTO table_name SELECT...FROM source_table
```

比如创建一张表结构和 user_ini 相同的表并插入 user_ini 表的全量数据，代码如下所示：

```
mydb=> CREATE TABLE tbl_batch1(user_id int8,user_name text);
CREATE TABLE
```



```
mydb=> INSERT INTO tbl_batch1(user_id,user_name)
SELECT user_id,user_name FROM user_ini;
INSERT 0 1000000
```

以上示例将表 user_ini 的 user_id、user_name 字段所有数据插入表 tbl_batch1，也可以插入一部分数据，插入时指定 where 条件即可。

通过函数进行批量插入，如下所示：

```
mydb=> CREATE TABLE tbl_batch2 (id int4,info text);
CREATE TABLE

mydb=> INSERT INTO tbl_batch2(id,info)
SELECT generate_series(1,5), 'batch2';
INSERT 0 5
```

通过 SELECT 表数据批量插入的方式大多关系型数据库都支持，接下来看看 PostgreSQL 支持的其他批量插入方式。

4.2.2 方式二：INSERT INTO VALUES (),(),...()

PostgreSQL 的另一种支持批量插入的方法为在一条 INSERT 语句中通过 VALUES 关键字插入多条记录，通过一个例子就很容易理解，如下所示：

```
mydb=> CREATE TABLE tbl_batch3(id int4,info text);
CREATE TABLE

mydb=> INSERT INTO tbl_batch3(id,info) VALUES (1,'a'),(2,'b'),(3,'c');
INSERT 0 3
```

数据如下所示：

```
mydb=> SELECT * FROM tbl_batch3;
 id | info
-----+
 1 | a
 2 | b
 3 | c
(3 rows)
```

这种批量插入方式非常独特，一条 SQL 插入多行数据，相比一条 SQL 插入一条数据的方式能减少和数据库的交互，减少数据库 WAL（Write-Ahead Logging）日志的生成，提升插入效率，通常很少有开发人员了解 PostgreSQL 的这种批量插入方式。

4.2.3 方式三：COPY 或 \COPY 元命令

2.2.3 节介绍了 psql 导入、导出表数据，使用的是 COPY 命令或 \copy 元命令，copy 或 \copy 元命令能够将一定格式的文件数据导入到数据库中，相比 INSERT 命令插入效率更高，通常大数据量的文件导入一般在数据库服务端主机通过 PostgreSQL 超级用户使用



COPY 命令导入，下面通过一个例子简单看看 COPY 命令的效率，测试机为一台物理机上的虚机，配置为 4 核 CPU，8GB 内存。

首先创建一张测试表并插入一千万数据，如下所示：

```
mydb=> CREATE TABLE tbl_batch4(
    id int4,
    info text,
    create_time timestamp(6) with time zone default clock_timestamp());
CREATE TABLE

mydb=> INSERT INTO tbl_batch4(id,info) SELECT n,n||'_batch4'
FROM generate_series(1,10000000) n;
INSERT 0 10000000
```

以上示例通过 INSERT 插入一千万数据，将一千万数据导出到文件，如下所示：

```
[postgres@pghost1 ~]$ psql mydb postgres
psql (10.0)
Type "help" for help.

mydb=# \timing
Timing is on.

mydb=# COPY pguser.tbl_batch4 TO '/home/pg10/tbl_batch4.txt';
COPY 10000000
Time: 6575.787 ms (00:06.576)
```

一千万数据导出花了 6575 毫秒，之后清空表 `tbl_batch4` 并将文件 `tbl_batch4.txt` 的一千万数据导入到表中，如下所示：

```
mydb=# TRUNCATE TABLE pguser.tbl_batch4;
TRUNCATE TABLE
mydb=# COPY pguser.tbl_batch4 FROM '/home/pg10/tbl_batch4.txt';
COPY 10000000
Time: 15663.834 ms (00:15.664)
```

一千万数据通过 COPY 命令导入执行时间为 15 663 毫秒。

4.3 RETURNING 返回修改的数据

PostgreSQL 的 RETURNING 特性可以返回 DML 修改的数据，具体为三个场景：INSERT 语句后接 RETURNING 属性返回插入的数据；UPDATE 语句后接 RETURNING 属性返回更新后的newValue；DELETE 语句后接 RETURNING 属性返回删除的数据。这个特性的优点在于不需要额外的 SQL 获取这些值，能够方便应用开发，下面通过示例演示。

4.3.1 RETURNING 返回插入的数据

INSERT 语句后接 RETURNING 属性返回插入的值，下面的代码创建测试表，并返回

已插入的整行数据。

```
mydb=> CREATE TABLE test_r1(id serial,flag char(1));
CREATE TABLE
mydb=> INSERT INTO test_r1(flag) VALUES ('a') RETURNING *;
      id | flag
-----+-----
      1 | a
(1 row)
INSERT 0 1
```

RETURNING * 表示返回表插入的所有字段数据，也可以返回指定字段，RETURNING 后接字段名即可，如下代码仅返回插入的 id 字段：

