
Mastering PostgreSQL 11 2nd Edition.Pdf

精通 PostgreSQL 11 第二版

关于作者

Hans-JürgenSchönig 拥有 18 年的 PostgreSQL 经验。他是 PostgreSQL 咨询和支持公司 CybertecSchönig 和 SchönigGmbH 的首席执行官。它成功地为全球无数客户提供服务。在创立 Cybertec 之前 Schönig 和 SchönigGmbH 于 2000 年在一家专注于奥地利劳动力市场的私人研究公司担任数据库开发人员，主要从事数据挖掘和预测模型研究。他还写了几本关于 PostgreSQL 的书。

关于校验者

Sheldon Strauch 是一位拥有二十年软件咨询经验的资深人士，曾为 IBM, Sears, Ernst and Young 和 Kraft Foods 等公司提供咨询服务。他拥有工商管理学士学位，并利用其技术技能提高企业的自我意识。他的兴趣包括数据收集，管理和挖掘；地图和绘图；商业智能；和数据分析的应用，以持续改进。他目前专注于位于芝加哥的金融服务公司 Enova International 的端到端数据管理和采矿开发。在他的业余时间，他喜欢表演艺术，特别是音乐，和他的妻子玛丽莲一起旅行。

关于译者

网名：JavanChueng，10 年+DBA 从业者，擅长于 Oracle、Postgresql、MySQL 数据库，曾就职于亚信科技、南瑞等知名企业。

译者水平有限，书中错误在所难免，有问题可以联系译者：javanchueng@163.com

前言

第二版 Mastering PostgreSQL 11 使用最新版本的 PostgreSQL 帮助您为企业应用程序构建动态数据库解决方案，使数据库分析师能够轻松地设计系统架构的物理和技术方面。

本书首先介绍了 PostgreSQL 11 中最新发布的特性，以帮助您构建高效和容错的 PostgreSQL 应用程序。您将详细研究 PostgreSQL 的所有高级方面，包括逻辑复制、数据

库集群、性能调整、监视和用户管理。您还将使用 PostgreSQL 优化器，配置 PostgreSQL 以实现高速查询，并了解如何从 Oracle 迁移到 PostgreSQL。在本章中，您将介绍事务、锁定、索引和优化查询以提高性能。

此外，您将学习如何管理网络安全并探索备份和复制，同时了解 PostgreSQL 的有用扩展，以便您可以优化使用大型数据库的速度和性能。

在本书的最后，通过轻松实现高级管理任务，您将能够最大限度地使用数据库。

这本书是给谁的

本书介绍了 PostgreSQL 11 中新引入的功能，并向您展示了如何构建更好的 PostgreSQL 应用程序，以及如何有效地管理 PostgreSQL 数据库。您将掌握 PostgreSQL 的高级功能，并获得构建高效数据库解决方案所需的技能。、

这本书涵盖了什么

第 1 章，*PostgreSQL 概述*，介绍了 PostgreSQL 以及 PostgreSQL 11 及更高版本中提供的新功能。

第 2 章，了解事务和锁定，探索锁定和事务，并以最有效的方式利用 PostgreSQL 事务。

第 3 章，使用索引，讨论索引，它们的类型，用例以及如何实现我们自己的索引策略。

第 4 章，处理高级 SQL，是关于现代 SQL 及其功能。我们将探索集合及其各种类型，并编写自己的聚合。

第 5 章，日志文件和系统统计信息，介绍了如何理解数据库统计信息。

第 6 章，优化查询以获得良好性能，解释了如何编写更好，更快的查询。我们还将专注于理解查询本质上是好还是坏的原因。

第 7 章，编写存储过程，仔细研究了过程和函数之间的基本差异。还将讨论存储过程，使用扩展和 PL / pgSQL 的一些更高级的功能。

第 8 章，管理 PostgreSQL 安全性，介绍了作为 PostgreSQL 开发人员和 DBA 将面临的最常见的安全问题。

第 9 章，处理备份和恢复，介绍如何还原备份和处理部分转储数据。

第 10 章，了解备份和复制，查看 PostgreSQL 的事务日志，并解释我们可以用它来改进我们的设置并使事情更安全。

第 11 章，决定有用的扩展，讨论了 PostgreSQL 的一些最广泛的扩展。

第 12 章，*PostgreSQL 故障排除*，重点是接近未知数据库，识别关键瓶颈，处理存储损坏以及检查损坏的副本。

第 13 章，*迁移到 PostgreSQL*，是关于从其他数据库迁移到 PostgreSQL。

为了充分利用这本书

本书是为广大读者编写的。为了遵循本书中提供的示例，至少要有一些 SQL 经验，甚至可能是 PostgreSQL（尽管这不是一个严格的要求）。通常，熟悉 UNIX 命令行是个好

主意。

使用的约定

本书中使用了许多文本约定。

CodeInText: 表示文本，数据库表名，文件夹名，文件名，文件扩展名，路径名，虚拟 URL，用户输入和 Twitter 句柄中的代码字。下面是一个示例：“我将使用一个简单的 `INSERT` 命令向表中添加一行。”

任何命令行输入或输出都写成如下：

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test VALUES (0);
INSERT 0 1
```

粗体: 表示您在屏幕上看到的新术语，重要单词或单词。例如，菜单或对话框中的单词会出现在文本中。下面是一个示例：“**从管理面板中选择系统信息**”。



警告或重要说明如下所示。



TIP 提示和技巧看起来像这样。

保持联系

欢迎来自我们读者的反馈。

一般反馈: 如果您对本书的任何方面有疑问，请在邮件主题中提及书名，并发送电子邮件至 customercare@packtpub.com。

勘误: 虽然我们已尽力确保内容的准确性，但确实会发生错误。如果您在本书中发现错误，我们将非常感谢您向我们报告此事。请访问 www.packtpub.com/submit-errata，选择您的图书，点击勘误表提交表格链接，然后输入详细信息。

版权: 如果您在互联网上以任何形式发现我们作品的任何非法副本，我们将非常感谢您提供我们的位置地址或网站名称。请通过 copyright@packt.com 与我们联系，并提供链接材料。

如果您有兴趣成为作者: 如果有一个您具有专业知识的主题。如果您对书籍的撰写或撰稿感兴趣，请访问 authors.packtpub.com。

评论

请留下评论。阅读并使用本书后，为什么不在您购买的网站上留下评论？潜在的读者可以查看并使用您的公正意见来做出购买决定，Packt 可以了解您对我们产品的看法，我们的作者可以看到您对其图书的反馈。谢谢！

有关 Packt 的更多信息，请访问 packt.com。

Javanchueng

1. Postgresql 概览

从我准备编写一本关于 PostgreSQL 书已经有一段时间了。目前我已经取得了很大的进展并且以完成第三版的《精通 PostgreSQL》而自豪，这本书包含了当前 PostgreSQL 11 版本所有的新特性。

PostgreSQL 是世界上开源的顶级数据库中的一员，它包含了很多被程序开发人员和系统管理员使用的特性。从 PostgreSQL 11 开始，很多新特性被添加，为这款卓越的开源产品的成功做出了巨大贡献。

本书将详细介绍和讨论许多这些很酷的功能。

在本章中，您将了解 PostgreSQL 11 及其他版本可用的酷炫新功能。所有相关的新功能都将详细介绍。鉴于对代码所做的大量更改以及 PostgreSQL 开发项目的规模，这个功能列表当然远非完整，所以我试着专注于最重要的、与大多数人使用息息相关的方面。

本章概述的功能将分为以下几类：

- PostgreSQL 11 版本的新功能
- SQL 和开发人员相关
- 备份、恢复和复制
- 性能优化相关

PostgreSQL 11 版本的新功能有哪些？

PostgreSQL 11 于 2018 年秋季发布，为用户提供了几个现代特性的版本。这些特性对于数据库专业人士和初学者都很有用。PostgreSQL 11 是第二个在 PostgreSQL 社区引入的新编号方案之后的主要版本。版本 11 之后的 PostgreSQL 的下一个主要版本将是 12. 该服务版本后续将以 PostgreSQL 11.1, 11.2, 11.3 依此类推命名。与之前的 10 版本系列相比这是一个应该特别指出的重大变化。

你应该使用哪个版本？建议始终使用最新的发布版本。开始使用 PostgreSQL 9.6 已经没有任何意义了。如果你是 PostgreSQL 的初学者，从版本 11 开始。这个版本没有已知的 BUG 并且 PostgreSQL 社区将持续为您解决发现的 BUG，因此你不必害怕使用 PostgreSQL 10 或 PostgreSQL 11 版本。它们会工作的很好。

了解新的数据库管理功能

PostgreSQL 11 具有许多新功能，可以帮助管理员减少工作量并且使数据库系统运行得更加可靠，更加稳健。

最应该帮助人们使得数据库运行更高效的功能之一应该是能够配置数据库实例运行时的参数的大小，通常认为最应该被调整是 WAL 段。

使用可配置的 WAL 段大小

自从 20 年前推出 PostgreSQL 以来，单个 WAL 文件的大小始终如此是 16 MB。最开始，它甚至是一个可编译限制项，后来改为编译时的选项。从 PostgreSQL 11 开始，这些 WAL 段的大小可以在实例创建时更改，这为管理员提供了额外的配置和优化 PostgreSQL 的方法。下面是它的工作方式。以下示例说明在数据库 `initdb` 的时候如何配置 WAL 段大小：

```
initdb -D /pgdata --wal-segsize=32
```

`initdb` 命令是用来调用创建数据库实例的工具。通常是你看到的是什么就是什么，即使被您使用的某些 Linux 发布版、Windows 操作系统所隐藏。然而，`initdb` 现在有一个新的选项就是将所需的 WAL 段大小直接传递给程序。

正如我之前提到的，WAL 段默认大小是 16 MB；因此，在大多数情况下，使用更大的段大小用来提高性能是显而易见的。除非你是在一个嵌入式系统中运行非常非常小的数据实例。

那么对 WAL 段大小真正产生性能影响的会是什么？一如既往，这真的取决于你在做什么。如果您正在运行一个 99% 都是读的数据库系统，对大的 WAL 段的影响将为零。是的，你听到了- 零。如果你面对的写应用系统负载 95% 空闲并且没有严重负载，影响仍为零或接近零。只有你运行着一个沉重，写密集型工作负载的系统，你才会看到效果。然后，只有这样，改变 WAL 段大小才值得。如果你只是运行一个客户偶尔才会访问的几个在线表格-何必呢？该新特性只有在导致产生大量 WAL 变化时才会显示其优点。

pg_stat_statements 更大的 queryid 值

如果你真的想深入了解 PostgreSQL 的性能，那么 `pg_stat_statements` 就是其中可以查看的工具。就个人而言，我认为它是黄金标准，也就是说，如果你真的想弄清楚系统中发生了什么。一旦 PostgreSQL 启动，`pg_stat_statements` 命令就会通过加载 `shared_preload_libraries` 并汇总有关服务器中运行的查询的统计信息。如果出现问题，它会立即显示。

`pg_stat_statements` 命令提供了一个名为 `queryid` 的字段，该字段过去一直是 32 位长度。在某些情况下，这会导致问题，因为在某些情况下键值可能会发生冲突。Magnus Hagander 在他的一篇论文中计算过，在运行 30 亿个不同的查询，预计会发生大约 50,000 次冲突。通过引入一个 64 位长度的 `queryid`，在 30 亿种不同类型的查询之后，这个数字预计会下降到大约 0.25 个冲突，这是一个重大改进。

请记住，如果你要迁移到 PostgreSQL 11，如果使用 `pg_stat_statements` 来跟踪性能问题，则可能需要更新脚本。

改进的索引和更好的优化

PostgreSQL 11 提供的不仅仅是一些用于管理方面改进的功能。索引的功能也有所改进。

其中一个最重要的功能是与索引和统计信息相关。

表达式索引统计信息

如果您正在运行一个简单的查询，PostgreSQL 将通过查看内部统计信息来优化它。如下示例：

```
SELECT * FROM person WHERE gender = 'female';
```

在这种情况下，PostgreSQL 将查询内部统计数据并估算表中女孩的数量。如果数量很少，PostgreSQL 会考虑走索引查找。如果大多数记录是女性，PostgreSQL 将考虑表顺序扫描而不是索引。表的每列都有统计信息数据。此外，还可以查看 PostgreSQL 10 版本中引入的跨列统计信息（查看 `CREATE STATISTICS` 命令以了解更多信息）。好消息是 PostgreSQL 还会跟踪基于函数的索引的统计信息：

```
CREATE INDEX idx_cos ON t_data (cos(data));
```

目前尚无法实现的是对函数索引使用更复杂的统计信息数据。

请考虑以下包含各种列的索引示例：

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));
```

```
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

在这个案例中，两列上有一个索引，它还提供由表达式表示的虚拟第三列。新功能允许我们为第三列显式创建更多统计信息，否则这些信息将被次最优地覆盖。在我的例子中，我们将明确告诉 PostgreSQL 我们希望第三列在系统统计信息中有 1,000 个条目。它将允许优化器作出更好的估计，从而可能创建更好的执行计划。这将对一些特殊的应用提高效率作出非常宝贵的贡献。

包含索引或覆盖索引

许多其他数据库系统长期以来提供了一种称为**覆盖索引**的功能。

这是什么意思？请考虑以下示例，该示例只从表中选择两列：

```
SELECT id, name FROM person WHERE id = 10;
```

假设我们只有 `id` 列上的索引。在这种情况下，PostgreSQL 将查找索引并在表中执行查找以得到额外的其他字段。这通常被称为索引扫描。它包括检查索引和基础表以组成一行。这里可以通过创建一个由两列组成的索引去解决。这个方法允许 PostgreSQL 执行仅索引扫描而不是索引扫描。如果索引具有所需的所有列，则无需再在表中执行额外的查找（对于大多数情况）。



TIP 查询列只写你真正需要的列--否则你可能会触发无意义的表查找。因此，通常认为以下类型的查询性能非常差：

```
SELECT * FROM person WHERE id = 10;
```

这里的问题是，如果您需要对 ID 进行主键约束，并且仍希望在读取其他列时执行仅索引扫描。这是新特性能用于实现的方式：

```
CREATE UNIQUE INDEX some_name ON person USING btree (id) INCLUDE (name);
```

PostgreSQL 将确保 `id` 值是唯一的，但仍会在索引中存储其他字段，以便在询问两列时执行仅索引扫描。在大容量 OLTP 环境中，这将显著提高性能。当然，要提供查询效率提升的具体数字是很困难的，因为每个表和每种类型的查询都是不同的。但是，成果是

非常明显的。PostgreSQL 11 将为我们提供更多选项来实现更多的仅索引扫描。

并行索引创建

在 PostgreSQL 中创建索引时，数据库传统上使用一个 CPU 核心来完成工作。在许多情况下，这不是问题。但是，PostgreSQL 用于持续增长的系统，因此在许多情况下索引创建开始成为一个问题。此刻社区也在努力改善排序的性能。因此，第一步是允许并行创建 **btrees**，已在 PostgreSQL 11 版本实现。PostgreSQL 的未来版本也将允许为正常操作提供并行排序功能，但遗憾的是，PostgreSQL 11 尚未支持。

并行索引创建可以显著加快索引创建速度，我们渴望看到此领域在未来的改进（可能支持其他索引类型，等等）。

更好的缓存管理

PostgreSQL 11 还将为您提供更好的方法来管理 I/O 缓存（共享缓冲区）。`pg_prewarm` 命令特别值得注意。

改进 `pg_prewarm`

`pg_prewarm` 命令允许您在重新启动数据库实例后恢复 PostgreSQL I/O 缓存的内容。它已经存在了相当长的一段时间，并被 PostgreSQL 用户群广泛使用。在 PostgreSQL 11 中，`pg_prewarm` 已被扩展并允许以固定间隔时间自动转储缓冲区内容列表。

`pg_prewarm` 还可以自动预加载旧的缓存内容，以便用户在重新启动后具有更好的数据库性能。特别是，具有大量内存的系统可以从这些新的改进中受益。

窗口函数增强

窗口函数和分析是任何现代 SQL 实现的基石，因此被专业人士广泛使用。PostgreSQL 已经为窗口函数提供了很长一段时间的支持。但是，对比 SQL 标准仍然存在一些小功能缺失。PostgreSQL 11 现在完全支持 SQL: 2011 提出的标准。

添加了以下功能：

- 范围区间（Range between）：
 - Previously just ROWS
 - Now handles values
- 排除语句（Exclusion clauses）：
 - 排除当前行（Exclude current row）
 - 排除关系（Exclude ties）

为了演示新功能的工作原理，我决定加入一个例子。该代码包含两个窗口函数。它们解释如下：

- 第一个使用 PostgreSQL 10 和之前已有的内容。
- 第二个 `array_agg` 排除当前行，这是 PostgreSQL 11 提供的新功能。

以下代码生成五行并包含两个窗口函数：

```
test=# SELECT *,  
          array_agg(x) OVER (ORDER BY x ROWS BETWEEN  
                               1 PRECEDING AND 1 FOLLOWING),  
          array_agg(x) OVER (ORDER BY x ROWS BETWEEN  
                               1 PRECEDING AND 1 FOLLOWING EXCLUDE CURRENT ROW)  
      FROM generate_series(1, 5) AS x;  
x | array_agg | array_agg  
---+-----+-----  
1 | {1,2}    | {2}  
2 | {1,2,3}  | {1,3}  
3 | {2,3,4}  | {2,4}  
4 | {3,4,5}  | {3,5}  
5 | {4,5}    | {4}  
(5 rows)
```

排除当前行是一个非常常见的要求，因此不应低估。

介绍即时编译

实时编译（JIT）实际上是 PostgreSQL 11 的亮点之一。为了支持将来更多的 JIT 编译，添加了许多基础架构，PostgreSQL 11 是第一个充分利用这一现代技术的版本。在我们深入了解 JIT 之前，我们需要知道 JIT 编译是关于什么的？在运行一个查询时，很多东西实际上只在运行时才会知道，而不是在编译时（编译 PostgreSQL 时）。因此，传统的编译器总是处于劣势，因为它不知道在运行时会发生什么。但是 JIT 编译器已经知道更多内容并且可以做出相应的反应。

从 PostgreSQL 11 开始，您可以使用 JIT 编译，这对于大型的查询尤其有用。我们将在本书后面的章节中深入研究细节。

增强的分区

PostgreSQL 10 是 PostgreSQL 引入了分区技术的第一个版本。当然，我们曾经使用继承的方式来实现。但是，PostgreSQL 10 确实是第一个提供现代分区功能的版本。PostgreSQL 11 将通过引入一些新亮点为这个功能强大的功能添加一些新功能，例如，如果现有分区都不匹配，则可以创建默认分区。

下面是它的工作原理：

```
postgres=# CREATE TABLE default_part PARTITION OF some_table DEFAULT;  
CREATE TABLE
```

在这个案例中，所有在任何分区条件都不匹配的行将最终出现在默认分区中。

不仅仅如此。在 PostgreSQL 中，一行不能（轻松的）从一个分区移动到另一个分区。假设你有一个按照国家/地区区分的分区。如果一个人从法国搬到爱沙尼亚，你就不能用一个简单的 UPDATE 语句来实现。你必须删除旧的行并插入一个新的。在 PostgreSQL 11 中，这个问题已经解决了。现在可以以完全透明的方式将行从一个分区移动到另一个分区。

PostgreSQL 仍然还有很多缺点。在旧版本中，所有分区都必须单独编制索引。无法为所有分区创建单个索引。在 PostgreSQL 11 中，为父表创建的索引将自动确保所有子表都被索引。这真的很有用，因为索引被遗忘创建的可能性降低。此外，在 PostgreSQL 11 中，您实际上可以添加全局唯一索引。因此，分区表可以强制执行唯一约束。

截止到 PostgreSQL 10，我们实现了范围分区和列表分区。PostgreSQL 11 增加了散列分区的功能。这是一个显示散列分区如何工作的示例：

```
test=# CREATE TABLE tab(i int, t text) PARTITION BY HASH (i);
CREATE TABLE
test=# CREATE TABLE tab_1 PARTITION OF tab
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE
```

但是，不仅仅是有更多的功能。还有很多新东西可以提高性能。分区裁剪现在要快得多，PostgreSQL 还能够智能的思考跨分区关联以及跨分区聚合，这正是数据分析和数据仓库所需要的。

添加对存储过程的支持

PostgreSQL 一直支持函数功能，这些函数通常被称为存储过程。但是，存储过程和函数之间存在区别。如前所述，直到 PostgreSQL 10，我们只有函数而没有存储过程。

关键点是函数是一个结构体的部分，一个单独的事务。一个存储过程可以包含多个如上的事务。因此，它不能被更大的事务调用，而是一个独立的结构体。

这是 CREATE PROCEDURE 的语法：

```
test=# \h CREATE PROCEDURE
Command: CREATE PROCEDURE
Description: define a new procedure
Syntax:
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ]
        argtype [ { DEFAULT | = } default_expr ] [, ...] )
    { LANGUAGE lang_name
        | TRANSFORM { FOR TYPE type_name } [, ... ]
        | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
        | SET configuration_parameter { TO value | = value | FROM CURRENT }
        | AS 'definition'
        | AS 'obj_file', 'link_symbol'
    } ...
```

以下存储过程说明了如何在同一过程中执行两个事务：

```
test=# CREATE PROCEDURE test_proc()
    LANGUAGE plpgsql
AS $$

BEGIN
    CREATE TABLE a (aid int);
    CREATE TABLE b (bid int);
    COMMIT;
```

```
CREATE TABLE c (cid int);
    ROLLBACK;
END;
$$;
```

CREATE PROCEDURE

请注意，在第二个事务中止时，前两个语句已提交。稍后您将在此示例中看到此更改的效果。

要运行该过程，我们可以使用 CALL:

```
test=# CALL test_proc();
```

```
CALL
```

前两个表已提交--由于存储过程内部的回滚，尚未创建第三个表：

```
test=# \d
```

List of relations

Schema	Name	Type	Owner
public	a	table	hs
public	b	table	hs
(2 rows)			

存储过程是迈向具备完整且功能齐全的数据库系统的重要一步。

改进 ALTER TABLE

ALTER TABLE 命令可用于更改表的定义。在 PostgreSQL 11 中，ALTER TABLE ... ADD COLUMN 的执行得到了显著改善。我们来看看细节。以下示例显示了如何将列添加到表中以及 PostgreSQL 如何处理这些新列：

```
ALTER TABLE x ADD COLUMN y int;
```

```
ALTER TABLE x ADD COLUMN z int DEFAULT 57;
```

列表中的第一个命令一直很快，原因是在 PostgreSQL 中，列的默认值为 NULL。所以 PostgreSQL 所做的就是在系统目录中添加一列而不实际存储。该列将添加到表的末尾，即使如果磁盘上的行太短，我们也知道无论如何它的值都将为 NULL。换句话说，即使您向 10 TB 表添加列，操作也会非常快，因为不必更新磁盘上的行。

在第二个案例中，情况则完全不同。DEFAULT 57 实际上确实在行中添加了实际数据，在 PostgreSQL 10 及更早版本中，这意味着数据库必须重写整个表以添加这个新的默认值。如果你有一张小表，这不是什么大问题。但是，如果您的表包含数十亿行记录，则无法将其锁定并重写--在专业的在线事务处理（OLTP）系统中，停机时间是不可能的。

从 PostgreSQL 11 开始，可以在不重写整个表的情况下向表中添加固定的默认值，这大大减轻了更改数据结构的负担。

小结

在 PostgreSQL 11 中，添加了许多功能，使人们能够更快，更高效地运行更专业的应用程序。数据库服务器很多领域都得到了改进，并增加了许多新的专业特性。将来，更多的改进都将实现。当然，本章中列出的更改尚未完全完成，只进行了许多小的更改。

在下一章中，您将学习索引和 PostgreSQL 基于成本规则的优化模型，这对于想保持数据库良好性能的你非常重要。

QA

PostgreSQL 11 最重要的特性是什么？

实际上，这很难说。这实际上取决于您使用数据库的方式以及哪些功能对您的应用程序最重要。然而，每个人都有个人喜好。就我而言，是并行索引创建，对于运行巨大的数据库的客户来说非常重要。不过，喜欢什么不喜欢什么完全取决于你自己。

PostgreSQL 11 可以在我的平台上运行吗？

PostgreSQL 11 适用于所有常见平台，包括但不限于 Linux, Windows, Solaris, AIX 和 macOS X。PG 社区试图覆盖尽可能多的平台，以免影响人们使用 PostgreSQL。对于大多数常见系统，PostgreSQL 甚至会被预先打包安装。

许可证模式是否变更过？

不，没有任何改变，并且将来也不会改变。

我们什么时候可以期待 PostgreSQL 12？

通常，每年预期发布一次主要版本。所以 PostgreSQL 12 的下一个主要版本将在 2019 年秋季推出。

2. 了解事务和锁

在第一次介绍 PostgreSQL 11 之后，我们希望将注意力集中在下一个重要主题上。锁是任何类型数据库的重要概念。仅仅了解如何编写适当或更好的应用是不够的，从性能的角度来看它也是必不可少的。如果不正确处理锁，您的应用程序可能不仅速度慢，而且可能也会出错，并且会出现意想不到的问题。在我看来，锁是性能的关键，对锁有全面的认知肯定会有帮助。因此，了解锁和事务对于数据库管理员和开发人员都很重要。在本章中，您将了解以下主题：

- 使用 PostgreSQL 事务
- 理解锁的基本信息
- 使用 FOR SHARE 和 FOR UPDATE
- 了解事务隔离级别
- 考虑可序列化快照隔离（SSI）事务
- 观察死锁和类似问题
- 优化存储和管理数据清理

在本章的最后，您将能够以最有效的方式理解和利用 PostgreSQL 事务。

使用 PostgreSQL 事务

PostgreSQL 为您提供了一种高级的事务处理机制，为开发人员和管理员提供了无数的功能。在本节中，是时候看一下事务的基本概念。

首先要知道的是，在 PostgreSQL 中，一切都是事务。如果您向服务器发送一个简单的查询，它已经是一个事务。这是一个例子：

```
test=# SELECT now(), now();
           now          |           now
-----+-----
 2018-08-24 16:03:27.174253+02 | 2018-08-24 16:03:27.174253+02
(1 row)
```

在这种情况下，SELECT 语句将是一个单独的事务。如果再次执行相同的命令，将返回不同的时间戳。



TIP 请记住，now() 函数将返回事务执行时的时间。因此，SELECT 语句将始终返回两个相同的时间戳。如果您想要查询“实时”时间，请考虑使用 clock_timestamp() 而不是 now()。

如果多个语句必须是同一事务的一部分，则必须使用 BEGIN 语句，如下所示：

```
test=# \h BEGIN
Command: BEGIN
Description: Start a transaction block
Syntax:
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
where transaction_mode is one of:
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ }
```

```
| READ COMMITTED | READ UNCOMMITTED }
```

```
READ WRITE | READ ONLY
```

```
[ NOT ] DEFERRABLE
```

BEGIN 语句将确保将多个命令打包到一个事务中。下面是它的工作原理：

```
test=# BEGIN;
BEGIN
test=# SELECT now();
      now
-----
2018-08-24 16:04:08.105131+02
(1 row)
test=# SELECT now();
      now
-----
2018-08-24 16:04:08.105131+02
(1 row)
test=# COMMIT;
COMMIT
```

这里重要的是两个时间戳都是相同的。如前所述，我们谈论的是事务时间。

要结束事务，可以使用 COMMIT：

```
test=# \h COMMIT
Command: COMMIT
Description: Commit the current transaction
Syntax:
COMMIT [ WORK | TRANSACTION ]
```

这里有几个语法元素。您可以使用 COMMIT, COMMIT WORK 或 COMMIT TRANSACTION。

这三个选项具有相同的含义。如果这还不够，还有更多：

```
test=# \h END
Command: END
Description: commit the current transaction
Syntax:
END [ WORK | TRANSACTION ]
```

END 子句与 COMMIT 子句相同。

ROLLBACK 是 COMMIT 的相反操作。它不会成功结束事务，而只是停止事务，从而让其他事务看不见：

```
test=# \h ROLLBACK
Command: ROLLBACK
Description: Abort the current transaction
Syntax:
ROLLBACK [ WORK | TRANSACTION ]
```

某些应用程序使用 ABORT 而不是 ROLLBACK。意思是一样的。

处理一个事务内部的错误

一个事务并非总是从头到尾都是正确执行的。无论出于何种原因，事务可能会出错。但是，在 PostgreSQL 中，只能提交无错误的事务。以下清单显示了一个失败的事务，该事务由于除数为零而失败：

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?

-----
1
(1 row)
test=# SELECT 1 / 0;
ERROR: division by zero
test=# SELECT 1;
ERROR: current transaction is aborted, commands ignored until end of
transaction block
test=# COMMIT;
ROLLBACK
```

请注意，division by zero 无效。



TIP 在任何恰当的数据库中，类似于此的指令将立即输出错误并使语句失败。

需要重点指出的是 PostgreSQL 会出错，不像 MySQL 那样不那么严格。发生错误后，即使这些后续指令在语义和语法上都是正确的，也不会再被执行。但是仍然可以发出 COMMIT 命令。然而最后唯一正确的事情是 PostgreSQL 将回滚事务。

使用 SAVEPOINT

在专业应用程序中，编写一个合理的长的事务而不遇到某个错误可能非常困难。为了解决这个问题，用户可以使用一种名为 **SAVEPOINT** 的命令。顾名思义，保存点是事务中的一个安全位置，如果该事务出现严重错误，应用程序可以返回到此安全位置。下面是一个例子：

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?

