决定有用的扩展

在第10章 "理解备份和复制 "中，我们的重点是复制、交易日志运输和逻辑解码。在看了大部分与管理有关的主题后，现在的目标是瞄准一个更广泛的主题。在PostgreSQL的世界里，许多事情是通过扩展来完成的。 扩展的好处是，可以在不影响PostgreSQL核心的情况下增加功能。用户可以从相互竞争的扩展中进行选择，找到最适合他们的东西。其理念是保持核心的纤细，相对容易维护，并为未来做好准备。

在这一章中，将讨论一些最常用的PostgreSQL的扩展。然而，在深入探讨这个问题之前，我想说明的是，本章只介绍了我个人认为有用的扩展列表。现在有这么多的模块，不可能以合理的方式将它们全部涵盖。每天都有信息发布，有时对于一个专业人士来说，甚至很难了解所有的信息。新的扩展正在发布，看看PostgreSQL扩展网络（PGXN）（https://pgxn.org/）可能是一个好主意，它包含了大量PostgreSQL的扩展。

在这一章中，我们将介绍以下内容。

- 了解扩展是如何工作的
- 利用contrib模块
- 其他有用的扩展

请注意，只有最重要的扩展将被涵盖。

1. 了解扩展是如何工作的

在深入研究现有的扩展设备之前，首先看一看扩展设备是如何工作的，这是一个好主意。了解扩展机制的内部运作可能是相当有益的。

我们先来看看语法：

```
test=# \h CREATE EXTENSION
Command: CREATE EXTENSION
Description: install an extension
Syntax:
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
 [ WITH ] [ SCHEMA schema_name ]
 [ VERSION version ]
 [ CASCADE ]
URL: https://www.postgresql.org/docs/13/sql-createextension.html
```

当你想部署一个扩展时，只需调用CREATE EXTENSION命令。它将检查该扩展并将其加载到你的数据库中。注意，扩展将被加载到数据库中，而不是整个数据库实例中。

如果我们正在加载一个扩展，我们可以决定我们要使用的模式。许多扩展可以被重新定位，以便用户可以选择使用哪种模式。然后，可以决定扩展的特定版本。通常情况下，我们不想部署最新版本的扩展，因为客户可能正在运行过时的软件。在这种情况下，能够部署系统上可用的任何版本可能是很方便的。

FROM old_version子句需要多注意一些。在过去，PostgreSQL并不支持扩展，所以很多未打包的代码仍然存在。这个选项使CREATE EXTENSION子句运行一个替代的安装脚本，将现有的对象吸收到扩展中，而不是创建新的对象。请确保SCHEMA子句指定了包含这些预先存在的对象的模式。只有当你周围有旧模块时才使用它。

最后，还有CASCADE条款。一些扩展依赖于其他扩展。CASCADE选项也会自动部署这些软件包。下面是一个例子。

```
test=# CREATE EXTENSION earthdistance;
ERROR: required extension "cube" is not installed
HINT: Use CREATE EXTENSION ... CASCADE to install required extensions too.
```

earthdistance模块实现了大圆圈距离的计算。你可能已经知道，地球上两点之间的最短距离不能以直线方式进行；相反，当从一个点飞到另一个点时，飞行员必须不断调整他们的航线以找到最快的路线。问题是，地球距离扩展依赖于立方体扩展，它允许你在球体上进行操作。

为了自动部署这种依赖关系，可以使用CASCADE子句，正如我们之前描述的那样。

```
test=# CREATE EXTENSION earthdistance CASCADE;
NOTICE: installing required extension "cube"
CREATE EXTENSION
```

在这种情况下，如NOTICE消息所示，两个扩展都将被部署。在下一节中，我们将弄清楚系统提供的是哪个扩展。

1.1 检查可用的扩展

PostgreSQL提供了各种视图，以便我们可以弄清楚哪些扩展在系统上，哪些扩展被实际部署。其中一个视图是pg_available_extensions。

```
test=# \d pg_available_extensions
 View "pg_catalog.pg_available_extensions"
 Column | Type | Collation | Nullable | Default
------------------+------+-----------+----------+---------
 name | name | | |
 default_version | text | | |
 installed_version | text | C | |
 comment | text | | |
```

这包含一个所有可用扩展的列表，包括它们的名称、默认版本和当前安装的版本。为了使终端用户更方便，还有一个描述，告诉我们更多关于扩展的信息

下面的列表包含了从pg_available_extensions中抽取的两行。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_available_extensions LIMIT 2;
-[ RECORD 1 ]-----+--------------------------------------------------
name | unaccent
default_version | 1.1
installed_version |
comment | text search dictionary that removes accents
-[ RECORD 2 ]-----+--------------------------------------------------
name | tsm_system_time
default_version | 1.0
installed_version |
comment | TABLESAMPLE method which accepts time in milliseconds as a
limit
```

正如你在前面的代码块中看到的，在我的数据库中，earthdistance和plpgsql扩展都是启用的。 plpgsql扩展是默认存在的，earthdistance也被添加了。这个视图的好处是，你可以迅速了解哪些东西已经安装，哪些可以安装。

然而，在某些情况下，扩展的版本不止一个。要了解更多关于版本划分的信息，请查看以下试图。