```
mydb=> INSERT INTO test_r1(flag) VALUES ('b') RETURNING id;
      id
-----
      2
(1 row)
INSERT 0 1
```

4.3.2 RETURNING 返回更新后数据

UPDATE 后接 RETURNING 属性返回 UPDATE 语句更新后的值，如下所示：

```
mydb=> SELECT * FROM test_r1 WHERE id=1;
      id | flag
-----+-----
      1 | a
(1 row)
mydb=> UPDATE test_r1 SET flag='p' WHERE id=1 RETURNING *;
      id | flag
-----+-----
      1 | p
(1 row)
UPDATE 1
```

4.3.3 RETURNING 返回删除的数据

DELETE 后接 RETURNING 属性返回删除的数据，如下所示：

```
mydb=> DELETE FROM test_r1 WHERE id=2 RETURNING *;
      id | flag
-----+-----
      2 | b
(1 row)
DELETE 1
```

4.4 UPSERT

PostgreSQL 的 UPSERT 特性是指 INSERT ... ON CONFLICT UPDATE，用来解决在数据插入过程中数据冲突的情况，比如违反用户自定义约束，在日志数据应用场景中，通常会在事务中批量插入日志数据，如果其中有一条数据违反表上的约束，则整个插入事务将会回滚，PostgreSQL 的 UPSERT 特性能解决这一问题。

4.4.1 UPSERT 场景演示

接下来通过例子来理解 UPSERT 的功能，定义一张用户登录日志表并插入一条数据，如下所示：

```
mydb=> CREATE TABLE user_logins(user_name text primary key,
    login_cnt int4,
    last_login_time timestamp(0) without time zone);
CREATE TABLE

mydb=> INSERT INTO user_logins(user_name,login_cnt) VALUES ('francs',1);
INSERT 0 1
```

在 user_logins 表 user_name 字段上定义主键，批量插入数据中如有重复会报错，如下所示：

```
mydb=> INSERT INTO user_logins(user_name,login_cnt)
VALUES ('matiler',1),('francs',1);
ERROR: duplicate key value violates unique constraint "user_logins_pkey"
DETAIL: Key (user_name)=(francs) already exists.
```

上述 SQL 试图插入两条数据，其中 matiler 这条数据不违反主键冲突，而 francs 这条数据违反主键冲突，结果两条数据都不能插入。PostgreSQL 的 UPSERT 可以处理冲突的数据，比如当插入的数据冲突时不报错，同时更新冲突的数据，如下所示：

```
mydb=> INSERT INTO user_logins(user_name,login_cnt)
VALUES ('matiler',1),('francs',1)
ON CONFLICT(user_name)
DO UPDATE SET
login_cnt=user_logins.login_cnt+EXCLUDED.login_cnt,last_login_time=now();
INSERT 0 2
```

上述 INSERT 语句插入两条数据，并设置规则：当数据冲突时将登录次数字段 login_cnt 值加 1，同时更新最近登录时间 last_login_time，ON CONFLICT(user_name) 定义冲突类型为 user_name 字段，DO UPDATE SET 是指冲突动作，后面定义了一个 UPDATE 语句。注意上述 SET 命令中引用了 user_loins 表和内置表 EXCLUDED，引用原表 user_loins 访问表中已存在的冲突记录，内置表 EXCLUDED 引用试图插入的值，再次查询表 user_login，如下所示：

```
mydb=> SELECT * FROM user_logins ;
```



```
user_name | login_cnt | last_login_time
-----+-----+-----
matiler | 1 |
francs | 2 | 2017-08-08 15:23:13
(2 rows)
```

一方面冲突的 frans 这条数据被更新了 login_cnt 和 last_login_time 字段，另一方面新的数据 matiler 记录已正常插入。

也可以定义数据冲突后啥也不干，这时需指定 DO NOTHING 属性，如下所示：

```
mydb=> INSERT INTO user_logins(user_name,login_cnt)
VALUES ('tutu',1),('francs',1)
ON CONFLICT(user_name) DO NOTHING;
INSERT 0 1
```

再次查询表数据，新的数据 tutu 这条已插入到表中，冲突的数据 frans 这行啥也没变，结果如下所示：

```
mydb=> SELECT * FROM user_logins ;
user_name | login_cnt | last_login_time
-----+-----+-----
matiler | 1 |
francs | 2 | 2017-08-08 15:23:13
tutu | 1 |
(3 rows)
```

4.4.2 UPSERT 语法

PostgreSQL 的 UPSERT 语法比较复杂，通过以上演示后再来查看语法则轻松些，语法如下：