-----
1
(1 row)
test=# SAVEPOINT a;
SAVEPOINT
test=# SELECT 2 / 0;
ERROR: division by zero
```

```
test=# SELECT 2;
ERROR: current transaction is aborted, commands ignored until end of
transaction block
test=# ROLLBACK TO SAVEPOINT a;
ROLLBACK
test=# SELECT 3;
?column?
-----
      3
(1 row)
test=# COMMIT;
COMMIT
```

在第一个 SELECT 子句之后，我决定创建一个保存点，以确保应用程序始终可以返回到事务内的这一点。如您所见，保存点有一个名称，稍后会提到。

返回到名为 a 的保存点后，事务可以正常进行。代码在错误发生前已经跳回来，所以一切都执行得很好。

事务中的保存点数实际上是无限的。我们已经看到客户在一次操作中拥有超过 250,000 个保存点。PostgreSQL 可以轻松处理这个问题。

如果要从事务内部删除保存点，可以使用 RELEASE SAVEPOINT 命令：

```
test=# \h RELEASE SAVEPOINT
Command: RELEASE SAVEPOINT
Description: Destroy a previously defined SAVEPOINT
Syntax:
RELEASE [ SAVEPOINT ] savepoint_name
```

许多人会问，如果您在事务结束后尝试回到某个保存点会发生什么？答案是，一旦事务结束，保存点的生命周期就会结束。换句话说，在交易完成后无法返回某个时间点。

DDLs 事务

PostgreSQL 有一个非常好的功能，遗憾的是在许多商业数据库系统中都没有。在 PostgreSQL 中，可以在事务块内运行 DDL（更改数据结构的命令）。在典型的商业系统中，DDL 将隐式提交当前事务。这在 PostgreSQL 中不会发生。

除了一些小的例外（DROP DATABASE, CREATE TABLESPACE, DROP TABLESPACE 等）之外，PostgreSQL 中的所有 DDL 都是事务性的，这对最终用户来说有巨大的优势和真正的好处。如下是一个示例：

```
test=# \d
No relations found.
test=# BEGIN;
BEGIN
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# ALTER TABLE t_test ALTER COLUMN id TYPE int8;
ALTER TABLE
test=# \d t_test
```

```

Table "public.t_test"
Column | Type | Modifiers
-----+-----+
  id   | bigint |
test=# ROLLBACK;
ROLLBACK
test=# \d
No relations found.

```

在此示例中，已创建并修改了表，然后中止整个事务。如您所见，没有隐式 COMMIT 或任何其他奇怪的行为。PostgreSQL 只是按照预期的情况运行。

如果您要部署软件，事务性 DDLs 尤其重要。试想一下在运行的内容管理系统（CMS）。如果发布了新版本，您将需要升级。运行旧版本仍然可以；运行新版本也行，但你真的不希望新旧系统混杂运行。因此，在完成升级操作时，在单个事务中部署升级肯定是非常有益的。



为了更好促进的软件更新实践，我们可以将从我们的源代码控制系统中生成的多个模块编码打包成一个单独的事务。

理解锁的基本信息

在本节中，您将学习基本的锁原理机制。目标是了解锁的一般工作方式以及如何正确获取简单的应用程序锁权限。

为了展示其工作原理，可以创建一个简单的表。出于演示目的，我会使用简单的 INSERT 命令向表中添加一行：

```

test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test VALUES (0);
INSERT 0 1

```

第一个重要的事情是表可以同时被读取。许多用户同时读取相同的数据不会相互阻塞。这使得 PostgreSQL 可以毫无问题地处理数千个用户读取表。



多个用户可以同时读取相同的数据而不会相互阻塞。

现在的问题是如果读取和写入同时发生会发生什么？这是一个例子。我们假设该表包含一行，其 id = 0：

事务 1	事务 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
用户将会看到结果 1	SELECT * FROM t_test;
	用户将会看到结果 0
COMMIT;	COMMIT;

打开两个事务。第一个将改变一行。但是，这不是问题，因为第二个事务可以继续。它

将返回 UPDATE 之前的旧行。此行为称为多版本并发控制（MVCC）。



只有在写入事务（在启动读取事务之前）提交数据时，事务才会看到数据。一个事务无法查看另一个活动连接事务所做的更改。事务只能看到已提交的更改。

还有第二个重要现实--许多商业或开源数据库仍然（截至 2018 年）无法处理并发读写。在 PostgreSQL 中，这绝对不是问题。读写可以共存。



写事务不会阻塞读事务。

提交事务后，该表值将变为 1。

如果两个人同时更改数据会发生什么？如下示例：

事务 1	事务 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
将会返回结果 2	UPDATE t_test SET id = id + 1 RETURNING *;
	等待事务 1 执行完成
COMMIT;	等待事务 1 执行完成
	将会重新读取记录，找到记录 2，并执行并 返回结果 3
	COMMIT;

假设您要计算网站上的点击次数。如果您按照刚刚概述的那样运行代码，则不会丢失点击次数的计算，因为 PostgreSQL 保证 UPDATE 语句执行一个接一个。



PostgreSQL 只会锁定受 UPDATE 影响的行。因此，如果您有 1,000 行，理论上可以在同一个表上运行 1,000 个并发更新。

值得注意的是，您始终可以运行并发读取。我们的两次写入不会阻塞读取。

避免典型错误和显式锁定

在我作为专业 PostgreSQL 顾问（<https://www.cybertec-postgresql.com>）的职业生涯中，我看到了一些经常重复犯的错误。如果生活中存在常数，那么这些典型的错误肯定是一些永不改变的事情。

这是我最喜欢的：

事务 1	事务 2
BEGIN;	BEGIN;
SELECT max(id) FROM product;	SELECT max(id) FROM product;
用户看到结果 17	用户看到结果 17
用户决定写入 18	用户决定写入 18

INSERT INTO product ... VALUES (18, ...)	INSERT INTO product ... VALUES (18, ...)
COMMIT;	COMMIT;

在这种情况下，将存在重复的主键冲突或两个相同的条目。这个问题的变化都没有吸引力。

解决问题的一种方法是使用显式表锁定。以下清单显示了 LOCK 的语法定义：

test=# \h LOCK

Command: LOCK

Description: lock a table

Syntax:

LOCK [TABLE] [ONLY] name [*] [, ...] [IN lockmode MODE]

[NOWAIT]

where lockmode is one of the following:

ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE |

SHARE UPDATE EXCLUSIVE | SHARE |

SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE

如您所见，PostgreSQL 提供了八种类型的锁来锁定整个表。在 PostgreSQL 中，锁可以像 ACCESS SHARE 锁一样轻，也可以像 ACCESS EXCLUSIVE 锁一样重。以下列表显示了这些锁的作用：

- **ACCESS SHARE:** 这种类型的锁仅由读取产生并且和 ACCESS EXCLUSIVE 冲突，后者由 DROP TABLE 等设置。实际上，这意味着如果要删除表，SELECT 将无法启动。这也意味着 DROP TABLE 必须等到读取事务完成。
- **ROW SHARE:** 在 SELECT FOR UPDATE/SELECT FOR SHARE 的情况下，PostgreSQL 采用这种锁。它与 EXCLUSIVE 和 ACCESS EXCLUSIVE 冲突。
- **ROW EXCLUSIVE:** 此锁由 INSERT, UPDATE 和 DELETE 执行产生。它与 SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE 和 ACCESS EXCLUSIVE 冲突。
- **SHARE UPDATE EXCLUSIVE:** 这种锁是由 CREATE INDEX CONCURRENTLY, ANALYZE, ALTER TABLE, VALIDATE 和其他一些类型的 ALTER TABLE 以及 VACUUM (不是 VACUUM FULL) 产生。它与 SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE 和 ACCESS EXCLUSIVE 锁定模式冲突。
- **SHARE:** 创建索引时，将设置 SHARE 锁。它与 ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE 和 ACCESS EXCLUSIVE 冲突。
- **SHARE ROW EXCLUSIVE:** 这个是由 CREATE TRIGGER 和某些形式的 ALTER TABLE 并与除了 ACCESS SHARE 之外的所有类型冲突。
- **EXCLUSIVE:** 这种类型的锁是迄今为止最严格的锁。它阻塞读写操作。如果事务占用此锁，则其他任何人都无法读取或写入受影响的表。
- **ACCESS EXCLUSIVE:** 此锁可阻塞并发事务的读写。

鉴于 PostgreSQL 锁的基础概念，前面概述的最大问题的一种解决方案如下。如下示例显示了如何锁定表：

```
BEGIN;
LOCK TABLE product IN ACCESS EXCLUSIVE MODE;
INSERT INTO product SELECT max(id) + 1, ... FROM product;
COMMIT;
```

请记住，这是一种非常讨厌的方式来执行此类操作，因为在您的操作过程中没有其他人

可以读取或写入表。因此，应不惜一切代价避免 ACCESS EXCLUSIVE 类型的锁定。

考虑替代解决方案

有一个解决问题的替代方案。考虑一个例子，要求您编写生成发票编号的应用程序。税务局可能会要求您创建没有间隔且没有重复的发票编号。你会怎么做？当然，一种解决方案是使用表锁。但是，你真的可以做得更好。我将采取以下措施来处理我们要解决的编号问题：

```
test=# CREATE TABLE t_invoice (id int PRIMARY KEY);
CREATE TABLE
test=# CREATE TABLE t_watermark (id int);
CREATE TABLE
test=# INSERT INTO t_watermark VALUES (0);
INSERT 0
test=# WITH x AS (UPDATE t_watermark SET id = id + 1 RETURNING *)
      INSERT INTO t_invoice
      SELECT * FROM x RETURNING ;
id
---
1
(1 row)
```

在这种情况下，我引入了一个名为 `t_watermark` 的表。它只包含一行。先执行 `WITH` 命令。该行将被锁定并递增，并返回新值。一次只能有一个人这样做。然后，`CTE` 返回的值将用于发票表中。它保证是独一无二的。美妙的是 `t_watermark` 表上只有一个简单的行锁，并且发票表中没有任何读取被阻塞。总的来说，这种方式更具可扩展性。

使用 FOR SHARE 和 FOR UPDATE

有时候，从数据库中选择数据，然后在应用程序中进行一些处理，最后，在数据库端进行一些更改。这是 `SELECT FOR UPDATE` 的典型示例。

如下是一个例子，显示了 `SELECT` 经常以错误的方式执行的方式：

```
BEGIN;
SELECT * FROM invoice WHERE processed = false;
** application magic will happen here **
UPDATE invoice SET processed = true ...
COMMIT;
```

这里的问题是两个人可能会选到相同的未处理数据。对这些数据的变更可能会被覆盖。简而言之，将出现竞争情况。

要解决此问题，开发人员可以使用 `SELECT FOR UPDATE`。以下是它的使用方法。以下示例将显示一个的典型场景：

```
BEGIN;
SELECT * FROM invoice WHERE processed = false FOR UPDATE;
** application magic will happen here **
```

```
UPDATE invoice SET processed = true ...
```

```
COMMIT;
```

SELECT FOR UPDATE 将像 UPDATE 一样锁定行。这意味着数据不会同时发生变化。所有锁将像往常一样在 COMMIT 时释放。

如果一个 SELECT FOR UPDATE 命令正在等待某个其他 SELECT FOR UPDATE 命令，则必须等到另一个完成（COMMIT 或 ROLLBACK）。如果第一个事务不结束（任何情况），那么第二个事务可能会永远等待。为避免这种情况的发生，可以使用 SELECT FOR UPDATE NOWAIT 替代。

下面是它的工作方式：

事务 1	事务 2
BEGIN;	BEGIN;
SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;	
一些其他处理...	SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;
一些其他处理...	ERROR: could not obtain lock on row in relation tab

如果 NOWAIT 不够灵活，请考虑使用 lock_timeout 参数。它将保持您想要等待锁定的时间。您可以在每个会话级别设置此参数：

```
test=# SET lock_timeout TO 5000;
```

```
SET
```

在此，值被设置为 5 秒。

虽然 SELECT 查询操作基本上没有锁，但 SELECT FOR UPDATE 可能非常苛刻。想象一下以下的业务流程：我们想要填满一个提供 200 个座位的飞机。许多人想要同时预订座位。在这种情况下，可能会发生以下情况：

事务 1	事务 2
BEGIN;	BEGIN;
SELECT ... FROM flight LIMIT 1 FOR UPDATE;	
等待用户输入...	SELECT ... FROM flight LIMIT 1 FOR UPDATE
等待用户输入...	它必须等待

麻烦的是，一次只能预订一个座位。可能有 200 个座位，但每个人都必须等待第一个人。

虽然第一个座位被阻塞，但即使人们不关心他们最终获得哪个座位，也没有其他人可以预订座位。

SELECT FOR UPDATE SKIP LOCKED 将解决问题。我们先创建一些示例数据：

```
test=# CREATE TABLE t_flight AS
```

```
SELECT * FROM generate_series(1, 200) AS id;
```

```
SELECT 200
```

现在魔术来了：

事务 1	事务 2
BEGIN;	BEGIN;
SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;	SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;
返回值 1 和 2	返回值 3 和 4

如果每个人都想获取两行，我们可以一次提供 100 个并发事务，而不必担心事务阻塞。



TIP 请记住，等待是最慢的执行形式。如果一次只能激活一个事务，如果您的真正问题通常是由锁和事务冲突引起的，购买更多昂贵服务器是没有意义的。

但是，还有更多状况。在某些情况下，`FOR UPDATE` 会产生意想不到的后果。大多数人都不知道 `FOR UPDATE` 会对外键产生影响。假设我们有两个表：一个用于存储货币而另一个用于存储账户。如下清单显示了一个示例：

```
CREATE TABLE t_currency (id int, name text, PRIMARY KEY (id));
INSERT INTO t_currency VALUES (1, 'EUR');
INSERT INTO t_currency VALUES (2, 'USD');
CREATE TABLE t_account (
    id          int,
    currency_id int      REFERENCES t_currency (id)
                          ON UPDATE CASCADE
                          ON DELETE CASCADE,
    balance     numeric);
INSERT INTO t_account VALUES (1, 1, 100);
INSERT INTO t_account VALUES (2, 1, 200);
```

现在，我们要在帐户表上运行 `SELECT FOR UPDATE`:

事务 1	事务 2
BEGIN;	
SELECT * FROM t_account FOR UPDATE;	BEGIN;
等待用户处理...	UPDATE t_currency SET id = id * 10;
等待用户处理...	将等待事务 1

虽然帐户表上有 `SELECT FOR UPDATE` 命令，但是将阻止货币表上的 `UPDATE` 命令。这是必要的，因为否则，有可能完全违反外键约束条件。因此，在相当复杂的数据结构中，您很容易在最不期望它们的区域（一些非常重要的查找表）中出现争用。

在 `FOR UPDATE` 之上，有 `FOR SHARE`, `FOR NO KEY UPDATE` 和 `FOR KEY SHARE`。以下列表描述了这些模式的实际含义：

- `FOR NO KEY UPDATE`: 这个与 `FOR UPDATE` 非常相似。但是，锁程度较弱，因此它可以与 `SELECT FOR SHARE` 共存。
- `FOR SHARE`: `FOR UPDATE` 相当强大，并且作用于预期您肯定会进行数据变更的情况下。`FOR SHARE` 是不同的，因为多个事务可以同时持有 `FOR SHARE` 锁
- `FOR KEY SHARE`: 其行为与 `FOR SHARE` 类似，只是锁程度较弱。它将阻塞 `FOR UPDATE`，但不会阻塞 `FOR NO KEY UPDATE`。

这里重要的事情是简单地尝试并观察发生的事情。改进锁的行为非常重要，因为它可以显著提高应用程序的可伸缩性。

理解事务隔离级别

到目前为止，您已经了解了如何处理锁以及一些基本的并发操作。在本节中，您将学习事务隔离级别。对我而言，这是现代软件开发中最被忽视的主题之一。只有一小部分软

件开发人员真正意识到这个问题，这反过来会导致令人难以置信的错误。

以下是可能发生的事情的示例：

事务 1	事务 2
BEGIN;	
SELECT sum(balance) FROM t_account;	
用户会看到结果 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	
用户会看到结果 400	
COMMIT;	

大多数用户实际上都希望左侧交易总是返回 300，而不管第二次交易。但是，事实并非如此。默认情况下，PostgreSQL 是以 READ COMMITTED 事务隔离级别模式运行的。这意味着事务中的每个语句都将获得数据的新快照，这将在整个查询中保持不变。



SQL 语句将在同一个快照上运行并将被忽略并发事务在执行时的变更。

如果您想避免这种情况，可以使用 TRANSACTION ISOLATION LEVEL REPEATABLE READ。在此事务隔离级别中，事务将在整个事务中使用相同的快照。以下是将要发生的事情：

事务 1	事务 2
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
SELECT sum(balance) FROM t_account;	
用户会看到结果 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	SELECT sum(balance) FROM t_account;
用户会看到结果 300	用户会看到结果 400
COMMIT;	

如上所述，第一个事务将冻结其数据快照，并在整个事务中为我们提供一致的结果。如果你要生成报表，此功能尤为重要。报表的第一页和最后一页应该始终如一，并且一致对相同的数据进行操作。因此，可重复读是报表生成的关键。

请注意，隔离级别相关的错误不会立即显现。可能在将应用程序转移到生产环境之后的几年才会发现问题。



可重复读（repeatable read）并不比读已提交（read committed）更昂贵。没有必要担心性能损失。对于正常的在线事务处理（OLTP）系统，读已提交具有各种优点，因为可以更早地看到更改，并且由于意外导致错误的几率通常更低。

考虑 SSI 事务

除了读已提交和可重复读取之外，PostgreSQL 还提供 SSI (serializable snapshot isolation) 事务隔离级别。总而言之，PostgreSQL 支持三种隔离级别。请注意，不支持 `read uncommitted` (读未提交，在某些商业数据库中仍然是默认值)：如果尝试启动 `read uncommitted` 事务，PostgreSQL 将默认设置为 `read committed`。让我们回到可序列化快照读的隔离级别。

可序列化快照读隔离级别的想法很简单；如果已知某个事务在只有一个用户时正常工作，那么在选择此隔离级别时，它也可以在并发的情况下工作。但是，用户必须做好准备：交易可能会失败（按设计要求的）并导致错误。除此之外，会产生较大性能损失。如果您想了解有关此隔离级别的更多信息，请考虑查看 <https://wiki.postgresql.org/wiki/Serializable>。



TIP 只有当您对数据库引擎内部发生的情况有一个合理的了解时，才考虑使用可序列化快照读隔离级别。

观察死锁和类似问题

死锁是一个重要的问题，可能发生在我所知道的每个数据库中。基本上，如果两个事务必须等待彼此，就会发生死锁。

在本节中，您将看到这是如何发生的。假设我们有一个包含两行的表：

```
CREATE TABLE t_deadlock (id int);
INSERT INTO t_deadlock VALUES (1), (2);
```

以下示例显示了可能发生的情况：

事务 1	事务 2
BEGIN;	BEGIN;
UPDATE t_deadlock SET id = id * 10 WHERE id = 1;	UPDATE t_deadlock SET id = id * 10 WHERE id = 2;
UPDATE t_deadlock SET id = id * 10 WHERE id = 2;	
等待事务 2	UPDATE t_deadlock SET id = id * 10 WHERE id = 1;
等待事务 2	等待事务 1
	Deadlock will be resolved after one second (deadlock_timeout)
COMMIT;	ROLLBACK;

一旦检测到死锁，就会显示以下错误消息：

```
ERROR: deadlock detected
DETAIL: Process 91521 waits for ShareLock on transaction 903;
blocked by process 77185.
```

Process 77185 waits for ShareLock on transaction 905;

blocked by process 91521.

HINT: See server log for query details.

CONTEXT: while updating tuple (0,1) in relation "t_deadlock"

PostgreSQL 甚至可以告诉我们哪一行导致了冲突。在我的例子中，所有错误的根源是一个元组编号 (0,1)。你在这里可以看到的是 ctid，它是表中行的唯一标识符。它告诉我们表格中行的物理位置。在此示例中，它是第一个块 (0) 中的第一行。

如果该行对您的事务仍然可见，您可以查询此行。下面是它的工作方式：

```
test=# SELECT ctid, * FROM t_deadlock WHERE ctid = '(0, 1)';
```

ctid	id
(0,1)	10

请记住，如果已经删除或修改了此查询，则该查询可能不会返回该行。

但是，不仅是死锁可能导致潜在的事务失败。还可能由于各种原因未对事务进行序列化。

以下示例显示了可能发生的情况。为了使示例有效，我假设你表中还有两行，`id = 1` 和 `id = 2`：

事务 1	事务 2
BEGIN ISOLATION LEVEL REPEATABLE READ;	
SELECT * FROM t_deadlock;	
返回两行	
	DELETE FROM t_deadlock;
SELECT * FROM t_deadlock;	
返回两行	
DELETE FROM t_deadlock;	
事务报错	
ROLLBACK; - we cannot COMMIT anymore	

在此示例中，两个并发事务正在运行。只要事务 1 只是查询数据，一切都很好，因为 PostgreSQL 可以很容易地保留静态数据。但是如果第二个事务提交 `DELETE` 命令会发生什么？只要是读查询事务，就没有问题。当事务 1 尝试删除或修改数据时，问题就开始了，此时数据已经真正死亡。PostgreSQL 的唯一解决方案是由于我们的事务冲突而提示错误：

```
test=# DELETE FROM t_deadlock;
```

ERROR: could not serialize access due to concurrent update

实际上，这意味着最终用户必须准备好错误事务的处理。如果出现问题，正确编写的应用程序必须具备再次尝试的功能。

利用 advisory 锁

PostgreSQL 具有高效和复杂的事务机制，能够以非常精细和高效的方式处理锁。几年前，有些人想出了使用此代码来相互同步应用程序的想法。

因此，advisory 锁就诞生了。

使用 `advisory` 锁时，重要的是要提到它们不会像普通锁一样在 `COMMIT` 上消失。因此，确保以完全可靠的方式正确完成解锁是非常重要的。

如果您决定使用 `advisory` 锁，那么您真正锁定的是一个数字。因此，这锁定的内容不是关于行或数据记录；它实际上只是一个数字。下面是它的工作方式：

事务 1	事务 2
BEGIN;	
SELECT pg_advisory_lock(15);	
	SELECT pg_advisory_lock(15);
	它必须等待
COMMIT;	仍然必须等待
SELECT pg_advisory_unlock(15);	仍然等待中
	获得锁

第一个交易将锁定 15 这个数值。第二个交易必须等到此数值被解锁。第二个会话甚至会在第一个会话执行提交之后继续等待。这非常重要，因为您不会相信事务的结束会很好地且奇迹般地为您解决解锁的问题。

如果你想解锁所有被锁定的数字，PostgreSQL 会提供 `pg_advisory_unlock_all()` 函数去实现：

```
test=# SELECT pg_advisory_unlock_all();
pg_advisory_unlock_all
-----
(1 row)
```

有时，您可能希望查看是否可以获得锁定并在可能的情况下输出错误。为实现这一目标，PostgreSQL 提供了一些功能；要查看所有此类可用功能的列表，请在命令行输入：`\df * try * advisory *`。

优化存储和管理数据清理

事务是 PostgreSQL 系统不可或缺的一部分。但是，事务的开销成本很低。如本章所示，可能会出现不同的并发用户产生不同的数据。每个人都会获得查询返回的不同数据。除此之外，`DELETE` 和 `UPDATE` 操作不允许实际意义上的物理删除更新数据，因为 `ROLLBACK` 将不会起作用。如果您恰好处于大型 `DELETE` 事务操作的中间，则无法确定是否能够进行 `COMMIT`。除此之外，当你执行 `DELETE` 的时候，数据仍然存在并可见，有时，一旦修改完成，数据甚至可以看到。

因此，这意味着数据清理必须异步进行。一个事务无法清理自己的产生的混乱数据，而 `COMMIT / ROLLBACK` 操作对于死行数据来说可能还为时尚早，无法及时有效处理。

这个问题的解决方案是 `VACUUM`。以下代码块为您提供语法概述：

```
test=# \h VACUUM
Command: VACUUM
Description: garbage-collect and optionally analyze a database
Syntax:
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [,...] ) ]
      [ table_name [ (column_name [,...]) ] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table_name ]
```

VACUUM [FULL] [FREEZE] [VERBOSE]

ANALYZE [table_name [(column_name [, ...])]]

VACUUM 将访问可能包含修改的所有页面并找到所有标记为 `dead` 的空间。然后通过数据库对象的 Free Space Map (FSM) 跟踪找到的自由可释放空间。

请注意，在大多数情况下，VACUUM 不会缩小表的大小。相反，它将跟踪并查找现有存储文件中的可用空间。



在 VACUUM 之后，表通常具有相同的大小。如果表的末尾没有有效行，则在极少数情况下文件大小可能会下降。这不是规则，而是例外。

这对最终用户意味着什么，将在本章“*查看 Vacuum 工作*”一节中概述。

配置 VACUUM 和 autovacuum

早在 PostgreSQL 项目诞生的早期，人们不得不手动运行 VACUUM。幸运的是，这已经成为过去式了。如今，管理员可以依赖一个名为 autovacuum 的工具，它是 PostgreSQL Server 基础架构的一部分。它会自动完成数据清理并在后台运行。它每分钟唤醒一次（参见 `postgresql.conf` 中的 `autovacuum_naptime = 1`）并检查是否有工作要做。如果有活干，autovacuum 将分叉三个工作进程（请参阅 `postgresql.conf` 中的 `autovacuum_max_workers`）。

主要问题是 autovacuum 何时触发创建工作进程？



实际上，autovacuum 进程本身并不分叉进程。相反，它告诉数据库主要进程这样做。这样做是为了在发生故障时避免僵尸进程并提高稳健性。

这个问题的答案可以在 `postgresql.conf` 中找到：

```
autovacuum_vacuum_threshold = 50
autovacuum_analyze_threshold = 50
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
```

`autovacuum_vacuum_scale_factor` 参数告诉 PostgreSQL 如果表 20% 的数据已被更改，则该表需要清理。麻烦的是，如果一个表只由一行组成，则一行变化率就已经是 100%。分叉一个完整的进程来清理一行是完全没有意义的。因此，`autovacuum_vacuum_threshold` 表示我们需要 20%，而这 20% 必须至少为 50 行。否则，VACUUM 将不会启动。在优化统计信息创建方面使用相同的机制。我们需要 10% 和至少 50 行来证明新的优化器统计信息的合理性。理想情况下，autovacuum 在正常的 VACUUM 期间创建新的统计信息，以避免不必要的表访问。

深入研究事务循环的相关问题

`postgresql.conf` 中还有两个设置非常重要，以便真正使用好 PostgreSQL。正如我已经说过的，理解 VACUUM 是性能的关键：

```
autovacuum_freeze_max_age = 200000000
```

autovacuum_multixact_freeze_max_age = 400000000

要理解整体问题，了解 PostgreSQL 如何处理并发非常重要。PostgreSQL 事务机制基于事务 ID 和状态事务的比较。

我们来看一个例子。如果我是事务 ID 4711 并且如果您恰好是 4712，我将不会看到您因为您仍在运行中。如果我是事务 ID 4711，但您是事务 ID 3900，我会看到您。如果您的事务失败，我可以安全地忽略失败事务产生的所有行。

麻烦如下：事务 ID 是有限的，而不是无限制的。在某些时候，他们将开始循环。实际上，这意味着事务 ID 5 实际上可能是在事务 ID 为 8 亿之后。那么 PostgreSQL 如何知道谁是第一个？它通过存储水印来实现。在某些时候，这些水印将被调整，这正是 VACUUM 开始运行相关内容的时候。通过运行 VACUUM（或 autovacuum），您可以确保将来总是有足够的事务 ID 可供使用。



并非每个事务都会增加事务 ID 计数器。只要事务仍在读取，它就只有一个虚拟事务 ID。这可确保事务 ID 不会过快消耗。

`autovacuum_freeze_max_age` 参数定义表的 `pg_class.relfrozenid` 字段在强制执行 VACUUM 操作以防止表中的事务 ID 环绕之前可以达到的最大事务数（年龄）。此值比较低，因为它还会影响 clog 清理（clog 或事务提交日志是一种数据结构，每个事务存储两 bits，用来指示事务是否正在运行，中止，已提交或仍在子事务中）。

`autovacuum_multixact_freeze_max_age` 配置表的 `pg_class.relminmxid` 字段在强制执行 VACUUM 操作之前可以达到的最大年龄以防止表中的多重事务 ID 环绕。冻结元组是一个重要的性能问题，在第 6 章优化查询以获得良好性能我们讨论查询优化方面将会有更多关于此过程的内容介绍。

通常，在保持操作安全性的同时尝试降低 VACUUM 负载是一个好主意。在大表上运行 VACUUM 的开销可能很昂贵，因此密切关注这些设置非常有意义。

关于 VACUUM FULL 的一句话

您也可以使用 VACUUM FULL，而不是普通的 VACUUM。但是，我真的想指出 VACUUM FULL 实际上锁定了表并重写了整个对象。对于小表，这可能不是问题。但是，如果你的表很大，表锁可以在几分钟内杀死你！VACUUM FULL 阻塞即将来临的数据写入，因此使用您的数据库的人可能会感觉它实际上已经被关闭。因此，建议谨慎行事。

为了摆脱 VACUUM FULL，我建议您查看 `pg_squeeze` (http://www.cybertec.at/introducing-pg_squeeze-a-postgresql-extension-to-auto-rebuild-bloated-tables/)，它可以没有阻塞的重写表。

查看 Vacuum 工作

在介绍完之后，是时候看着 VACUUM 工作了。我在这里包含了这一部分，因为我作为

PostgreSQL 顾问和支持者的实际工作 (<http://postgresql-support.de/>) 揭示了大多数人对存储方面只有非常模糊的理解。

再次强调这一点，在大多数情况下，**VACUUM** 不会缩小你的表；文件系统空间通常不会回收。

这是我的示例，演示如何使用自定义的 **autovacuum** 设置创建一个小表。该表插入了 100,000 行：

```
CREATE TABLE t_test (id int) WITH (autovacuum_enabled = off);
INSERT INTO t_test
SELECT * FROM generate_series(1, 100000);
```

我们的想法是创建一个包含 100,000 行的简单表。请注意，可以关闭指定表的 **autovacuum**。通常，对于大多数应用程序来说，这不是一个好主意。但是，有一个极端情况，其中设置 **autovacuum_enabled = off** 是有道理的。只考虑一个生命周期很短的表。如果开发人员已经知道整个表将在几秒钟内被删除，那么清除元组是没有意义的。在数据仓库中，如果您将表用作暂存区域，则可能出现这种情况。这个例子中 **VACUUM** 已关闭，可以确保后台没有操作产生。你所看到的一切都是由我而不是某个数据库进程触发的。

首先，考虑使用以下命令检查表大小：

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
3544 kB
(1 row)
```

pg_relation_size 命令以字节为单位返回表的大小。

pg_size.pretty 命令将获取此数字并将其转换为人更容易读的内容。

然后，将使用如下一个清单中所示的简单 **UPDATE** 语句更新表中的所有行：

```
test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
```

发生什么了对于理解 PostgreSQL 非常重要。数据库引擎必须复制所有行。为什么？

首先，我们不知道事务执行是否会成功，因此数据无法被全部覆盖（擦除重写）。第二个重要方面是其他并发事务可能仍然在看旧版本的数据。

UPDATE 操作将复制行。

从逻辑上讲，更改后表的大小会更大：

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
```

7080 kB

(1 row)

在 UPDATE 之后，人们可能会尝试将空间释放到文件系统：

```
test=# VACUUM t_test;
```

```
VACUUM
```

如前所述，在大多数情况下，VACUUM 不会向文件系统返回空间。相反，它将允许空间被重用。因此，该表根本没有缩小：

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
```

```
pg_size.pretty
```

```
-----
```

7080 kB

(1 row)

但是，下一个 UPDATE 不会使表增长，因为它将占用表内的空闲空间。只有第二次 UPDATE 会使表再次增长，因为所有空间都消失了，因此需要额外的存储空间：

```
test=# UPDATE t_test SET id = id + 1;
```

```
UPDATE 100000
```

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
```

```
pg_size.pretty
```

```
-----
```

7080 kB

(1 row)

```
test=# UPDATE t_test SET id = id + 1;
```

```
UPDATE 100000
```

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
```

```
pg_size.pretty
```

```
-----
```

10 MB

(1 row)

如果我必须在阅读本书后决定你应该记住的一件事，那就是它。通常来说了解存储结构是性能调整和数据库管理的关键。

让我们再运行一些查询：

```
VACUUM t_test;
```

```
UPDATE t_test SET id = id + 1;
```

```
VACUUM t_test;
```

尺寸再次保持不变。 让我们看看表中的内容：

```
test=# SELECT ctid, * FROM t_test ORDER BY ctid DESC;
```

ctid		id
------	--	----

```
-----+-----
```

```
...
(1327, 46) | 112
(1327, 45) | 111
(1327, 44) | 110
...
(884, 20) | 99798
(884, 19) | 99797
...
```

Ctid 列是行在磁盘上存储的物理位置。使用 ORDER BY ctid DESC，您基本上将按物理顺序向后读取表内容。你为什么要关心？原因是在表的末尾有一些非常小的值和一些非常大的值。以下清单显示了删除数据时表的大小如何变化：

```
test=# DELETE FROM t_test
      WHERE id > 99000
            OR id < 1000;
DELETE 1999
test=# VACUUM t_test;
VACUUM
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
3504 kB
(1 row)
```

尽管只有 2% 的数据被删除，但表格的大小却减少了三分之二。原因是如果 VACUUM 只在表中的某个位置之后找到标记为 `dead` 的行，它可以释放文件系统占用的空间。这是您看到表的大小下降的唯一情况。当然，普通用户无法控制磁盘上数据的物理位置。因此，除非删除所有行，否则存储空间很可能保持不变。



为什么表格末尾有这么多小和大的值？在表最初填充 100,000 行之后，最后一个块未完全填满，因此第一个 `UPDATE` 将填充最后一个块并进行更改。这自然会使表的末端稍微清洗了一下。在这个精心设计的示例中，这就是表末端奇怪布局的原因。

在实际应用中，这种观察的影响无法重点突出。没有真正理解存储就不能更好的调优性能。

通过使用快照过旧来限制事务

VACUUM 做得很好，它会根据需要收回自由空间。但是，什么时候 VACUUM 可以实际清理行并将它们变成自由空间？规则如下：如果任何人都不能看到一行，则可以回收它。实际上，这意味着即使是最老的活动事务也不再能看到的所有东西都可以被认为是真的死了。

这也意味着真正的长事务可以推迟清理很长一段时间。然后合乎逻辑的结果是表膨胀。表将超比例增长，性能将趋于下滑。幸运的是，从 PostgreSQL 9.6 开始，数据库有一个很好的功能，允许管理员智能地限制事务的持续时间。Oracle 数据库管理员很熟悉快照过旧的错误。自 PostgreSQL 9.6 以来，此错误消息也可用。然而，它更多的是一个特性，而不是错误配置的意外副作用（实际上它在 Oracle 中）。

要限制快照的生命周期，可以使用 PostgreSQL 的配置文件 `postgresql.conf` 中的设置，该配置文件包含所需的所有配置参数：

```
old_snapshot_threshold = -1
# 1min-60d; -1 disables; 0 is immediate
```

如果设置了此变量，则事务将在一定时间后失败。请注意，此设置是实例级别的，无法在会话中设置。通过限制事务的年龄，疯狂长期事务运行产生的风险将急剧下降。

小结

在本章中，您了解了事务，锁及其逻辑含义，以及 PostgreSQL 事务机制可用于存储，并发和管理的一般体系结构。您了解了如何锁定行以及 PostgreSQL 中可用的一些功能。

在第 3 章“使用索引”中，您将学习数据库工作中最重要的主题之一：索引。您还将了解 PostgreSQL 查询优化器，以及各种类型的索引及其行为。

QA

事务的目的是什么？

事务是任何现代关系数据库的核心。我们的想法是能够使操作成为原子。换句话说，你想要“一切或全无”。例如，如果要删除一百万行，你希望所有行或没有一行全部消失 - 您不希望被剩下的其他几行所困扰。

PostgreSQL 中的事务可以有多长时间？

最重要的是 PostgreSQL 的配置并不会真正影响事务的最大长度。因此，您可以运行基本上（几乎）无限长的事务，通过数亿个语句改变数十亿行数据。

什么是事务隔离？

并非所有事务都是平等创造的。因此，在许多情况下，您必须控制事务内数据的可见性。这正是事务隔离发挥作用的时候。 事务隔离级别允许您实现此操作。

我们应该避免表锁吗？

当然是。表锁会锁定阻塞其他所有人，这可能会导致性能问题。

您可以避免的表锁越多，从长远来看越好。 性能会受到严重影响。

事务与 VACUUM 有什么关系？

事务与 VACUUM 密切相关。整个 VACUUM 功能是在数据记录“过期”后清除它们。如

果最终用户不理解事务，则表膨胀对他们来说就是问题。

Javanchueng

3. 使用索引

在第 2 章“了解事务和锁”中，您了解了并发和锁。在本章中，是时候向索引进攻了。这个主题的重要性不言而喻 - 索引是（并且很可能仍然是）每个数据库工程师生涯中的最重要的一个主题。

经过 18 年的专业，全职 PostgreSQL 咨询和 PostgreSQL 24x7 支持 (www.cybertec-postgresql.com)，我可以说有一点肯定 - 糟糕的索引是糟糕性能的主要来源。当然，内存参数调整很重要。但是，如果没有正确使用索引，那都是徒劳的。索引的缺失没有替代方案。为了说明我的观点：如果没有适当的索引，就无法获得良好的性能 - 因此，如果性能不好，请务必检查索引。

这就是将整章专门用于索引的原因。还将为您提供尽可能多的见解。

在本章中，我们将介绍以下主题：

- PostgreSQL 什么时候使用索引？
- 优化器如何处理事情？
- 有哪些类型的索引以及它们如何工作？
- 使用您自己的索引策略

在本章的最后，您将能够理解如何在 PostgreSQL 中有效地使用索引。

了解简单查询和成本模型

在本节中，我们将开始使用索引。要了解工作原理，需要一些测试数据。以下代码段显示了如何轻松创建数据：

```
test=# DROP TABLE IF EXISTS t_test;
DROP TABLE
test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test (name) SELECT 'hans'
      FROM generate_series(1, 2000000);
INSERT 0 2000000
test=# INSERT INTO t_test (name) SELECT 'paul'
      FROM generate_series(1, 2000000);
INSERT 0 2000000
```

在第一行中，创建一个简单的表。使用两列；第一个是自增列，它不断的生成数字，第二个是一个填充静态值的列。



generate_series 函数将生成从 1 到 2 百万的数字。因此，在此示例中，为 hans 创建了 200 万个静态值，为 paul 创建了 200 万个静态值。

总共增加了 400 万行：

```
test=# SELECT name, count(*) FROM t_test GROUP BY 1;
   name | count
-----+
  hans | 2000000
  paul | 2000000
(2 rows)
```