```
test=# \d pg_available_extension_versions
View "pg_catalog.pg_available_extension_versions"
 Column | Type | Modifiers
--------------+---------+-----------
 name | name |
 version | text |
 installed | boolean |
 superuser | boolean |
 trusted | boolean |
 relocatable | boolean |
 schema | name |
 requires | name[] |
 comment | text |
```

系统视图显示了您需要了解的有关扩展的所有信息。

这里有一些更详细的信息，如以下列表所示。

```
test=# SELECT * FROM pg_available_extension_versions LIMIT 1;
-[ RECORD 1 ]-----------------------------------------------------------
name | isn
version | 1.1
installed | f
superuser | t
trusted | t
relocatable | t
schema |
requires |
comment | data types for international product numbering standards
```

PostgreSQL也会告诉你这个扩展是否可以被重新定位，它被部署在哪个模式中，以及需要什么其他的扩展。然后，是描述扩展的注释，如前所示。

你可能想知道PostgreSQL在哪里找到所有这些关于系统中扩展的信息。假设你已经从官方的PostgreSQL RPM存储库中部署了PostgreSQL 10.0，/usr/pgsql-13/share/extension目录将包含几个文件。

```
...
-bash-4.3$ ls -l citext*
ls -l citext*
-rw-r--r--. 1 hs hs 1028 Sep 11 19:53 citext--1.0--1.1.sql
-rw-r--r--. 1 hs hs 2748 Sep 11 19:53 citext--1.1--1.2.sql
-rw-r--r--. 1 hs hs 307 Sep 11 19:53 citext--1.2--1.3.sql
-rw-r--r--. 1 hs hs 668 Sep 11 19:53 citext--1.3--1.4.sql
-rw-r--r--. 1 hs hs 2284 Sep 11 19:53 citext--1.4--1.5.sql
-rw-r--r--. 1 hs hs 13466 Sep 11 19:53 citext--1.4.sql
-rw-r--r--. 1 hs hs 158 Sep 11 19:53 citext.control
-rw-r--r--. 1 hs hs 9781 Sep 11 19:53 citext--unpackaged--1.0.sql
...
```

citext（不区分大小写的文本）扩展的默认版本是1.4，所以有一个文件叫citext--1.3.sql。除此之外，还有一些文件用于从一个版本转移到下一个版本（1.0→1.1，1.1→1.2，以此类推）。

然后是 .control 文件：

```
-bash-4.3$ cat citext.control
# citext extension
comment = 'data type for case-insensitive character strings'
default_version = '1.4'
module_pathname = '$libdir/citext'
relocatable = true
```

这个文件包含与扩展名相关的所有元数据；第一个条目包含注释。注意，这些内容就是我们刚才讨论的系统视图中要显示的内容。当你访问这些视图时，PostgreSQL将进入这个目录并读取所有的.control文件。然后，是默认版本和二进制文件的路径。

如果你从RPM安装一个典型的扩展，目录将是$libdir，它在你的PostgreSQL二进制目录内。然而，如果你已经写了你自己的商业扩展，它很可能位于其他地方

最后一项设置将告诉PostgreSQL，该扩展是否可以驻留在任何模式中，或者是否必须在一个固定的、预定义的模式中。

最后，是解压后的文件。下面是它的一个摘录。

```
...
ALTER EXTENSION citext ADD type citext;
ALTER EXTENSION citext ADD function citextin(cstring);
ALTER EXTENSION citext ADD function citextout(citext);
ALTER EXTENSION citext ADD function citextrecv(internal);
...
```

解除包装的文件将把任何现有的代码变成一个扩展。 因此，在你的数据库中整合现有的代码是很重要的。 在这个关于一般扩展的基本介绍之后，我们现在将研究一些额外的扩展。

2.利用contrib模块

现在我们已经看了关于扩展的理论介绍，现在是时候看一下一些最重要的扩展了。在这一节中，你将了解到作为PostgreSQL contrib模块的一部分提供给你的那些模块。当你安装PostgreSQL时，我建议你总是安装这些contrib模块，因为它们包含重要的扩展，可以真正使你的生活更容易。

在接下来的章节中，我们将引导你了解一些我认为最有趣的、对各种原因最有用的扩展（用于调试、性能调整，等等）。

2.1 使用 adminpack 模块

adminpack模块背后的想法是给管理员提供一种无需SSH访问的方法来访问文件系统。该包包含几个功能来实现这一点。

要将模块加载到数据库中，请运行以下命令：

```
test=# CREATE EXTENSION adminpack;
CREATE EXTENSION
```

adminpack模块最有趣的功能之一是检查日志文件的能力。pg_logdir_ls函数检查了日志目录并返回一个日志文件列表。

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
ERROR: the log_filename parameter must equal 'postgresql-%Y-%m-%d_%H%M%S.log'
```

这里重要的是，log_filename参数必须根据adminpack模块的需要进行调整。如果你碰巧运行从PostgreSQL仓库下载的RPM，log_filename参数被定义为postgresql-%a，这必须被改变，以便避免错误。

进行此更改后，将返回日志文件名列表：

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
 a | b
---------------------+---------------------------------------
 2020-09-15 08:13:30 | log/postgresql-2020-09-15_081330.log
 2020-09-15 08:13:23 | log/postgresql-2020-09-15_081323.log
 2020-09-15 08:13:21 | log/postgresql-2020-09-15_081321.log
(3 rows)
```

除了这些功能外，还有一些进一步的功能是由该模块提供的。