```
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
[ ON CONFLICT [ conflict_target ] conflict_action ]

where conflict_target can be one of:
( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ]
[, ...] ) [ WHERE index_predicate ]
ON CONSTRAINT constraint_name

and conflict_action is one of:
DO NOTHING
DO UPDATE SET { column_name = { expression | DEFAULT } |
( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT }
[, ...] ) |
( column_name [, ...] ) = ( sub-SELECT )
} [, ...]
[ WHERE condition ]
```

以上语法主要注意 [ON CONFLICT [conflict_target] conflict_action] 这行，conflict_target 指选择仲裁索引判定冲突行为，一般指定被创建约束的字段；conflict_action 指冲突



动作，可以是 DO NOTHING，也可以是用户自定义的 UPDATE 语句。

4.5 数据抽样

数据抽样 (TABLESAMPLE) 在数据处理方面经常用到，特别是当表数据量比较大时，随机查询表中一定数量记录的操作很常见，PostgreSQL 早在 9.5 版时就已经提供了 TABLESAMPLE 数据抽样功能，9.5 版前通常通过 ORDER BY random() 方式实现数据抽样，这种方式虽然在功能上满足随机返回指定行数据，但性能很低，如下所示：

```
mydb=> SELECT * FROM user_ini ORDER BY random() LIMIT 1;
      id | user_id | user_name |          create_time
-----+-----+-----+-----+
    500449 | 768810 | 2TY6P4 | 2017-08-05 15:59:32.294761+08
(1 row)

mydb=> SELECT * FROM user_ini ORDER BY random() LIMIT 1;
      id | user_id | user_name |          create_time
-----+-----+-----+-----+
    324823 | 740720 | 07SKCU | 2017-08-05 15:59:29.913984+08
(1 row)
```

执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM user_ini ORDER BY random() LIMIT 1;
          QUERY PLAN
-----
Limit  (cost=25599.98..25599.98 rows=1 width=35) (actual time=367.867..367.868 rows=1
loops=1)
->  Sort   (cost=25599.98..28175.12 rows=1030056 width=35) (actual time=
367.866..367.866 rows=1 loops=1)
    Sort Key: (random())
    Sort Method: top-N heapsort  Memory: 25kB
        ->  Seq Scan on user_ini  (cost=0.00..20449.70 rows=1030056 width=35)
            (actual time=0.012..159.569 rows=1000000 loops=1)
Planning time: 0.083 ms
Execution time: 367.909 ms
(7 rows)
```

表 user_ini 数据量为 100 万，从 100 万随机取一条上述 SQL 的执行时间为 367ms，这种方法进行了全表扫描和排序，效率非常低，当表数据量大时，性能几乎无法接受。

9.5 版本以后 PostgreSQL 支持 TABLESAMPLE 数据抽样，语法如下所示：

```
SELECT ...
FROM table_name
TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ]
```

sampling_method 指抽样方法，主要有两种：SYSTEM 和 BERNOULLI，接下来详细介绍这两种抽样方式，argument 指抽样百分比。





注 explain analyze 命令表示实际执行这条 SQL，同时显示 SQL 执行计划和执行时间，Planning time 表示 SQL 语句解析生成执行计划的时间，Execution time 表示 SQL 的实际执行时间。

4.5.1 SYSTEM 抽样方式

SYSTEM 抽样方式为随机抽取表上数据块上的数据，理论上被抽样表的每个数据块被检索的概率是一样的，SYSTEM 抽样方式基于数据块级别，后接抽样参数，被选中的块上的所有数据将被检索，下面使用示例进行说明。

创建 test_sample 测试表，并插入 150 万数据，如下所示：

```
mydb=> CREATE TABLE test_sample(id int4,message text,
create_time timestamp(6) without time zone default clock_timestamp());
CREATE TABLE

mydb=> INSERT INTO test_sample(id,message)
SELECT n, md5(random()::text) FROM generate_series(1,1500000) n;
INSERT 0 1500000

mydb=> SELECT * FROM test_sample LIMIT 1;
 id |          message          |      create_time
----+-----+-----+
 1 | 58f2506410be948963d6d9adf4b4e0c2 | 2017-08-08 21:17:20.984481
(1 row)
```

抽样因子设置成 0.01，意味着返回 $1500\ 000 \times 0.01\% = 150$ 条记录，执行如下 SQL：

```
EXPLAIN ANALYZE SELECT * FROM test_sample TABLESAMPLE SYSTEM(0.01);
```

执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM test_sample TABLESAMPLE SYSTEM(0.01);
          QUERY PLAN
-----
 Sample Scan on test_sample  (cost=0.00..3.50 rows=150 width=45) (actual
   time=0.099..0.146 rows=107 loops=1)
   Sampling: system ('0.01'::real)
   Planning time: 0.053 ms
   Execution time: 0.166 ms
(4 rows)
```

以上执行计划主要有两点，一方面进行了 Sample Scan 扫描（抽样方式为 SYSTEM），执行时间为 0.166 毫秒，性能较好，另一方面优化器预计访问 150 条记录，实际返回 107 条，为什么会返回 107 条记录呢？接着查看表占用的数据块数量，如下所示：

```
mydb=> SELECT relname,relpages FROM pg_class WHERE relname='test_sample';
 relname | relpages
-----+-----
```