这 400 万行有一些很好的属性，我们将在本章中使用它们。ID 是升序的，只有两个不同的名称。

我们现在运行一个简单的查询：

```
test=# \timing
Timing is on.
test=# SELECT * FROM t_test WHERE id = 432332;
   id  | name
-----+
 432332 | hans
(1 row)
Time: 176.949 ms
```

在这种案例下， timing 命令将告诉 psql 显示查询的运行时间。



这不是服务器上的实际执行时间，而是 psql 测量的时间。如果查询非常短，网络延迟可能是总时间的重要部分，因此必须将其考虑在内。

使用 EXPLAIN

在这个例子中，读取 400 万行的时间超过 100 毫秒。从性能的角度来看，这是一场灾难。

为了弄清楚出了什么问题，PostgreSQL 提供了 EXPLAIN 命令：

```
test=# \h EXPLAIN
Command: EXPLAIN
Description: show the execution plan of a statement
Syntax:
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
where option can be one of:
  ANALYZE [ boolean ]
  VERBOSE [ boolean ]
  COSTS [ boolean ]
  BUFFERS [ boolean ]
  TIMING [ boolean ]
  FORMAT { TEXT | XML | JSON | YAML }
```

当您感觉查询效果不佳时， EXPLAIN 将帮助您揭示真正的性能问题。

下面是它的工作原理：

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 432332;
```

QUERY PLAN

Gather (cost=1000.00..43463.92 rows=1 width=9)

Workers Planned: 2

-> Parallel Seq Scan on t_test

(cost=0.00..42463.82 rows=1 width=9)

Filter: (id = 432332)

(4 rows)

您在此列表中看到的是执行计划。在 PostgreSQL 中，SQL 语句将分四个阶段执行。以下组件负责工作：

1. 解析器将检查语法错误和明显的问题
2. 重写系统负责规则（视图和其他事情）
3. 优化器将弄清楚如何以最有效的方式执行查询并制定计划
4. 优化程序提供的计划将由执行程序用于最终创建结果

EXPLAIN 的目的是查看 planner 提出的有效运行查询的内容。在我的例子中，PostgreSQL 将使用并行顺序扫描。这意味着两个 workers 将合作并共同处理过滤条件。部分结果然后通过称为 gather 节点的东西（这已经在 PostgreSQL 9.6 中引入（它是并行查询基础体系的一部分））联合起来。如果你更精确地看一下这个计划，你会看到 PostgreSQL 在计划的每个阶段都期望有多少行（在此例如，rows = 1；也就是说，将返回一行）。



TIP 在 PostgreSQL 9.6 到 10.0 中，并行工作者的数量将是表的大小决定。执行的操作越大，PostgreSQL 引发的并行工作就越多。对于一个非常小的表，不使用并行，因为它会产生太多的开销。

并行操作不是必须的。通过将以下变量设置为 0，始终可以减少并行工作者的数量以模仿 PostgreSQL 9.6 之前的行为：

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
```

请注意，此更改没有副作用，因为它仅在您的会话中设置。当然，您也可以在 postgresql.conf 文件中进行更改，但我不建议您这样做，因为您可能会因此失去并行查询提供的额外性能。

深入研究 PostgreSQL 成本模型

如果只使用一个 CPU，执行计划将如下所示：

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 432332;
                                         QUERY PLAN
```

Seq Scan on t_test (cost=0.00..71622.00 rows=1 width=9)

Filter: (id = 432332)

(2 rows)

PostgreSQL 将按顺序读取（顺序扫描）整个表并应用过滤器。它预计该行动将花费 71622

点罚分。现在，这是什么意思？ 罚分（或成本）主要是抽象概念。需要使用它们来比较执行不同的查询方式。如果 excutoer 可以通过多种不同的方式执行查询，PostgreSQL 将决定承诺使用尽可能低的成本的执行计划。现在的问题是 PostgreSQL 如何得到 71622 分？

下面是它的工作原理：

```
test=# SELECT pg_relation_size('t_test') / 8192.0;  
?column?  
-----  
21622.000000  
(1 row)
```

`pg_relation_size` 函数将返回表的大小（以字节为单位）。给出该示例，您可以看到该对象由 21622 个块（每个 8K）组成。根据成本模型，PostgreSQL 将为它必须按顺序读取的每个块增加一个成本。

影响的配置参数如下：

```
test=# SHOW seq_page_cost;  
seq_page_cost  
-----  
1  
(1 row)
```



```
test=# SHOW cpu_tuple_cost;  
cpu_tuple_cost  
-----  
0.01  
(1 row)
```

```
test=# SHOW cpu_operator_cost;  
cpu_operator_cost  
-----  
0.0025  
(1 row)
```

这导致以下计算：

```
test=# SELECT 21622*1 + 4000000*0.01 + 4000000*0.0025;  
?column?  
-----  
71622.0000  
(1 row)
```

如您所见，这正是执行计划中的数字。成本将包括 CPU 部分和 I/O 部分，它们将全部转换为单个数字。这里重要的是成本与实际执行无关，因此无法将成本转换为毫秒。`planner` 提出的数字实际上只是一个估计值。

当然，这个简短的例子中还列出了一些参数。PostgreSQL 还有与索引相关的操作的特殊

参数，如下所示：

- `random_page_cost = 4`: 如果 PostgreSQL 使用索引，通常会涉及很多随机 I/O 访问。在传统的 HDD 磁盘上，随机读取比顺序读取更重要，因此 PostgreSQL 会相应地考虑它们。请注意，在 SSD 上，随机读取和顺序读取之间的差异不再存在，因此在 `postgresql.conf` 文件中设置 `random_page_cost = 1` 是有意义的。
- `cpu_index_tuple_cost = 0.005`: 如果使用索引，PostgreSQL 也会考虑有一些 CPU 成本。

如果您正在使用并行查询，则会有更多成本参数：

- `parallel_tuple_cost = 0.1`: 这定义了将一个元组从并行工作进程转移到另一个进程的成本。它基本上解释了在进程体系内部移动行的开销。
- `parallel_setup_cost = 1000.0`: 这可以调整启动 `worker` 进程的成本。当然，启动并行运行查询的过程并不是免费的，因此，此参数会尝试对与流程管理相关的成本进行建模。
- `min_parallel_tables_scan_size = 8 MB`: 这定义了为并行查询考虑的表的最小大小。表越大，PostgreSQL 将使用的 CPU 越多。表的大小必须增加三倍才能再增加一个 `worker`。
- `min_parallel_table_scan_size = 512kB`: 这定义了索引的大小，这是考虑并行扫描所必需的。

部署简单索引

启动更多 `worker` 进程来扫描更大的表有时候不是最好的解决方案。读取整个表的记录来查找某行通常不是一个好的主意。

因此，创建索引就变得有意义了：

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
test=# SELECT * FROM t_test WHERE id = 43242;
 id | name
----+
 43242 | hans
(1 row)
Time: 0.259 ms
```

PostgreSQL 使用 Lehman-Yao 的高并发 btree 作为标准索引 (<https://www.csd.uoc.gr/~hy460/pdf/p650-lehman.pdf>)。伴随着 PostgreSQL 特定的优化措施，这些 btree 索引最终用户提供了出色的性能。最重要的是 Lehman-Yao 索引允许您同时在同一个索引上运行许多操作（读取和写入），这有助于显著提高吞吐量。

但是，索引不是无偿的：

```
test=# \di+
      List of relations
 Schema | Name   | Type  | Owner | Table | Size | Description
-----+-----+-----+-----+-----+-----+
```

```
public | idx_id | index | hs      | t_test | 86 MB |
(1 row)
```

如您所见，包含 400 万行的索引将占用 86 MB 的磁盘空间。除此之外，对表的写入速度会变慢，因为索引必须始终保持数据同步。

换句话说，如果您往一个包含 20 个索引的表中插入记录，您还必须记住，我们必须在 `INSERT` 的同时写入所有这些索引，这会严重降低写入速度。



随着版本 11 的推出，PostgreSQL 现在支持并行索引创建。可以利用多个 CPU core 来构建索引，从而大大加快了创建进程。目前，这只有在你想创建一个普通的 btree 时才有可能，还不支持其他的索引类型。但是，这很可能在未来发生变化。控制创建并行度的参数是 `max_parallel_maintenance_workers`。它告诉 PostgreSQL 最多有多少进程可以用来并行创建。

使用排序输出

Btree 索引不仅用于查找行；它们还用于将已排序的数据提供给流程中的下一个阶段：

```
test=# EXPLAIN SELECT *
  FROM  t_test
  ORDER BY id DESC
  LIMIT 10;
                                         QUERY PLAN
-----
Limit (cost=0.43..0.74 rows=10 width=9)
  -> Index Scan Backward using idx_id on t_test
      (cost=0.43..125505.43 rows=4000000 width=9)
(2 rows)
```

在这个案例中，索引已经以正确的排序顺序返回数据，因此无需对整个数据集进行排序。读取索引的最后十行将满足此查询的要求。实际上，这意味着可以在几分之一毫秒内找到表的前 N 行记录。

但是，`ORDER BY` 并不是唯一需要排序输出的操作。`min` 和 `max` 函数也都是关于排序输出的，因此索引也可用于加速这两个操作。这里是一个例子：

```
test=# explain SELECT min(id), max(id) FROM t_test;
                                         QUERY PLAN
-----
Result (cost=0.93..0.94 rows=1 width=8)
  InitPlan 1 (returns $0)
    -> Limit (cost=0.43..0.46 rows=1 width=4)
      -> Index Only Scan using idx_id on t_test
          (cost=0.43..135505.43 rows=4000000 width=4)
          Index Cond: (id IS NOT NULL)
```

```
InitPlan 2 (returns $1)
-> Limit (cost=0.43..0.46 rows=1 width=4)
    -> Index Only Scan Backward using idx_id on t_test t_test_1
        (cost=0.43..135505.43 rows=4000000 width=4)
        Index Cond: (id IS NOT NULL)
(9 rows)
```

在 PostgreSQL 中，索引（btree，更准确）可以按照正常顺序读取或者向后读取。现在可以将 btreet 看作是一个排序过的列表。因此，很自然地，最小值在列表的开始处，最大值在最后处。因此，min 和 max 是通过 btreet 查询加速的受益者。值得注意的是，在这种情况下，根本不需要查询主表的记录。

在 SQL 中，许多操作依赖于排序输入；因此，了解这些操作至关重要，因为对索引有严重影响。

一次使用多个索引

到目前为止，您已经看到一次使用了一个索引。然而，在许多现实世界的情况下，这远远不够。有些情况需要在数据库中使用更多逻辑。

PostgreSQL 允许在单个查询中使用多个索引。当然，如果同时查询许多列，这是有意义的。然而，情况并非总是如此。也可能会多次使用单个索引来处理同一列。

这是一个例子：

```
test=# explain SELECT * FROM t_test WHERE id = 30 OR id = 50;
          QUERY PLAN
-----
Bitmap Heap Scan on t_test (cost=8.88..16.85 rows=2 width=9)
  Recheck Cond: ((id = 30) OR (id = 50))
    -> BitmapOr (cost=8.88..8.88 rows=2 width=0)
      -> Bitmap Index Scan on idx_idv
          (cost=0.00..4.44 rows=1 width=0)
          Index Cond: (id = 30)
      -> Bitmap Index Scan on idx_id (cost=0.00..4.44 rows=1 width=0)
          Index Cond: (id = 50)
(7 rows)
```

这里的要点是 id 列需要两次查找。首先，查找 30，然后查找 50。如您所见，PostgreSQL 将进行位图扫描。



位图扫描与位图索引不同，具有 Oracle 使用背景的人可能知道。它们是两个完全不同的东西，没有任何共同之处。位图索引是 Oracle 中的索引类型，而位图扫描是扫描方法。

位图扫描背后的实现原理是 PostgreSQL 将扫描第一个索引，收集包含数据的块列表（= 表的页块）。然后，将扫描下一个索引以再次汇集块列表。该操作取决于扫描所需索引数量。在 OR 的情况下，统一这些块列表，给我们留下了包含数据的大量块列表。通过使用此列表，将扫描表以检索需要的数据块。

现在麻烦的是 PostgreSQL 检索到的数据超过了需要的数量。在我们的例子中，查询将查找两行；但是，位图扫描可能会返回几个块。因此，executor 将重新检查以过滤掉那些不满足我们条件的行。

位图扫描也适用于 AND 条件或 AND 和 OR 的混合。但是，如果 PostgreSQL 看到一个 AND 查询条件，它不一定就会强迫使用位图扫描。假设我们有一个查询，寻找生活在奥地利的每个人和一个具有特定 ID 的人。在这里使用两个索引真的没有意义，因为在完成 ID 搜索之后，实际上没有太多数据。扫描两个索引会更加昂贵，因为有八百万人（包括我）住在奥地利，从性能的角度来看，阅读这么多行只找一个人是没有意义的。好消息是 PostgreSQL 优化器将通过比较不同选项和潜在索引的成本为您做出所有这些决定，因此无需担心。

有效地使用位图扫描

现在自然产生的问题是，位图扫描何时最有益，何时由优化器选择？从我的角度来看，实际上只有两个情况：

- 避免一遍又一遍地取出同一个块
- 结合相对恶劣的条件

第一种情况很常见。假设您正在寻找说某种语言的所有人。为了举例，我们可以假设所有人中有 10% 的人说所需的语言。扫描索引意味着必须再次扫描表中的块，因为许多熟练的演讲者（注：说某种语言的人）可能存储在同一块中。通过使用位图扫描，确保特定块仅使用一次，这当然会导致更好的性能。

第二个常见用例是将相对较小的查询准则联合一起使用。让我们假设我们正在寻找 20 到 30 岁之间拥有一件黄色衬衫的人。现在，可能有 15% 的人在 20 到 30 岁之间，可能有 15% 的人实际拥有黄色衬衫。顺序扫描表是很昂贵的，因此 PostgreSQL 可能决定选择两个索引，因为最终结果可能只包含 1% 的数据。扫描两个索引可能比读取所有数据开销更低。

在 PostgreSQL 10.0 中，支持并行位图扫描。通常，位图扫描由相对昂贵的查询使用。因此，在这一领域增加并行性是向前迈出的一大步，绝对有益。

以相对智能的方式使用索引

到目前为止，应用索引感觉就像圣杯一样，它总能神奇地改善性能。然而，这种情况并非一直如此。在某些情况下，索引使用也可能毫无意义。

在深入研究之前，这里是我们用于此示例的数据结构。请记住，只有两个不同的列名和唯一 IDs：

```
test=# \d t_test
          Table "public.t_test"
  Column | Type      | Modifiers
-----+-----+
    id   | integer   | not null default nextval('t_test_id_seq'::regclass)
   name  | text      |
Indexes:
"idx_id" btree (id)
```

此时，已定义一个索引，其中包含 id 列。在下一步中，将查询 name 列。在执行此操作之前，将创建基于 name 列的索引：

```
test=# CREATE INDEX idx_name ON t_test (name);
CREATE INDEX
```

现在，是时候查看索引是否正确使用了：

```
test=# EXPLAIN SELECT * FROM t_test WHERE name = 'hans2';
          QUERY PLAN
-----
Index Scan using idx_name on t_test
(cost=0.43..4.45 rows=1 width=9)
  Index Cond: (name = 'hans2'::text)
(2 rows)
```

正如所料，PostgreSQL 将决定使用索引查找。大多数用户都会期待这一点 但请注意，我的查询条件是 hans2。请记住，表中不存在 hans2，查询计划完美地反映了这一点。rows = 1 表示 planner 预期查询返回的数据集非常小。



表中没有一行记录，但 PostgreSQL 永远不会估计零行，因为它会使后续估计变得更加困难，执行计划中其他节点的成本计算变得几乎是不可能的。

让我们看看如果我们查询更多数据会发生什么：

```
test=# EXPLAIN SELECT *
  FROM t_test
 WHERE name = 'hans'
   OR name = 'paul';
          QUERY PLAN
-----
Seq Scan on t_test (cost=0.00..81622.00 rows=3000011 width=9)
  Filter: ((name = 'hans'::text) OR (name = 'paul'::text))
(2 rows)
```

在这种情况下，PostgreSQL 将进行直接顺序扫描。这是为什么？为什么系统会忽略所有索引？原因很简单：hans 和 paul 组成整个数据记录，因为没有其他值（PostgreSQL 通过检查系统统计信息知道）。因此，PostgreSQL 认为无论如何都必须读取整个表。如果只读取表就够了，那就没有理由去读取索引再整个表了。

换句话说，PostgreSQL 不会因为有一个索引就使用该索引。PostgreSQL 会在有意义的时候使用索引。如果数据行数较小，PostgreSQL 将会考虑使用位图扫描和正常索引扫描：

```
test=# EXPLAIN SELECT *  
      FROM t_test  
     WHERE name = 'hans2'  
          OR name = 'paul2';
```

Bitmap Heap Scan on t_test (cost=8.88..12.89 rows=1 width=9)
Recheck Cond: ((name = 'hans2'::text) OR (name = 'paul2'::text))
-> BitmapOr (cost=8.88..8.88 rows=1 width=0)
 -> Bitmap Index Scan on idx_name
 (cost=0.00..4.44 rows=1 width=0)
 Index Cond: (name = 'hans2'::text)
 -> Bitmap Index Scan on idx_name
 (cost=0.00..4.44 rows=1 width=0)
 Index Cond: (name = 'paul2'::text)

这里要学习的最重要的一点是执行计划的生成取决于输入的值。

它们不是静态的，不依赖于表中的数据。这是一个非常重要的现象，必须始终牢记在心。在实际示例中，执行计划更改通常是数据库运行变幻莫测的原因。

使用群集表提高速度

在本节中，您将了解相关性（correlation）的强大功能和集群表的强大功能。整个想法是什么？考虑您想要读取整个数据区域。这可能是某个时间范围，某些块，ID 等。

此类查询的运行时间将根据数据量和磁盘上数据的物理排列而有所不同。因此，即使您正在运行返回相同行数的查询，两个系统也可能无法在同一时间范围内给出结果，如物理磁盘布局可能会有所不同。

这是一个例子：

```
test=# EXPLAIN (analyze true, buffers true, timing true)
      SELECT *
        FROM t_test
       WHERE id < 10000;
```

```
Index Scan using idx_id on t_test
  (cost=0.43..370.87 rows=10768 width=9)
  (actual time=0.011..2.897 rows=9999 loops=1)
    Index Cond: (id < 10000)
    Buffers: shared hit=85
Planning time: 0.078 ms
Execution time: 4.081 ms
(5 rows)
```

您可能还记得，数据已经按照有组织的顺序方式加载。数据在 ID 之后不断添加新的 ID，因此可以预期数据将按顺序排列在磁盘上。如果使用某些自增列将数据加载到空表中则会实现。

你已经看到 EXPLAIN 的操作了。在此示例中，使用了 EXPLAIN(analyze true, buffers true, timing true)。我们的想法是，分析选项 (analyze) 不仅会显示计划，还会执行查询并告诉我们发生了什么。

EXPLAIN 分析非常适合将 planner 估算值与实际情况进行比较。这是了解 planner 是否正确或偏离的最佳方法。buffers true 参数将告诉我们查询遍历了多少 8 k 块。在此示例中，遍历了总共 85 个块。共享命中意味着数据来自 PostgreSQL I/O 缓存（共享缓冲区）。总而言之，PostgreSQL 用了 4 毫秒来检索数据。

如果表中的数据存储有些随机，会发生什么？事情会改变吗？

要创建一个包含相同数据但以随机顺序排列的表，您只需使用 ORDER BY random() 即可。它将确保数据确实在磁盘上杂乱无序：

```
test=# CREATE TABLE t_random AS SELECT * FROM t_test ORDER BY random();
SELECT 4000000
```

为确保公平比较，将创建索引相同的列：

```
test=# CREATE INDEX idx_random ON t_random (id);
CREATE INDEX
```

要准确运行，PostgreSQL 需要优化器的统计信息。这些统计信息将告诉 PostgreSQL 有多少数据，数据值的分布情况，以及数据是否在磁盘上相关。为了加快速度，我执行了一个 VACUUM 操作。请注意 VACUUM 将在本书后面更深入地讨论：

```
test=# VACUUM ANALYZE t_random;
VACUUM
```

现在，让我们运行与以前相同的查询：

```
test=# EXPLAIN (analyze true, buffers true, timing true)
SELECT * FROM t_random WHERE id < 10000;
QUERY PLAN
```

```
Bitmap Heap Scan on t_random
  (cost=203.27..18431.86 rows=10689 width=9)
  (actual time=5.087..13.822 rows=9999 loops=1)
    Recheck Cond: (id < 10000)
    Heap Blocks: exact=8027
    Buffers: shared hit=8057
    -> Bitmap Index Scan on idx_random
        (cost=0.00..200.60 rows=10689 width=0)
        (actual time=3.558..3.558 rows=9999 loops=1)
        Index Cond: (id < 10000)
        Buffers: shared hit=30
Planning time: 0.075 ms
Execution time: 14.411 ms
(9 rows)
```

这里有几点需要注意。首先，需要总共 8,057 个块，运行时间已超过 14 毫秒。这里唯一不同的事情是，有些挽回颜面的表现是数据是从内存访问的而非磁盘。试想一下，如果你需要访问磁盘 8,057 次来获得查询结果意味着什么。这将是一场彻底的灾难，因为磁盘等待延迟肯定会大大降低查询速度。

但是，还有更多可以关注的内容。您甚至可以看到执行计划已变更。PostgreSQL 现在使用位图扫描而不是普通的索引扫描。这样做是为了减少查询中所需的块数，以防止更糟糕的行为。

planner 如何知道数据如何存储在磁盘上？ pg_stats 是一个系统视图，包含有关列内容的所有统计信息。以下查询显示相关内容：

```
test=# SELECT tablename, attname, correlation
  FROM pg_stats
 WHERE tablename IN ('t_test', 't_random')
 ORDER BY 1, 2;
tablename | attname | correlation
-----+-----+
t_random | id      | -0.0114944
t_random | name    |  0.493675
t_test   | id      |       1
t_test   | name    |       1
(4 rows)
```

你可以看到 PostgreSQL 处理每一列。视图的内容由名为 ANALYZE 的功能创建，这对性能至关重要：

```
test=# \h ANALYZE
Command: ANALYZE
Description: Collect statistics about a database
Syntax:
```

ANALYZE [VERBOSE] [table_name [(column_name [, ...])]]

通常, ANALYZE 会在后台使用 auto-vacuum 守护程序自动执行, 本书稍后将对此进行介绍。

回到我们的查询。如您所见, 两个表都有两列 (`id` 和 `name`)。在 `t_test.id` 的案例中, 相关性为 1, 这意味着下一个值在某种程度上取决于前一个值。在我的例子中, 数字只是简单的升序递增。这同样适用于 `t_test.name`。首先, 我们有包含 `hans` 的条目, 然后我们有包含 `paul` 的条目。因此, 所有相同的名称都存储在一起。

在 `t_random` 中, 情况完全不同; 负相关性意味着数据被打乱。您还可以看到名称列的相关性约为 0.5。实际上, 这意味着表中通常没有直接排序的相同名称的列值, 这意味着当按照物理顺序读取表时, 查询的列名称值会一直变化。

为什么这会导致查询中命中如此多的块? 答案相对简单。如果我们需要的数据没有紧密地存储在一起, 而是在表中均匀分布, 则需要更多的块来获取相同数量的内容, 这反过来又会导致性能下降。

群集表

在 PostgreSQL 中, 有一个名为 CLUSTER 的命令允许我们以所需的顺序重写表。可以指向索引并以与索引相同的顺序存储数据:

```
test=# \h CLUSTER
Command: CLUSTER
Description: cluster a table according to an index
Syntax:
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

CLUSTER 命令已存在多年, 并且很好地发挥其作用。但是, 在生产系统上盲目运行之前, 有一些事情需要考虑:

- CLUSTER 命令将在表运行时锁定该表。在 CLUSTER 运行时, 您无法插入或修改数据。这在生产系统上可能是不可接受的。
- 数据只能根据一个索引进行组织。您不能同时按邮政编码, 姓名, 身份证, 生日等作为表排序的关键字。这意味着使用在大多数时间作为搜索条件的列, CLUSTER 将更有意义。
- 请记住, 本书中概述的示例更多是最糟糕的场景。实际上, 群集表和非群集表之间的性能差异将取决于工作负载, 检索的数据量, 缓存命中率以及其他更多。
- 在正常操作期间对表进行更改时, 将不会保留表的聚簇状态。随着时间的推移, 相关性通常会恶化。

以下是如何运行 CLUSTER 命令的示例:

```
test=# CLUSTER t_random USING idx_random;
CLUSTER
```

根据表的大小，`cluster` 命令所需的执行时间会有所不同。

使用仅索引扫描

到目前为止，您已经看到了何时使用索引以及何时使用索引。除此之外，还讨论了位图扫描。

但是，索引还有很多内容。以下两个示例仅略有不同，但性能差异可能相当大。这是第一个查询：

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 34234;
                                QUERY PLAN
-----
Index Scan using idx_id on t_test
(cost=0.43..8.45 rows=1 width=9)
Index Cond: (id = 34234)
```

这里没什么不寻常的。PostgreSQL 使用索引来查找单行。如果只查询一列，又会发生什么呢？

```
test=# EXPLAIN SELECT id FROM t_test WHERE id = 34234;
                                QUERY PLAN
-----
Index Only Scan using idx_id on t_test
(cost=0.43..8.45 rows=1 width=4)
Index Cond: (id = 34234)
(2 rows)
```

如您所见，该计划已从索引扫描更改为仅索引扫描。在我们的示例中，`id` 列已被索引，因此其内容自然位于索引中。如果所有数据都已从索引中取出，则在大多数情况下无需再回到表中查找。如果查询了其他字段，则（几乎）需要回表查询访问该表，但这不是这个例子讨论的情况。因此，仅索引扫描将比正常索引扫描明显具备更好的性能。

实际上，在这里或者那里添加一个额外的列到索引是有意义的，可以享受到使用此功能的好处。在 MS SQL 中，我们都应该添加其他列到索引被称为覆盖索引。从 PostgreSQL 11 开始，我们具有相同的功能，它在 `CREATE INDEX` 中使用 `INCLUDE` 关键字。

理解其他 btree 功能

在 PostgreSQL 中，索引是一个很大的领域，涵盖了数据库工作的许多方面。正如我在本书中已经概述的那样，索引是性能的关键。没有适当的索引就没有好的表现。因此，值得更详细地检查这些索引相关的功能。

组合索引

在我的工作中，作为专业的 PostgreSQL 支持供应商，我经常被问及组合索引和单个索引之间的区别。在本节中，我将尝试阐明这个问题。

一般规则是，如果单个索引可以满足您的查询问题，通常是最佳选择。但是，您无法索引人们查询过滤的所有可能的字段组合。您可以做的是使用组合索引的属性来获得尽可能多的收益。

假设我们有一个包含三列的表：`postal_code`, `last_name` 和 `first_name`。电话簿将使用这样的组合索引。您将看到数据按位置排序。在同一位置，数据将按姓氏和名字排序。

下表显示了如果给定三列组合索引的可能操作：

Query	Possible	Remarks
<code>postal_code = 2700 AND last_name = 'Schönig' AND first_name = 'Hans'</code>	Yes	这是该索引的理想用例。
<code>postal_code = 2700 AND last_name = 'Schönig'</code>	Yes	无限制。
<code>last_name = 'Schönig AND postal_code = 2700</code>	Yes	PostgreSQL 将简单地交换条件。
<code>postal_code = 2700</code>	Yes	这就像在 <code>postal_code</code> 列的索引一样；只是组合索引需要更多磁盘空间。
<code>first_name = 'Hans'</code>	Yes, but a different use case	PostgreSQL 不能使用的排序属性索引了。但是，在一些罕见的情况下（通常非常广泛的表，包括无数的列）PostgreSQL 将扫描整个索引如果它和读取宽表的成本一样。

如果列被单独创建索引，您很可能最终会看到位图扫描。当然，手工定制的单个索引更好。

添加函数索引

到目前为止，您已经了解了如何对列的内容进行索引。但是，这可能并不总是你真正想要的。因此，PostgreSQL 允许创建函数索引。基本思路很简单；相比较索引一个列的值而言，函数的输出值存储在索引中。

以下示例显示如何索引 id 列的余弦值：

```
test=# CREATE INDEX idx_cos ON t_random (cos(id));
CREATE INDEX
test=# ANALYZE;
ANALYZE
```

您所要做的就是将该函数放在索引列列表中，您就完成了。当然，这不适用于所有类型的函数。只有有固定输出值的函数才能使用函数索引。

```
test=# SELECT age('2010-01-01 10:00:00'::timestamptz);
          age
-----
 6 years 9 mons 14:00:00
(1 row)
```

像 `age` 这样的函数并不适合索引，因为它们的输出不是常量。时间一直在变化，因此，`age` 函数的输出也会改变。PostgreSQL 将明确禁止在给定相同输入的情况下有可能改变其结果的函数创建函数索引。`cos` 函数在这方面就很好，因为从现在起 1000 年内，一个值的余弦仍然是相同的。

为了测试函数索引，我编写了一个简单的查询来显示将要发生的事情：

```
test=# EXPLAIN SELECT * FROM t_random WHERE cos(id) = 10;
               QUERY PLAN
-----
 Index Scan using idx_cos on t_random (cost=0.43..8.45 rows=1 width=9)
   Index Cond: (cos((id)::double precision) = '10'::double precision)
(2 rows)
```

正如所料，函数索引将像任何其他索引一样使用。

减少空间消耗

索引很好，其主要目的是尽可能加快查询速度。与所有好东西一样，索引也附带有额外的开销：空间消耗。为了发挥其魔力，索引必须以有组织的方式存储值。如果您的表包含 1000 万个整数值，则属于该表的索引将在逻辑上包含这些 1000 万个整数值以及额外的开销。

`btree` 将包含指向表中每一行的指针，因此它肯定不是免费的。要确定索引需要多少空间，可以使用 `\di+` 命令询问 `psql`：

```
test=# \di+
              List of relations
 Schema | Name    | Type  | Owner | Table | Size
-----+-----+-----+-----+-----+
 public | idx_cos | index | hs    | t_random | 86 MB
 public | idx_id  | index | hs    | t_test   | 86 MB
 public | idx_name | index | hs    | t_test   | 86 MB
```

```
public | idx_random | index | hs      | t_random | 86 MB
(4 rows)
```

在我的数据库中，为了存储这些索引，已经消耗了惊人的 344 MB。现在，将其与索引基表的存储量进行比较：

```
test=# \d+
                                         List of relations
 Schema | Name           | Type   | Owner | Size
-----+-----+-----+-----+
 public | t_random      | table  | hs    | 169 MB
 public | t_test        | table  | hs    | 169 MB
 public | t_test_id_seq | sequence | hs    | 8192 bytes
(3 rows)
```

两个表的总大小仅为 338 MB。换句话说，我们的索引需要比实际数据更多的存储空间。在现实世界中，这是常见的，实际上非常可能。最近，我访问了德国的一个 Cybertec 客户，我看到了一个数据库，其中 64% 的数据库大小由从未使用过的索引组成（不是一次而是超过几个月的时间）。因此，过度索引可能就像索引不足一样。请记住，这些索引不仅消耗空间。每个 INSERT 或 UPDATE 都必须维护这些值，在索引中也是如此。在极端情况下，例如我们的示例，这极大地降低了写入吞吐量。

如果表中只有少数不同的值，则部分索引是一种解决方案：

```
test=# DROP INDEX idx_name;
DROP INDEX
test=# CREATE INDEX idx_name ON t_test (name)
      WHERE name NOT IN ('hans', 'paul');
CREATE INDEX
```

在这种情况下，大部分值已被排除在索引之外，可以享受一个小而有效的索引带来的收益：

```
test=# \di+ idx_name
                                         List of relations
 Schema | Name           | Type   | Owner | Table | Size
-----+-----+-----+-----+-----+
 public | idx_name      | index  | hs    | t_test | 8192 bytes
(1 row)
```

请注意，排除构成表的大部分的非常频繁的值（至少 25% 左右）才有意义。理想的部分排除值可能是性别（我们假设大多数人是男性或女性），国籍（假设您所在国家的大多数人具有相同的国籍），等等。当然，应用这种技巧需要对您的数据有一些深入的理解，但它肯定会有所回报。

索引时添加数据

创建索引很容易。但是，请记住，在构建索引时无法修改表。`CREATE INDEX` 命令将使用 `SHARE` 锁定锁定表，以确保不会发生任何更改。虽然这对于小型表来说显然没有问题，但它会对生产系统上的大型表造成问题。索引数 `TB` 左右的数据需要一些时间，因此，长时间阻塞表可能会成为一个问题。

该问题的解决方案是 `CREATE INDEX CONCURRENTLY` 命令。构建索引将花费更长的时间（通常至少两倍），但您可以在索引创建期间正常使用该表。

下面是它的工作方式：

```
test=# CREATE INDEX CONCURRENTLY idx_name2 ON t_test (name);
CREATE INDEX
```

请注意，如果您使用 `CREATE INDEX CONCURRENTLY` 命令，PostgreSQL 不保证能执行成功。如果系统上正在进行的操作与索引创建发生冲突，则索引最终可能会被标记为无效。如果要检查索引是否无效，请在表对象上使用 `\d` 命令。

介绍运算符类

到目前为止，目标是找出要索引的内容并盲目地在此列或一组列上创建索引。然而，有一个假设是，我们默默地接受了这项工作。到目前为止，我们已经假设必须按序排序的数据是一个有点固定的常数。实际上，这种假设可能不成立。当然，数字将始终处于相同的顺序，但其他类型的数据很可能没有按照预定义的固定顺序排序。

为了证明我的观点，我编写了一个真实的例子。看看以下两条记录：

```
1118 09 08 78
2345 01 05 77
```

我现在的问题是，这两行是否正确排序？他们可能因为一个比另外一个先到。但是，这是错误的，因为这两行确实有一些隐藏的语义。你在这里看到的是两个奥地利社会安全号码。`09 08 78` 实际上是指 1978 年 8 月 9 日，`01 05 77` 实际上是指 1977 年 5 月 1 日。前四个数字由校验和和某种自动递增的三位数字组成。所以实际上，1977 年是在 1978 年之前，我们可能会考虑交换这两行来实现所需的排序顺序。