```
test=# SELECT proname FROM pg_proc WHERE proname ~ 'pg_file_.*';
 proname
----------------
 pg_file_write
 pg_file_rename
 pg_file_unlink
 pg_file_read
 pg_file_length
(5 rows)
```

在这里，您可以读取、写入、重命名或简单地删除文件。

当然，这些函数只能由超级用户调用。

2.2 应用布隆过滤器

从PostgreSQL 9.6开始，就可以使用扩展来即时添加索引类型。新的CREATE ACCESS METHOD命令，以及一些额外的功能，使我们有可能在飞行中创建功能齐全和有事务日志的索引类型。
bloom扩展为PostgreSQL用户提供了bloom过滤器，即 前置过滤器，帮助我们尽快有效地减少数据量。布隆过滤器背后的想法是，我们可以计算一个位掩码，并将该位掩码与查询进行比较。布隆过滤器可能会产生一些误报，但它仍然会大大减少数据量。
当一个表由几百个列和几百万行组成时，这一点特别有用。数以百万计的行。用Btrees来索引几百个列是不可能的，所以Bloom filter是一个很好的选择，因为它可以让我们一次性索引 一次性索引所有内容

为了了解事情是如何运作的，我们将安装该扩展。

```
test=# CREATE EXTENSION bloom;
CREATE EXTENSION
```

现在，我们需要创建一个包含各种列的表：

```
test=# CREATE TABLE t_bloom
(
 id serial,
 col1 int4 DEFAULT random() * 1000,
 col2 int4 DEFAULT random() * 1000,
 col3 int4 DEFAULT random() * 1000,
 col4 int4 DEFAULT random() * 1000,
 col5 int4 DEFAULT random() * 1000,
 col6 int4 DEFAULT random() * 1000,
 col7 int4 DEFAULT random() * 1000,
 col8 int4 DEFAULT random() * 1000,
 col9 int4 DEFAULT random() * 1000
);
CREATE TABLE
```

为了使之更容易，这些列有一个默认值，这样就可以使用一个简单的SELECT子句轻松地添加数据。

```
test=# INSERT INTO t_bloom (id)
 SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
```

前面的查询向表中添加了 100 万行。现在，该表可以被索引：

```
test=# CREATE INDEX idx_bloom ON t_bloom
 USING bloom(col1, col2, col3, col4, col5, col6, col7, col8, col9);
CREATE INDEX
```

请注意，该索引一次包含九列。与B树相比，这些列的顺序其实并没有什么区别。

> 请注意，我们刚刚创建的表在没有索引的情况下大约是65MB。

该索引又增加了 15 MB 的存储空间：

```
test=# \di+ idx_bloom
 List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
--------+-----------+-------+-------+---------+-------------+-------+------------
-
 public | idx_bloom | index | hs | t_bloom | permanent | 15 MB |
(1 row)
```

布隆过滤器的魅力在于它可以寻找任何列的组合。

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
test=# explain SELECT count(*)
 FROM t_bloom
 WHERE col4 = 454 AND col3 = 354 AND col9 = 423;
 QUERY PLAN
------------------------------------------------------------------------------
 Aggregate (cost=20352.02..20352.03 rows=1 width=8)
 -> Bitmap Heap Scan on t_bloom
 (cost=20348.00..20352.02 rows=1 width=0)
 Recheck Cond: ((col3 = 354) AND (col4 = 454) AND (col9 = 423))
 -> Bitmap Index Scan on idx_bloom
 (cost=0.00..20348.00 rows=1 width=0)
 Index Cond: ((col3 = 354) AND (col4 = 454)
 AND (col9 = 423))
(5 rows)
```

到目前为止，你所看到的感觉都很特别。一个可能出现的自然问题是：为什么不总是使用布隆过滤器？原因很简单--为了使用它，数据库必须读取整个布隆过滤器。在B树的情况下，这是不需要的。

在未来，更多的索引类型可能会被添加，以确保PostgreSQL可以覆盖更多的用例。

> 如果你想了解更多关于bloom过滤器的信息，可以考虑阅读我们的博文：https://www.cybertec-postgresql.com/en/trying-out-postgres-bloom-indexes/。

2.3 部署 btree_gist 和 btree_gin

甚至还有更多与索引有关的功能可以被添加。在PostgreSQL中，有一个运算符类的概念，我们在第3章"利用索引 "中讨论过这个概念。

contrib模块提供了两个扩展（即btree_gist和btree_gin），这样我们就可以为GiST和GIN索引添加B-tree功能。为什么会有这样的作用呢？GiST索引提供了B树所不支持的各种功能。其中一个功能是能够进行K-近邻（KNN）搜索。

为什么会有这种关系？想象一下，有人正在寻找昨天中午左右添加的数据。那么，那是什么时候？在某些情况下，可能很难想出边界，例如，如果有人在寻找一个价格在70欧元左右的产品。在这里，KNN可以起到拯救的作用。这里有一个例子。

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
```

需要添加一些简单的数据：

```
test=# INSERT INTO t_test SELECT * FROM generate_series(1, 100000);
INSERT 0 100000
```

现在，可以添加扩展：

```
test=# CREATE EXTENSION btree_gist;
CREATE EXTENSION
```

为该列添加gist索引很容易，只要使用USING gist子句即可。 注意，只有在扩展名存在的情况下，给整数列添加gist索引才有效。否则，PostgreSQL会报告说没有合适的操作符类。