```
test_sample | 14019
(1 row)
```

表 test_sample 物理上占用 14 019 个数据块，也就是说每个数据块存储 $1\ 000\ 000 / 14\ 019 = 107$ 条记录。

查看抽样数据的 ctid，如下所示：

```
mydb=> SELECT ctid,* FROM test_sample TABLESAMPLE SYSTEM(0.01);
+-----+-----+-----+-----+
| ctid | id   | message          | create_time      |
+-----+-----+-----+-----+
| (5640,1) | 603481 | 385484b3452b245e46388d71ce4ea928 | 2017-08-08 21:17:23.32394
| (5640,2) | 603482 | e09c526118f1d4b3c391d59ae915c4e8 | 2017-08-08 21:17:23.32396
...省略很多行
| (5640,107) | 603587 | c33875a052f4ca63c4b38c649fb6bcc3 | 2017-08-08 21:17:23.324336
(107 rows)
```

ctid 是表的隐藏列，括号里的第一位表示逻辑数据块编号，第二位表示逻辑块上的数据的逻辑编号，从以上看出，这 107 条记录都存储在逻辑编号为 5640 的数据块上，也就是说抽样查询返回了一个数据块上的所有数据，抽样因子固定为 0.01，多次执行以下查询，如下所示：

```
mydb=> SELECT count(*) FROM test_sample TABLESAMPLE SYSTEM(0.01);
+-----+
| count |
+-----+
| 214   |
(1 row)

mydb=> SELECT count(*) FROM test_sample TABLESAMPLE SYSTEM(0.01);
+-----+
| count |
+-----+
| 107   |
(1 row)
```

再次查询发现返回的记录为 214 或 107，由于一个数据块存储 107 条记录，因此查询结果有时返回了两个数块上的所有数据，这是因为抽样因子设置成 0.01，意味着返回 $1\ 500\ 000 \times 0.01\% = 150$ 条记录，150 条记录需要两个数据块存储，这也验证了 SYSTEM 抽样方式返回的数据以数据块为单位，被抽样的块上的所有数据被检索。

4.5.2 BERNOULLI 抽样方式

BERNOULLI 抽样方式随机抽取表的数据行，并返回指定百分比数据，BERNOULLI 抽样方式基于数据行级别，理论上被抽样表的每行记录被检索的概率是一样的，因此 BERNOULLI 抽样方式抽取的数据相比 SYSTEM 抽样方式具有更好的随机性，但性能上相比 SYSTEM 抽样方式低很多，下面演示下 BERNOULLI 抽样方式，同样基于 test_sample 测试表。



设置抽样方式为 BERNOULLI，抽样因子为 0.01，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM test_sample TABLESAMPLE BERNOULLI (0.01);
          QUERY PLAN
-----
      Sample Scan on test_sample  (cost=0.00..14020.50 rows=150 width=45) (actual
      time=0.025..22.541 rows=152 loops=1)
      Sampling: bernoulli ('0.01'::real)
      Planning time: 0.063 ms
      Execution time: 22.569 ms
(4 rows)
```

从以上执行计划看出进行了 Sample Scan 扫描（抽样方式为 BERNOULLI），执行计划预计返回 150 条记录，实际返回 152 条，从返回的记录数来看，非常接近 150 条 ($1\ 000\ 000 \times 0.01\%$)，但执行时间却要 22.569 毫秒，性能相比 SYSTEM 抽样方式 0.166 毫秒差了 136 倍。

多次执行以下查询，查看返回记录数的变化，如下所示：

```
mydb=> SELECT count(*) FROM test_sample TABLESAMPLE BERNOULLI(0.01);
      count
-----
      151
(1 row)

mydb=> SELECT count(*) FROM test_sample TABLESAMPLE BERNOULLI(0.01);
      count
-----
      147
(1 row)
```

从以上看出，BERNOULLI 抽样方式返回的数据量非常接近抽样数据的百分比，而 SYSTEM 抽样方式数据返回以数据块为单位，被抽样的块上的所有数据都被返回，因此 SYSTEM 抽样方式返回的数据量偏差较大。

由于 BERNOULLI 抽样基于数据行级别，猜想返回的数据应该位于不同的数据块上，通过查询表的 ctid 进行验证，如下所示：

```
mydb=> SELECT ctid,id,message
      FROM test_sample TABLESAMPLE BERNOULLI(0.01) LIMIT 3;
      ctid | id | message
-----+---+-
(55, 30) | 5915 | f3803f234f6cf6cdd276d9d027487582
(240, 23) | 25703 | c04af69ac76f6465832e0cd87939a1af
(318, 3) | 34029 | dd35438b24980d1a8ed2d3f5edd5ca1c
```

从以上三条记录的 ctid 信息看出，三条数据分别位于数据块 55、240、318 上，因此 BERNOULLI 抽样方式随机性相比 SYSTEM 抽样方式更好。