问题是 PostgreSQL 不知道这两行究竟是什么意思。如果列被标记为文本，PostgreSQL 将应用标准规则对文本进行排序。如果列被标记为数字，PostgreSQL 将应用标准规则来排序数字。在任何情况下，它都不会使用像我所描述的奇怪的东西。如果您认为我之前概述的事实是处理这些数字时唯一需要考虑的事情，那么您就错了。一年有几个月？12？远非真实。在奥地利社会保障体系里面，这些数字可以持有长达 14 个月。为什么？记住，……三位数字只是一个自动增量值。麻烦的是，如果移民或难民没有有效的纸质文书，如果他的生日不知道，他将会在第 13 个月被分配一个人工生日。

在 1990 年的巴尔干战争期间，奥地利向超过 115,000 名难民提供了庇护。当然，这个

三位数字还不够，还增加了第 14 个月。现在，从 20 世纪 70 年代早期开始，哪种标准数据类型可以处理这种类型的 COBOL-leftover（引入社会安全号码的时候）？答案是，没有。

为了以合理的方式处理特殊用途的字段，PostgreSQL 提供了运算符类：

```
test=# \h CREATE OPERATOR CLASS
Command: CREATE OPERATOR CLASS
Description: define a new operator class
Syntax:
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
    USING index_method [ FAMILY family_name ] AS
        { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ]
            [ FOR SEARCH | FOR ORDER BY sort_family_name ]
            | FUNCTION support_number [ ( op_type [, op_type ] ) ]
                function_name ( argument_type [, ...] )
            | STORAGE storage_type
        } [, ... ]
```

运算符类将告诉索引如何表现。我们来看一下标准的二叉树。它可以执行五个操作：

Strategy	Operator	Description
1	<	Less than
2	<=	Less than or equal to
3	=	Equal to
4	>=	Greater than or equal to
5	>	Greater than

标准运算符类支持我们在本书中使用的标准数据类型和标准运算符。如果您想处理社会安全号码，有必要提供自己的操作符，为您提供所需的处理逻辑。然后可以使用这些自定义运算符来形成运算符类，这只不过是传递给索引以配置其行为方式的策略。

实现一个 btree 的运算符类

为了给你一个运算符类的实际例子，我已经破解了一些代码来处理社会安全号码。为了简单起见，我没有注意校验和等细节。

创建新的操作符

必须要做的第一件事是提出所需的运算符。请注意，需要五个操作员。每种策略都有一个操作符。索引的策略就像一个插件，允许您输入自己的代码。

在开始之前，我编译了一些测试数据：

```
CREATE TABLE t_sva (sva text);
INSERT INTO t_sva VALUES ('1118090878');
INSERT INTO t_sva VALUES ('2345010477');
```

现在测试数据已存在，是时候创建一个运算符了。为此，PostgreSQL 提供了 CREATE OPERATOR 命令：

```
test=# \h CREATE OPERATOR
Command: CREATE OPERATOR
Description: define a new operator
Syntax:
CREATE OPERATOR name (
    PROCEDURE = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

基本上，概念如下：运算符调用一个函数，该函数获取一个或两个参数，一个用于左参数，一个用于操作符的右参数。

如您所见，运算符只不过是函数调用。因此，有必要在运算符隐藏的那些函数中实现所需的逻辑。为了修复排序顺序，我编写了一个名为 normalize_si 的函数：

```
CREATE OR REPLACE FUNCTION normalize_si(text) RETURNS text AS $$

BEGIN
    RETURN substring($1, 9, 2) ||
           substring($1, 7, 2) ||
           substring($1, 5, 2) ||
           substring($1, 1, 4);
END; $$

LANGUAGE 'plpgsql' IMMUTABLE;
```

调用该函数将返回以下结果：

```
test=# SELECT normalize_si('1118090878');
normalize_si
-----
7808091118 (1 row)
```

如您所见，我们所做的就是交换一些数字。现在可以使用普通的字符串排序顺序。在下一步中，此功能已经可以用于直接比较社会安全号码。

所需的第一个功能是小于功能，这是第一个策略所需要的：

```
CREATE OR REPLACE FUNCTION si_lt(text, text) RETURNS boolean AS $$

BEGIN
    RETURN normalize_si($1) < normalize_si($2);
END;

$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

这里有两件重要的事情需要注意：

- 该函数不能用 SQL 编写。它仅适用于程序语言或编译语言。原因是 SQL 函数在某些情况下可以内联，这会削弱整个执行过程。
- 第二个问题是你应该坚持本章中使用的命名约定 - 它被社区广泛接受。少于函数应该被称为`_lt`，少于或等于函数应该被称为`_le`，依此类推。

有了这些知识，我们可以定义未来操作符所需的下一个功能：

```
-- lower equals
CREATE OR REPLACE FUNCTION si_le(text, text)
RETURNS boolean AS
$$
BEGIN
    RETURN normalize_si($1) <= normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
-- greater equal
CREATE OR REPLACE FUNCTION si_ge(text, text)
RETURNS boolean AS
$$
BEGIN
    RETURN normalize_si($1) >= normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
-- greater
CREATE OR REPLACE FUNCTION si_gt(text, text)
RETURNS boolean AS
$$
BEGIN
    RETURN normalize_si($1) > normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

到目前为止，已经定义了四个功能。不需要`equals`运算符的第五个函数。我们可以简单地使用现有的运算符，因为`equals`不依赖于排序顺序。

现在所有功能都已到位，是时候定义这些运算符了：

```
-- define operators
CREATE OPERATOR <# ( PROCEDURE=si_lt,
LEFTARG=text,
RIGHTARG=text);
```

操作符的设计实际上非常简单。 运算符需要一个名称（在我的情况下是<#），一个应该被调用的存储过程，以及左右参数的数据类型。 调用运算符时，左参数将是 si_lt 的第一个参数，右参数将是第二个参数。

其余三个运算符遵循相同的原则：

```
CREATE OPERATOR <#= ( PROCEDURE=si_le,
                      LEFTARG=text,
                      RIGHTARG=text);
CREATE OPERATOR >#= ( PROCEDURE=si_ge,
                      LEFTARG=text,
                      RIGHTARG=text);
CREATE OPERATOR ># ( PROCEDURE=si_gt,
                      LEFTARG=text,
                      RIGHTARG=text);
```

根据您使用的索引类型，需要一些支持功能。 在标准 btree 的情况下，只需要一个支持函数，用于内部加速：

```
CREATE OR REPLACE FUNCTION si_same(text, text) RETURNS int AS $$  
BEGIN  
    IF normalize_si($1) < normalize_si($2)  
    THEN  
        RETURN -1;  
    ELSIF normalize_si($1) > normalize_si($2)  
    THEN  
        RETURN +1;  
    ELSE  
        RETURN 0;  
    END IF;  
END;  
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

如果第一个参数较小，则 si_same 函数将返回-1，如果两个参数相等则返回 0，如果第一个参数更大，则返回 1。 在内部，_same 函数是主函数，所以你应该确保你的代码经过优化。

创建运算符类

最后，所有组件都已就绪，可以创建索引所需的运算符类：

```
CREATE OPERATOR CLASS sva_special_ops  
FOR TYPE text USING btree  
AS  
OPERATOR 1 <#,
```

```
OPERATOR 2 <#=,
OPERATOR 3 =,
OPERATOR 4 >#=,
OPERATOR 5 >#,
FUNCTION 1 si_same(text, text);
```

CREATE OPERATOR CLASS 命令连接策略和运算符 OPERATOR 1。

<# 表示策略 1 将使用<# 运算符。 最后，_same 函数与运算符类连接。

请注意，运算符类具有名称，并且已明确定义它以使用 btree。在创建索引期间，已经可以使用运算符类：

```
CREATE INDEX idx_special ON t_sva (sva sva_special_ops);
```

创建索引的方式与以前略有不同：sva sva_special_ops 表示使用索引编制 sva 列 sva_special_ops 运算符类。如果没有显式使用 sva_special_ops，那么 PostgreSQL 不会用于我们的特殊排序顺序，而是决定使用默认的运算符类。

测试自定义运算符类

在我们的示例中，测试数据仅包含两行。因此，PostgreSQL 永远不会使用索引，因为该表太小，无法验证使用索引的额外开销。为了能够在不加载太多数据的情况下进行测试，您可以告诉优化器使得顺序扫描成本更加昂贵。

使用以下指令可以在会话级别中完成顺序扫描成本更昂贵的操作：

```
SET enable_seqscan TO off;
```

索引按预期的工作：

```
test=# explain SELECT * FROM t_sva WHERE sva = '0000112273';
          QUERY PLAN
```

```
Index Only Scan using idx_special on t_sva
(cost=0.13..8.14 rows=1 width=32)
Index Cond: (sva = '0000112273'::text)
(2 rows)

test=# SELECT * FROM t_sva;
sva
-----
2345010477
1118090878
(2 rows)
```

了解 PostgreSQL 索引类型

到目前为止，我们只讨论了 `btree` 类型的索引。但是，在许多情况下，`btree` 还不够。为什么会这样？正如本章所讨论的，`btree` 基本上是基于排序。可以使用 `btree` 处理运算符`<`，`<=`，`=`，`>=`和`>`。问题是，并非所有数据类型都可以以有用的方式排序。想象一下多边形。你会如何以有用的方式对这些对象进行排序？当然，您可以按照覆盖的区域，长度等进行排序，但这样做不会让您使用几何方式搜索找到它们。

该问题的解决方案是提供多于一种索引类型。每个索引都将用于特殊目的，并完全满足需要。可以使用以下索引类型（从 PostgreSQL 10.0 开始）：

```
test=# SELECT * FROM pg_am;
 amname | amhandler | amtype
-----+-----+-----+
 btree  | bthandler | i
 hash   | hashhandler | i
 GiST   | gisthandler | i
 gin    | ginhandler | i
 spGiST | spghandler | i
 brin   | brinhandler | i
(6 rows)
```

这里有六种类型的索引。已经对 `btree` 进行了非常详细的讨论，但那些其他索引类型有哪些优点呢？以下部分将概述 PostgreSQL 中可用的每种索引类型的用途。

请注意，有一些扩展类型可以在您在此处看到的内容之上使用。网上提供的其他索引类型包括 `rum`，`vodka` 以及未来的 `cognac`。

哈希索引

哈希索引已存在多年。我们的想法是对输入值进行哈希处理并将其存储起来以供以后查找。哈希索引实际上是有存在的意义的。但是，在 PostgreSQL 10.0 之前，不建议使用哈希索引，因为 PostgreSQL 没有实现 WAL 支持他们。在 PostgreSQL 10.0 中，这已经发生了变化。哈希索引现在已被 WAL 完全记录，因此可以进行复制，可以 100% 安全的从崩溃中恢复。

哈希索引通常比 `btree` 索引大一些。假设您要索引 4 百万整数值。`btree` 需要大约 90 MB 的存储才能完成此任务。哈希索引在磁盘上需要大约 125 MB。因此，在许多情况下，许多人认为磁盘上的哈希索引非常小是错误的。

GiST 紴引

Generalized Search Tree (GiST) 紴引是非常重要的索引类型，因为它们用于各种不同的

事物。GiST 索引可用于实现 R-tree 行为，甚至可以充当 btree。但是，不推荐滥用 GiST 用于 btree 索引。

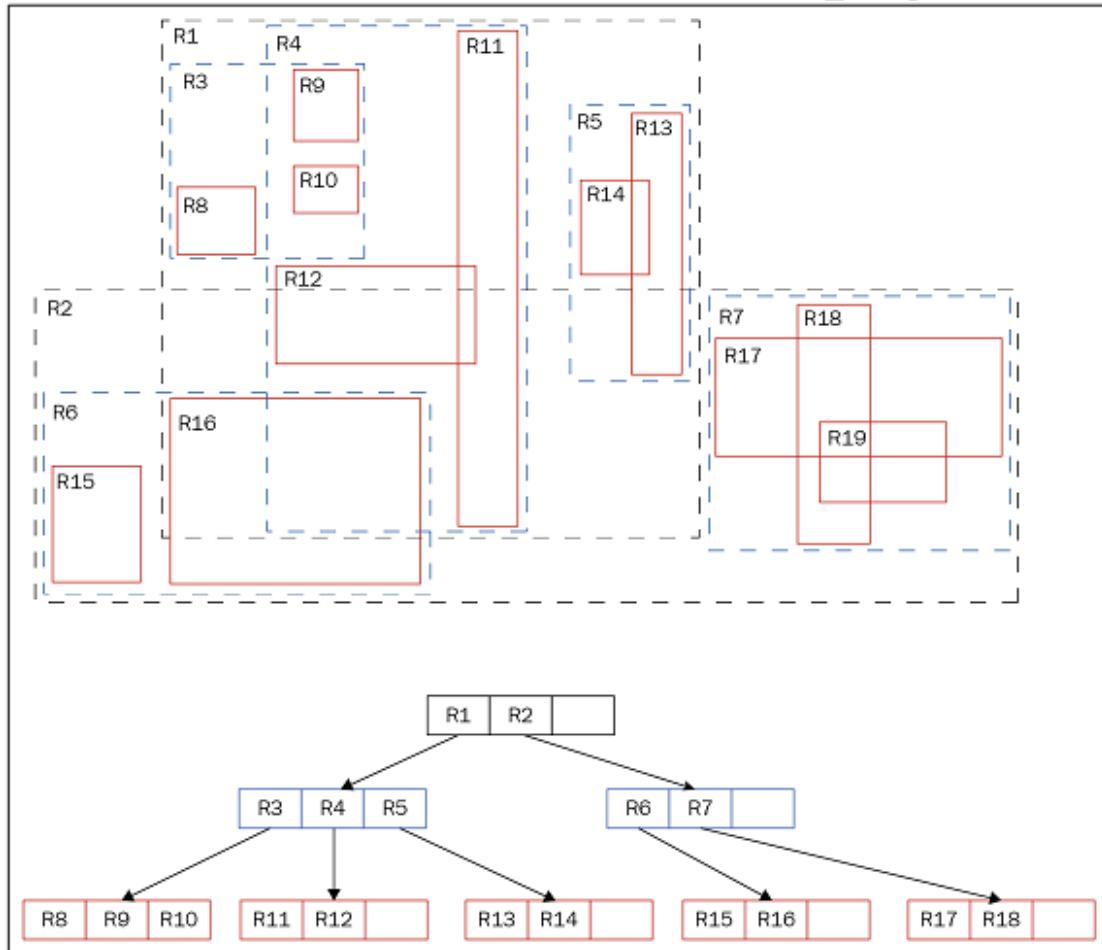
GiST 的典型用例如下：

- 范围类型
- 几何索引（例如，由非常受欢迎的 PostGIS 扩展使用）
- 模糊搜索

了解 GiST 的工作原理

对许多人来说，GiST 仍然是一个黑盒。因此，我决定在本章中添加一节，概述 GiST 如何在内部工作。

请考虑以下图表：



看看树。您将看到 R1 和 R2 位于顶部。R1 和 R2 是包含其他所有内容的边界框。R3 包含 R3, R4 和 R5。R8, R9 和 R10 包含在 R3 中，依此类推。因此，GiST 索引是分层组织的。您可以在图中看到的是，支持某些在 btree 中不可用的操作。其中一些操作是重叠的，左边的，右边的，等等。GiST 树的布局是几何索引的理想选择。

扩展 GiST

当然，也可以提出自己的运算符类。支持以下策略：

Operation	Strategy number
Strictly left of	1
Does not extend to right of	2
Overlaps	3
Does not extend to left of	4
Strictly right of	5
Same	6
Contains	7
Contained by	8
Does not extend above	9
Strictly below	10
Strictly above	11
Does not extend below	12

如果要为 GiST 编写运算符类，则必须提供一些支持函数。在 btree 的情况下，只有相同的函数 - GiST 索引提供了更多：

Function	Description	Support function number
consistent	这些函数确定关键字是否满足查询限定符。在内部，查找和检查策略	1
union	计算一组键值的并集。在数值的情况下，仅计算上限值和下限值或范围。这对几何形状尤为重要。	2
compress	计算键或值的压缩表示。	3
decompress	这是压缩功能的相反功能。	4
penalty	在插入期间，将计算插入二叉树中的成本。成本决定了新条目在二叉树中的位置。因此，良好的惩罚函数是索引具备良好性能的关键。	5
picksplit	确定在页面分裂的情况下移动条目的位置。某些条目必须保留在旧页上，而其他条目将转到正在创建的新页上。具有良好的 picksplit 功能对于良好的索引性能至关重要。	6
equal	相等功能类似于你在 btree 中看到的相同功能。	7
distance	计算关键字和查询值之间的距离（数字）。距离函数是可选的，在支持 KNN 搜索时是必需的。	8
fetch	确定压缩关键字的原始表示形式。这个函数是最新版本的 PostgreSQL 支持处理仅索引扫描必须要的。	9

为 GiST 索引实现运算符类通常用 C 语言完成。如果您对一个好的示例感兴趣，我建议您查看 contrib 目录中的 btree_GiST 模块。它展示了如何使用 GiST 索引标准数据类型，并且是一个很好的信息来源和灵感来源。

GIN 索引

Generalized inverted (GIN) 索引是索引文本的好方法。假设您要索引一百万个文本文档。某个词可能会出现数百万次。在普通的 `btree` 索引中，这意味着关键词存储了数百万次。在 GIN 中不是这种情况。每个键（或单词）存储一次并分配给文档列表。键值按标准 `b-tree` 组织。每个条目都有一个文档列表，指向表中具有相同键的所有条目。GIN 指数非常小且紧凑。但是，它缺少 `btree` 排序数据中的重要特征。在 GIN 中，与某个键关联的项目内容列表按表中行的位置排序，而不是按某些任意标准排序。

扩展 GIN

就像任何其他索引一样，GIN 可以扩展。可以使用以下策略：

Operation	Strategy number
Overlap	1
Contains	2
Is contained by	3
Equal	4

除此之外，还提供以下支持功能：

Function	Description	Support function number
compare	比较功能类似于您在 <code>btree</code> 中看到的相同功能。如果比较两个键，则返回 -1 (低), 0 (相等) 或 1 (高)。	1
extractValue	从要编制索引的值中提取键值。值可以有很多键。例如，文本值可能包含多个单词。	2
extractQuery	从查询条件中提取键。	3
consistent	检查值是否与查询条件匹配。	4
comparePartial	比较查询中的部分键和索引中的键。返回 -1, 0 或 1 (类似于 <code>btree</code> 支持的相同函数)	5
triConsistent	确定值是否与查询条件匹配 (三元变体)。如果存在 <code>consistent</code> 函数，则它是可选的。	6

如果您正在寻找一个如何扩展 GIN 的例子，请考虑查看 PostgreSQL contrib 目录中的 `btree_gin` 模块。它是宝贵的信息来源，也是开始实现自己功能的好方法。

如果您对全文搜索感兴趣，本章稍后将提供更多信息。

SP-GiST 索引

Space partitioned GiST (SP-GiST) 主要用于内存使用。原因是存储在磁盘上的 SP-GiST 需要相当多的磁盘命中才能运行。磁盘命中比仅仅遵循 RAM 中的几个指针成本更昂贵。

美妙的是 SP-GiST 可用于实现各种类型的树，例如四叉树，k-d 树和 radix 树（尝试）。

提供以下策略：

Operation	Strategy number
Strictly left of	1
Strictly right of	5
Same	6
Contained by	8
Strictly below	10
Strictly above	11

要为 SP-GiST 编写自己的运算符类，必须提供几个功能：

Function	Description	Support function number
config	提供有关正在使用的运算符类的信息	1
choose	弄清楚如何将新值插入内部元组	2
picksplit	弄清楚如何分区/拆分一组值	3
inner_consistent	确定需要在查询中搜索哪些子分区	4
leaf_consistent	确定键是否满足查询限定符	5

BRIN 索引

块范围索引 (BRIN) 具有很大的实用价值。到目前为止所讨论的所有索引都需要相当多的磁盘空间。虽然很多工作已经开展用于缩小 GIN 索引等，但它们仍然需要很多，因为每个条目都需要一个索引指针。所以，如果有 1000 万个条目，那么将有 1000 万个索引指针。空间是 BRINs 索引所关注的主要问题。BRIN 索引不会为每个元组保留索引条目，但会存储 128 (默认) 数据块 (1 MB) 的最小值和最大值。因此索引非常小但有损。扫描索引将返回比我们要求的更多的数据。PostgreSQL 必须在后面的步骤中过滤这些额外的行。

以下示例演示了 BRIN 索引到底有多小：

```
test=# CREATE INDEX idx_brin ON t_test USING brin(id);
CREATE INDEX
test=# \di+ idx_brin
```

List of relations

Schema Name	Type	Owner	Table	Size
---------------	------	-------	-------	------

```
-----+-----+-----+-----+
 public | idx_brin | index | hs      | t_test | 48 KB
(1 row)
```

在我的例子中，BRIN 索引比标准 btree 小 2,000 倍。现在自然产生的问题是，为什么我们不总是使用 BRIN 索引？要回答这类问题，重要的是要反思 BRIN 的布局；存储 1 MB 的最小值和最大值。如果对数据进行排序（高相关性），BRIN 非常有效，因为我们可以获取 1 MB 的数据，扫描它，然后就完成了。但是，如果数据被清洗过怎么办？在这种情况下，BRIN 将无法再排除数据块，因为接近整体高位和总体低位的数据很可能在 1 MB 的数据范围内。因此，BRIN 主要用于高度相关的数据。实际上，数据仓库应用程序中很可能存在相关数据。通常，数据每天都会加载，因此日期数据可以高度相关。

扩展 BRIN 索引

BRIN 支持与 btree 相同的策略，因此需要相同的运算符集。代码可以很好地重用：

Operation	Strategy number
Less than	1
Less than or equal	2
Equal	3
Greater than or equal	4
Greater than	5

BRIN 所需的支持功能如下：

Function	Description	Support function number
opcInfo	提供有关索引的内部信息	1
add_value	向现有摘要元组添加条目	2
consistent	检查值是否与条件匹配	3
union	计算两个汇总值的并集（最小值/最大值）	4

添加其他索引

从 PostgreSQL 9.6 开始，有一种简单的方法可以将全新的索引类型部署为扩展。这非常酷，因为如果 PostgreSQL 提供的那些索引类型不够，可以添加额外的扩展索引以满足您的需求。执行此操作的说明是 CREATE ACCESS METHOD：

```
test=# \h CREATE ACCESS METHOD
Command: CREATE ACCESS METHOD
Description: define a new access method
Syntax:
```

```
CREATE ACCESS METHOD name  
  TYPE access_method_type  
  HANDLER handler_function
```

不要太担心这个命令 - 只要你部署自己的索引类型，它将作为一个随时可用的扩展。

其中一个扩展实现了 bloom 过滤器。 Bloom 过滤器是概率数据结构。 它们有时会返回太多行但从不会太少。 因此， bloom 过滤器是预过滤数据的好方法。

它是如何工作的？ bloom 过滤器定义在几列上。 根据输入值计算位掩码，然后将其与查询进行比较。 bloom 过滤器的优点是您可以根据需要索引任意数量的列。 缺点是必须读取整个 bloom 过滤器。 当然， bloom 过滤器比基表数据小，因此在许多情况下非常有益。

要使用 bloom 过滤器，只需激活扩展，这是 PostgreSQL contrib 包的一部分：

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

如前所述， bloom 过滤器背后的想法是，它允许您根据需要索引尽可能多的列。 在许多实际应用程序中，挑战在于索引许多列而不知道用户在运行时实际需要哪些组合。 对于大型表，在 80 个字段或更多字段上创建标准 btree 索引是完全不可能的。 在如下案例中， bloom 过滤器可以替代：

```
test=# CREATE TABLE t_bloom (x1 int, x2 int, x3 int, x4 int,  
x5 int, x6 int, x7 int);
```

CREATE TABLE

创建索引很简单：

```
test=# CREATE INDEX idx_bloom ON t_bloom USING bloom(x1, x2, x3, x4,  
x5, x6, x7);
```

CREATE INDEX

如果关闭顺序扫描，则可以看到索引：

```
test=# SET enable_seqscan TO off;
```

SET

```
test=# explain SELECT * FROM t_bloom WHERE x5 = 9 AND x3 = 7;
```

QUERY PLAN

```
Bitmap Heap Scan on t_bloom (cost=18.50..22.52 rows=1 width=28)
```

```
  Recheck Cond: ((x3 = 7) AND (x5 = 9))
```

```
  -> Bitmap Index Scan on idx_bloom (cost=0.00..18.50 rows=1 width=0)
```

```
    Index Cond: ((x3 = 7) AND (x5 = 9))
```

请注意，我查询了随机列的组合；它们与索引中的实际顺序无关。 bloom 过滤器仍然是有效的。

如果您对 bloom 过滤器感兴趣，请考虑查看以下网站：https://en.wikipedia.org/wiki/Bloom_filter。

通过模糊搜索获得更好的答案

如今，执行精确搜索并不是用户唯一期望的。现代网站以一种他们总是期望结果的方式教育用户，无论用户输入什么。如果你在谷歌上搜索，即使用户输入错误，充满拼写错误或者毫无意义，也会有答案。无论输入数据如何，人们都期待会有好的搜索结果。

利用 pg_trgm

要使用 PostgreSQL 进行模糊搜索，可以添加 pg_trgm 扩展名。要激活扩展，只需运行以下指令即可：

```
test=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

pg_trgm 扩展非常强大，为了展示它的功能，我编写了一些样本数据，包括欧洲奥地利的 2,354 个村庄和城市名称。

我们的样本数据可以存储在一个简单的表中：

```
test=# CREATE TABLE t_location (name text);
CREATE TABLE
```

我的公司网站拥有所有数据，PostgreSQL 使您可以直接加载数据：

```
test=# COPY t_location FROM PROGRAM
      'curl https://www.cybertec-postgresql.com/secret/orte.txt';
COPY 2354
```



必须安装 curl（用于获取数据的命令行工具）。如果你没有这个工具，正常下载文件并从您的文件中导入本地文件系统。

加载数据后，可以检查表的内容：

```
test=# SELECT * FROM t_location LIMIT 4;
          name
-----
Eisenstadt
Rust
Breitenbrunn am Neusiedler See
Donnerskirchen
(4 rows)
```

如果德语不是你的母语，就不可能在没有严重错误的情况下拼写这些地点的名字。

`pg_trgm` 为我们提供了一个计算两个字符串之间距离的距离算子：

```
test=# SELECT 'abcde' <-> 'abdeacb';
?column?
-----
0.833333
(1 row)
```

距离是 0 到 1 之间的数字。数字越小，两个字符串越相似。

这是如何运作的？ `Trigrams` 取一个字符串并将其分解为三个字符的序列：

```
test=# SELECT show_trgm('abcdef');
show_trgm
-----
{" a"," ab",abc,bcd,cde,def,"ef "}
(1 row)
```

然后将使用这些序列来确定您刚刚看到的距离。当然，可以在查询中使用距离运算符来查找最接近的匹配：

```
test=# SELECT *
FROM t_location
ORDER BY name <-> 'Kramertneusiedel'
LIMIT 3;
name
-----
Gramatneusiedl
Klein-Neusiedl
Potzneusiedl
(3 rows)
```

`Gramatneusiedl` 非常靠近 `Kramertneusiedel`。听起来很相似，使用 `K` 而不是 `G` 是一个非常常见的错误。在谷歌上，你有时会看到“Did you mean”。谷歌很可能在这里使用 n-gram 来做到这一点。

在 PostgreSQL 中，可以使用 `GiST` 对使用 `trigrams` 的文本进行索引：

```
test=# CREATE INDEX idx_trgm ON t_location
      USING GiST(name GiST_trgm_ops);
CREATE INDEX
```

`pg_trgm` 为我们提供了 `GiST_trgm_ops` 运算符类，用于进行相似性搜索。以下清单显示了索引按预期使用的效果：

```
test=# explain SELECT *
      FROM t_location
```

```
ORDER BY name <-> 'Kramertneusiedel'  
LIMIT 5;  
QUERY PLAN  
-----  
Limit (cost=0.14..0.58 rows=5 width=17)  
  -> Index Scan using idx_trgm on t_location  
    (cost=0.14..207.22 rows=2354 width=17)  
      Order By: (name <-> 'Kramertneusiedel'::text)  
(3 rows)
```

加速 LIKE 查询

LIKE 查询肯定会导致这些天全球人们面临的一些最糟糕的性能问题。在大多数数据库系统中，LIKE 非常慢并且需要顺序扫描。除此之外，最终用户很快发现模糊搜索在许多情况下会比精确查询返回更好的结果。因此，如果经常在整个数据库服务器大型表上的调用单一类型的 LIKE 查询通常会削弱整个数据库服务器的性能。

幸运的是，PostgreSQL 提供了解决问题的方法，并且解决方案恰好已经安装：

```
test=# explain SELECT * FROM t_location WHERE name LIKE '%neusi%';  
QUERY PLAN  
-----  
Bitmap Heap Scan on t_location  
(cost=4.33..19.05 rows=24 width=13)  
  Recheck Cond: (name ~~ '%neusi%'::text)  
    -> Bitmap Index Scan on idx_trgm (cost=0.00..4.32 rows=24 width=0)  
      Index Cond: (name ~~ '%neusi%'::text)  
(4 rows)
```

上一节中部署的三元组索引也适用于加速 LIKE 查询。请注意，% 符号可以在搜索字符串中的任何位置使用。这也是比较标准 btree 的一个主要优势，它恰好在查询时加速结尾通配符。

处理正则表达式

然而，这仍然不是一切。Trigram 索引甚至能够加速简单的正则表达式。以下示例显示了如何完成此操作：

```
test=# SELECT * FROM t_location WHERE name ~ '[A-C].*neu.*';  
name  
-----  
Bruckneudorf  
(1 row)  
test=# explain SELECT * FROM t_location WHERE name ~ '[A-C].*neu.*';  
QUERY PLAN
```

```
-----  
Index Scan using idx_trgm on t_location (cost=0.14..8.16  
    rows=1 width=13)  
  Index Cond: (name ~ '[A-C].*neu.*'::text)  
(2 rows)
```

PostgreSQL 将检查正则表达式并使用索引来回答问题。



TIP 在内部，PostgreSQL 可以将正则表达式转换为图形并相应地遍历索引。

了解全文搜索

如果您要查找名称或者某个简单字符串，通常会查询字段的整个内容。在全文搜索(FTS)中，这是不同的。全文搜索的目的是查找可在文本中找到的单词或单词组。因此，FTS更像是一个包含操作，因为您基本上从不查找精确的字符串。

在 PostgreSQL 中，可以使用 GIN 索引完成 FTS。这个想法是剖析文本，提取有价值的词汇(=“预处理的词语”)，并索引这些元素而不是基础文本。为了使您的搜索更加成功，这些字词已经过预处理。

这是一个例子：

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even  
mind having many cars');  
          to_tsvector  
-----  
'car':2,6,14 'even':10 'mani':13 'mind':11 'want':4 'would':8  
(1 row)
```

该示例显示了一个简单的句子。`to_tsvector` 函数将获取字符串，应用英语规则并执行词干过程。根据配置(英文)，PostgreSQL 将解析字符串，丢弃停用词，并词干单个单词。例如，`car` 和 `cars` 将转变为汽车。请注意，这不是关于找到词干。在许多情况下，PostgreSQL 将简单地将字符串转换为 `mani`，方法是应用适用于英语的标准规则。

请注意，`to_tsvector` 函数的输出高度依赖于语言配置。如果你告诉 PostgreSQL 将字符串视为荷兰语，结果将完全不同：

```
test=# SELECT to_tsvector('dutch', 'A car, I want a car. I would not even  
mind having many cars');  
          to_tsvector  
-----  
'a':1,5 'car':2,6,14 'even':10 'having':12 'i':3,7 'many':13  
 'mind':11 'not':9 'would':8  
(1 row)
```

要确定支持哪些配置，请考虑运行以下查询：

```
SELECT cfgname FROM pg_ts_config;
```

比较字符串

在简要了解词干过程后，是时候弄清楚如何将词干文本与用户查询进行比较。以下代码段检查所需的单词：

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even
mind having many cars') @@ to_tsquery('english', 'wanted');
?column?
-----
t
(1 row)
```

请注意，想要的内容实际上并未显示在原始文本中。仍然，PostgreSQL 将返回 `true`。原因是 `want` 和 `wanted` 都被转换成相同的词汇，所以结果是真的。实际上，这很有道理。想象一下，您正在 Google 上寻找汽车。如果你找到销售汽车的网页，这是完全没问题的。因此，寻找共同的词汇是一个明智的想法。

有时，人们不仅要找一个单词，而且想要找到一组单词。使用 `to_tsquery`，这是可能的，如下一个示例所示：

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even
mind having many cars') @@ to_tsquery('english', 'wanted & bmw');
?column?
-----
f
(1 row)
```

在这种情况下，返回 `false`，因为在我们的输入字符串中找不到 `bmw`。在 `to_tsquery` 函数中，`&` 的意思是并且，`|` 的意思是或者。因此很容易构建复杂的搜索字符串。

定义 GIN 索引

如果要将文本搜索应用于列或列组，则基本上有两种选择：

- 使用 GIN 创建功能索引
- 添加包含即用型 `tsvectors` 和触发器的列以使它们保持同步

在本节中，将概述两个选项。为了说明事情是如何工作的，我创建了一些示例数据：

```
test=# CREATE TABLE t_fts AS SELECT comment
      FROM pg_available_extensions;
SELECT 43
```

使用函数索引直接索引列绝对是一种更慢但更节省空间的方法：

```
test=# CREATE INDEX idx_fts_func ON t_fts
```

```
    USING gin(to_tsvector('english', comment));
CREATE INDEX
```

在函数上部署索引很容易，但可能会导致一些开销。添加物理列需要更多空间，但会带来更好的运行效率：

```
test=# ALTER TABLE t_fts ADD COLUMN ts tsvector;
ALTER TABLE
```

唯一的麻烦是，如何保持此列同步？答案是使用触发器。

```
test=# CREATE TRIGGER tsvectorupdate
BEFORE INSERT OR UPDATE ON t_fts
FOR EACH ROW
EXECUTE PROCEDURE
tsvector_update_trigger(somename, 'pg_catalog.english', 'comment');
```