```
test=# CREATE INDEX idx_id ON t_test USING gist(id);
CREATE INDEX
```

一旦部署了索引，就有可能按距离排序。

```
test=# SELECT *
 FROM t_test
 ORDER BY id <-> 100
 LIMIT 6;
 id
-----
 100
 101
 99
 102
 98
 97
(6 rows)
```

正如你所看到的，第一行是一个完全匹配。接下来的匹配已经不太精确了，而且越来越差。该查询将始终返回固定数量的行。

这里重要的是执行计划：

```
test=# explain SELECT *
 FROM t_test
 ORDER BY id <-> 100
 LIMIT 6;
 QUERY PLAN
-------------------------------------------------------------
 Limit (cost=0.28..0.64 rows=6 width=8)
 -> Index Only Scan using idx_id on t_test
 (cost=0.28..5968.28 rows=100000 width=8)
 Order By: (id <-> 100)
(3 rows)
```

正如你所看到的，PostgreSQL直接进行了索引扫描，这大大加快了查询的速度。

在未来的PostgreSQL版本中，B-trees很可能也会支持KNN搜索。一个添加这个功能的补丁已经被添加到开发邮件列表中。也许它最终会进入核心版本。 将KNN作为B-tree的一个功能，最终会导致标准数据类型上的GiST索引减少。

2.4 dblink——考虑逐步淘汰

使用数据库链接的愿望已经存在了很多年。然而，在世纪之交，PostgreSQL的外来数据封装器甚至还没有出现，传统的数据库链接实现也绝对没有出现。大约在这个时候，一位来自加利福尼亚的PostgreSQL开发者（Joe Conway）将dblink的概念引入到PostgreSQL中，从而开创了数据库连接的工作。虽然多年来dblink为人们提供了很好的服务，但它已经不再是最先进的了。

因此，建议我们从dblink转向更现代的SQL/MED实现（这是一个定义了外部数据与关系型数据库集成方式的规范）。 postgres_fdw扩展是建立在SQL/MED之上的，它提供的不仅仅是数据库连接，因为它允许你连接到基本上任何数据源。

2.5 使用 file_fdw 获取文件

在某些情况下，从磁盘上读取一个文件并将其作为一个表暴露给PostgreSQL是有意义的。这正是你可以通过file_fdw扩展实现的。我们的想法是，有一个模块允许你从磁盘上读取数据，并使用SQL进行查询。

安装该模块的工作原理与预期一致

```
CREATE EXTENSION file_fdw;
```

现在，我们需要创建一个虚拟服务器：

```
CREATE SERVER file_server  FOREIGN DATA WRAPPER file_fdw;
```

file_server是基于file_fdw扩展的外来数据包装器，它告诉PostgreSQL如何访问文件。

要把一个文件暴露为一个表，使用下面的命令。

```
CREATE FOREIGN TABLE t_passwd
(
 username text,
 passwd text,
 uid int,
 gid int,
 gecos text,
 dir text,
 shell text
) SERVER file_server
OPTIONS (format 'text', filename '/etc/passwd', header 'false', delimiter ':');
```

在这个例子中，/etc/passwd文件将被暴露。所有的字段都要被列出，数据类型也要相应地被映射。所有额外的重要信息都用OPTIONS传递给模块。在这个例子中，PostgreSQL必须知道文件的类型（文本），文件的名称和路径，以及分隔符。也可以告诉PostgreSQL是否有一个头文件。如果设置为真，第一行将被跳过，并被认为是不重要的。如果你碰巧加载了一个CSV文件，跳过标题是特别重要的。

一旦表被创建，就可以读取数据了。

```
SELECT * FROM t_passwd;
```

不出所料，PostgreSQL 返回 /etc/passwd 的内容：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM t_passwd LIMIT 1;
-[ RECORD 1 ]-------
username | root
passwd | x
uid | 0
gid | 0
gecos | root
dir | /root
shell | /bin/bash
```

当看执行计划时，你会看到PostgreSQL使用所谓的外来扫描来从文件中获取数据。

```
test=# explain (verbose true, analyze true) SELECT * FROM t_passwd;
 QUERY PLAN
------------------------------------------------------------------------
 Foreign Scan on public.t_passwd (cost=0.00..2.80 rows=18 width=168)
 (actual time=0.022..0.072 rows=61 loops=1)
 Output: username, passwd, uid, gid, gecos, dir, shell
 Foreign File: /etc/passwd
 Foreign File Size: 3484
 Planning time: 0.058 ms
 Execution time: 0.138 ms
(6 rows)
```

执行计划还告诉我们文件的大小等等。既然我们在谈论计划器，有一个侧面值得一提。PostgreSQL甚至会获取文件的统计数据。规划器检查文件大小并将相同的成本分配给文件，就像分配给相同大小的普通 PostgreSQL 表一样

2.6 使用 pageinspect 检查存储

如果你正面临存储损坏或其他一些与存储有关的问题，可能与表中的坏块有关，pageinspect扩展可能是你正在寻找的模块。我们将从创建扩展开始，如下面的例子所示。

```
test=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

pageinspect背后的想法是为你提供一个模块，允许你在二进制层面上检查一个表。

当使用这个模块时，最重要的事情是获取一个块。