本节演示了 SYSTEM 和 BERNOULLI 抽样方式，SYSTEM 抽样方式基于数据块级别，



随机抽取表数据块上的记录，因此这种方式抽取的记录的随机性不是很好，但返回的数据以数据块为单位，抽样性能很高，适用于抽样效率优先的场景，例如抽样大小为上百 GB 的日志表；而 BERNOULLI 抽样方式基于数据行，相比 SYSTEM 抽样方式所抽样的数据随机性更好，但性能相比 SYSTEM 差很多，适用于抽样随机性优先的场景。读者可根据实际应用场景选择抽样方式。

4.6 聚合函数

聚合函数可以对结果集进行计算，常用的聚合函数有 avg()、sum()、min()、max()、count() 等，本节将介绍 PostgreSQL 两个特殊功能的聚合函数并给出测试示例。

在介绍两个聚合函数之前，先来看一个应用场景，假如一张表有以下数据：

中国	台北
中国	香港
中国	上海
日本	东京
日本	大阪

要求得到如下结果集：

中国	台北, 香港, 上海
日本	东京, 大阪

读者想想这个 SQL 如何写？

4.6.1 string_agg 函数

首先介绍 string_agg 函数，此函数语法如下所示：

```
string_agg(expression, delimiter)
```

简单地说 string_agg 函数能将结果集某个字段的所有行连接成字符串，并用指定 delimiter 分隔符分隔，expression 表示要处理的字符类型数据；参数的类型为 (text, text) 或 (bytea, bytea)，函数返回的类型同输入参数类型一致，bytea 属于二进制类型，使用情况不多，我们主要介绍 text 类型输入参数，本节开头的场景正好可以用 string_agg 函数处理。

首先创建测试表并插入以下数据：

```
CREATE TABLE city (country character varying(64),city character varying(64));
INSERT INTO city VALUES ('中国','台北');
INSERT INTO city VALUES ('中国','香港');
INSERT INTO city VALUES ('中国','上海');
INSERT INTO city VALUES ('日本','东京');
INSERT INTO city VALUES ('日本','大阪');
```

数据如下所示：



```
mydb=> SELECT * FROM city;
      country | city
-----+-----
    中国    | 台北
    中国    | 香港
    中国    | 上海
    日本    | 东京
    日本    | 大阪
(5 rows)
```

将 city 字段连接成字符串的代码如下所示：

```
mydb=> SELECT string_agg(city,',') FROM city;
      string_agg
-----+
台北,香港,上海,东京,大阪
(1 row)
```

可见 string_agg 函数将输出的结果集连接成了字符串，并用指定的逗号分隔符分隔，回到本文开头的问题，通过 SQL 实现，如下所示：

```
mydb=> SELECT country,string_agg(city,',') FROM city GROUP BY country;
      country | string_agg
-----+-----
    日本    | 东京,大阪
    中国    | 台北,香港,上海
(2 rows)
```

4.6.2 array_agg 函数

array_agg 函数和 string_agg 函数类似，最主要的区别为返回的类型为数组，数组数据类型同输入参数数据类型一致，array_agg 函数支持两种语法，第一种如下所示：

```
array_agg(expression) --输入参数为任何非数组类型
```

输入参数可以是任何非数组类型，返回的结果是一维数组，array_agg 函数将结果集某个字段的所有行连接成数组，例如执行以下查询：

```
mydb=> SELECT country,array_agg(city) FROM city GROUP BY country;
      country | array_agg
-----+-----
    日本    | {东京,大阪}
    中国    | {台北,香港,上海}
```

array_agg 函数输出的结果为字符类型数组，其他无明显区别，使用 array_agg 函数主要优点在于可以使用数组相关函数和操作符。

第二种 array_agg 函数语法如下所示：

```
array_agg(expression) --输入参数为任何数组类型
```

第一种 array_agg 函数的输入参数为任何非数组类型，这里输入参数为任何数组类型，



返回类型为多维数组：

首先创建数组表。

```
mydb=> CREATE TABLE test_array3(id int4[]);
CREATE TABLE
mydb=> INSERT INTO test_array3(id) VALUES (array[1,2,3]);
INSERT 0 1
mydb=> INSERT INTO test_array3(id) VALUES (array[4,5,6]);
INSERT 0 1
```

数据如下所示：

```
mydb=> SELECT * FROM test_array3;
      id
-----
 {1,2,3}
 {4,5,6}
(2 rows)
```

使用 array_agg 函数，如下所示：

```
mydb=> SELECT array_agg(id) FROM test_array3;
      array_agg
-----
 {{1,2,3},{4,5,6}}
(1 row)
```

也可以将 array_agg 函数输出类型转换成字符串，并用指定分隔符分隔，使用 array_to_string 函数，如下所示：

```
mydb=> SELECT array_to_string( array_agg(id),',') FROM test_array3;
      array_to_string
-----
 1,2,3,4,5,6
(1 row)
```