幸运的是，PostgreSQL 已经提供了一个 C 函数，触发器可以使用它来同步 tsvector 列。只需传递一个名称，所需的语言，以及函数的几列，您就已经完成了。触发功能将处理所需的一切。请注意，触发器将始终在与进行修改的语句相同的事务中运行。因此，不存在数据不一致的风险。

调试您的搜索

有时，查询与给定搜索字符串匹配的原因尚不清楚。为了调试您的查询，PostgreSQL 提供了 `ts_debug` 函数。从用户的角度来看，它可以像 `to_tsvector` 一样使用。它揭示了很多关于 FTS 基础架构的内部工作原理：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM ts_debug('english', 'go to
www.postgresql-support.de');
-[ RECORD 1 ]+-----
alias      | asciiword
description | Word, all ASCII
token      | go
dictionaries | {english_stem}
dictionary   | english_stem
lexemes     | {go}
-[ RECORD 2 ]+-----
alias      | blank
description | Space symbols
token      |
dictionaries | {}
dictionary   |
lexemes     |
-[ RECORD 3 ]+-----
```

```

alias      | asciiword
description | Word, all ASCII
token      | to
dictionaries | {english_stem}
dictionary   | english_stem
lexemes     | {}
-[ RECORD 4 ]+-----
alias      | blank
description | Space symbols
token      |
dictionaries | {}
dictionary   |
lexemes     |
-[ RECORD 5 ]+-----
alias      | host
description | Host
token      | www.postgresql-support.de
dictionaries | {simple}
dictionary   | simple
lexemes     | {www.postgresql-support.de}

```

`ts_debug` 将列出找到的每个令牌并显示有关令牌的信息。您将看到解析器找到的标记，使用的字典以及对象的类型。在我的例子中，找到了空格，单词和主机。您可能还会看到数字，电子邮件地址等等。根据字符串的类型，PostgreSQL 会以不同的方式处理事情。例如，阻止主机名和电子邮件地址是完全没有意义的。

收集单词统计信息

全文搜索可以处理大量数据。为了让最终用户更深入地了解他们的文本，PostgreSQL 提供了 `pg_stat` 函数（译者注：此处应该是笔误，正确内容应该是 `ts_stat`），它返回一个单词列表：

```

SELECT * FROM ts_stat('SELECT to_tsvector("english", comment)
                      FROM pg_available_extensions')
ORDER BY 2 DESC
LIMIT 3;
word      | ndoc | nentry
-----+-----+
function  | 10   | 10
data      | 10   | 10
type      | 7    | 7
(3 rows)

```

单词列包含词干；`ndoc` 告诉我们某个单词出现的文档数量。`nentry` 表示一个单词被发现的频率。

利用排除运操作

到目前为止，索引已被用于加速查询并确保唯一性。然而，几年前，有人提出了使用索引的想法。正如您在本章中看到的那样，GiST 支持诸如交叉，重叠，包含等操作。那么，为什么不使用这些操作来管理数据完整性呢？

这里是一个例子：

```
test=# CREATE EXTENSION btree_gist;
test=# CREATE TABLE t_reservation (
    room int,
    from_to tsrange,
    EXCLUDE USING GiST (room with =,
        from_to with &&)
);
CREATE TABLE
```

EXCLUDE USING GiST 子句定义了其他约束。如果您在销售房间，您可能希望同时预订不同的房间。但是，您不希望在同一时期内两次出售同一个房间。什么是 EXCLUDE，在我的例子中是：如果房间同时预订了两次，则会弹出一个错误（如果与同一房间有关，则 from_to 中的数据不得重叠（&&））。

以下两行不会违反约束：

```
test=# INSERT INTO t_reservation
VALUES (10, '['"2017-01-01", "2017-03-03"]');
INSERT 0 1
test=# INSERT INTO t_reservation
VALUES (13, '['"2017-01-01", "2017-03-03"]);
INSERT 0 1
```

但是，下一个 INSERT 会导致违规，因为数据重叠：

```
test=# INSERT INTO t_reservation
VALUES (13, '['"2017-02-02", "2017-08-14"]);
ERROR: conflicting key value violates exclusion constraint
"t_reservation_room_from_to_excl"
DETAIL: Key (room, from_to)=(13, ["2017-02-02 00:00:00","2017-08-14
00:00:00"]) conflicts with existing key (room, from_to)=(13, ["2017-01-01
00:00:00","2017-03-03 00:00:00"]).
```

使用排除运算符非常有用，可以为您提供处理数据完整性的高级方法。

小结

本章全是关于索引的。我们了解 PostgreSQL 何时决定索引以及存在哪些类型的索引。

除了使用索引之外，还可以实现自己的索引策略，以使用自定义运算符和索引策略加速应用程序的运行。

对于那些真正想要挑战极限的人来说，PostgreSQL 提供了自定义访问方法的功能。

在第 4 章 “处理高级 SQL” 中，完全是关于高级 SQL 的部分。很多人都不知道 SQL 真正能做什么，因此，我将向您展示一些高效，更高级的 SQL 内容。

QA

索引总能提高性能吗？

答案肯定是否定的。如果某些东西总是好的话，它会默认存在。索引可以加速许多操作，但它们也可以大大减慢速度。唯一的规则是：关于你在做什么以及你想要实现什么。

索引是否占用了大量空间？

这取决于索引的类型。BRIN 索引非常小而且相当便宜，而其他索引通常需要更多空间。例如，btree 比 btree 大约 2000 倍（译者注：此处应该是笔误，原作者正确的意思应该是 BRIN 比 btrees 索引小 2000 倍）。在大多数情况下，基于 Trigram 的索引甚至更大。

如何找到丢失的索引？

我判断的最好的方法是查看 `pg_stat_statements` 和 `pg_stat_user_tables`。特别是 `seq_tup_read` 是一个非常有价值的列。如果您正在读取非常多的行，则说明可能缺少索引。一般来说，深入了解查询是必要的。简而言之：在任何情况下，`EXPLAIN` 都是你的朋友。

索引可以并行构建吗？

是的，自从 PostgreSQL 11 以来我们支持并行索引创建。它可以大大加快索引创建速度。

4. 处理高级 SQL

在第3章“使用索引”中，您了解了索引以及 PostgreSQL 运行自定义索引代码以加快查询的能力。在本章中，您将了解高级 SQL。本书的大多数读者都会有一些使用 SQL 的经验。但是，经验表明，本书中概述的高级功能并不广为人知，因此在此背景下涵盖它们以帮助人们更快，更有效地实现目标是有意义的。关于数据库是否只是一个简单地数据存储工具，还是业务逻辑是否应该存在数据库中的讨论，已经进行了很长时间。也许这一章将揭示一些亮点，并展示现代关系数据库的真正能力。

本章介绍现代 SQL 及其功能。包含各种不同且复杂的 SQL 功能并详细介绍。涵盖的主题如下：

- 分组集合
- 有序集合
- 假设聚合
- 窗口函数和分析

在本章的最后，您将能够理解和使用高级 SQL。

介绍分组集合

SQL 的每个高级用户都应该熟悉 GROUP BY 和 HAVING 子句。但他们是否也知道 CUBE, ROLLUP 和 GROUPING SETS？如果没有，本章是必读的。

加载一些示例数据

为了使本章成为您愉快的学习经历，我们将编制一些从 BP 能源报告中获取的样本数据：
<http://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy/downloads.html>.

这是将使用的数据结构：

```
test=# CREATE TABLE t_oil (
    region      text,
    country     text,
    year        int,
    production  int,
    consumption int
);
CREATE TABLE
```

测试数据可以直接使用 curl 从我们的网站下载：

```
test=# COPY t_oil FROM PROGRAM '
curl https://www.cybertec-postgresql.com/secret/oil_ext.txt ';
```

COPY 644

与前一章一样，我们可以在导入文件之前下载该文件。在某些操作系统上，默认情况下不存在 curl 或尚未安装 curl，因此先下载文件可能对许多人来说更容易一些。

世界上两个地区的 14 个国家有 1965 年至 2010 年的数据：

```
test=# SELECT region, avg(production) FROM t_oil GROUP BY region;
      region      |      avg
-----+-----
Middle East    | 1992.6036866359447005
North America | 4541.3623188405797101
(2 rows)
```

应用分组集合

GROUP BY 子句将每组转换为多行。但是，如果您在现实生活中进行报道，他们可能也会对整体平均水平感兴趣。可能还需要一行。

以下是如何实现这一目标：

```
test=# SELECT region, avg(production)
      FROM t_oil
      GROUP BY ROLLUP (region);
      region      |      avg
-----+-----
Middle East    | 1992.6036866359447005
North America | 4541.3623188405797101
                  | 2607.5139860139860140
(3 rows)
```

ROLLUP 将注入一个额外的行，它将包含整体平均值。如果您要做报告，则很可能需要汇总行。PostgreSQL 可以通过只运行一个查询来提供数据，而不是运行两个查询。你可能还会注意到第二件事；不同版本的 PostgreSQL 可能以不同的顺序返回数据。原因是，在 PostgreSQL 10.0 中，实现这些分组集的方式已经显着改善。回到 9.6 和之前，PostgreSQL 必须进行大量的排序。从版本 10.0 开始，已经可以对这些操作使用哈希查询，这在很多情况下会显着加快速度：

```
test=# explain SELECT region, avg(production)
      FROM t_oil
      GROUP BY ROLLUP (region);
                                         QUERY PLAN
-----
MixedAggregate (cost=0.00..17.31 rows=3 width=44)
  Hash Key: region
  Group Key: ()
    -> Seq Scan on t_oil  (cost=0.00..12.44 rows=644 width=16)
```

(4 rows)

如果我们希望对数据进行排序，并确保所有版本以完全相同的顺序返回数据，则必须向查询语句添加 `ORDER BY` 子句。

当然，如果您只按一列分组，也可以使用这种操作：

(7 rows)

在这个例子中，PostgreSQL 会在结果集中注入三行。一条线将注入中东，一条线注入北美。最重要的是，我们将获得整体平均线。如果我们正在构建 Web 应用程序，那么当前结果是很理想的，因为您可以通过过滤掉空值来轻松构建 GUI 界面以展示结果集。

当您想立即显示结果时，**ROLLUP** 是合适的。就个人而言，我一直用它来向最终用户展示最终结果。但是，如果您正在做报告，他们可能希望预先计算更多数据以确保具备更大的灵活性。**CUBE** 关键字是您可能一直在寻找的：

USA	9141.3478260869565217
(11 rows)	

请注意，结果中添加了更多行。 CUBE 将创建与 GROUP BY region, country + GROUP BY region + GROUP BY country + 总体平均值相同的数据。因此，整个想法是立即提取许多结果和各种级别的聚合。生成的多维数据集包含所有可能的分组组合。

ROLLUP 和 CUBE 实际上只是 GROUPING SETS 子句之上便利的功能。

使用 GROUPING SETS 子句，您可以显式列出所需的聚合：

```
test=# SELECT region, country, avg(production)
  FROM t_oil
 WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
 GROUP BY GROUPING SETS ( (), region, country);
region      | country |      avg
-----+-----+-----+
Middle East |         | 2142.911111111111111111
North America |         | 5632.2826086956521739
              |         | 3906.7692307692307692
              | Canada | 2123.2173913043478261
              | Iran   | 3631.6956521739130435
              | Oman   | 586.4545454545454545
              | USA    | 9141.3478260869565217
```

(7 rows)

在本节中，我选择了三个分组集：总体平均值，GROUP BY region 和 GROUP BY country。如果您希望 region 和 country 合并，请使用 (region, country)。

性能考察

分组集是一个强大的功能；它们有助于减少昂贵的查询数量。在内部，PostgreSQL 基本上会转向传统的 GroupAggregates 来使事情发挥作用。GroupAggregate 节点需要排序数据，因此要做好准备，因为 PostgreSQL 可能使用以下方法进行大量临时排序：

```
test=# explain SELECT region, country, avg(production)
  FROM t_oil
 WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
 GROUP BY GROUPING SETS ( (), region, country);
          QUERY PLAN
-----
GroupAggregate (cost=22.58..32.69 rows=34 width=52)
  Group Key: region
  Group Key: ()
  Sort Key: country
  Group Key: country
```

```
-> Sort (cost=22.58..23.04 rows=184 width=24)
   Sort Key: region
-> Seq Scan on t_oil
   (cost=0.00..15.66 rows=184 width=24)
   Filter: (country = ANY
   ('{USA,Canada,Iran,Oman}'::text[]))
(9 rows)
```

在 PostgreSQL 中，哈希聚合仅支持不涉及分组集的普通 GROUP BY 子句。在 PostgreSQL 10.0 中，规划器已经拥有比 PostgreSQL 9.6 更多的选项。预计新版本中的分组集会更快。

将分组集与 FILTER 子句组合在一起

在实际应用程序中，分组集通常可以与 FILTER 子句结合使用。FILTER 子句背后的想法是能够运行部分聚合。

这是一个例子：

```
test=# SELECT region,
    avg(production) AS all,
    avg(production) FILTER (WHERE year < 1990) AS old,
    avg(production) FILTER (WHERE year >= 1990) AS new
FROM t_oil
GROUP BY ROLLUP (region);
      region      | all          | old          | new
-----+-----+-----+-----+
Middle East | 1992.603686635 | 1747.325892857 | 2254.233333333
North America | 4541.362318840 | 4471.653333333 | 4624.349206349
               | 2607.513986013 | 2430.685618729 | 2801.183150183
(3 rows)
```

这里的看法是，并非所有列都将使用相同的数据进行聚合。FILTER 子句允许您有选择地将数据传递给这些聚合。在此示例中，第二个聚合将仅考虑 1990 年之前的数据，而第三个聚合将处理更新的数据，而第一个聚合将获取所有数据。



TIP 如果可以将条件移动到 WHERE 子句，这总是必须的，因为可以从表中获取更少的数据。FILTER 仅在每个聚合不需要 WHERE 子句留下的数据时才有用。

FILTER 适用于各种聚合，并提供一种简单地数据透视方式。此外，FILTER 比使用 CASE WHEN ... THEN NULL ... ELSE END 模仿相同的行为更快。您可以在这里找到一些真实的性能比 较： <https://www.cybertec-postgresql.com/en/postgresql-9-4-aggregation-filters-they-do-pay-off/>。

使用有序集合

有序集合是强大的功能，但在开发人员社区中并没有被广泛认为并且不广为人知。这个想法实际上非常简单：数据被正常分组，然后在给定一定条件的情况下对每个组内的数据进行排序。然后对该排序数据执行计算。

一个典型的例子是计算中位数。



中位数是中间值。例如，如果你赚钱的收入是中位数，收入少于和超过你的人数相同；50% 的人做得更好，50% 的人做得更差。

获得中位数的一种方法是获取排序数据并将 50% 移动到数据集中。这是一个要求 PostgreSQL 执行 WITHIN GROUP 子句的示例：

```
test=# SELECT region,
           percentile_disc(0.5) WITHIN GROUP (ORDER BY production)
      FROM t_oil
     GROUP BY 1;
region          | percentile_disc
-----+-----
Middle East    |        1082
North America |        3054
(2 rows)
```

percentile_disc 函数将跳过该组的 50% 并返回所需的值



请注意，中位数可能会显著偏离平均值。

在经济学中，中位数和平均收入之间的偏差甚至可以用于社会平等或不平等的指标。中位数与平均值相比越高，收入不平等程度越高。为了提供更大的灵活性，ANSI 标准不仅提出中值函数。相反的，percentile_disc 允许您使用 0 到 1 之间的任何值。

美妙之处在于您甚至可以使用有序集合和分组集合：

```
test=# SELECT region,
           percentile_disc(0.5) WITHIN GROUP (ORDER BY production)
      FROM   t_oil
     GROUP BY ROLLUP (1);
region          | percentile_disc
-----+-----
Middle East    |        1082
North America |        3054
                 |
                 |        1696
(3 rows)
```

在这种情况下，PostgreSQL 将再次向结果集中注入额外的行。

正如 ANSI SQL 标准所提出的，PostgreSQL 为您提供了两个百分位函数。percentile_disc 函数将返回数据集实际包含的值。如果未找到完全匹配的值，则 percentile_cont 函数将插值。以下示例显示了这是如何工作的：

```
test=# SELECT percentile_disc(0.62) WITHIN GROUP (ORDER BY id),
           percentile_cont(0.62) WITHIN GROUP (ORDER BY id)
      FROM generate_series(1, 5) AS id;
           percentile_disc | percentile_cont
-----+-----
        4             |       3.48
(1 row)
```

4 是实际存在的值 - 3.48 是被插入的值。百分位函数不是 PostgreSQL 提供的唯一函数。要查找组中最常见的值，可以使用 mode 函数功能。在展示如何使用 mode 函数的示例之前，我已经编写了一个查询，告诉我们更多关于表的内容：

```
test=# SELECT production, count(*)
      FROM t_oil
     WHERE country = 'Other Middle East'
    GROUP BY production
   ORDER BY 2 DESC
  LIMIT 4;
      production | count
-----+-----
        50         |      5
        48         |      5
        52         |      5
        53         |      4
(4 rows)
```

三个不同的值恰好出现五次。当然，mode 函数只能给我们一个：

```
test=# SELECT country, mode() WITHIN GROUP (ORDER BY production)
      FROM t_oil
     WHERE country = 'Other Middle East'
    GROUP BY 1;
      country          | mode
-----+-----
 Other Middle East | 48
(1 row)
```

最常见的值被查询并返回，但 SQL 不会告诉我们这个数字实际显示的频率。可能是这个数字只出现了一次。

理解假设的聚合

假设聚合与标准有序集合很相似。但是，它们有助于回答一个不同类型的问题：如果数据中存在值，结果会是什么？正如您所看到的，这不是关于数据库中的值，而是关于实际存在某个值的结果。

PostgreSQL 提供的唯一假设函数是 rank:

```
test=# SELECT region,
           rank(9000) WITHIN GROUP
                     (ORDER BY production DESC NULLS LAST)
      FROM t_oil
     GROUP BY ROLLUP (1);
    region      | rank
-----+-----
Middle East   |  21
North America |  27
               |  47
(3 rows)
```

它告诉我们：如果有人每天生产 9,000 桶，它将在北美排名第 27 位，在中东排名第 21 位。



TIP 在这个例子中，我使用了 `NULLS LAST`。对数据进行排序时，空值通常在最后。但是，如果排序顺序相反，则空值仍应位于列表的末尾。`NULLS LAST` 准备地保证了这一点。

利用窗口函数和分析

现在我们已经讨论了有序集合，现在是时候看一下窗口函数了。

聚合遵循一个相当简单的原则；占用很多行并将它们转换为更少的聚合行。窗口函数是不同的。它将当前行与分组中的所有行进行比较。返回的行数不会更改。

这是一个例子：

```
test=# SELECT avg(production) FROM t_oil;
      avg
-----
2607.5139
(1 row)

test=# SELECT country, year, production,
           consumption, avg(production) OVER ()
      FROM t_oil
     LIMIT 4;
country  | year  | production | consumption |      avg
-----+-----+-----+-----+-----+
USA      | 1980  | 10000000 | 10000000 | 2607.5139
Canada   | 1980  | 10000000 | 10000000 | 2607.5139
Mexico   | 1980  | 10000000 | 10000000 | 2607.5139
Brazil   | 1980  | 10000000 | 10000000 | 2607.5139
(4 rows)
```

```

-----+-----+-----+-----+
USA    | 1965 |      9014 |     11522 | 2607.5139
USA    | 1966 |      9579 |     12100 | 2607.5139
USA    | 1967 |     10219 |     12567 | 2607.5139
USA    | 1968 |     10600 |     13405 | 2607.5139
(4 rows)

```

我们数据集中的平均产量约为每天 260 万桶。此查询的目标是将此平均值添加为列。现在可以轻松地将当前行与总体平均值进行比较。

请记住，OVER 子句是必不可少的。没有它，PostgreSQL 无法处理查询：

```

test=# SELECT country, year, production, consumption, avg(production) FROM
t_oil;
ERROR: column "t_oil.country" must appear in the GROUP BY clause or be
used
      in an aggregate function
LINE 1: SELECT country, year,  production, consumption, avg(productio...

```

这实际上是有道理的，因为必须精确定义平均值。数据库引擎不能靠猜测一个任意值。



其他数据库引擎可以接受聚合函数而无需 OVER 甚至 GROUP BY 子句。但是，从逻辑的角度来看，这是错误的，最重要的是，违反了 SQL 的标准。

分区数据

到目前为止，使用子查询也可以容易地实现相同的结果。但是，如果您想要的不仅仅是整体平均值，那么由于子查询的复杂性，会把你的查询变成噩梦。假设，您不仅想要整体平均值，还想要处理的 country 的平均值。那么您需要一个 PARTITION BY 子句：

```

test=# SELECT country, year, production, consumption,
          avg(production) OVER (PARTITION BY country)
FROM t_oil;
country | year | production | consumption |      avg
-----+-----+-----+-----+
Canada  | 1965 |      920 |      1108 | 2123.2173
Canada  | 2010 |     3332 |      2316 | 2123.2173
Canada  | 2009 |     3202 |      2190 | 2123.2173
...
Iran    | 1966 |     2132 |       148 | 3631.6956
Iran    | 2010 |     4352 |      1874 | 3631.6956
Iran    | 2009 |     4249 |      2012 | 3631.6956
...

```

这里的要点是每个国家将生成该国的平均水平值。OVER 子句定义了我们正在查看的窗口。在这种情况下，窗口是行所属的国家。换句话说，查询返回与此国家中的所有行

相比较的行。



年份列未被排序。查询不包含显式排序顺序，因此可能是以随机顺序返回数据。请记住，除非您明确说明您想要的排序的内容，否则 SQL 不会排序输出。

基本上，PARTITION BY 子句接受任何表达式。通常，大多数人会使用列来分区数据。这是一个例子：

```
test=# SELECT year, production,
           avg(production) OVER (PARTITION BY year < 1990)
      FROM t_oil
     WHERE country = 'Canada'
    ORDER BY year;
year | production |      avg
-----+-----+
 1965 |        920 | 1631.6000000000000000
 1966 |       1012 | 1631.6000000000000000
 ...
 1990 |      1967 | 2708.4761904761904762
 1991 |      1983 | 2708.4761904761904762
 1992 |      2065 | 2708.4761904761904762
 ...
...
```

关键是使用表达式拆分数据。`year < 1990` 可以返回两个值：`true` 或 `false`。根据每年的组别，它将被分配到 1990 年以前的平均值或 1990 年后的平均值。PostgreSQL 在这里非常灵活。在实际应用程序中使用函数来确定分组成员并不罕见。

在窗口函数内排序数据

PARTITION BY 子句不是唯一可以放入 OVER 子句的东西。有时需要对窗口内的数据进行排序。ORDER BY 将以某种方式向您的聚合函数提供数据。这是一个例子：

```
test=# SELECT country, year, production,
           min(production) OVER (PARTITION BY country ORDER BY year)
      FROM t_oil
     WHERE year BETWEEN 1978 AND 1983 AND country IN ('Iran', 'Oman');
country | year | production | min
-----+-----+-----+
  Iran  | 1978 |      5302 | 5302
  Iran  | 1979 |      3218 | 3218
  Iran  | 1980 |      1479 | 1479
  Iran  | 1981 |      1321 | 1321
  Iran  | 1982 |      2397 | 1321
  Iran  | 1983 |      2454 | 1321
  Oman  | 1978 |        314 | 314
```

Oman	1979	295	295
Oman	1980	285	285
Oman	1981	330	285
...			

从 1978 年到 1983 年的数据集中选择了两个国家（伊朗和阿曼）的数据。请记住，1979 年伊朗发生了一场革命，因此这对石油的生产产生了一些影响。查询出的数据反映了这一点。

查询的作用是计算我们时间序列中某个点的最小生产量。此时，对于 SQL 学生来说，记住在 OVER 子句中使用 ORDER BY 子句排序是一个不错的方式。在此示例中，PARTITION BY 子句将为每个 country 创建一个组，并在分组内获取数据。min 函数将遍历排序过的数据并获取所需的小值。

如果您不熟悉窗口函数，那么您应该注意一些事项。无论您是否使用 ORDER BY 子句，它确实是有效的：

```
test=# SELECT country, year, production,
           min(production) OVER (),
           min(production) OVER (ORDER BY year)
      FROM t_oil
     WHERE year BETWEEN 1978 AND 1983
       AND country = 'Iran';
country | year | production | min | min
-----+-----+-----+-----+
Iran   | 1978 |      5302 | 1321 | 5302
Iran   | 1979 |      3218 | 1321 | 3218
Iran   | 1980 |      1479 | 1321 | 1479
Iran   | 1981 |      1321 | 1321 | 1321
Iran   | 1982 |      2397 | 1321 | 1321
Iran   | 1983 |      2454 | 1321 | 1321
(6 rows)
```

如果在没有 ORDER BY 的情况下使用聚合功能，它将自动获取窗口内整个数据集的最小值。如果存在 ORDER BY 子句，则不会发生这种情况。在这种情况下，根据您定义的顺序，它始终是到此为止的最小值。

使用变动的窗口函数

到目前为止，我们在查询中使用的窗口函数是静态的。但是，对于诸如移动平均线之类的计算，这还不够。移动平均线需要一个随着数据处理而移动的变动的统计窗口。

以下是如何实现移动平均线的示例：

```
test=# SELECT country, year, production,
           min(production)
```

```

OVER (PARTITION BY country
      ORDER BY year ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
FROM  t_oil
WHERE year BETWEEN 1978 AND 1983
      AND country IN ('Iran', 'Oman');
country | year | production | min
-----+-----+-----+
Iran    | 1978 |      5302 | 3218
Iran    | 1979 |      3218 | 1479
Iran    | 1980 |      1479 | 1321
Iran    | 1981 |      1321 | 1321
Iran    | 1982 |      2397 | 1321
Iran    | 1983 |      2454 | 2397
Oman   | 1978 |       314  | 295
Oman   | 1979 |       295  | 285
Oman   | 1980 |       285  | 285
Oman   | 1981 |       330  | 285
Oman   | 1982 |       338  | 330
Oman   | 1983 |       391  | 338
(12 rows)

```

最重要的是移动窗口应该与 `ORDER BY` 子句一起使用。否则，将会出现重大问题。PostgreSQL 会完成查询请求，但是结果确是完全错误的。请记住，将数据提供给移动窗口而不先进行排序会导致随机结果。

`ROWS BETWEEN 1 PRECEDING 和 1 FOLLOWING` 定义了数据窗口。在此示例中，最多将使用三行：当前行，前一行和当前行之后的行。要说明移动窗口的工作原理，请查看以下示例：

```

test=# SELECT *, array_agg(id)
          OVER (ORDER BY id ROWS BETWEEN 1 PRECEDING AND 1
          FOLLOWING)
     FROM  generate_series(1, 5) AS id;
id  | array_agg
----+-----
1  | {1,2}
2  | {1,2,3}
3  | {2,3,4}
4  | {3,4,5}
5  | {4,5}
(5 rows)

```

`array_agg` 函数将值列表转换为 PostgreSQL 数组。它将有助于解释移动窗口的运行方式。

实际上，这个简单的查询有一些非常重要的方面。你看到的是第一个数组只包含两个

值。 1 之前没有条目，因此数组未满。 PostgreSQL 不会添加空条目，因为无论如何它们都会被聚合忽略。 在数据结尾处是同样的情况。

但是，移动窗口提供更多。 有多个关键字可用于指定移动窗口查询：

```
test=# SELECT *,  
          array_agg(id) OVER (ORDER BY id ROWS BETWEEN  
                                UNBOUNDED PRECEDING AND 0 FOLLOWING)  
FROM generate_series(1, 5) AS id;  
id | array_agg  
----+-----  
 1 | {1}  
 2 | {1,2}  
 3 | {1,2,3}  
 4 | {1,2,3,4}  
 5 | {1,2,3,4,5}  
(5 rows)
```

UNBOUNDED PRECEDING 关键字表示当前行之前的所有内容都将在窗口中。 UNBOUNDED PRECEDING 的对应物是 UNBOUNDED FOLLOWING。 我们来看下面的例子：

```
test=# SELECT *,  
          array_agg(id) OVER (ORDER BY id ROWS BETWEEN 2 FOLLOWING AND  
                                UNBOUNDED  
                                FOLLOWING)  
FROM generate_series(1, 5) AS id;  
id | array_agg  
----+-----  
 1 | {3,4,5}  
 2 | {4,5}  
 3 | {5}  
 4 |  
 5 |  
(5 rows)
```

如您所见，也可以使用将来的窗口。 PostgreSQL 在这里非常灵活。

抽象窗口函数子句

窗口函数允许我们将列添加到已动态计算的结果集中。 但是，许多列基于同一窗口是一种常见现象。 将相同的查询子句一遍又一遍地放入查询中绝对不是一个好主意，因为您的查询难以阅读，因此难以维护。

WINDOW 子句允许开发人员预先定义窗口并在查询中的各个位置使用它。 下面是它的工作原理：

```
SELECT country, year, production,
```

```

        min(production) OVER (w),
        max(production) OVER (w)
FROM t_oil
WHERE country = 'Canada'
    AND year BETWEEN 1980
    AND 1985
WINDOW w AS (ORDER BY year);
country | year   | production | min   | max
-----+-----+-----+-----+
Canada  | 1980   |      1764 | 1764 | 1764
Canada  | 1981   |      1610 | 1610 | 1764
Canada  | 1982   |      1590 | 1590 | 1764
Canada  | 1983   |      1661 | 1590 | 1764
Canada  | 1984   |      1775 | 1590 | 1775
Canada  | 1985   |      1812 | 1590 | 1812
(6 rows)

```

该示例显示 min 和 max 将使用相同的窗口查询子句。

当然，有可能有不止一个 WINDOW 子句 - PostgreSQL 在这方面不会对用户施加严格的限制。

利用系统自带的窗口函数

在介绍了基本概念之后，现在是时候看看 PostgreSQL 开箱即用的窗口函数了。您已经看到窗口适用于所有标准聚合函数。除了这些功能之外，PostgreSQL 还提供了一些专门用于窗口和分析的附加功能。

在本节中，将解释和讨论一些非常重要的功能。

rank 和 dense_rank 函数

在我看来，rank () 和 dense_rank () 函数是 SQL 中最突出的函数。rank () 函数返回其窗口中当前行的编号。计数从 1 开始。

这是一个例子：

```

test=# SELECT year, production,
           rank() OVER (ORDER BY production)
FROM   t_oil
WHERE country = 'Other Middle East'
ORDER BY rank
LIMIT 7;
year | production | rank
-----+-----+-----+
2001 |    47      | 1

```

2004		48		2
2002		48		2
1999		48		2
2000		48		2
2003		48		2
1998		49		7

(7 rows)

Rank 函数将对数据集中的元组进行编号。请注意，我的示例中的许多行是相等的。因此，等级将直接从 2 跳到 7，因为许多值是相同的。如果你想避免这种情况，可以使 use dense_rank () 函数：

```
test=# SELECT year, production,
           dense_rank() OVER (ORDER BY production)
      FROM t_oil
     WHERE country = 'Other Middle East'
   ORDER BY dense_rank
  LIMIT 7;
```

year		production		dense_rank
2001		47		1
2004		4		2
...				
2003		48		2
1998		49		3

(7 rows)

PostgreSQL 将更紧密地生成数值，不会有任何间隔。

Ntile () 函数

某些应用程序要求将数据拆分为理想相同的组。 ntile () 函数将完全适合您。

以下示例显示如何将数据拆分为组：

```
test=# SELECT year, production,
           ntile(4) OVER (ORDER BY production)
      FROM t_oil
     WHERE country = 'Iraq'
       AND year BETWEEN 2000 AND 2006;
```

year		production		ntile
2003		1344		1
2005		1833		1
2006		1999		2

2004	2030	2
2002	2116	3
2001	2522	3
2000	2613	4
(7 rows)		

查询将数据拆分为四个组。麻烦的是只选择了七行，这使得无法创建四个偶数组。正如你所看到的，PostgreSQL 将填满前三组并使最后一组变小。你可以依赖这样一个事实，即最后的组总是比其他组小一点。



在此示例中，仅使用了少量的行。在实际应用程序中，将涉及数百万行，因此如果组不完全相等则没有问题。

`ntile()` 函数通常不单独使用。当然，将一个组 ID 分配给一行是有帮助的。但是，在实际应用程序中，人们希望在这些组之上执行计算。假设您要为数据创建四分位分布。下面是它的工作原理：

```
test=# SELECT grp, min(production), max(production), count(*)
  FROM (
    SELECT year, production,
           ntile(4) OVER (ORDER BY production) AS grp
      FROM t_oil
     WHERE country = 'Iraq'
  ) AS x
 GROUP  BY ROLLUP (1);
grp | min   | max   | count
----+-----+-----+
 1  | 285   | 1228  | 12
 2  | 1313  | 1977  | 12
 3  | 1999  | 2422  | 11
 4  | 2428  | 3489  | 11
      | 285   | 3489  | 46
(5 rows)
```

最重要的是，计算不能一步完成。在 Cybertec (<https://www.cybertec-postgresql.com>) 上进行 SQL 培训课程时，我尝试向学生解释，如果您不知道如何一次完成所有操作，请考虑使用子查询。在数据分析中，这通常是个好主意。在此示例中，在子查询中完成的第一件事是将组标签附加到每个组。然后在主查询中获取并处理这些组。

结果已经可以在实际应用程序中使用（可能是图表旁边的图例，依此类推）。

lead() 和 lag() 函数

虽然 `ntile()` 函数对于将数据集拆分成组是必不可少的，但是 `lead()` 和 `lag()` 函数

在此处移动结果集中的行。一个典型的用例是计算从一年到下一年的生产差异，如下例所示：

```
test=# SELECT year, production,
           lag(production, 1) OVER (ORDER BY year)
      FROM t_oil
     WHERE country = 'Mexico'
       LIMIT 5;
year | production | lag
-----+-----+
1965 |      362 |
1966 |      370 | 362
1967 |      411 | 370
1968 |      439 | 411
1969 |      461 | 439
(5 rows)
```

在实际计算生产变化之前，有必要坐下来看看 `lag()` 函数实际上做了什么。您可以看到该列移动了一行。数据按 `ORDER BY` 子句中的定义移动。在我的例子中，它意味着下移。如果是 `ORDER BY DESC` 子句当然会将数据上移。

从现在开始，查询很简单：