```
test=# SELECT * FROM get_raw_page('pg_class', 0);
...
```

这个函数将返回一个单一的块。在前面的例子中，它是pg_class参数中的第一个块，它是一个系统表。当然，用户可以自行选择任何其他的表。

接下来，你可以提取页头。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
-[ RECORD 1 ]--------
lsn | 0/8A8E310
checksum | 0
flags | 1
lower | 212
upper | 6880
special | 8192
pagesize | 8192
version | 4
prune_xid | 0
```

页头已经包含了很多关于该页的信息。如果你想了解更多，你可以调用heap_page_items函数，该函数对页面进行剖析，每一个元组返回一行。

```
test=# SELECT *
 FROM heap_page_items(get_raw_page('pg_class', 0))
 LIMIT 1;
-[ RECORD 1 ]---
lp | 1
lp_off | 49
lp_flags | 2
lp_len | 0
t_xmin |
t_xmax |
t_field3 |
t_ctid |
t_infomask2 |
t_infomask |
t_hoff |
t_bits |
t_oid |
t_data | ...
```

您还可以将数据拆分为各种元组：

```
test=# SELECT tuple_data_split('pg_class'::regclass,
 t_data, t_infomask, t_infomask2, t_bits)
 FROM heap_page_items(get_raw_page('pg_class', 0))
 LIMIT 2;
-[ RECORD 1 ]----+--------------------------------
tuple_data_split |
-[ RECORD 2 ]----+--------------------------------
tuple_data_split |
{"\\x610000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00
000000000000000000000000000000000000000000000000000000","\\x98080000","\\x50ac0c00"
,"
\\x00000000","\\x01400000","\\x00000000","\\x4eac0c00","\\x00000000","\\xbb01000
0"
,"\\x0050c347","\\x00000000","\\x00000000","\\x01","\\x00","\\x70","\\x72","\\x0
10
0","\\x0000","\\x00","\\x00","\\x00","\\x00","\\x00","\\x00","\\x01","\\
x6
4","\\xc3400900","\\x01000000",NULL,NULL}
```

为了读取数据，我们必须熟悉PostgreSQL的盘上格式。否则，数据可能会显得相当晦涩难懂。

pageinspect为所有可能的访问方法（表、索引等）提供了函数，并允许我们剖析存储，以便提供更多的细节。

2.7 使用 pg_buffercache 研究缓存

在简单介绍了pageinspect扩展之后，我们将把注意力转向pg_buffercache扩展，它允许你详细查看I/O缓存的内容。

```
test=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

pg_buffercache扩展为你提供了一个包含以下字段的视图。

```
test=# \d pg_buffercache
 View "public.pg_buffercache"
 Column | Type | Collation | Nullable | Default
-----------------+----------+-----------+----------+---------
 bufferid | integer | | |
 relfilenode | oid | | |
 reltablespace | oid | | |
 reldatabase | oid | | |
 relforknumber | smallint | | |
 relblocknumber | bigint | | |
 isdirty | boolean | | |
 usagecount | smallint | | |
 pinning_backends | integer | | |
```

bufferid字段只是一个数字；它标识了缓冲区。然后，还有relfilenode字段，它指向磁盘上的文件。如果我们想查询一个文件属于哪个表，我们可以查看pg_class模块，它也包含一个叫做relfilenode的字段。然后，还有reldatabase和reltablespace字段。注意，所有的字段都被定义为oid类型，所以为了以更有用的方式提取数据，有必要将系统表连接起来。

relforknumber字段告诉我们该表的哪一部分被缓存了。它可能是堆，自由空间图，或其他一些组件，如可见性图。在未来，肯定会有更多类型的关系分叉。 下一个字段，relblocknumber，告诉我们哪个块已经被缓存了。最后是isdirty标志，它告诉我们一个区块已经被修改了，以及关于使用计数器和钉住该区块的后端数量。

如果你想了解pg_buffercache扩展的意义，增加额外的信息很重要。要弄清楚哪个数据库使用缓存最多，下面的查询可能会有帮助。

```
test=# SELECT datname,
 count(*),
 count(*) FILTER (WHERE isdirty = true) AS dirty
 FROM pg_buffercache AS b, pg_database AS d
 WHERE d.oid = b.reldatabase
 GROUP BY ROLLUP (1);
 datname | count | dirty
-----------+-------+-------
 abc | 132 | 1
 postgres | 30 | 0
 test | 11975 | 53
 | 12137 | 54
(4 rows)
```

在这种情况下，pg_database扩展必须被连接。我们可以看到，oid是连接的标准，这对刚接触PostgreSQL的人来说可能不是很明显。

有时，我们可能想知道数据库中哪些与我们相连的块被缓存了。下面是它的工作原理。