4.7 窗口函数

上一节介绍了聚合函数，聚合函数将结果集进行计算并且通常返回一行。窗口函数也是基于结果集进行计算，与聚合函数不同的是窗口函数不会将结果集进行分组计算并输出一行，而是将计算出的结果合并到输出的结果集上，并返回多行。使用窗口函数能大幅简化 SQL 代码。

4.7.1 窗口函数语法

PostgreSQL 提供内置的窗口函数，例如 row_num()、rank()、lag() 等，除了内置的窗口函数外，聚合函数、自定义函数后接 OVER 属性也可作为窗口函数。

窗口函数的调用语法稍复杂，如下所示：

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ]
    OVER ( window_definition )
```

其中 window_definition 语句如下：

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

说明如下：

- OVER 表示窗口函数的关键字。
- PARTITION BY 属性对查询返回的结果集进行分组，之后窗口函数处理分组的数据。
- ORDER BY 属性设定结果集的分组数据排序。

后续小节将介绍常用窗口函数的使用。

4.7.2 avg() OVER()

聚合函数后接 OVER 属性的窗口函数表示在一个查询结果集上应用聚合函数，本节将演示 avg() 聚合函数后接 OVER 属性的窗口函数，此窗口函数用来计算分组后数据的平均值。

创建一张成绩表并插入测试数据，如下所示：

```
CREATE TABLE score (
    id serial primary key,
    subject character varying(32),
    stu_name character varying(32),
    score numeric(3,0) );
INSERT INTO score ( subject,stu_name,score ) VALUES ('Chinese','francs',70);
INSERT INTO score ( subject,stu_name,score ) VALUES ('Chinese','matiler',70);
INSERT INTO score ( subject,stu_name,score) VALUES ('Chinese','tutu',80);
INSERT INTO score ( subject,stu_name,score ) VALUES ('English','matiler',75);
INSERT INTO score ( subject,stu_name,score ) VALUES ('English','francs',90);
INSERT INTO score ( subject,stu_name,score ) VALUES ('English','tutu',60);
INSERT INTO score ( subject,stu_name,score ) VALUES ('Math','francs',80);
INSERT INTO score ( subject,stu_name,score ) VALUES ('Math','matiler',99);
INSERT INTO score ( subject,stu_name,score ) VALUES ('Math','tutu',65);
```

查询每名学生学习成绩并且显示课程的平均分，通常是先计算出课程的平均分，然后用 score 表与平均分表关联查询，如下所示：

```
mydb=> SELECT s.subject, s.stu_name,s.score, tmp.avgscore
  FROM score s
  LEFT JOIN (SELECT subject, avg(score) avgscore FROM score GROUP BY subject) tmp
    ON s.subject = tmp.subject;
      subject | stu_name | score |      avgscore
```

```
-----+-----+-----+
Chinese | francs   |    70 | 73.333333333333333333
Chinese | matiler  |    70 | 73.333333333333333333
Chinese | tutu     |    80 | 73.333333333333333333
English | matiler  |    75 | 75.000000000000000000
English | francs   |    90 | 75.000000000000000000
English | tutu     |    60 | 75.000000000000000000
Math   | francs   |    80 | 81.333333333333333333
Math   | matiler  |    99 | 81.333333333333333333
Math   | tutu     |    65 | 81.333333333333333333
(9 rows)
```

使用窗口函数很容易实现以上需求，如下所示：

```
mydb=> SELECT subject,stu_name, score, avg(score) OVER(PARTITION BY subject) FROM score;
subject | stu_name | score |      avg
-----+-----+-----+
Chinese | francs   |    70 | 73.333333333333333333
Chinese | matiler  |    70 | 73.333333333333333333
Chinese | tutu     |    80 | 73.333333333333333333
English | matiler  |    75 | 75.000000000000000000
English | francs   |    90 | 75.000000000000000000
English | tutu     |    60 | 75.000000000000000000
Math   | francs   |    80 | 81.333333333333333333
Math   | matiler  |    99 | 81.333333333333333333
Math   | tutu     |    65 | 81.333333333333333333
(9 rows)
```

以上查询前三列来源于表 score，第四列表示取课程的平均分，PARTITION BY subject 表示根据字段 subject 进行分组。

4.7.3 row_number()

row_number() 窗口函数对结果集分组后的数据标注行号，从 1 开始，如下所示：

```
mydb=> SELECT row_number() OVER (partition by subject ORDER BY score desc),* FROM score;
row_number| id | subject | stu_name | score
-----+-----+-----+-----+
      1 |  3 | Chinese | tutu     |    80
      2 |  1 | Chinese | francs   |    70
      3 |  2 | Chinese | matiler  |    70
      1 |  5 | English | francs   |    90
      2 |  4 | English | matiler  |    75
      3 |  6 | English | tutu     |    60
      1 |  8 | Math    | matiler  |    99
      2 |  7 | Math    | francs   |    80
      3 |  9 | Math    | tutu     |    65
(9 rows)
```

以上 row_number() 窗口函数显示的是分组后记录的行号，如果不指定 partition 属性，row_number() 窗口函数显示表所有记录的行号，类似 oracle 里的 ROWNUM，如下所示：