```
test=# SELECT year, production,
           production - lag(production, 1) OVER (ORDER BY year)
      FROM t_oil
     WHERE country = 'Mexico'
       LIMIT 3;
year | production | ?column?
-----+-----+
1965 |      362 |
1966 |      370 |      8
1967 |      411 |      41
(3 rows)
```

您所要做的就是像计算任何其他列一样计算差异。请注意，`lag()` 函数有两个参数。第一个表示要显示的列。第二列告诉 PostgreSQL 你要移动多少行。因此，输入 7 表示一切都是七行。

请注意，第一个值为 Null（与没有先前值的所有其他滞后行一样）。

`lead()` 函数是 `lag()` 函数的对应部分；它将向上移动行而不是向下移动：

```
test=# SELECT year, production,
           production - lead(production, 1) OVER (ORDER BY year)
      FROM t_oil
     WHERE country = 'Mexico'
```

```

LIMIT 3;
year | production | ?column?
-----+-----+
1965 | 362 | -8
1966 | 370 | -41
1967 | 411 | -28
(3 rows)

```

基本上，PostgreSQL 也会接受 lead 和 lag 的负值。因此，`lag (production, -1)` 是 `lead (production, 1)` 的替代品。但是，使用正确的功能以您想要的方向移动数据肯定更清晰。

到目前为止，您已经看到了如何滞后于一列。在大多数应用程序中，滞后单个值将是大多数开发人员使用的标准情况。关键是，PostgreSQL 可以做更多的事情。可能会滞后延迟整行：

```

test=# \x
Expanded display is on.
test=# SELECT year, production,
          lag(t_oil, 1) OVER (ORDER BY year)
     FROM t_oil
    WHERE country = 'USA'
   LIMIT 3;
-[ RECORD 1 ]-----
year      | 1965
production | 9014
lag       |
-[ RECORD 2 ]-----
year      | 1966
production | 9579
lag       | ("North America",USA,1965,9014,11522)
-[ RECORD 3 ]-----
year      | 1967
production | 10219
lag       | ("North America",USA,1966,9579,12100)

```

这里的美妙之处在于，不仅可以将单个值与前一行进行比较。但是，问题是 PostgreSQL 将整个行作为复合类型返回，因此难以使用。要剖析复合类型，可以使用括号和星号：

```

test=# SELECT year, production,
          (lag(t_oil, 1) OVER (ORDER BY year)).*
     FROM t_oil
    WHERE country = 'USA'
   LIMIT 3;
year | prod | region      | country | year | prod | consumption
-----+-----+-----+-----+-----+-----+
1965 | 9014 |           |        | 1965 | 9014 |           |

```

```
1966 | 9579 | N. America |      USA | 1965 | 9014 | 11522
1967 | 10219 | N. America |      USA | 1966 | 9579 | 12100
(3 rows)
```

为什么这有用？ 滞后整行将使得可以查看数据是否已被多次插入。在时间序列中检测重复行（或接近重复的行）非常简单。

看看以下示例：

```
test=# SELECT *
  FROM (SELECT t_oil, lag(t_oil) OVER (ORDER BY year)
        FROM   t_oil
       WHERE country = 'USA'
      ) AS x
 WHERE t_oil = lag;
t_oil  | lag
-----+-----
(0 rows)
```

当然，样本数据不包含重复项。但是，在实际示例中，重复项很容易发生，即使没有主键，也很容易检测到它们。



t_oil 行实际上就是整行。子查询返回的滞后也是一个完整的行。在 PostgreSQL 中，可以直接比较复合类型，以防字段相同。PostgreSQL 将简单地比较一个字段。

first_value () , nth_value () 和 last_value () 函数

有时，有必要根据数据窗口的第一个值计算数据。不出所料，执行此操作的函数是 first_value ()：

```
test=# SELECT year, production,
            first_value(production) OVER (ORDER BY year)
  FROM   t_oil
 WHERE  country = 'Canada'
 LIMIT  4;
year  | production | first_value
-----+-----+-----
1965 |      920 |      920
1966 |     1012 |      920
1967 |     1106 |      920
1968 |     1194 |      920
(4 rows)
```

同样，需要一个排序顺序来告诉系统第一个值实际在哪里。然后 PostgreSQL 将相同的

值放入最后一列。如果要在窗口中找到最后一个值，只需使用 `last_value()` 函数而不是 `first_value()` 函数。

如果你对第一个或最后一个值不感兴趣但是正在寻找中间的东西，PostgreSQL 提供了 `nth_value()` 函数：

```
test=# SELECT year, production,
           nth_value(production, 3) OVER (ORDER BY year)
      FROM t_oil
     WHERE country = 'Canada';
   year | production | nth_value
-----+-----+
  1965 |       920 |
  1966 |      1012 |
  1967 |      1106 |      1106
  1968 |      1194 |      1106
...

```

在此，第三个值将被放入最后一列。但是，请注意前两行是空的。问题是，当 PostgreSQL 开始浏览数据时，第三个值还不知道。因此，添加 `null`。现在的问题是，我们怎么样使时间序列更完整，并用数据替换这两个空值？

这是一种方法：

```
test=# SELECT *, min(nth_value) OVER ()
      FROM (
        SELECT year, production,
               nth_value(production, 3) OVER (ORDER BY year)
          FROM t_oil
         WHERE country = 'Canada'
      ) AS x
     LIMIT 4;
   year | production | nth_value | min
-----+-----+-----+
  1965 |       920 |          | 1106
  1966 |      1012 |          | 1106
  1967 |      1106 |      1106 | 1106
  1968 |      1194 |      1106 | 1106
(4 rows)
```

子查询将创建不完整的时间序列。最上层的 `SELECT` 子句将完成数据查询。这里的情况是完成数据查询语句可能会更复杂，因此一个子查询语句可能会使得添加更复杂的逻辑更有效，而不是一步完成。

row_number () 函数

本节讨论的最后一个函数是 `row_number()` 函数，它可以简单地用于返回虚拟 ID。听起来很简单？这里是：

```
test=# SELECT country, production,  
           row_number() OVER (ORDER BY production)  
      FROM t_oil  
     LIMIT 3;  
  
country | production | row_number  
-----+-----+  
Yemen   |      10 |      1  
Syria   |      21 |      2  
Yemen   |      26 |      3  
(3 rows)
```

`row_number()` 函数只是为行分配一个数字。绝对没有重复值。

这里有趣的一点是，即使没有顺序也可以这样做（如果它与你无关）：

```
test=# SELECT country, production,  
           row_number() OVER()  
      FROM t_oil  
     LIMIT 3;  
  
country | production | row_number  
-----+-----+  
USA     |    9014 |      1  
USA     |    9579 |      2  
USA     |   10219 |      3  
(3 rows)
```

编写自己的聚合函数

在本书中，您可以了解 PostgreSQL 提供的大多数自带函数。但是，SQL 提供的功能可能还不够。好消息是可以将自己实现的聚合函数添加到数据库引擎中。在本节中，您将了解如何完成。

创建简单的聚合

出于本示例的目的，目标是解决一个非常简单的问题。如果客户乘坐出租车，他们通常需要为乘坐出租车付费 - 例如 2.50 欧元。然后，让我们假设每公里，客户必须支付 2.20 欧元。现在的问题是，旅行的总价是多少？

当然，这个例子很简单，无需自定义聚合即可解决；但是，让我们看看它是如何工作的。

首先，创建一些测试数据：

```
test=# CREATE TABLE t_taxi (trip_id int, km numeric);
CREATE TABLE
test=# INSERT INTO t_taxi
VALUES (1, 4.0), (1, 3.2), (1, 4.5), (2, 1.9), (2, 4.5);
INSERT 0 5
```

要创建聚合，PostgreSQL 提供 CREATE AGGREGATE 命令。这个命令的语法已经变得如此强大并且随着时间的推移而变得如此强大，以至于在本书中包含它的输出就没有意义了。相反，我建议去查看 PostgreSQL 文档，可以在 <https://www.postgresql.org/docs/devel/static/sql-createaggregate.html> 找到。

编写聚合时需要的第一件事是一个函数，它为每一行调用。它将采用中间值和从处理的行中获取的数据。这是一个例子：

```
test=# CREATE FUNCTION taxi_per_line (numeric, numeric)
RETURNS numeric AS
$$
BEGIN
    RAISE NOTICE 'intermediate: %, per row:  %', $1, $2;
    RETURN $1 + $2*2.2;
END;
$$
LANGUAGE 'plpgsql';
CREATE FUNCTION
```

现在，已经可以创建一个简单的聚合：

```
test=# CREATE AGGREGATE taxi_price (numeric)
(
    INITCOND = 2.5,
    SFUNC = taxi_per_line,
    STYPE = numeric
);
CREATE AGGREGATE
```

如上所述，每次出行都是以 2.50 欧元的价格进入出租车，这是由 INITCOND（初始条件）定义的。它代表每个组的起始值。然后为组中的每一行调用一个函数。在我的例子中，这个函数是 taxi_per_line 并且已经被定义了。如您所见，它需要两个参数。第一个参数是中间值。这些附加参数是用户传递给函数的参数。

以下语句显示了传递的数据，时间和方式：

```
test=# SELECT trip_id, taxi_price(km) FROM t_taxi GROUP  BY 1;
NOTICE:  intermediate: 2.5, per row:  4.0
NOTICE:  intermediate: 11.30, per row:  3.2
NOTICE:  intermediate: 18.34, per row:  4.5
```

```

NOTICE: intermediate: 2.5, per row: 1.9
NOTICE: intermediate: 6.68, per row: 4.5
trip_id | taxi_price
-----+
 1 |    28.24
 2 |    16.58
(2 rows)

```

系统以行程 1 和 € 2.50 (初始条件) 开始。然后增加 4 公里。总的来说，现在的价格是 $2.50 + 4 \times 2.2$ 。然后，添加下一行，将添加 3.2×2.2 ，依此类推。因此，第一次旅行的费用为 28.24 欧元。

然后下一次旅行开始。同样，有一个新的 init 条件，PostgreSQL 将每行调用一个函数。

在 PostgreSQL 中，聚合也可以自动用作窗口函数。无需其他步骤 - 您可以直接使用聚合：

```

test=# SELECT *, taxi_price(km) OVER (PARTITION BY trip_id ORDER BY km)
  FROM t_taxi;
NOTICE: intermediate: 2.5, per row: 3.2
NOTICE: intermediate: 9.54, per row: 4.0
NOTICE: intermediate: 18.34, per row: 4.5
NOTICE: intermediate: 2.5, per row: 1.9
NOTICE: intermediate: 6.68, per row: 4.5
trip_id | km    | taxi_price
-----+-----+
 1     | 3.2  |    9.54
 1     | 4.0  |   18.34
 1     | 4.5  |   28.24
 2     | 1.9  |    6.68
 2     | 4.5  |   16.58
(5 rows)

```

查询的作用是为我们提供行程中给定点的价格。

我们定义的聚合将每行调用一个函数。但是，用户如何计算平均值？如果不添加 FINALFUNC 函数，则无法进行此类计算。为了演示 FINALFUNC 如何工作，我们必须扩展我们的例子。假设客户一离开出租车就想给出租车司机 10% 的小费。一旦知道总价，就必须在最后添加 10%。这就是 FINALFUNC 开始的地方。以下是它的工作原理：

```

test=# DROP AGGREGATE taxi_price(numeric);
DROP AGGREGATE

```

首先，旧聚合被删除。然后，定义 FINALFUNC。它将获得中间结果作为参数并发挥其魔力：

```

test=# CREATE FUNCTION taxi_final (numeric)

```

```
RETURNS numeric AS
$$
SELECT $1 * 1.1;
$$
LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

计算非常简单，在这种情况下 - 如前所述，10%被添加到最终总和中。

部署该功能后，就可以重新创建聚合：

```
test=# CREATE AGGREGATE taxi_price (numeric)
(
    INITCOND = 2.5,
    SFUNC   = taxi_per_line,
    STYPE   = numeric,
    FINALFUNC = taxi_final
);
CREATE AGGREGATE
```

最后，价格会比以前略高一些：

```
test=# SELECT trip_id, taxi_price(km) FROM t_taxi GROUP BY 1;
NOTICE:  intermediate: 2.5, per row: 4.0
...
trip_id | taxi_price
-----+-----
 1 |     31.064
 2 |     18.238
(2 rows)
```

PostgreSQL 会自动处理所有分组等工作。

对于简单计算，可以将简单数据类型用于中间结果。但是，并非所有操作都可以通过传递简单的数字和文本来完成。幸运的是，PostgreSQL 允许使用复合数据类型，可以用作中间结果。

想象一下，你想要计算一些数据的平均值，也许是一个时间序列。中间结果可能如下所示：

```
test=# CREATE TYPE my_intermediate AS (c int4, s numeric);
CREATE TYPE
```

随意组成任何符合您目的的任意类型。只需将其作为第一个参数传递，并根据需要添加数据作为附加参数。

添加对并行查询的支持

你刚刚看到的是一个简单的聚合，它不支持并行查询等等。为了解决这些挑战，接下来的几个例子都是关于改进和加速的。

创建聚合时，您可以选择定义以下内容：

```
[, PARALLEL = { SAFE | RESTRICTED | UNSAFE }]
```

默认情况下，聚合不支持并行查询。但是，出于性能原因，明确说明聚合的能力是有意义的：

- **UNSAFE**: 在此模式下，不允许并行查询
- **RESTRICTED**: 在这种模式下，聚合可以并行模式执行，但执行仅限于并行组的领导者
- **SAFE**: 在此模式下，它为并行查询提供完全支持

如果将函数标记为 **SAFE**，则必须记住该函数不得有副作用。执行顺序不得对查询结果产生影响。只有这样，PostgreSQL 才能被允许并行执行操作。没有副作用的函数的例子是 `sin(x)` 和 `length(s)`。**IMMUTABLE** 函数是很好的选择，因为它们可以保证在给定相同输入的情况下返回相同的结果。如果适用某些限制，**STABLE** 功能可以工作。

提高效率

到目前为止定义的聚合已经可以实现很多。但是，如果您使用移动窗口，函数调用的数量将会爆炸。这里是发生的事情：

```
test=# SELECT taxi_price(x::numeric)
          OVER (ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
  FROM generate_series(1, 5) AS x;
NOTICE: intermediate: 2.5, per row: 1
NOTICE: intermediate: 4.7, per row: 2
NOTICE: intermediate: 9.1, per row: 3
NOTICE: intermediate: 15.7, per row: 4
NOTICE: intermediate: 2.5, per row: 2
NOTICE: intermediate: 6.9, per row: 3
NOTICE: intermediate: 13.5, per row: 4
NOTICE: intermediate: 22.3, per row: 5
...
...
```

对于每一行，PostgreSQL 将处理完整的窗口。如果移动窗口范围很大，效率会降低。为了解决这个问题，可以扩展我们的聚合。在此之前，可以删除旧聚合：

```
DROP AGGREGATE taxi_price(numeric);
```

基本上，需要两个功能。`msfunc` 函数会将窗口中的下一行添加到中间结果中：

```
CREATE FUNCTION taxi_msfunc(numeric, numeric)
RETURNS numeric AS
```

```
$$
BEGIN
    RAISE NOTICE 'taxi_msfunc called with % and %', $1, $2;
    RETURN $1 + $2;
END;
$$ LANGUAGE 'plpgsql' STRICT;
```

minvfunc 函数将从中间结果中删除窗口中掉出的值:

```
CREATE FUNCTION taxi_minvfunc(numeric, numeric) RETURNS numeric AS
$$
BEGIN
    RAISE NOTICE 'taxi_minvfunc called with % and %', $1, $2;
    RETURN $1 - $2;
END;
$$
LANGUAGE 'plpgsql' STRICT;
```

在这个例子中，我们所做的只是加减。在更复杂的示例中，计算可以是任意的、复杂的。

下一个语句显示了如何重新创建聚合:

```
CREATE AGGREGATE taxi_price (numeric)
(
    INITCOND = 0,
    STYPE   = numeric,
    SFUNC   = taxi_per_line,
    MSFUNC  = taxi_msfunc,
    MINVFUNC = taxi_minvfunc,
    MSTYPE  = numeric
);
```

现在让我们再次运行相同的查询:

```
test# SELECT taxi_price(x::numeric)
           OVER (ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
FROM    generate_series(1, 5) AS x;
NOTICE: taxi_msfunc called with 1 and 2
NOTICE: taxi_msfunc called with 3 and 3
NOTICE: taxi_msfunc called with 6 and 4
NOTICE: taxi_minfunc called with 10 and 1
NOTICE: taxi_msfunc called with 9 and 5
NOTICE: taxi_minfunc called with 14 and 2
NOTICE: taxi_minfunc called with 12 and 3
NOTICE: taxi_minfunc called with 9 and 4
```

函数调用的数量急剧减少。每行只需执行少量固定的调用。不再需要再次计算相同的层。

引用假设的聚合

编写聚合并不难，对于执行更复杂的操作非常有用。在本节中，计划是编写一个假设的聚合，这已在本章中讨论过。

实现假设聚合与编写普通聚合没有太大区别。

真正困难的部分是弄清楚何时实际使用一个。为了使这一部分尽可能易于理解，我决定包含一个简单的例子：给定一个特定的顺序，如果我们将 abc 添加到字符串的末尾，结果会是什么？

下面是它的工作方式：

```
CREATE AGGREGATE name ([ [ argmode ] [ argname ] arg_data_type [, ...
]
    ORDER BY [ argmode ] [ argname ] arg_data_type
    [, ...])
(
    SFUNC = sfunc,
    STYPE = state_data_type
    [, SSPACE = state_data_size ] [, FINALFUNC = ffunc ]
    [, FINALFUNC_EXTRA ]
    [, INITCOND = initial_condition ]
    [, PARALLEL = { SAFE | RESTRICTED | UNSAFE } ] [, HYPOTHETICAL ]
)
```

将需要两个功能。将为每一行调用 sfunc 函数：

```
CREATE FUNCTION hypo_sfunc(text, text)
RETURNS text AS
$$
BEGIN
    RAISE NOTICE 'hypo_sfunc called with % and %', $1, $2;
    RETURN $1 || $2;
END;
$$ LANGUAGE 'plpgsql';
```

两个参数将传递给该过程。逻辑与以前相同。正如我们之前所做的那样，可以定义最终的函数调用：

```
CREATE FUNCTION hypo_final(text, text, text)
RETURNS text AS
$$
BEGIN
    RAISE NOTICE 'hypo_final called with %, %, and %',
    $1, $2, $3;
```

```
RETURN $1 || $2;
END;
$$ LANGUAGE 'plpgsql';
```

一旦这些功能到位，就可以创建假设聚合：

```
CREATE AGGREGATE whatif(text ORDER BY text)
(
    INITCOND = 'START',
    STYPE   = text,
    SFUNC   = hypo_sfunc,
    FINALFUNC = hypo_final,
    FINALFUNC_EXTRA = true,
    HYPOTHETICAL
);
```

请注意，聚合已被标记为假设，因此 PostgreSQL 将知道它实际上是什么类型的聚合。

创建聚合后，可以运行它：

```
test=# SELECT whatif('abc'::text) WITHIN GROUP (ORDER BY id::text)
      FROM generate_series(1, 3) AS id;
NOTICE:  hypo_sfunc called with START and 1
NOTICE:  hypo_sfunc called with START1 and 2
NOTICE:  hypo_sfunc called with START12 and 3
NOTICE:  hypo_final called with START123, abc, and <NULL>
whatif
-----
START123abc
(1 row)
```

理解所有这些聚合的关键是真正完全了解何时调用各种函数以及整个机器如何工作。

小结

在本章中，您了解了 SQL 提供的高级功能。在简单聚合之上，PostgreSQL 提供有序集，分组集，窗口函数和递归，以及创建自定义聚合的接口。在数据库中运行聚合的优点是代码易于编写，并且数据库引擎通常在效率方面具有优势。

在第 5 章“日志文件和系统统计信息”中，我们将把注意力转向更多管理任务，例如日志文件处理，理解系统统计信息和实施监控。

5. 日志文件和系统统计信息

在第4章“处理高级SQL”中，您学到了很多关于高级SQL以及以不同方式查看SQL的方法。但是，数据库工作不仅包括优化各种类型的SQL。有时，它也是为了让数据库的事情以专业的方式运行。要做到这一点，关注系统统计信息，日志文件等非常重要。监控是如何专业运行数据库的关键。幸运的是，PostgreSQL有许多功能可以帮助您监视数据库，您将学习如何在本章中使用它们。

在本章中，您将了解以下主题：

- 收集运行时统计信息
- 创建日志文件
- 收集重要信息
- 理解数据库统计信息

到本章结束时，您将能够正确配置PostgreSQL的日志记录基础结构，并以最专业的方式处理日志文件。

收集运行时统计信息

您真正需要了解的第一件事是使用和理解PostgreSQL自带的统计数据。在我个人看来，如果没有先收集必要的数据来做出谨慎的决定，就无法提高性能和可靠性。

本节将指导您完成PostgreSQL的运行时统计信息，并详细说明如何从数据库设置中提取更多运行时信息。

使用PostgreSQL系统视图

PostgreSQL提供了大量系统视图，允许管理员和开发人员深入了解系统中的实际情况。麻烦的是很多人实际上收集了所有这些数据但却无法真正理解它。一般规则是这样的：无论如何都不能为你不理解的东西绘制图形。因此，本节的目标是阐明PostgreSQL提供的内容，以期让人们更容易利用他们可以使用的内容。

检查实时流量

每当我检查一个系统时，都会有一个系统视图，我希望在深入挖掘之前先检查一下。当然，我在谈论`pg_stat_activity`。这种观点背后的想法是让你有机会弄清楚现在发生了什么。

下面是它的工作方式：

```
test=# \d pg_stat_activity
View "pg_catalog.pg_stat_activity"
```

Column	Type	Collation	Nullable
Default			
<hr/>			
datid	oid		
datname	name		
pid	integer		
usesysid	oid		
username	name		
application_name	text		
client_addr	inet		
client_hostname	text		
client_port	integer		
backend_start	timestamp with time zone		
xact_start	timestamp with time zone		
query_start	timestamp with time zone		
state_change	timestamp with time zone		
wait_event_type	text		
wait_event	text		
state	text		
backend_xid	xid		
backend_xmin	xid		
query	text		
backend_type	text		

此外，`pg_stat_activity` 将为每个活动连接提供一行。

您将看到数据库的内部对象 ID (`datid`)，某人所连接的数据库的名称以及为此连接提供的进程 ID (`pid`)。最重要的是，PostgreSQL 会告诉你谁连接 (`username`; 注意丢失的 `r`) 和该用户的内部对象 ID (`usesysid`)。

然后，有一个名为 `application_name` 的字段，值得更广泛地评论。通常，终端用户可以自由设置 `application_name`:

```
test=# SET application_name TO 'www.cybertec-postgresql.com';
SET
test=# SHOW application_name;
 application_name
-----
 www.cybertec-postgresql.com
(1 row)
```

关键在于：让我们假设有数千个连接来自单个 IP。作为管理员，您能说出现在特定连接的实际情况吗？您可能不会全心全意地了解所有 SQL。如果客户端足够友好以设置 `application_name` 参数，则可以更容易地了解连接的目的是什么。在我的示例中，我将名称设置为连接所属的域。这样可以轻松找到可能导致类似问题的类似连接。

接下来的三列（client_）将告诉您连接的来源。PostgreSQL 将显示 IP 地址和（如果已配置）甚至主机名。

此外，backend_start 将告诉您何时启动某个连接，xact_start 指示事务何时启动。然后，有 query_start 和 state_change。回到过去的黑暗时代，PostgreSQL 只会显示活动查询。在查询比今天花费更长时间的时候，这是有意义的。在现代硬件上，OLTP 查询可能只消耗几分之一毫秒，因此很难捕获此类查询，从而造成潜在危害。解决方案是显示活动查询或您正在查看的连接执行的上一个查询。

您可能会看到以下内容：

```
test=# SELECT pid, query_start, state_change, state, query
  FROM pg_stat_activity;
...
-[ RECORD 2 ] +-----
pid          | 28001
query_start   | 2018-11-05 10:03:57.575593+01
state_change  | 2018-11-05 10:03:57.575595+01
state         | active
query         | SELECT pg_sleep(10000000);
```

在这种情况下，您可以看到 pg_sleep 正在第二个连接中执行。一旦此查询终止，输出将更改：

```
-[ RECORD 2 ]+-----
pid          | 28001
query_start   | 2018-11-05 10:03:57.575593+01
state_change  | 2018-11-05 10:05:10.388522+01
state         | idle
query         | SELECT pg_sleep(10000000);
```

该查询现在标记为空闲。state_change 和 query_start 之间的区别是查询需要执行的时间。

因此，pg_stat_activity 将为您提供有关系统当前状态的概述。新的 state_change 字段使得更有可能发现昂贵的查询。

现在的问题是：一旦你发现了错误的查询，你怎么能真正摆脱它们？PostgreSQL 提供了两个函数来处理这些事情：pg_cancel_backend 和 pg_terminate_backend。pg_cancel_backend 函数将终止查询，但会保持连接。

pg_terminate_backend 函数有点激进，它将与查询一起终止整个数据库连接。

如果您想断开除自己以外的所有其他用户，请按以下步骤操作：

```
test=# SELECT pg_terminate_backend(pid)
  FROM pg_stat_activity
```

```
WHERE pid <> pg_backend_pid()
      AND backend_type = 'client backend'
      pg_terminate_backend
-----
t
t
(2 row)
```

如果您碰巧被踢出，将显示以下消息：

```
test=# SELECT pg_sleep(10000000);
FATAL: terminating connection due to administrator command server closed
the connection unexpectedly
```

这可能意味着服务器在处理请求之前或处理时异常终止。与服务器的连接丢失了。尝试重置：成功。



TIP 只有 psql 会尝试重新连接。对于大多数其他客户来说并非如此 - 尤其不适用于客户端驱动库连接。

检查数据库

检查了活动数据库连接后，可以深入挖掘并检查数据库级统计信息。pg_stat_database 将在 PostgreSQL 实例中的每个数据库返回一行。

这是你在那里可以找到的：

```
test=# \d pg_stat_database
          View "pg_catalog.pg_stat_database"
   Column   |      Type       | Collation | Nullable | Default
-----+-----+-----+-----+
datid    | oid           |           |           |
datname   | name          |           |           |
numbackends | integer       |           |           |
xact_commit | bigint        |           |           |
xact_rollback | bigint       |           |           |
blkss_read | bigint        |           |           |
blkss_hit  | bigint        |           |           |
tup_returned | bigint       |           |           |
tup_fetched  | bigint        |           |           |
tup_inserted | bigint        |           |           |
tup_updated  | bigint        |           |           |
tup_deleted  | bigint        |           |           |
conflicts   | bigint        |           |           |
```

temp_files	bigint			
temp_bytes	bigint			
deadlocks	bigint			
blk_read_time	double precision			
blk_write_time	double precision			
stats_reset	timestamp with time zone			

在数据库 ID 和数据库名称旁边，有一个名为 `numbackends` 的列，显示当前打开的数据连接数。

然后，有 `xact_commit` 和 `xact_rollback`。这两列表明您的应用程序是倾向于提交还是回滚。`blk_hit` 和 `blk_read` 将告诉您有关缓存命中和缓存未命中的信息。在检查这两列时，请记住我们主要讨论共享缓冲区命中和共享缓冲区未命中。在数据库级别上，没有合理的方法来区分文件系统缓存命中和实际磁盘命中。在数据库层面上，没有合理的方法来区分文件系统缓存命中和实际磁盘命中。在 Cybertec (<https://www.cybertec-postgresql.com>)，我们希望在 `pg_stat_database` 中同时查看是否存在磁盘等待和缓存未命中，以了解系统中的实际情况。

`tup`_列将告诉您系统中是否有大量读数或大量写入内容。

然后，我们有 `temp_files` 和 `temp_bytes`。这两列非常重要，因为它们会告诉您数据库是否必须将临时文件写入磁盘，这将不可避免地降低操作速度。临时文件使用高的原因是？主要原因如下：

- 参数设置不佳：如果你的 `work_mem` 设置太低，就无法在 RAM 中做任何事情，因此 PostgreSQL 将转到磁盘。
- 愚蠢的操作：人们常常用相当耗费资源，毫无意义的查询来折磨他们的系统。如果在 OLTP 系统上看到许多临时文件，请考虑检查这些耗费资源的查询。
- 索引和其他管理任务：偶尔，可能会创建索引或者人们可能会运行某些 DDLs。这些操作可能导致临时文件 I/O，但不一定被视为问题（在许多情况下）。

简而言之，系统可能会产生临时文件。即使你的系统完全没问题。但是，关注它们并确保不经常产生临时文件肯定是有意义的。

最后，还有两个更重要的字段：`blk_read_time` 和 `blk_write_time`。默认情况下，这两个字段为空，不会收集任何数据。这些列背后的设计想法是让您了解在 I/O 上花费了多少时间。这些字段为空的原因是默认情况下 `track_io_timing` 处于关闭状态。这是有充分理由的。想象一下，您想要检查读取 100 万个块需要多长时间。要做到这一点，你必须在 C 语言编译库中调用 `time` 函数两次，这导致 200 万个额外的函数调用只是为了读取 8 GB 的数据。这是否会导很多开销真正的取决于你的系统运行速度。

幸运的是，有一个工具可以帮助您确定 timing 的执行成本：

```
[hs@zenbook ~]$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 23.16 nsec
```

Histogram of timing durations:

< usec	%	of total	count
1	97.70300	126549189	
2	2.29506	2972668	
4	0.00024	317	
8	0.00008	101	
16	0.00160	2072	
32	0.00000	5	
64	0.00000	6	
128	0.00000	4	
256	0.00000	0	
512	0.00000	0	
1024	0.00000	4	
2048	0.00000	2	

在我的例子中，为会话或 postgresql.conf 文件打开 track_io_timing 的开销大约是 23 纳秒，这很好。专业的高端服务器可以为您提供低至 14 纳秒的数字，而真正糟糕的虚拟化设备可以返回高达 1,400 纳秒甚至 1,900 纳秒的值。如果您使用的是云服务，可以预期的值大约是 100-120 纳秒（在大多数情况下）。如果您遇到了四位数值，测量 I/O 延时肯定会导致实际可测量的额外开销，这将拖慢您的系统速度。一般规则如下：

在真正的物理硬件设备上，timing 不是问题；在虚拟设备系统上，请在打开它之前检查一下。



TIP 还可以通过使用 ALTER DATABASE, ALTER USER 等选择性地打开事物。

检查表

一旦您了解了数据库中发生的情况，就可以深入挖掘并查看各个表中发生的情况。这里有两个系统视图可以帮助您：pg_stat_user_tables 和 pg_statio_user_tables。这是第一个：

```
test=# \d pg_stat_user_tables
          View "pg_catalog.pg_stat_user_tables"
   Column    |      Type      | Collation | Nullable |
Default
-----+-----+-----+-----+
relid      | oid          |           |           |
schemaname | name         |           |           |
relname    | name         |           |           |
seq_scan   | bigint       |           |           |
seq_tup_read | bigint     |           |           |
```

<code>idx_scan</code>	bigint			
<code>idx_tup_fetch</code>	bigint			
<code>n_tup_ins</code>	bigint			
<code>n_tup_upd</code>	bigint			
<code>n_tup_del</code>	bigint			
<code>n_tup_hot_upd</code>	bigint			
<code>n_live_tup</code>	bigint			
<code>n_dead_tup</code>	bigint			
<code>n_mod_since_analyze</code>	bigint			
<code>last_vacuum</code>	timestamp with time zone			
<code>last_autovacuum</code>	timestamp with time zone			
<code>last_analyze</code>	timestamp with time zone			
<code>last_autoanalyze</code>	timestamp with time zone			
<code>vacuum_count</code>	bigint			
<code>autovacuum_count</code>	bigint			
<code>analyze_count</code>	bigint			
<code>autoanalyze_count</code>	bigint			

根据我的判断，`pg_stat_user_tables` 是一个最重要但也是最容易被误解甚至被忽略的系统视图之一。我有一种感觉，许多人阅读过它，但未能充分挖掘这里可以看到的东西的全部潜力。只要使用得当，在某些情况下，`pg_stat_user_tables` 可以启发一些东西。

在深入研究数据解释之前，了解哪些字段实际存在是非常重要的。首先，每个表都有一个条目，它将向我们显示表中发生的顺序扫描的数量（`seq_scan`）。然后，我们有 `seq_tup_read`，它告诉我们在这些顺序扫描期间系统必须读取多少个元组。



记住 `seq_tup_read` 列；它包含有助于发现性能问题的重要信息。

然后，`idx_scan` 是列表中的下一个。它将向我们显示索引用于此表的频率。PostgreSQL 还会向我们展示这些扫描返回的行数。然后，有几列，从 `n_tup_` 开始。这些将告诉我们插入，更新和删除了多少。这里最重要的是 HOT UPDATE。运行 UPDATE 时，PostgreSQL 必须复制一行以确保 ROLLBACK 正常工作。HOT UPDATE 非常好，因为它允许 PostgreSQL 确保一行不必留下一个块。该行的副本保留在同一块内，一般来说这对于性能是有益的。相当数量的 HOT UPDATE 表示如果这是 UPDATE 可能更新较多的系统，那么这些值是正确被跟踪并显示的。正常和 HOT UPDATE 之间的完美的比例对于大数据用户情况来说是无法被精确衡量的。人们必须自己思考：有多少操作就有多少压力。一般规则是这样的：工作中越多预期的 UPDATE 操作，那么越多的 HOT UPDATE 就越好。

最后，还有一些 VACUUM 统计数据，这些统计数据主要说明一切。

理解 `pg_stat_user_tables`

阅读所有这些数据可能很有趣；但是，除非你能够理解它，否则它是毫无意义的。使

用 `pg_stat_user_tables` 的一种方法是检测哪些表可能需要索引。 获得关于正确方向的线索的一种方法是使用以下查询，这些年来一直很好用：

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
       seq_tup_read / seq_scan AS avg, idx_scan
  FROM pg_stat_user_tables
 WHERE seq_scan > 0
 ORDER BY seq_tup_read DESC
LIMIT 25;
```

我们的想法是找到在顺序扫描中经常使用的大型表。 这些表格自然会出现在列表的顶部，以极高的 `seq_tup_read` 值来提示我们，这是令人兴奋的。



TIP 从上到下工作，寻找耗费昂贵的扫描。 请记住，顺序扫描不一定是坏事。 它们在备份，分析语句等中自然出现而不会造成任何伤害。但是，如果您一直在运行大型顺序扫描，那么您的性能将会下降。

请注意，此查询价值千金 - 它将帮助您查找缺少索引的表。近二十年的实践经验一再表明，索引的确实是性能表现不佳的最重要原因。 因此，您正在查看的查询真的是价值千金。

完成查找可能缺少的索引后，请考虑简要了解表的缓存行为。 为此，`pg_statio_user_tables` 包含有关各种事物的信息，例如表的缓存行为 (`heap_blks_`)，索引的缓存行为 (`idx_blks_`) 以及 The Oversized Attribute Storage Technique (TOAST) 表。 最后，您可以找到有关 TID 扫描的更多信息，这些扫描通常与系统的整体性能无关：