```
test=# SELECT relname,
 relkind,
 count(*),
 count(*) FILTER (WHERE isdirty = true) AS dirty
 FROM pg_buffercache AS b, pg_database AS d, pg_class AS c
 WHERE d.oid = b.reldatabase
 AND c.relfilenode = b.relfilenode
 AND datname = 'test'
 GROUP BY 1, 2
 ORDER BY 3 DESC
 LIMIT 7;
 relname | relkind| count| dirty
--------------------------+--------+-------+-------
 t_bloom | r | 8338 | 0
 idx_bloom | i | 1962 | 0
 idx_id | i | 549 | 0
 t_test | r | 445 | 0
 pg_statistic | r | 90 | 0
 pg_depend | r | 60 | 0
 pg_depend_reference_index | i | 34 | 0
(7 rows)
```

在这种情况下，我们过滤了当前的数据库，并与包含对象列表的pg_class模块连接。relkind列特别值得注意：r指的是表（关系），i指的是索引。这告诉我们，我们正在看哪个对象。

2.8 使用 pgcrypto 加密数据

在整个contrib模块部分，最强大的模块之一是pgcrypto。它最初是由Skype的一个系统管理员编写的，提供了无数的功能，使我们能够加密和解密数据。

它提供了对称和非对称加密的功能。由于提供了大量的功能，绝对建议查看文档页面：https://www.postgresql.org/docs/current/static/pgcrypto.html。

由于本章的范围有限，我们不可能挖掘pgcrypto模块的所有细节。

2.9 使用 pg_prewarm 预热缓存

当PostgreSQL正常运行时，它试图对重要数据进行缓存。shared_buffers这个变量很重要，因为它定义了由PostgreSQL管理的缓存的大小。现在的问题是这样的：如果你重新启动数据库服务器，由PostgreSQL管理的缓存就会丢失。也许操作系统还有一些数据可以减少对磁盘等待的影响，但在很多情况下，这是不足够的。解决这个问题的方法叫做pg_prewarm扩展。现在让我们安装pg_prewarm，看看我们能用它做什么。

```
test=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
```

这个扩展部署了一个函数，允许我们在需要时明确地预热缓存。该列表显示了与该扩展相关的函数。

```
test=# \x
Expanded display is on.
test=# \df *prewa*
List of functions
-[ RECORD 1 ]
Schema | public
Name | autoprewarm_dump_now
Result data type | bigint
Argument data types |
Type | func
-[ RECORD 2 ]
Schema | public
Name | autoprewarm_start_worker
Result data type | void
Argument data types |
Type | func
-[ RECORD 3 ]
Schema | public
Name | pg_prewarm
Result data type | bigint
Argument data types | regclass, mode text DEFAULT 'buffer'::text,
 fork text DEFAULT 'main'::text,
 first_block bigint DEFAULT NULL::bigint,
 last_block bigint DEFAULT NULL::bigint
Type | func
```

调用 pg_prewarm 扩展最简单和最常见的方法是要求它缓存整个对象：

```
test=# SELECT pg_prewarm('t_test');
 pg_prewarm
------------
 443
(1 row)
```

> 请注意，如果表太大以至于无法放入缓存，则只有部分表会保留在缓存中，这在大多数情况下都很好。

该函数返回被该函数调用处理的8KB块的数量。 如果你不想缓存一个对象的所有块，你也可以在表中选择一个特定的范围。

在下面的例子中，我们可以看到10到30块被缓存在主分叉中。

```
test=# SELECT pg_prewarm('t_test', 'buffer', 'main', 10, 30);
 pg_prewarm
------------
 21
(1 row)
```

在这里，很明显缓存了 21 个块。

2.10 使用 pg_stat_statements 检查性能

pg_stat_statements是目前最重要的contrib模块。它应该总是被启用，并且是为了提供卓越的性能数据。 如果没有pg_stat_statements模块，真的很难追踪到性能问题。

2.11 使用 pgstattuple 检查存储

有时，PostgreSQL中的表可能会增长得不成比例。表增长过多的技术术语是表膨胀。现在出现的问题是：哪些表已经膨胀了，有多少膨胀了？pgstattuple扩展将帮助我们回答这些问题。

```
test=# CREATE EXTENSION pgstattuple;
CREATE EXTENSION
```

正如我们之前所说，该模块部署了几个函数。在pgstattuple扩展的情况下，这些函数返回一个由复合类型组成的行。因此，必须在FROM子句中调用该函数，以确保有一个可读的结果。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pgstattuple('t_test');
-[ RECORD 1 ]
--------------------+-------------
 table_len | 3629056
 tuple_count | 100000
 tuple_len | 2800000
 tuple_percent | 77.16
 dead_tuple_count | 0
 dead_tuple_len | 0
 dead_tuple_percent | 0
 free_space | 16652
 free_percent | 0.46
```

在这个例子中，用于测试的表似乎处于一个相当好的状态：表的大小为3.6MB，不包含任何死行。 自由空间也是有限的。如果对你的表的访问因表的膨胀而变慢，那么这意味着死行的数量和自由空间的数量将不成比例地增长。一些自由空间和少量的死行是正常的；但是，如果表已经增长到大部分由死行和自由空间组成，就需要采取果断的行动，使情况再次得到控制。

pgstattuple扩展也提供了一个函数，我们可以用它来检查索引。

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
```

pgstattindex函数返回了很多关于我们要检查的索引的信息。

```
test=# SELECT * FROM pgstatindex('idx_id');
-[ RECORD 1 ]
----------------------+---------------
version | 2
tree_level | 1
index_size | 2260992
root_block_no | 3
internal_pages | 1
leaf_pages | 274
empty_pages | 0
deleted_pages | 0
avg_leaf_density | 89.83
leaf_fragmentation | 0
```

我们的指数是相当密集的（89%）。这是一个好的迹象。索引的默认FILLFACTOR设置是90%，所以接近90%的数值表明索引非常好。

有时，你不想检查一个表；相反，你想检查所有的表，或者只检查一个模式中的所有表。如何实现这一点呢？通常情况下，你想处理的对象的列表在FROM子句中。然而，在我的例子中，函数已经在FROM子句中了，那么我们怎样才能使PostgreSQL在表的列表中循环呢？ 答案是使用一个LATERAL连接。

请记住，pgstattuple必须要读取整个对象。如果我们的数据库很大，它可能需要相当长的时间来处理。因此，存储我们刚刚看到的查询结果可能是一个好主意，这样我们就可以彻底检查它们，而不必一次又一次地重新运行该查询

2.12 使用 pg_trgm 进行模糊搜索

pg_trgm模块允许你进行模糊搜索。 这个模块在第三章 "利用索引 "中讨论过。

2.13 使用 postgres_fdw 连接到远程服务器

数据并不总是只在一个地方可用。更多的时候，数据分布在基础设施中，可能驻留在不同地方的数据必须被整合。

解决这个问题的方法就是SQL/MED标准所定义的外来数据包装器。

在这一节中，我们将讨论postgres_fdw扩展。它是一个允许我们从PostgreSQL数据源动态获取数据的模块。 我们需要做的第一件事是部署外部数据包装器。