```
mydb=> SELECT row_number() OVER (ORDER BY id) AS rounum,* FROM score;
      rounum | id | subject | stu_name | score
-----+-----+-----+-----+
      1 | 1 | Chinese | francs | 70
      2 | 2 | Chinese | matiler | 70
      3 | 3 | Chinese | tutu | 80
      4 | 4 | English | matiler | 75
      5 | 5 | English | francs | 90
      6 | 6 | English | tutu | 60
      7 | 7 | Math | francs | 80
      8 | 8 | Math | matiler | 99
      9 | 9 | Math | tutu | 65
(9 rows)
```

4.7.4 rank()

rank() 窗口函数和 row_number() 窗口函数相似，主要区别为当组内某行字段值相同时，行号重复并且行号产生间隙（手册上解释为 gaps），如下所示：

```
mydb=> SELECT rank() OVER(PARTITION BY subject ORDER BY score),* FROM score;
      rank | id | subject | stu_name | score
-----+-----+-----+-----+
      1 | 2 | Chinese | matiler | 70
      1 | 1 | Chinese | francs | 70
      3 | 3 | Chinese | tutu | 80
      1 | 6 | English | tutu | 60
      2 | 4 | English | matiler | 75
      3 | 5 | English | francs | 90
      1 | 9 | Math | tutu | 65
      2 | 7 | Math | francs | 80
      3 | 8 | Math | matiler | 99
(9 rows)
```

以上示例中，Chinese 课程前两条记录的 score 字段值都为 70，因此前两行的 rank 字段值为 1，而第三行的 rank 字段值为 3，产生了间隙。

4.7.5 dense_rank ()

dense_rank () 窗口函数和 rank () 窗口函数相似，主要区别为当组内某行字段值相同时，虽然行号重复，但行号不产生间隙，如下所示：

```
mydb=> SELECT dense_rank() OVER(PARTITION BY subject ORDER BY score),* FROM score;
      dense_rank | id | subject | stu_name | score
-----+-----+-----+-----+
      1 | 2 | Chinese | matiler | 70
      1 | 1 | Chinese | francs | 70
      2 | 3 | Chinese | tutu | 80
      1 | 6 | English | tutu | 60
      2 | 4 | English | matiler | 75
      3 | 5 | English | francs | 90
(9 rows)
```



```
1 | 9 | Math      | tutu      | 65
2 | 7 | Math      | francs    | 80
3 | 8 | Math      | matiler   | 99
(9 rows)
```

以上示例中，Chinese 课程前两行的 rank 字段值 1，而第三行的 rank 字段值为 2，没有产生间隙。

4.7.6 lag()

另一重要窗口函数为 lag()，可以获取行偏移 offset 那行某个字段的数据，语法如下：

```
lag(value anyelement [, offset integer [, default anyelement ]])
```

其中：

- value 指定要返回记录的字段。
- offset 指行偏移量，可以是正整数或负整数，正整数表示取结果集中向上偏移的记录，负整数表示取结果集中向下偏移的记录，默认值为 1。
- default 是指如果不存在 offset 偏移的行时用默认值填充，default 值默认为 null。

例如，查询 score 表并获取向上偏移一行记录的 id 值，如下所示：

```
mydb=> SELECT lag(id,1)OVER(),* FROM score;
 lag | id | subject | stu_name | score
-----+---+-----+-----+-----+
     | 1 | Chinese | francs  | 70
 1 | 2 | Chinese | matiler | 70
 2 | 3 | Chinese | tutu    | 80
 3 | 4 | English | matiler | 75
 4 | 5 | English | francs  | 90
 5 | 6 | English | tutu    | 60
 6 | 7 | Math    | francs  | 80
 7 | 8 | Math    | matiler | 99
 8 | 9 | Math    | tutu    | 65
(9 rows)
```

查询 score 表并获取向上偏移两行记录的 id 值，并指定默认值，代码如下所示：

```
mydb=> SELECT lag(id,2,1000)OVER(),* FROM score;
 lag | id | subject | stu_name | score
-----+---+-----+-----+-----+
 1000 | 1 | Chinese | francs  | 70
 1000 | 2 | Chinese | matiler | 70
     | 3 | Chinese | tutu    | 80
 2 | 4 | English | matiler | 75
 3 | 5 | English | francs  | 90
 4 | 6 | English | tutu    | 60
 5 | 7 | Math    | francs  | 80
 6 | 8 | Math    | matiler | 99
 7 | 9 | Math    | tutu    | 65
(9 rows)
```



以上演示了 lag() 窗口函数取向上偏移记录的字段值，将 offset 设置成负整数可以取向下偏移记录的字段值，有兴趣的读者自行测试。

4.7.7 first_value()

first_value() 窗口函数用来取结果集每一个分组的第一行数据的字段值。

例如 score 表按课程分组后取分组的第一行的分数，如下所示：