```
test=# \d pg_statio_user_tables
          View "pg_catalog.pg_statio_user_tables"
   Column    | Type   | Collation | Nullable | Default
-----+-----+-----+-----+
  relid   | oid    |           |           |
schemaname | name   |           |           |
  relname  | name   |           |           |
heap_blks_read | bigint |           |           |
heap_blks_hit | bigint |           |           |
idx_blks_read | bigint |           |           |
idx_blks_hit | bigint |           |           |
toast_blks_read | bigint |           |           |
toast_blks_hit | bigint |           |           |
tidx_blks_read | bigint |           |           |
tidx_blks_hit | bigint |           |           |
```

尽管 `pg_statio_user_tables` 包含重要信息，但通常情况下 `pg_stat_user_tables` 更有可能为您提供真正相关的信息（例如缺少索引等）。

深入挖掘索引

虽然 `pg_stat_user_tables` 对于发现丢失的索引很重要，但有时确是需要找到真正不应该存在的索引。最近，我正在德国出差，发现了一个主要包含无意义索引的系统（占总存储消耗的 74%）。虽然如果您的数据库非常小，这可能不是问题，但它确实会对大型系统产生影响 - 拥有数百 GB 的无意义索引会严重损害您的整体性能。

幸运的是，可以检查 `pg_stat_user_indexes` 以找到那些无意义的索引：

```
test=# \d pg_stat_user_indexes
          View "pg_catalog.pg_stat_user_indexes"
      Column      | Type   | Collation | Nullable | Default
-----+-----+-----+-----+
    relid      | oid    |           |           |
indexrelid     | oid    |           |           |
 schemaname    | name   |           |           |
    relname    | name   |           |           |
indexrelname   | name   |           |           |
    idx_scan   | bigint |           |           |
    idx_tup_read | bigint |           |           |
    idx_tup_fetch | bigint |           |           |
```

该视图告诉我们每个模式中每个表上的每个索引的使用频率 (`idx_scan`)。为了浓缩这个视图的精华，我建议使用以下 SQL：

```
SELECT schemaname, relname, indexrelname, idx_scan,
       pg_size.pretty(pg_relation_size(indexrelid)) AS idx_size,
       pg_size.pretty(sum(pg_relation_size(indexrelid)))
             OVER (ORDER BY idx_scan, indexrelid) AS total
  FROM   pg_stat_user_indexes
 ORDER BY 6 ;
```

该语句的输出非常有用。它不仅包含有关索引使用频率的信息 - 它还告诉我们每个索引浪费了多少空间。最后，它将第 6 列中的所有空间消耗相加。您现在可以浏览该表并重新考虑所有那些很少使用的索引。关于何时删除索引很难得出一般规则，因此一些手动检查很有意义。



TIP 不要盲目放弃索引。在某些情况下，根本不必使用索引，因为最终用户使用应用程序与预期的不同。如果最终用户改变（雇用新的秘书等等），索引很可能再次成为有用的对象。

还有一个名为 `pg_statio_user_indexes` 的视图，其中包含有关索引缓存的信息。虽然它很有意思，但它通常不包含会导致索引性能质变的信息。

跟踪后台工作进程

在本节中，是时候看看后台统计信息写进程了。您可能已经知道，在许多情况下，数据库连接不会直接将磁盘写入磁盘。相反，数据由后台写进程或检查点写入。

要查看数据写入的方式，可以检查 pg_stat_bgwriter 视图：

Column	Type	Collation	Nullable
Default			
checkpoints_timed	bigint		
checkpoints_req	bigint		
checkpoint_write_time	double precision		
checkpoint_sync_time	double precision		
buffers_checkpoint	bigint		
buffers_clean	bigint		
maxwritten_clean	bigint		
buffers_backend	bigint		
buffers_backend_fsync	bigint		
buffers_alloc	bigint		
stats_reset	timestamp with time zone		

这里应该引起你注意的第一件事是前两列。您将在本书的后面部分了解到 PostgreSQL 将执行常规检查点，这些检查点是确保数据真正写入磁盘所必需的。如果您的检查点时间彼此距离太近，checkpoint_req 可能会给你指明正确的方向。如有请求的检查点很高，这可能意味着写入了大量数据，并且由于吞吐量高，始终会触发检查点。除此之外，PostgreSQL 还会告诉您在检查点期间写入数据所需的时间以及同步所需的时间。除此之外，buffers_checkpoint 指示在检查点期间写入了多少缓冲区，以及后台写进程器（buffers_clean）写入了多少缓冲区。

但还有更多：maxwritten_clean 告诉我们后台写进程停止清理扫描的次数，因为它写了太多缓冲区。

最后，有 buffers_backend（由后端数据库连接直接写入的缓冲区数），buffers_backend_fsync（由数据库连接刷新的缓冲区数），buffers_alloc（包含分配的缓冲区数）。一般来说，如果数据库连接开始自己写入自己的东西，这不是一件好事。

跟踪，归档和流式传输

在本节中，我们将介绍与复制和事务日志归档相关的一些功能。要检查的第一件事是

`pg_stat_archiver`, 它告诉我们归档过程将事务日志（WAL）从主服务器移动到某个备份设备：

```
test=# \d pg_stat_archiver
          View "pg_catalog.pg_stat_archiver"
      Column |           Type           | Collation | Nullable |
Default
```

Column	Type	Collation	Nullable
Default			
archived_count	bigint		
last_archived_wal	text		
last_archived_time	timestamp with time zone		
failed_count	bigint		
last_failed_wal	text		
last_failed_time	timestamp with time zone		
stats_reset	timestamp with time zone		

此外，`pg_stat_archiver` 包含有关归档过程的重要信息。首先，它将告知您已归档的事务日志文件的数量（`archived_count`）。它还将知道归档的最后一个文件以及何时发生（`last_archived_wal` 和 `last_archived_time`）。

虽然知道 WAL 文件的数量肯定很有趣，但它并不是那么重要。因此，请考虑看看 `failed_count` 和 `last_failed_wal`。如果您的事务日志归档失败，它将告诉您失败的最新文件以及发生的时间。建议密切注意这些字段，否则归档可能会在您没有注意到的情况下停止工作。

如果您正在运行流式复制，以下两个视图对您来说非常重要。第一个称为 `pg_stat_replication`，它将提供有关从主服务器到从服务器的流式处理的信息。每个 WAL 发送者进程一个条目将可见。如果没有单个条目，则没有事务日志流，这可能不是您想要看到的。

我们来看看 `pg_stat_replication`：

```
test=# \d pg_stat_replication
          View "pg_catalog.pg_stat_replication"
      Column |           Type           | Collation | Nullable |
Default
```

Column	Type	Collation	Nullable
Default			
pid	integer		
usesysid	oid		
username	name		
application_name	text		
client_addr	inet		
client_hostname	text		
client_port	integer		

backend_start	timestamp with time zone			
backend_xmin	xid			
state	text			
sent_lsn	pg_lsn			
write_lsn	pg_lsn			
flush_lsn	pg_lsn			
replay_lsn	pg_lsn			
write_lag	interval			
flush_lag	interval			
replay_lag	interval			
sync_priority	integer			
sync_state	text			

您将找到通过流复制连接的用户名的列。然后是应用程序名称以及连接数据(`client_`)。然后，PostgreSQL 告诉我们何时开始的流式复制。在生产环境中，年轻的连接可能指向网络问题或更糟糕的事情(可靠性问题等)。状态列显示流复制的另一侧是哪个状态。请注意，第 10 章“理解备份和复制”将提供更多相关信息。

有些字段告诉我们通过网络连接发送了多少事务日志(`sent_lsn`)，已经发送到数据库内核的数量(`write_lsn`)，刷新到磁盘的数量(`flush_lsn`)，以及已经有多少被重演(`replay_lsn`)。最后，列出复制同步的状态。从 PostgreSQL 10.0 开始，还有其他字段已经包含主服务器和从服务器之间的复制时间差。`*_lag` 字段包含间隔，可以显示服务器之间的实际时间差异。

虽然可以在复制设置的发送服务器(主库)上查询 `pg_stat_replication`，但也可以在接收端(从库)查询 `pg_stat_wal_receiver`。它提供类似的信息，并允许在从库上提取此信息。

以下是视图的定义：

```
test=# \d pg_stat_wal_receiver
          View "pg_catalog.pg_stat_wal_receiver"
 Column |          Type          | Collation | Nullable |
-----+-----+-----+-----+
Default
-----+-----+-----+-----+
pid      | integer          |           |           |
status    | text             |           |           |
receive_start_lsn | pg_lsn          |           |           |
receive_start_tli | integer          |           |           |
received_lsn   | pg_lsn          |           |           |
received_tli   | integer          |           |           |
```

<code>last_msg_send_time</code>	<code>timestamp with time zone</code>		
<code>last_msg_receipt_time</code>	<code>timestamp with time zone</code>		
<code>latest_end_lsn</code>	<code>pg_lsn</code>		
<code>latest_end_time</code>	<code>timestamp with time zone</code>		
<code>slot_name</code>	<code>text</code>		
<code>sender_host</code>	<code>text</code>		
<code>sender_port</code>	<code>integer</code>		
<code>conninfo</code>	<code>text</code>		

首先，PostgreSQL 将告诉我们 WAL 接收进程的进程 ID。然后，该视图向我们显示正在使用的连接的状态。`receive_start_lsn` 将告诉我们 WAL 接收器启动时使用的事务日志位置。除此之外，`receive_start_tli` 还包含 WAL 接收器启动时使用的时间线。在某些时候，您可能想知道最新的 WAL 位置和时间线。要获得这两个数字，请使用 `received_lsn` 和 `received_tli`。

在接下来的两列中，有两个时间戳：`last_msg_send_time` 和 `last_msg_receipt_time`。第一个说明上次发送消息的时间和收到消息的时间。

`latest_end_lsn` 包含在 `latest_end_time` 报告给 WAL 发送方进程的最后一个事务日志位置。然后，有 `slot_name` 和连接信息的模糊版本。在 PostgreSQL 11 中，最后添加了另外的字段 - `sender_host`, `sender_port` 和 `conninfo` 字段告诉我们 WAL 接收进程连接的主机。

检查 SSL 连接

许多运行 PostgreSQL 的人使用 SSL 来加密从服务器到客户端的连接。更新版本的 PostgreSQL 提供了一个视图，以获得这些加密连接的概述，即 `pg_stat_ssl`:

```
test=# \d pg_stat_ssl
          View "pg_catalog.pg_stat_ssl"
   Column  |  Type   | Collation | Nullable | Default
-----+-----+-----+-----+
    pid   | integer |           |           |
     ssl  | boolean |           |           |
  version | text    |           |           |
   cipher | text    |           |           |
    bits  | integer |           |           |
compression | boolean |           |           |
clientdn | text    |           |           |
```

每个进程都由进程 ID 表示。如果连接使用 SSL，则第二列设置为 `true`。第三和第四列

将定义版本和密码。 最后，加密算法使用的位数包括是否使用压缩的指示符，以及客户端证书的 Distinguished Name (DN) 字段。

实时事务检查

到目前为止，已经讨论了一些统计信息表。 所有这些背后的想法是看看整个系统中发生了什么。 但是，如果您是想要检查单个事务的开发人员，该怎么办？ `pg_stat_xact_user_tables` 随时为您提供帮助。 它不包含系统范围的事务；仅有关于您当前交易的数据：

```
test=# \d pg_stat_xact_user_tables
          View "pg_catalog.pg_stat_xact_user_tables"
      Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
    relid | oid  |           |           |
schemaname | name |           |           |
    relname | name |           |           |
    seq_scan | bigint |          |           |
    seq_tup_read | bigint |          |           |
    idx_scan | bigint |          |           |
    idx_tup_fetch | bigint |          |           |
    n_tup_ins | bigint |          |           |
    n_tup_upd | bigint |          |           |
    n_tup_del | bigint |          |           |
    n_tup_hot_upd | bigint |          |           |
```

因此，开发人员可以在事务提交之前查看事务是否已导致任何性能问题。 它有助于区分整体数据和应用程序刚刚提交产生的数据。

应用程序开发人员使用此视图的理想方式是在提交之前在应用程序中添加函数调用以跟踪事务的执行情况。

然后可以检查该数据，以便可以将当前事务的输出与总体工作负载区分开。

跟踪 vacuum 进度

在 PostgreSQL 9.6 中，社区引入了许多人一直在等待的系统视图。 多年来，人们希望跟踪 vacuum 进程的进展，看看它可能还需要多长时间。

因此，发明了 `pg_stat_progress_vacuum` 来解决这个问题：

```
test=# \d pg_stat_progress_vacuum
          View "pg_catalog.pg_stat_progress_vacuum"
      Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
```

pid	integer			
datid	oid			
datname	name			
relid	oid			
phase	text			
heap_blk_total	bigint			
heap_blk_scanned	bigint			
heap_blk_vacuumed	bigint			
index_vacuum_count	bigint			
max_dead_tuples	bigint			
num_dead_tuples	bigint			

大多数列都如字面意思描述的一样，因此我不会在这里详细介绍。应该记住几件事。首先，这个过程不是线性的 - 它是大跨度跳跃性的。除此之外，vacuum 通常非常快，因此进展很快且难以跟踪。

使用 pg_stat_statements

在讨论了前几个观点后，现在是时候将注意力转向最重要的观点之一，可用于发现性能问题。我当然是在谈论 pg_stat_statements。我们的想法是获取有关您的系统查询的信息。它可以帮助我们确定哪些类型的查询速度慢以及调用查询的频率。

要使用该模块，需要遵循三个步骤：

1. 将 pg_stat_statements 添加到 postgresql.conf 文件中的 shared_preload_libraries
2. 重启数据库服务
3. 在您选择的数据库中运行 CREATE EXTENSION pg_stat_statements

让我们检查一下视图的定义：

```
test=# \d pg_stat_statements
```

View "public.pg_stat_statements"					
Column	Type	Collation	Nullable	Default	
userid	oid				
dbid	oid				
queryid	bigint				
query	text				
calls	bigint				
total_time	double precision				
min_time	double precision				
max_time	double precision				
mean_time	double precision				
stddev_time	double precision				
rows	bigint				

<code>shared_blk_hit</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>shared_blk_read</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>shared_blk_dirtied</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>shared_blk_written</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>local_blk_hit</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>local_blk_read</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>local_blk_dirtied</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>local_blk_written</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>temp_blk_read</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>temp_blk_written</code>	<code> bigint</code>	<code> </code>	<code> </code>	<code> </code>
<code>blk_read_time</code>	<code> double precision</code>	<code> </code>	<code> </code>	<code> </code>
<code>blk_write_time</code>	<code> double precision</code>	<code> </code>	<code> </code>	<code> </code>

有趣的是，`pg_stat_statements` 提供了非常棒的信息。对于每个数据库中的每个用户，它为每个查询提供一行。默认情况下，它跟踪 5,000 个记录（可以通过设置 `pg_stat_statements.max` 来更改）。

查询和参数是分开的。PostgreSQL 会将占位符放入查询中。这允许聚合仅使用不同参数的相同查询。`SELECT ... FROM x WHERE y = 10` 将变为 `SELECT ... FROM x WHERE y = ?`。

对于每个查询，PostgreSQL 将告诉我们它消耗的总时间以及调用次数。在更新的版本中，添加了 `min_time`, `max_time`, `mean_time` 和 `stddev`。标准偏差尤其值得注意，因为它会告诉我们查询是否稳定或运行时的波动。不稳定的运行可能是由各种原因导致的：

- 如果数据没有完全缓存在 RAM 中，那么必须转到磁盘的查询将比缓存的对应文件花费更长的时间
- 不同的参数可能导致不同的计划和完全不同的结果集
- 并发和锁定可能会产生影响

PostgreSQL 还会告诉我们查询的缓存行为。`shared_` 列显示来自缓存 (`_hit`) 或操作系统 (`_read`) 的块数量。如果许多块来自操作系统，则查询运行时可能会发生波动。

下一块列是关于本地缓冲区的。本地缓冲区是由数据库连接直接分配的内存块。

除了所有这些信息之外，PostgreSQL 还提供有关临时文件 I/O 的信息。请注意，当构建大型索引或执行某些其他大型 DDL 时，自然会发生临时文件 I/O。然而，临时文件在 OLTP 中通常是一件非常糟糕的事情，因为它会通过阻塞磁盘来降低整个系统的速度。大量的临时文件 I/O 可能指向一些不受欢迎的事情。以下列表包含我总结的前三名：

- 使不得的 `work_mem` 设置 (OLTP)
- 次优的 `maintenance_work_mem` 设置 (DDL)
- 不应该首先运行的查询

最后，有两个字段包含有关 I/O 延迟的信息。默认情况下，这两个字段为空。这样做的原因是测量延时可能会在某些系统上产生相当多的开销。因此，`track_io_timing` 的默认值为 `false` - 请记住在需要此数据时将其打开。

启用此模块后，PostgreSQL 已经在收集数据，您可以使用该视图。



TIP 切勿在客户面前运行 `SELECT * FROM pg_stat_statements`。人们不止一次开始关注查询。他们碰巧知道并开始解释为什么，谁，什么，什么时候等等。使用此视图时，必须按条件排序输出，以便立即看到最相关的信息。

在 Cybertec，我们发现以下查询非常有助于概览数据库服务器上发生的情况：

```
test=# SELECT round((100 * total_time / sum(total_time)
                     OVER ())::numeric, 2) percent,
              round(total_time::numeric, 2) AS total,
              calls,
              round(mean_time::numeric, 2) AS mean,
              substring(query, 1, 40)
        FROM  pg_stat_statements
       ORDER BY total_time DESC
      LIMIT 10;
percent |    total    |   calls |   mean | substring
-----+-----+-----+-----+
 54.47 | 111289.11 | 122161 |  0.91 | UPDATE pgbench_branches SET
          bbalance = b
 43.01 |  87879.25 | 122161 |  0.72 | UPDATE pgbench_tellers SET
          tbalance = tb
 1.46 |   2981.06 | 122161 |  0.02 | UPDATE pgbench_accounts SET
          abalance = a
 0.50 |   1019.83 | 122161 |  0.01 | SELECT abalance FROM
          pgbench_accounts WH
 0.42 |    856.22 | 122161 |  0.01 | INSERT INTO pgbench_history
          (tid, bid, a
 0.04 |     85.63 |      1 |  85.63 | copy pgbench_accounts from
          stdin
 0.02 |    44.11 |      1 |  44.11 | vacuum analyze pgbench_accounts
 0.02 |    42.86 | 122161 |  0.00 | END;
 0.02 |    34.08 | 122171 |  0.00 | BEGIN;
 0.01 |    22.46 |      1 |  22.46 | alter table pgbench_accounts
          add primary
(10 rows)
```

它显示前 10 个查询及其运行时间，包括百分比。显示查询的平均执行时间也是有意义的，这样您就可以判断这些查询运行时间是否过高。

沿着列表以你的方式工作，检查平均运行时间过长的所有查询。

请记住，通过查询前 1000 个值通常是不必要的。在大多数情况下，上面第一个查询已经负责系统上的大部分负载。



TIP 在我的示例中，我使用 `substring` 函数来截取并缩短查询语句以适应页面。如果你真的想看看发生了什么，这没有任何意义。

请记住，默认情况下，`pg_stat_statements` 将以 1,024 字节截断显示查询信息：

```
test=# SHOW track_activity_query_size;  
track_activity_query_size  
-----  
1024  
(1 row)
```

考虑将此值增加到 16,384。如果您的客户端正在运行基于 Hibernate 的 Java 应用程序，则较大的 `track_activity_query_size` 值将确保在显示查询有趣的部分之前不会被截断。

在这一点上，我想用这种情况指出 `pg_stat_statements` 是有多重要。它是迄今为止追踪性能问题的最简单方法。慢查询日志永远不会像 `pg_stat_statements` 一样有用，因为慢查询日志只会指向单个慢查询 - 它不会向我们显示由大量中等查询引起的问题。因此，建议始终打开此模块。而且它的开销非常小，绝不会损害系统的整体性能。

默认情况下，会跟踪 5,000 种类型的查询。在合适的应用程序中，这就足够了。

要重置数据，请考虑使用以下指令：

```
test=# SELECT pg_stat_statements_reset();  
pg_stat_statements_reset  
-----  
(1 row)
```

创建日志文件

在深入了解 PostgreSQL 提供的系统视图后，是时候配置日志记录了。幸运的是，PostgreSQL 提供了一种简单的方法来处理日志文件，并帮助人们轻松设置好相关配置。

收集日志很重要，因为它可以指出错误和潜在的数据库问题。

`postgresql.conf` 文件包含为您提供所有必要信息所需的所有参数。

配置 `postgresql.conf` 文件

在本节中，我们将通过 `postgreql.conf` 文件中的一些最重要的条目来配置日志记录，并查看如何以最有利的方式使用日志记录。

在我们开始之前，我想说几句关于登录 PostgreSQL 的话。在 Unix 系统上，PostgreSQL 默认会将日志信息发送到 `stderr`。但是，`stderr` 并不适合日志存放，因为您肯定希望在

某些时候检查日志输出。因此，完成本章并根据您的需求调整内容确实很有意义。

定义日志存储目录和日志循环

让我们浏览一下 `postgresql.conf` 文件，看看可以做些什么：

```
#-----
# ERROR REPORTING AND LOGGING
#-----
# - Where to Log -
#log_destination = 'stderr'
    # Valid values are combinations of
    # stderr, csvlog, syslog, and eventlog,
    # depending on platform. csvlog
    # requires logging_collector to be on.

# This is used when logging to stderr:
#logging_collector = off
    # Enable capturing of stderr and csvlog
    # into log files. Required to be on for
    # csvlogs.
    # (change requires restart)
```

第一个配置选项定义了日志的处理方式。默认情况下，它将转到 `stderr`（在 Unix 上）。在 Windows 上，默认为 `eventlog`，它是用于处理日志记录的 Windows 自带工具。或者，您可以选择使用 `csvlog` 或 `syslog`。

如果你想生成 PostgreSQL 日志文件，你应该设置 `stderr` 并打开日志记录收集器。然后 PostgreSQL 将创建日志文件。

现在的逻辑问题是：这些日志文件的名称是什么以及这些文件将存储在何处？幸运的是，`postgresql.conf` 有答案：

```
# These are only used if logging_collector is on:
#log_directory = 'pg_log'
    # directory where log files are written,
    # can be absolute or relative to PGDATA
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
    # log file name pattern,
    # can include strftime() escapes
```

此外，`log_directory` 将告诉系统存储日志的位置。如果使用绝对路径，则可以显式得配置日志的位置。如果您希望日志直接位于 PostgreSQL 数据中，只需选择相对路径即可。优点是数据目录将是自包含的，您可以移动它而不必担心。

在下一步中，您可以定义 PostgreSQL 应该使用的文件名。PostgreSQL 非常灵活，允许

您使用 `strftime` 提供的所有快捷方式。为了让您了解此功能的强大功能，我的平台快速统计显示 `strftime` 提供 43 (!) 个占位符来创建文件名。人们通常需要的一切当然是可能的。

一旦定义了文件名，就可以简单地考虑一下如何清理。 将提供以下设置：

```
#log_truncate_on_rotation = off  
#log_rotation_age = 1d  
#log_rotation_size = 10MB
```

默认情况下，如果文件超过 1 天或大于 10 MB，PostgreSQL 将继续生成日志文件。 此外，`log_truncate_on_rotation` 指定是否要追加到已有日志文件。 有时，`log_filenames` 以其变为循环的方式定义。`log_truncate_on_rotation` 参数定义是覆盖还是追加到已存在的文件。 鉴于默认设置的日志文件，这当然不会发生。

处理自动循环的一种方法是使用类似 `postgresql_%a.log` 的东西，以及 `log_truncate_on_rotation = on`。`%a` 表示将在日志文件中使用星期几。 这里的优点是，星期几往往会重演每 7 天一次。 因此，日志将保留一周并回收。 如果您的目标是每周轮换，则 10 MB 的文件大小可能还不够。 可以考虑关闭最大文件大小。

配置 syslog

有些人更喜欢使用 `syslog` 来收集日志文件。 PostgreSQL 提供以下配置参数：

```
# These are relevant when logging to syslog:  
#syslog_facility = 'LOCAL0'  
#syslog_ident = 'postgres'  
#syslog_sequence_numbers = on  
#syslog_split_messages = on
```

一个 `syslog` 在 `sys` 管理员中非常流行。 幸运的是，它很容易配置。

基本上，您设置一下设施和标识符。 如果 `log_destination` 设置为 `syslog`，那么这就是它要做的所有事情。

记录慢查询

该日志还可用于跟踪单个慢速查询。 回到过去，这几乎是发现性能问题的唯一方法。

它是如何工作的？ 基本上，`postgresql.conf` 有一个名为 `log_min_duration_statement` 的变量。 如果将此值设置为大于零的值，则超出所设置时间的每个查询都将记录在日志中：

```
# log_min_duration_statement = -1
```

大多数人认为慢查询日志是充满智慧的设计。 但是，我想补充一点。 有很多慢查询，它们碰巧使用了很多 CPU：索引创建，数据导出，分析等等。

那些长期运行的查询完全是预期的，并且在许多情况下不是所有邪恶的根源。经常发生的并且较多较短的查询都应该受到指责。这是一个例子：

1,000 个查询 x 500 毫秒比 2 个查询 x 5 秒差。在某些情况下，慢查询日志可能会产生误导。

尽管如此，这并不意味着它毫无意义 - 它只是意味着它是获取慢查询信息的来源而不是慢查询产生的根源。

定义记录的内容和方式

在看了一些基本设置之后，是时候决定记录什么了。默认情况下，仅记录错误。但是，这可能还不够。在本节中，您将了解可以记录的内容以及日志记录行的内容。

默认情况下，PostgreSQL 不会记录有关检查点的信息。以下设置可以改变：

```
#log_checkpoints = off
```

这同样适用于连接；无论何时建立连接或正确销毁连接，PostgreSQL 都可以创建日志条目记录：

```
#log_connections = off  
#log_disconnections = off
```

在大多数情况下，记录连接没有意义，因为大量日志记录会显着降低系统速度。分析系统不会受到太大影响。但是，OLTP 可能会受到严重影响。

如果要查看语句需要执行多长时间，请考虑将以下设置切换为打开：

```
#log_duration = off
```

让我们继续讨论一个最重要的设置。我们还没有定义消息的布局，到目前为止，日志文件包含以下形式的错误：

```
test=# SELECT 1 / 0;  
ERROR: division by zero
```

日志将显示 ERROR 以及错误消息。在 PostgreSQL 10.0 之前，没有时间戳，用户名等。您必须立即更改该值以了解日志。在 PostgreSQL 10.0 中，默认值已更改为更合理的内容：要更改它，请查看 log_line_prefix：

```
#log_line_prefix = '%m [%p]'  
# special values:  
#   %a = application name  
#   %u = user name  
#   %d = database name  
#   %r = remote host and port
```

```
# %h = remote host
# %p = process ID
# %t = timestamp without milliseconds
# %m = timestamp with milliseconds
# %n = timestamp with milliseconds (as a Unix epoch)
# %i = command tag
# %e = SQL state
# %c = session ID
# %l = session line number
# %s = session start timestamp
# %v = virtual transaction ID
# %x = transaction ID (0 if none)
# %q = stop here in non-session processes
# %% = '%'
```

此外，`log_line_prefix` 非常灵活，允许您配置日志行以完全满足您的需求。通常，记录时间戳是个好主意。否则，当发生坏事时几乎不可能看到所需的信息。就个人而言，我也想知道用户名，交易 ID 和数据库。但是，您需要决定自己真正需要的是什么。

有时，缓慢的查询是由不良的锁引起的。相互阻塞的用户可能会导致性能下降，因此必须解决这些问题以确保高数据库的吞吐量。通常，与锁相关的问题很难追查。

基本上，`log_lock_waits` 可以帮助检测此类问题。如果锁的持有时间超过 `deadlock_timeout`，则只要打开以下配置变量，就会向日志发送一行：

```
#log_lock_waits = off
```

最后，是时候告诉 PostgreSQL 实际记录的日志内容了。到目前为止，只有错误，慢查询等已发送到日志。但是，`log_statement` 有四种可能的设置：

```
#log_statement = 'none'
# none, ddl, mod, all
```

请注意，`none` 表示仅记录错误。`ddl` 表示将记录错误以及 DDL（`CREATE TABLE`, `ALTER TABLE` 等）。`mod` 将包含数据更改，并且所有语句都将会被发送到日志并记录。

 **TIP** 请注意，所有这些都可能导致大量日志信息被记录，这可能使你的系统变慢。

为了给你展示可以产生多大的影响，我编写了一篇博文。它可以在 <https://www.cybertec-postgresql.com/en/logging-the-hidden-speedbrakes/> 找到。

如果要更详细地检查复制，请考虑打开以下设置：

```
#log_replication_commands = off
```

它会将与复制相关的命令发送到日志。有关更多信息，请访问以下网站：<https://www.postgresql.org/docs/current/static/protocol-replication.html>。

由临时文件 I/O 导致可能经常发生性能问题。要查看哪些查询导致问题，可以使用以下

设置：

```
#log_temp_files = -1  
# log temporary files equal or larger  
# than the specified size in kilobytes;  
# -1 disables, 0 logs all temp files
```

虽然 `pg_stat_statements` 包含聚合信息，但 `log_temp_files` 将指向导致问题的特定查询。将此值设置为相当低的值通常是有意义的。正确的值取决于您的数据库工作负载，但可能 4 MB 已经就是一个合适的值。

默认情况下，PostgreSQL 将服务器所在的时区写入日志文件。但是，如果您运行的是遍布全球的系统，则可以通过以下方式调整时区，并比较日志条目：

```
log_timezone = 'Europe/Vienna'
```

请记住，在 SQL 端，您仍将看到当地时区的时间。但是，如果设置了此变量，则日志条目将记录不同的时区。

小结

本章内容全部是关于系统统计信息的。您学习了如何从 PostgreSQL 中提取信息以及如何以有益的方式使用系统统计信息。并且详细讨论了重要的系统视图。

第 6 章，“优化查询以获得良好的性能”，完全是关于查询性能优化的。您将学习如何检查查询以及如何优化它们。

QA

与大多数章节一样，我们将查看刚刚涵盖的一些关键问题：

PostgreSQL 收集什么样的运行时统计信息？

PostgreSQL 中的统计信息收集器收集了大量信息。有关内容的完整概述可以在官方 PostgreSQL 文档中找到，该文档可以从 <https://www.postgresql.org/docs/11/static/monitoring-stats.html> 在线获得。

我怎样才能轻松发现性能问题？

有各种方法可以检测 PostgreSQL 中的性能问题。一种方法是使用 `pg_stat_statements`。其他选项是使用 `auto_explain` 或简单的标准 PostgreSQL 日志文件。根据您的需要，您可以决定哪种方法最适合您。以下博客向您展示了如何从系统中提取此类信息：<https://www.cybertec-postgresql.com/en/3-ways-to-detect-slow-queries-in-postgresql/>。

PostgreSQL 如何写日志文件？

PostgreSQL 有各种创建日志的选项。最常用的是简单地创建标准文本日志。但是，您也可以使用 `syslog` 或 Windows 提供的日志记录工具（`eventlog`）。

日志记录会对性能产生影响吗？

其实，是的。 如果您正在运行许多小型查询，并且您碰巧将数百万行日志写入系统，则会影响性能。

Javanchueng

6. 优化查询以获得良好性能

在第 5 章“日志文件和系统统计信息”中，您学习了如何阅读系统统计信息以及如何使用 PostgreSQL 提供的信息。有了这些知识，本章就是关于如何优化查询性能的。您将了解有关以下主题的更多信息：

- 优化器内部
- 执行计划
- 分区数据
- 启用和禁用优化设置
- 良好查询性能的参数
- 并行查询
- JIT 编译

到本章结束时，我们将能够编写更好，更快的查询。如果查询性能仍然不是很好，我们应该能够理解为什么会这样。我们还将能够使用我们学到的新技术来进行数据分区。

了解优化器的功能

在尝试考虑查询性能之前，熟悉查询优化器的功能是有意义的。深入了解幕后发生的事情很有意义，因为它可以帮助您了解数据库的真正含义以及它正在做什么。

通过示例进行优化

为了演示优化器的工作原理，我编写了一个示例。这是我多年来用于 PostgreSQL 培训的东西。我们假设有三个表，如下所示：

```
CREATE TABLE a (aid int, ...);          -- 100 million rows
CREATE TABLE b (bid int, ...);          -- 200 million rows
CREATE TABLE c (cid int, ...);          -- 300 million rows
```

让我们进一步假设这些表包含数百万或数亿个行。除此之外，还有索引：

```
CREATE INDEX idx_a ON a (aid);
CREATE INDEX idx_b ON b (bid);
CREATE INDEX idx_c ON c (cid);
CREATE VIEW v AS SELECT *
    FROM a, b
    WHERE aid = bid;
```

最后，有一个视图将前两个表连接在一起。

我们现在假设最终用户想要运行以下查询。优化器将对此查询执行什么操作？
planner 有哪些选择？

```
SELECT *
```

```
FROM v, c  
WHERE v.aid = c.cid  
AND cid = 4;
```

在查看真正的优化过程之前，我们将重点关注 `planner` 所具有的一些选项。

评估 JOIN 选项

`planner` 在这里有几个选项，所以让我们借此机会了解如果使用细小的方法会出现什么问题。

假设 `planner` 只是全力向前并且计算视图的结果输出。 联合查询 1 亿行和 2 亿行的最佳方法是什么？

在本节中，将讨论几个（并非所有）连接选项，以向您展示 PostgreSQL 能够执行的操作。

嵌套循环（Nested loops）

连接两个表的一种方法是使用嵌套循环。 原理很简单。 这是一些伪代码：