```
test=# \h CREATE FOREIGN DATA WRAPPER
Command: CREATE FOREIGN DATA WRAPPER
Description: define a new foreign-data wrapper
Syntax:
CREATE FOREIGN DATA WRAPPER name
 [ HANDLER handler_function | NO HANDLER ]
 [ VALIDATOR validator_function | NO VALIDATOR ]
 [ OPTIONS ( option 'value' [, ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-createforeigndatawrapper.html
```

幸运的是，CREATE FOREIGN DATA WRAPPER命令隐藏在一个扩展中。它可以很容易地使用正常程序安装，如下所示。

```
test=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

现在，必须要定义一个虚拟服务器。它将指向另一台主机，并告诉PostgreSQL从哪里获得数据。在数据的最后，PostgreSQL必须建立一个完整的连接字符串--服务器数据是PostgreSQL首先要知道的东西。用户信息将在以后添加。 服务器将只包含主机、端口，等等。CREATE SERVER的语法如下

```
test=# \h CREATE SERVER
Command: CREATE SERVER
Description: define a new foreign server
Syntax:
CREATE SERVER [ IF NOT EXISTS ] server_name [ TYPE 'server_type' ] [ VERSION
'server_version' ]
 FOREIGN DATA WRAPPER fdw_name
 [ OPTIONS ( option 'value' [, ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-createserver.html
```

为了理解这一点，我们将在同一主机上创建第二个数据库，并创建一个服务器。

```
[hs@zenbook~]$ createdb customer
[hs@zenbook~]$ psql customer
customer=# CREATE TABLE t_customer (id int, name text);
CREATE TABLE
customer=# CREATE TABLE t_company (
 country text,
 name text,
 active text
);
CREATE TABLE
customer=# \d
List of relations
 Schema | Name | Type | Owner
----------+------------+--------+-------
 public | t_company | table |
 hs public | t_customer | table | hs
(2 rows)
```

现在，应该将服务器添加到标准测试数据库：

```
test=# CREATE SERVER customer_server
 FOREIGN DATA WRAPPER postgres_fdw
 OPTIONS (host 'localhost', dbname 'customer', port '5432');
CREATE SERVER
```

请注意，所有的重要信息都是以OPTIONS子句的形式存储的。这一点有些重要，因为它给了用户很大的灵活性。有许多不同的外部数据包装器，每个包装器都需要不同的选项。

一旦服务器被定义，就到了映射用户的时候了。如果我们从一个服务器连接到另一个服务器，我们在两个地方的用户可能不是同一个人。因此，外部数据封装器需要用户定义实际的用户映射，如下所示。

```
test=# \h CREATE USER MAPPING
Command: CREATE USER MAPPING
Description: define a new mapping of a user to a foreign server
Syntax:
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { user_name | USER | CURRENT_USER |
PUBLIC }
 SERVER server_name
 [ OPTIONS ( option 'value' [ , ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-createusermapping.html
```

语法非常简单，可以很容易地使用：

```
test=# CREATE USER MAPPING
 FOR CURRENT_USER SERVER customer_server
 OPTIONS (user 'hs', password 'abc');
CREATE USER MAPPING
```

同样，所有的重要信息都隐藏在OPTIONS子句中。 根据外部数据封装器的类型，选项列表会有所不同。请注意，我们必须在这里使用适当的用户数据，这对我们的设置是有效的。在这种情况下，我们将简单地使用本地用户。

一旦基础设施到位，我们就可以创建外域表了。创建外域表的语法与我们创建普通本地表的方法非常相似。所有的列都必须被列出，包括它们的数据类型。

```
test=# CREATE FOREIGN TABLE f_customer (id int, name text)
 SERVER customer_server
 OPTIONS (schema_name 'public', table_name 't_customer');
CREATE FOREIGN TABLE
```

所有的列都被列出，就像在普通的CREATE TABLE子句中的情况。这里的特殊之处在于，外域表指向了远程端的一个表。模式的名称和表的名称必须在OPTIONS子句中指定。

一旦它被创建，该表就可以被使用

```
test=# SELECT * FROM f_customer ;
 id | name
----+------
(0 rows)
```

要检查PostgreSQL内部做了什么，最好是运行带有分析参数的EXPLAIN子句。它将揭示一些关于服务器中真正发生的事情的信息。

```
test=# EXPLAIN (analyze true, verbose true)
 SELECT * FROM f_customer ;
 QUERY PLAN
------------------------------------------------------------------------------
 Foreign Scan on public.f_customer
 (cost=100.00..150.95 rows=1365 width=36)
 (actual time=0.221..0.221 rows=0 loops=1)
 Output: id, name
 Remote SQL: SELECT id, name FROM public.t_customer
 Planning time: 0.067 ms
 Execution time: 0.451 ms
(5 rows)
```