```
mydb=> SELECT first_value(score) OVER( PARTITION BY subject ),* FROM score;
      first_value | id | subject | stu_name | score
-----+-----+-----+-----+-----+
        70 |  1 | Chinese | francs   |    70
        70 |  2 | Chinese | matiler  |    70
        70 |  3 | Chinese | tutu     |    80
        75 |  4 | English | matiler  |    75
        75 |  5 | English | francs   |    90
        75 |  6 | English | tutu     |    60
        80 |  7 | Math    | francs   |    80
        80 |  8 | Math    | matiler  |    99
        80 |  9 | Math    | tutu     |    65
(9 rows)
```

通过 first_value() 窗口函数很容易查询分组数据的最大值或最小值，例如 score 表按课程分组同时取每门课程的最高分，如下所示：

```
mydb=> SELECT first_value(score) OVER( PARTITION BY subject ORDER BY score
      desc ),* FROM score;
      first_value | id | subject | stu_name | score
-----+-----+-----+-----+-----+
        80 |  3 | Chinese | tutu     |    80
        80 |  1 | Chinese | francs   |    70
        80 |  2 | Chinese | matiler  |    70
        90 |  5 | English | francs   |    90
        90 |  4 | English | matiler  |    75
        90 |  6 | English | tutu     |    60
        99 |  8 | Math    | matiler  |    99
        99 |  7 | Math    | francs   |    80
        99 |  9 | Math    | tutu     |    65
(9 rows)
```

4.7.8 last_value()

last_value() 窗口函数用来取结果集每一个分组的最后一行数据的字段值。

例如 score 表按课程分组后取分组的最后一行的分数，如下所示：

```
mydb=> SELECT last_value(score) OVER( PARTITION BY subject ),* FROM score;
      last_value | id | subject | stu_name | score
-----+-----+-----+-----+-----+
        80 |  1 | Chinese | francs   |    70
```



```
80 | 2 | Chinese | matiler | 70
80 | 3 | Chinese | tutu | 80
60 | 4 | English | matiler | 75
60 | 5 | English | francs | 90
60 | 6 | English | tutu | 60
65 | 7 | Math | francs | 80
65 | 8 | Math | matiler | 99
65 | 9 | Math | tutu | 65
(9 rows)
```

4.7.9 nth_value()

nth_value() 窗口函数用来取结果集每一个分组的指定行数据的字段值，语法如下所示：

```
nth_value(value any, nth integer)
```

其中：

- value 指定表的字段。
- nth 指定结果集分组数据中的第几行，如果不存在则返回空。

例如 score 表按课程分组后取分组的第二行的分数，如下所示：

```
mydb=> SELECT nth_value(score,2) OVER( PARTITION BY subject ),* FROM score;
          nth_value | id | subject | stu_name | score
-----+-----+-----+-----+
    70 | 1 | Chinese | francs | 70
    70 | 2 | Chinese | matiler | 70
    70 | 3 | Chinese | tutu | 80
    90 | 4 | English | matiler | 75
    90 | 5 | English | francs | 90
    90 | 6 | English | tutu | 60
    99 | 7 | Math | francs | 80
    99 | 8 | Math | matiler | 99
    99 | 9 | Math | tutu | 65
(9 rows)
```

4.7.10 窗口函数别名的使用

如果 SQL 中需要多次使用窗口函数，可以使用窗口函数别名，语法如下：

```
SELECT .. FROM .. WINDOW window_name AS ( window_definition ) [, ...]
```

WINDOW 属性指定表的别名为 window_name，可以给 OVER 属性引用，如下所示：

```
mydb=> SELECT avg(score) OVER(r),sum(score) OVER(r),* FROM SCORE WINDOW r as (PARTITION
          BY subject);
          avg | sum | id | subject | stu_name | score
-----+-----+-----+-----+
  73.33333333333333 | 220 | 1 | Chinese | francs | 70
  73.33333333333333 | 220 | 2 | Chinese | matiler | 70
  73.33333333333333 | 220 | 3 | Chinese | tutu | 80
  75.00000000000000 | 225 | 4 | English | matiler | 75
```



```
75.0000000000000000 | 225 | 5 | English | francs | 90
75.0000000000000000 | 225 | 6 | English | tutu | 60
81.3333333333333333 | 244 | 7 | Math | francs | 80
81.3333333333333333 | 244 | 8 | Math | matiler | 99
81.3333333333333333 | 244 | 9 | Math | tutu | 65
(9 rows)
```

以上介绍了常用的窗口函数，读者可根据实际应用场景使用相应的窗口函数。

4.8 本章小结

本章介绍了 PostgreSQL 支持的一些高级 SQL 特性，了解这些功能可简化 SQL 代码，提升开发效率，并且实现普通查询不容易实现的功能，希望通过阅读本章读者能够在实际工作中应用 SQL 高级特性，同时挖掘 PostgreSQL 的其他高级 SQL 特性。