这里的重要部分是远程SQL。外部数据封装器将向对方发送查询，并尽可能少地获取数据，因为尽可能多的限制条件在远程端执行，以确保没有太多的数据被本地处理。过滤条件、连接、甚至聚合都可以在远程执行（从PostgreSQL 10.0开始）。

虽然CREATE FOREIGN TABLE子句肯定是个好东西，但反复列出所有这些列可能是相当麻烦的。

解决这个问题的方法是IMPORT子句。这使我们能够快速而方便地将整个模式导入本地数据库，并创建外域表。

```
test=# \h IMPORT
Command: IMPORT FOREIGN SCHEMA
Description: import table definitions from a foreign server
Syntax:
IMPORT FOREIGN SCHEMA remote_schema
 [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
 FROM SERVER server_name
 INTO local_schema
 [ OPTIONS ( option 'value' [, ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-importforeignschema.html
```

IMPORT使我们能够很容易地连接大型表格集。由于所有的信息都是直接从远程数据源获取的，所以它也减少了错别字和错误的机会。

下面是它的工作原理。

```
test=# IMPORT FOREIGN SCHEMA public
 FROM SERVER customer_server INTO public;
IMPORT FOREIGN SCHEMA
```

在这种情况下，之前在公共模式中创建的所有表都被直接链接。我们可以看到，所有的远程表现在都可以使用。

```
test=# \det
List of foreign tables
 Schema | Table | Server
---------+------------+-----------------
 public | f_customer | customer_server
 public | t_company | customer_server
 public | t_customer | customer_server
(3 rows)
```

\det列出了所有的外来表，如前面的代码所示。

2.13.1 处理错误和拼写错误

创建外域表其实并不难，然而，有时会发生人们犯错的情况，或者是已经使用的密码简单地改变了。为了处理这些问题，PostgreSQL提供了两个命令。ALTER SERVER和ALTER USER MAPPING。

ALTER SERVER允许你修改一个服务器。下面是它的语法

```
test=# \h ALTER SERVER
Command: ALTER SERVER
Description: change the definition of a foreign server
Syntax:
ALTER SERVER name [ VERSION 'new_version' ]
 [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
URL: https://www.postgresql.org/docs/13/sql-alterserver.html
```

我们可以使用这个命令来添加和删除特定服务器的选项，如果我们忘记了什么，这是件好事。

要修改用户信息，我们也可以改变用户的映射。

```
test=# \h ALTER USER MAPPING
Command: ALTER USER MAPPING
Description: change the definition of a user mapping
Syntax:
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER | PUBLIC
}
 SERVER server_name
 OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
URL: https://www.postgresql.org/docs/13/sql-alterusermapping.html
```

SQL/MED接口正在不断地被改进，在写这篇文章的时候，功能正在被添加。在未来，甚至更多的优化将进入核心，使SQL/MED接口成为提高可扩展性的良好选择。

到目前为止，你已经学会了如何使用外来数据包装器。现在让我们来看看一些更有用的扩展。

3.其他有用的扩展

到目前为止，我们所描述的扩展都是PostgreSQL contrib包的一部分，它是作为PostgreSQL源代码的一部分被运送的。 然而，我们在这里看到的包并不是PostgreSQL社区中唯一可用的包。还有很多包允许我们做各种各样的事情。

不幸的是，这一章太短了，无法深入研究目前存在的所有东西。模块的数量每天都在增长，不可能涵盖所有的模块。因此，我只想指出我认为最重要的那些。

PostGIS（http://postgis.net/）是开源世界中的地理信息系统（GIS）数据库接口。它已被全球采用，是关系型开源数据库世界的事实标准。它是一个专业和极其强大的解决方案。

如果你正在寻找地理空间路由，pgRouting就是你可能正在寻找的东西。它提供了各种算法，你可以用它来寻找地点之间的最佳连接，并在PostgreSQL的基础上工作。

在本章中，我们已经了解了postgres_fdw扩展，它允许我们连接到其他一些PostgreSQL数据库。周围还有许多外部数据包装器。其中最著名和最专业的是oracle_fdw扩展。它允许你与Oracle集成，并通过电线获取数据，这可以通过postgres_fdw扩展来完成。

在某些情况下，你也可能对用pg_crash（https://github.com/cybertec-postgresql/pg_crash）测试基础设施的稳定性感兴趣。这个想法是要有一个模块来不断地崩溃你的数据库。

pg_crash模块是测试和调试连接池的绝佳选择，它允许你重新连接到一个失败的数据库。 pg_crash模块会周期性地疯狂运行，杀死数据库会话或破坏内存。它是长期测试的理想选择。

4.总结

在这一章中，我们了解了一些最有前途的模块，这些模块是随PostgreSQL标准发布版一起提供的。这些模块相当多样化，提供了从数据库连接到区分大小写的文本和模块，以便我们可以检查服务器。然而，在本节中，你已经了解了周围最重要的模块。 这将帮助你部署更伟大的数据库设置

现在我们已经处理了扩展问题，在下一章，我们将把注意力转移到迁移上。在那里，我们将学习如何以最简单的方式迁移到PostgreSQL。