```
for x in table1:  
    for y in table2:  
        if x.field == y.field  
            issue row  
        else  
            keep doing
```

如果其中一边对象非常小并且仅包含有限的数据集，则经常使用嵌套循环。 在我们的示例中，嵌套循环将导致代码中的 1 亿次乘以 2 亿次的迭代循环。 这显然不是一种好的选择，因为运行时会撑爆。

嵌套循环复杂度通常为 $O(n^2)$ ，因此仅在连接的一侧非常小时才有效。 在此示例中，情况并非如此，因此可以排除嵌套循环以计算视图结果。

哈希连接（Hash Join）

第二个选项是哈希连接。 可以应用以下策略来解决我们的小问题。 以下清单将显示哈希连接的工作原理：

```
Hash join  
Sequential scan table 1  
Sequential scan table 2
```

可以对双方进行哈希处理，并且可以比较哈希键值，从而为我们提供连接的结果。这里的问题是所有的值都必须进行哈希并存储在某处。

合并连接（Merge Join）

最后，有一个合并连接。想法是使用排序列表来连接结果。如果连接的两侧都已排序，系统可以从顶部获取行，看看它们是否匹配并返回它们。这里的主要要求是列表已排序。这是一个示例计划：

```
Merge join
  Sort table 1
    Sequential scan table 1
  Sort table 2
    Sequential scan table 2
```

要连接这两个表（表 1 和表 2），必须按排序顺序提供数据。在许多情况下，PostgreSQL 只会对数据进行排序。但是，还有其他选项可以提供已排序数据的连接。一种方法是查询索引，如以下示例所示：

```
Merge join
  Index scan table 1
  Index scan table 2
```

连接的一侧或双方可以使用来自计划的较低级别的排序数据。如果直接访问该表，则索引是显而易见的选择，但前提是返回的结果集明显小于整个表。否则，我们会遇到几乎两倍的开销，因为首先我们必须读取整个索引，然后是整个表。如果结果集是表的大部分，则顺序扫描更有效，尤其是在以主键顺序访问的时候。

合并连接的优点在于它可以处理大量数据。缺点是必须在某个时刻对数据进行排序或从索引中获取数据。

排序复杂度为 $O(n * \log(n))$ 。因此，排序 3 亿行来执行连接也不具吸引力。



注意，自从引入 PostgreSQL 10.0 以来，所有的连接选项也支持并行。因此，优化器不仅会考虑哪些标准的连接选项，还会评估是否有必要进行并行查询。

应用转换

显然，做一件显而易见的事情（首先执行视图连接查询）毫无意义。嵌套循环将从顶层循环耗费执行时间。哈希连接必须哈希数百万行，嵌套循环（译者注：应该是合并连接）必须排序 3 亿行。这三个选项显然都不合适。出路是应用逻辑转换来执行快速查询。在本节中，您将了解 planner 如何加速查询。将执行几个步骤。

内联视图

优化程序完成的第一个转换是内联视图。如下是执行的内容：

```
SELECT *
FROM
(
  SELECT *
    FROM a, b
   WHERE aid = bid
) AS v, c
 WHERE v.aid = c.cid
   AND cid = 4;
```

视图被内联并转换为子查询。这个给我们带来了什么？其实，什么都没有。它所做的就是为进一步优化打开了大门，对于这个查询来说真的会改变查询规则。

平展子查询

接下来是平展子查询，这意味着将它们集成到主查询中。通过删除子查询，将出现更多选项来优化查询。

以下是查询的内容，例如在平展子查询之后：

```
SELECT * FROM a, b, c WHERE a.aid = c.cid AND aid = bid AND cid = 4;
```

它现在是一个普通的连接查询。



我们可以自己重写这个 SQL，但无论如何，planner 将为我们处理这些转变。
优化器现在可以执行进一步的优化。

应用等式约束

以下过程创建了等式约束。我们的想法是检测其他约束，连接选项和过滤器。让我们深呼吸，看看以下问题：如果 `aid = cid` 和 `aid = bid`，我们可以得出 `bid = cid`。如果 `cid = 4` 而所有其他人都相同，我们知道 `aid` 和 `bid` 也必须是 4，这导致我们进行以下查询：

```
SELECT *
FROM a, b, c
WHERE a.aid = c.cid
  AND aid = bid
  AND cid = 4
  AND bid = cid
  AND aid = 4
  AND bid = 4
```

不必特别强调这种优化的重要性。 planner 在这里做了什么，为原始查询中无法清晰显示的两个额外索引打开了大门。

通过能够在所有三列上使用索引，查询现在成本低得多。优化器可以选择从索引中检索几行并使用任何有意义的连接选项。

详尽的检索

现在已经完成了那些正式的转换，PostgreSQL 将执行详尽的检索。它会尝试所有可能的计划，并为您的查询提供成本最低的解决方案。PostgreSQL 知道哪些索引是可用的，只不过使用成本模型来确定如何以最佳方式执行操作。

在详尽的检索过程中，PostgreSQL 还将尝试确定最佳的连接顺序。在原始查询中，连接顺序固定为 A→B 和 A→C。但是，使用这些相等约束，我们可以加入 B→C 并稍后加入 A。所有选项均对 planner 透明。

尽管试一试

现在已经讨论了所有这些优化，现在是时候看看 PostgreSQL 生成了什么样的执行计划：

```
test=# explain SELECT * FROM v, c WHERE v.aid = c.cid AND cid = 4;
          QUERY PLAN
```

```
-----  
Nested Loop (cost=1.71..17.78 rows=1 width=12)  
  -> Nested Loop (cost=1.14..9.18 rows=1 width=8)  
    -> Index Only Scan using idx_a on a  
        (cost=0.57..4.58 rows=1 width=4)  
        Index Cond: (aid = 4)  
    -> Index Only Scan using idx_b on b  
        (cost=0.57..4.59 rows=1 width=4)  
        Index Cond: (bid = 4)  
  -> Index Only Scan using idx_c on c  
      (cost=0.57..8.59 rows=1 width=4)  
      Index Cond: (cid = 4)
```

(8 rows)



请注意，本章中显示的计划不一定是 100% 与您运行时将观察到的相同。根据您加载的数据量，可能会有轻微的变化。成本还可能取决于磁盘上数据的物理对齐（磁盘上的顺序）。运行这些示例时请记住这一点。

如您所见，PostgreSQL 将使用三个索引。有趣的是，PostgreSQL 决定采用嵌套循环来连接数据。这非常有意义，因为索引扫描几乎没有数据返回。因此，使用嵌套循环来连接数据是完全可行和高效的。

使优化过程失败

到目前为止，您已经了解了 PostgreSQL 可以为您做什么以及优化器如何帮助加快查询速度。 PostgreSQL 很聪明，但它更需要聪明的用户。 在某些情况下，最终用户通过做愚蠢的事情来削弱整个优化过程。 让我们使用以下命令删除视图：

```
test=# DROP VIEW v;  
DROP VIEW
```

现在，视图已重新创建。 请注意，OFFSET 0 已添加到视图的末尾。 我们来看看下面的例子：

```
test=# CREATE VIEW v AS SELECT *  
      FROM a, b  
     WHERE aid = bid  
   OFFSET 0;  
CREATE VIEW
```

虽然此视图在逻辑上等效于前面显示的示例，但优化器必须以不同方式处理。 除 0 以外的每个 OFFSET 都将更改结果，因此必须重新计算视图。 通过添加诸如 OFFSET 之类的东西来削弱整个优化过程。



TIP PostgreSQL 社区不敢优化视图中具有 OFFSET 0 的情况。人们也不应该这样做。

我们将仅以此为例来观察某些操作如何削弱性能，以及我们作为开发人员应该了解底层优化过程。 但是，如果你碰巧知道 PostgreSQL 是如何工作的，那么这个技巧可以用作优化。

这是新的执行计划：

```
test=# EXPLAIN SELECT * FROM v, c WHERE v.aid = c.cid AND cid = 4;  
QUERY PLAN  
-----  
Nested Loop (cost=120.71..7949879.40 rows=1 width=12)  
  -> Subquery Scan on v  
        (cost=120.13..7949874.80 rows=1 width=8)  
          Filter: (v.aid = 4)  
  -> Merge Join (cost=120.13..6699874.80 rows=100000000 width=8)  
    Merge Cond: (a.aid = b.bid)  
    -> Index Only Scan using idx_a on a  
        (cost=0.57..2596776.57 rows=100000000 width=4)  
    -> Index Only Scan using idx_b on b  
        (cost=0.57..5193532.33 rows=199999984 width=4)  
  -> Index Only Scan using idx_c on c  
        (cost=0.57..4.59 rows=1 width=4)  
        Index Cond: (cid = 4)  
(9 rows)
```

只需看看 `planner` 预测的成本。 成本从两位数急剧上升到惊人的数字。 显然，此查询将为您提供糟糕的性能。

有各种方法可以削弱性能，但是记住优化过程是有意义的。

常量折叠

但是，PostgreSQL 中还有许多优化方式在幕后进行，并有助于提高整体性能。 其中一个特征称为常量折叠。 想法是将表达式转换为常量，如以下示例所示：

```
test=# explain SELECT * FROM a WHERE aid = 3 + 1;
```

QUERY PLAN

```
-----  
Index Only Scan using idx_a on a  
(cost=0.57..4.58 rows=1 width=4)  
Index Cond: (aid = 4)  
(2 rows)
```

正如您所看到的，PostgreSQL 将尝试寻找 4。因为 `aid` 列存在索引，PostgreSQL 将进行索引扫描。请注意，我们的表只有一列，因此 PostgreSQL 甚至发现它所需的所有数据都可以在索引中找到（覆盖索引扫描）。

如果表达式位于查询条件左侧，会发生什么？

```
test=# explain SELECT * FROM a WHERE aid - 1 = 3;
```

QUERY PLAN

```
-----  
Seq Scan on a (cost=0.00..1942478.48 rows=500000 width=4)  
Filter: ((aid - 1) = 3)  
(2 rows)
```

在这种情况下，索引扫描将失败，PostgreSQL 必须进行顺序扫描。请记住，这是一个单核扫描计划。如果表的大小很大或者 PostgreSQL 配置不同，您可能会看到多核扫描的计划。为简单起见，本章仅包含单核的扫描计划，这样阅读起来更容易。

了解函数内联

如本节中所述，有许多优化可以帮助加快查询速度。其中一个称为函数内联。PostgreSQL 能够内联不可变的 SQL 函数。主要思想是减少必须进行的函数调用的数量，以加快速度。

以下是优化器可以内联的函数示例：

```
test=# CREATE OR REPLACE FUNCTION Id(int)  
RETURNS numeric AS
```

```
$$
    SELECT log(2, $1);
$$
LANGUAGE 'sql' IMMUTABLE;
CREATE FUNCTION
```

该函数将计算输入值的 \log 值:

test=# SELECT Id(1024);

Id

10.000000000000000
(1 row)

为了演示工作原理，我们将使用较少的内容重新创建表以加快索引创建：

```
test=# TRUNCATE a;  
TRUNCATE TABLE
```

然后，可以再次添加数据并创建索引：

```
test=# INSERT INTO a SELECT * FROM generate_series(1, 10000);
INSERT 0 10000
test=# CREATE INDEX idx_Id ON a (Id(aid));
CREATE INDEX
```

正如所料，在函数上创建的索引将像任何其他索引一样使用。

但是，让我们仔细看看索引条件：

test=# EXPLAIN SELECT * FROM a WHERE Id(aid) = 10;

QUERY PLAN

Index Scan using idx_Id on a (cost=0.29..8.30 rows=1 width=4)
Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(2 rows)

这里值得关注的情况是索引查找实际上是查找 `log` 函数而不是 `Id` 函数。优化器去掉了函数调用的方式而是直接使用了函数内部的 `log` 函数。

从逻辑上讲，这为以下查询打开了大门：

test=# EXPLAIN SELECT * FROM a WHERE log(2, aid) = 10;

QUERY PLAN

Index Scan using idx_id on a (cost=0.29..8.30 rows=1 width=4)
Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric
(2 rows)

连接裁剪

PostgreSQL 还提供了一个名为连接裁剪的优化。如果查询不需要连接，则可以删除连接。如果查询是由某些中间件或某些 ORM 生成的，则可以派上用场。如果可以删除连接，它自然会加快速度显著地减少了开销。

现在的问题是，连接裁剪是如何工作的？这里是一个例子：

```
CREATE TABLE x (id int, PRIMARY KEY (id));
CREATE TABLE y (id int, PRIMARY KEY (id));
```

首先，创建两个表。确保连接条件的两侧实际上是唯一的。这些限制很重要。

现在，我们可以编写一个简单的查询：

```
test=# EXPLAIN SELECT *
  FROM  x LEFT JOIN y ON (x.id = y.id)
 WHERE x.id = 3;
                                         QUERY PLAN
-----
Nested Loop Left Join  (cost=0.31..16.36 rows=1 width=8)
  Join Filter: (x.id = y.id)
    -> Index Only Scan using x_pkey on x
        (cost=0.15..8.17 rows=1 width=4)
          Index Cond: (id = 3)
    -> Index Only Scan using y_pkey on y
        (cost=0.15..8.17 rows=1 width=4)
          Index Cond: (id = 3)
(6 rows)
```

如您所见，PostgreSQL 将直接连接查询这些表。到目前为止，没有任何意外。

但是，稍微修改了以下查询。它不是选择所有列，而是仅选择连接左侧的那些列：

```
test=# EXPLAIN SELECT x.*
  FROM x LEFT JOIN y ON (x.id = y.id)
 WHERE x.id = 3;
                                         QUERY PLAN
-----
```

```
Index Only Scan using x_pkey on x  (cost=0.15..8.17 rows=1 width=4)
  Index Cond: (id = 3)
(2 rows)
```

PostgreSQL 将进行直接内部扫描并完全跳过表连接。有两个原因可以解释这个操作实际上的可行性和逻辑上的正确性：

- 没有从连接的右侧选择列；因此，查看这些列并不会带给我们任何东西
- 右侧数值是唯一的，这意味着由于即使右侧数据的重复，连接查询也不能增加

查询的行数

如果可以自动裁剪联接，则可能会发生查询速度更快的情况。这里的美妙之处就在于，只需删除应用程序可能不需要查询的列，即可实现速度的提高。

加速集合操作

集合操作允许将多个查询的结果组合到单个结果集中。

集合运算符包括 UNION, INTERSECT 和 EXCEPT。PostgreSQL 实现了所有这些功能，并提供了许多重要的优化来加速它们。

planner 能够将限制推进到集合操作中，总体上为不可思议的索引创建和加速打开了大门。让我们看看下面的查询，它向我们展示了它的工作原理：

```
test=# EXPLAIN SELECT *
  FROM
  (
    SELECT aid AS xid
      FROM a
     UNION ALL
    SELECT bid FROM b
  ) AS y
 WHERE xid = 3;
```

QUERY PLAN

```
Append (cost=0.29..12.89 rows=2 width=4)
  -> Index Only Scan using idx_a on a
      (cost=0.29..8.30 rows=1 width=4)
        Index Cond: (aid = 3)
  -> Index Only Scan using idx_b on b
      (cost=0.57..4.59 rows=1 width=4)
        Index Cond: (bid = 3)
(5 rows)
```

你在这里看到的是两个对象集相互交融。唯一的麻烦是限制条件在子查询之外。但是，PostgreSQL 指出过滤器可以在计划中下推。因此，`xid = 3` 条件被自动下推到查询列 `aid` 和 `bid`，激活了在查询的两个表上使用索引的选项。通过避免对两个表进行顺序扫描，查询运行速度会快得多。

请注意，UNION 子句和 UNION ALL 子句之间存在重复值的区别。UNION ALL 子句将盲目地追加数据并传递两个表的结果。

UNION 子句不同，因为它将过滤掉重复项。以下计划显示了它的工作原理：

```
test=# EXPLAIN SELECT *
```

```
FROM
(
    SELECT aid AS xid
    FROM a
    UNION SELECT bid
    FROM b
) AS y
WHERE xid = 3;
```

QUERY PLAN

```
-----
Unique (cost=12.92..12.93 rows=2 width=4)
-> Sort (cost=12.92..12.93 rows=2 width=4)
    Sort Key: a.aid
    -> Append (cost=0.29..12.91 rows=2 width=4)
        -> Index Only Scan using idx_a on a
            (cost=0.29..8.30 rows=1 width=4)
            Index Cond: (aid = 3)
        -> Index Only Scan using idx_b on b
            (cost=0.57..4.59 rows=1 width=4)
            Index Cond: (bid = 3)
(8 rows)
```

PostgreSQL 必须在 Append 操作之上添加一个 Sort 操作，以确保稍后可以过滤重复项。



TIP 许多开发人员并不完全了解 UNION 子句和 UNION ALL 子句之间的区别。他们抱怨性能不佳，因为他们不知道 PostgreSQL 必须过滤掉重复数据，这对于大型数据集来说尤其痛苦。

了解执行计划

现在我们已经深入研究了 PostgreSQL 中实现的一些重要优化，让我们继续深入了解执行计划。你已经在本书中看到了一些计划。但是，为了充分利用计划，必须建立一种系统的方法来阅读这些信息。系统地阅读计划完全在本节的范围内。

系统地接近计划

您必须要知道的第一件事是 EXPLAIN 子句可以为您做很多事情，我强烈建议您充分利用这些功能。

正如许多读者可能已经知道的那样，EXPLAIN ANALYZE 子句将执行查询并返回计划，包括真实的运行时信息。这是一个例子：

```
test=# EXPLAIN ANALYZE SELECT *
FROM
```

```

(
    SELECT *
    FROM b
    LIMIT 1000000
) AS b
ORDER BY cos(bid);

QUERY PLAN
-----
Sort (cost=146173.12..148673.12 rows=1000000)
(actual time=837.049..1031.587 rows=1000000)
  Sort Key: (cos((b.bid)::double precision))
  Sort Method: external merge Disk: 25408kB
-> Subquery Scan on b
  (cost=0.00..29424.78 rows=1000000 width=12)
  (actual time=0.011..352.717 rows=1000000)
    -> Limit (cost=0.00..14424.78 rows=1000000)
      (actual time=0.008..169.784 rows=1000000)
        -> Seq Scan on b b_1 (cost=0.00..2884955.84 rows=199999984
width=4)
          (actual time=0.008..85.710 rows=1000000)

Planning time: 0.064 ms
Execution time: 1159.919 ms
(8 rows)

```

该计划看起来有点可怕，但不要惊慌；我们将一步一步地完成它。 阅读计划时，请确保从内到外阅读。 在我们的示例中，执行从 `b` 上的顺序扫描开始。 这里实际上有两个信息块：成本块和实际时间块。 虽然成本块包含估算值，但实际时间段是确凿的证据。 它显示了实际执行时间。 在此示例中，顺序扫描花费了 85.7 毫秒。



请注意，在你的数据库系统上显示的成本可能不相同。 统计信息中的任何一个小小差异可能会导致成本统计的差异。 重要的是这个计划被阅读的方式。

然后，来自索引扫描的数据将传递到 `Limit`，从而确保没有多余的数据。 请注意，每个执行阶段也会向我们显示所涉及的行数。 正如您所看到的，PostgreSQL 首先只会从表中获取 100 万行；`Limit` 保证了数据记录的准确性。 但是，在这个阶段有一个成本耗费的标签，运行时间达到了 169 毫秒。 最后，数据被排序，这需要花费很多时间。 查看执行计划时最重要的是找出时间实际耗费的地方。 最好的方法是查看实际执行的时间部分，并尝试找出时间跳跃的位置。 在这个例子中，顺序扫描需要一些时间，但不能很容易的被加速。 相反，我们可以看到排序的地方时间耗费猛增。

当然，可以加快排序的过程，但我们在后面详细介绍。

使 EXPLAIN 更加冗长

在 PostgreSQL 中，EXPLAIN 子句的输出可以加强一些，以便为您提供更多信息。要从执行计划中尽可能多地获取信息，请考虑启用以下选项：

```
test=# EXPLAIN (analyze, verbose, costs, timing, buffers)
SELECT * FROM a ORDER BY random();
                                         QUERY PLAN
-----
Sort (cost=834.39..859.39 rows=10000 width=12)
    (actual time=6.089..7.199 rows=10000 loops=1)
        Output: aid, (random())
        Sort Key: (random())
        Sort Method: quicksort Memory: 853kB
        Buffers: shared hit=45
    -> Seq Scan on public.a
        (cost=0.00..170.00 rows=10000 width=12)
        (actual time=0.012..2.625 rows=10000 loops=1)
        Output: aid, random()
        Buffers: shared hit=45
Planning time: 0.054 ms
Execution time: 7.992 ms
(10 rows)
```

analyze true 实际上会执行查询，如前所示。verbose true 将为计划添加更多信息（例如列信息等）。costs true 将显示有关成本的信息。timing true 同样重要，因为它将为我们提供良好的运行时时间数据，以便我们可以看到计划的时间耗费的位置。最后，buffers true，这可能非常有启发性。在我的例子中，它揭示了我们需要访问 45 个缓冲区来执行查询。

发现问题

鉴于第 5 章“日志文件和系统统计”中显示的所有信息，已经可以发现一些在现实生活中非常重要的潜在性能问题。

发现运行时的变化

在查看计划时，您总会有两个问题要问自己：

- EXPLAIN ANALYZE 子句显示的运行时间是否真正反映了查询实际运行时的耗费？
- 如果查询很慢，运行时时间耗费在哪里？

在我的例子中，顺序扫描的额定值为 2.625 毫秒。排序在 7.199 毫秒左右完成，因此

排序大约需要 4.5 毫秒才能完成，导致了查询所需的大部分运行时间。

寻找查询执行时间的跳转将揭示真正发生的事情。针对哪种类型的操作会耗费多少时间，您必须采取相应的措施。在这里不太可能提供一些一般性的建议，因为有太多东西可能导致问题。

检查估计

但是，应该要始终做一些事情：确保估计值和实际值完全相近。在某些情况下，优化程序会由于某种原因导致的估计值不准确而给出糟糕的执行计划。由于系统统计信息不是最新的，因此可能会出现估计值已关闭的情况。因此，运行 `ANALYZE` 子句绝对是一个好的开始。但是，优化器统计数据主要由自动 `vacuum` 进程处理，因此绝对值得考虑导致错误估计值的其他可能的选项。看看下面的示例，它可以帮助我们向表中添加一些数据：

```
test=# CREATE TABLE t_estimate AS
    SELECT * FROM generate_series(1, 10000) AS id;
SELECT 10000
```

加载 10000 行后，将创建优化程序统计信息：

```
test=# ANALYZE t_estimate;
ANALYZE
```

我们现在来看看估算值：

```
test=# EXPLAIN ANALYZE SELECT * FROM t_estimate WHERE cos(id) < 4;
QUERY PLAN
-----
Seq Scan on t_estimate (cost=0.00..220.00 rows=3333 width=4)
(actual time=0.010..4.006 rows=10000 loops=1)
Filter: (cos((id)::double precision) < '4'::double precision)
Planning time: 0.064 ms
Execution time: 4.701 ms
(4 rows)
```

在许多情况下，PostgreSQL 可能无法正确处理 `WHERE` 子句，因为它只有列的统计信息，而不是表达式。我们在这里看到的是对 `where` 子句返回的数据严重低估。

当然，也可能会高估数据量：

```
test=# EXPLAIN ANALYZE
SELECT *
FROM t_estimate
WHERE cos(id) > 4;
QUERY PLAN
-----
Seq Scan on t_estimate (cost=0.00..220.00 rows=3333 width=4)
```

```
(actual time=3.802..3.802 rows=0 loops=1)
Filter: (cos((id)::double precision) > '4'::double precision)
Rows Removed by Filter: 10000
Planning time: 0.037 ms
Execution time: 3.813 ms
(5 rows)
```

如果这样的事情发生在执行计划的深处，那么这个过程很可能会产生一个糟糕的计划。因此，确保估计值在一定范围内是很有意义的。

幸运的是，有一种方法可以解决这个问题：

```
test=# CREATE INDEX idx_cosine ON t_estimate (cos(id));
CREATE INDEX
```

创建索引将使 PostgreSQL 跟踪表达式的统计信息：

```
test=# ANALYZE t_estimate;
ANALYZE
```

除了该计划将确保显著更好的性能之外，它还将修复统计数据，即使未使用索引：

```
test=# EXPLAIN ANALYZE SELECT * FROM t_estimate WHERE cos(id) > 4;
QUERY PLAN
-----
Index Scan using idx_cosine on t_estimate
(cost=0.29..8.30 rows=1 width=4)
(actual time=0.002..0.002 rows=0 loops=1)
Index Cond: (cos((id)::double precision) > '4'::double precision)
Planning time: 0.095 ms
Execution time: 0.011 ms
(4 rows)
```

然而，估计错误的数量远远超出了人们的视线。经常被低估的一个问题称为跨列相关。

考虑一个涉及两列的简单示例：

- 20% 的人喜欢滑雪
- 20% 的人来自非洲

如果我们想计算非洲滑雪者的数量，数学表明结果将是 $0.2 \times 0.2 =$ 总人口的 4%。但是，非洲没有降雪，这个国家的收入很低。因此，实际结果肯定会更低。观察非洲和观察滑雪在统计上不是独立的。在许多情况下，postgresql 保留不跨越多个列的列统计信息这一事实可能会导致不好的结果。

当然，planner 做了很多事情来防止这些事情尽可能频繁地发生。不过，这仍然可能是一个问题。

从 PostgreSQL 10.0 开始，我们有多变量统计数据，它一劳永逸地终结了跨列相关的问题。

题。

检查缓冲区使用情况

但是，执行计划本身并不是导致问题的唯一因素。在许多情况下，危险的东西隐藏在其他一些层面上。内存和缓存可能会导致意外行为，对于未经过培训以查看本节中描述的问题的最终用户而言，这通常很难理解。

这是一个描述随机将数据插入表中的示例。该查询将生成一些随机排序的数据并将其添加到新表：

```
test=# CREATE TABLE t_random AS
        SELECT * FROM generate_series(1, 10000000) AS id ORDER BY random();
SELECT 10000000
test=# ANALYZE t_random ;
ANALYZE
```

我们现在生成了一个包含 1000 万行的简单表，并创建了优化器统计信息。

在下一步中，执行仅检索少量行的简单查询：

```
test=# EXPLAIN (analyze true, buffers true, costs true, timing true)
        SELECT * FROM t_random WHERE id < 1000;
                                QUERY PLAN
-----
Seq Scan on t_random (cost=0.00..169248.60 rows=1000 width=4)
(actual time=1.068..685.410 rows=999 loops=1)
Filter: (id < 1000)
Rows Removed by Filter: 999001
Buffers: shared hit=2112 read=42136
Planning time: 0.035 ms
Execution time: 685.551 ms
(6 rows)
```

在检查数据之前，请确保已执行两次查询。当然，在这里使用索引是有意义的。但是，在此查询中，PostgreSQL 在缓存中找到了 2,112 个缓冲块，并且必须从操作系统中获取 421136 个缓冲块。现在，有两件事情可能发生。如果运气好的话，操作系统会出现几个缓存命中，查询速度很快。如果不幸运，文件系统缓存中没有，则必须从磁盘中获取这些块。这可能看起来很明显；然而，它可能导致执行时间的剧烈波动。完全在缓存中运行的查询比必须从磁盘缓慢获取随机块的查询快 100 倍。

让我们试着通过一个简单的例子来概述这个问题。假设我们有一个存储 100 亿行的电话系统（这对于大型电话运营商来说并不罕见）。数据快速流入，用户希望查询此数据。如果你有 100 亿行，那么只有部分数据存在于内存，因此很多其他数据自然会最终来自磁盘。

现在让我们运行一个简单的查询来展示 PostgreSQL 如何查找电话号码：

```
SELECT * FROM data WHERE phone_number = '+12345678';
```

即使您正在通话，您的数据也会遍布各地。如果你结束通话只是为了开始下一个电话，成千上万的人会这样做，所以你的两个电话将在 8,000 个社区中同一个结束的几率自然接近于零。想象一下，目前有 10 万个电话在同一时间进行。在磁盘上，数据将随机分布。如果您的电话号码经常出现，则意味着对于每一行，必须从磁盘中取出至少一个块（假设缓存命中率非常低）。假设将返回 5,000 行。假设您必须转到磁盘 5,000 次，这会导致诸如 $5,000 \times 5$ 毫秒 = 25 秒的执行时间。请注意，此查询的执行时间可能会在毫秒之间变化，例如 30 秒，具体取决于操作系统或 PostgreSQL 缓存了多少。

请记住，每次重启服务器都会自然地清除 PostgreSQL 和文件系统缓存，这可能会在节点发生故障后导致真正的问题。

解决高缓冲区使用率

需要回答的问题是，我如何才能改善这种情况？一种方法是运行 CLUSTER 子句：

```
test=# \h CLUSTER
Command: CLUSTER
Description: cluster a table according to an index
Syntax:
CLUSTER [VERBOSE] table_name
[ USING index_name ] CLUSTER [VERBOSE]
```

cluster 子句将按照与 (btree) 索引相同的顺序重写表。如果您正在运行分析工作负载，那么这是有意义的。但是，在 oltp 系统中，cluster 子句可能不可行，因为在重写表时需要表锁。

理解和解决连接

联接很重要；每个人都需要掌握它的基础。连接也是维持、实现良好性能相关的内容。为了确保您可以编写好的联接，我们还将在本书中学习连接。

使用正确的连接

在我们深入研究优化连接之前，重要的是要看一下连接时出现的一些最常见的问题，以及哪些问题应该为您敲响警钟。

下面是一个简单的表结构示例，用于演示联接的工作方式：

```
test=# CREATE TABLE a (aid int);
CREATE TABLE
```

```
test=# CREATE TABLE b (bid int);
CREATE TABLE
test=# INSERT INTO a VALUES (1), (2), (3);
INSERT 0 3
test=# INSERT INTO b VALUES (2), (3), (4);
INSERT 0 3
```

在以下示例中，您将看到一个简单的外连接：

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid);
aid | bid
-----+
 1 |
 2 |    2
 3 |    3
(3 rows)
```

您可以看到 PostgreSQL 将从左侧获取所有行，并仅列出适合连接的行。

以下示例可能会让很多人感到惊讶：

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid AND bid = 2);
aid | bid
-----+
 1 |
 2 |    2
 3 |
(3 rows)
```

不，行数不会减少 - 它会保持不变。 大多数人都认为联接中只有一行，但事实并非如此，并且会导致一些隐藏的问题。

请考虑以下查询，该查询执行简单连接：

```
test=# SELECT avg(aid), avg(bid)
  FROM a LEFT JOIN b
    ON (aid = bid AND bid = 2);
avg      |      avg
-----+
2.0000000000000000 | 2.0000000000000000
(1 row)
```

大多数人认为平均值是基于单行计算的。 但是，如前所述，情况并非如此，因此诸如此类的查询通常被认为是有性能问题的，因为由于某种原因，PostgreSQL 不会在连接的左侧索引表。当然，我们不是在考虑性能问题 - 我们在研究语义问题。它定期发生，人们编写外连接并不真正知道他们要求 PostgreSQL 做什么。因此，我个人的建议是在受影响的客户端报告性能问题之前始终质疑外连接语义的正确性。

我无法强调这种工作对于确保您的查询是正确的并且完全满足需要的重要性。

处理外连接

在从业务角度验证您的查询实际上是正确的之后，检查优化器可以做些什么来加速外连接是有意义的。最重要的是，在许多情况下，PostgreSQL 可以重新排序内部联接以显著加快速度。但是，在外连接的情况下，这并不总是可行的。实际上只允许少数重新排序操作：

- $(A \text{ leftjoin } B \text{ on } (Pab)) \text{ innerjoin } C \text{ on } (Pac) = (A \text{ innerjoin } C \text{ on } (Pac)) \text{ leftjoin } B \text{ on } (Pab)$

Pac 是一个关联 A 和 C 的谓词，依此类推（在这种情况下，显然， Pac 不能关联 B，否则转换是无意义的）：

- $(A \text{ leftjoin } B \text{ on } (Pab)) \text{ leftjoin } C \text{ on } (Pac) = (A \text{ leftjoin } C \text{ on } (Pac)) \text{ leftjoin } B \text{ on } (Pab)$
- $(A \text{ leftjoin } B \text{ on } (Pab)) \text{ leftjoin } C \text{ on } (Pbc) = (A \text{ leftjoin } (B \text{ leftjoin } C \text{ on } (Pbc))) \text{ on } (Pab)$

如果所有 B 列的行都是空值， Pbc 谓词肯定会失败（ Pbc 是有严格要求的即对于 B 列来说至少存在一行记录），则最后一条规则才成立。如果 Pbc 未严格要求，则第一个表单可能会生成一些具有非空 C 列的行，其中第二个表单将使这些条目为空。

虽然某些连接可以重新排序，但典型的查询类型无法从连接重新排序中受益：

```
SELECT ...
  FROM a LEFT JOIN b ON (aid = bid)
    LEFT JOIN c ON (bid = cid)
      LEFT JOIN d ON (cid = did)
...
...
```

解决这个问题的方法是检查是否真的需要所有外连接。在许多情况下，人们会在没有实际需要的情况下编写外连接。通常，业务案例甚至根本不需要使用外连接。

了解 `join_collapse_limit` 变量

在执行计划生产过程中，PostgreSQL 尝试检查所有可能的连接顺序。在许多情况下，这可能相当昂贵，因为可能存在许多排列，这自然会减慢执行计划生成的过程。

`join_collapse_limit` 变量用于为开发人员提供实际解决这些问题的工具，并以更直接的方式定义如何处理查询。

为了展示这个设置的全部内容，我们现在将编写一个小例子：

```
SELECT * FROM tab1, tab2, tab3
  WHERE tab1.id = tab2.id
    AND tab2.ref = tab3.id;
```

```
SELECT * FROM tab1 CROSS JOIN tab2
CROSS JOIN tab3
WHERE tab1.id = tab2.id;
    AND tab2.ref = tab3.id;
SELECT * FROM tab1 JOIN (tab2 JOIN tab3
    ON (tab2.ref = tab3.id))
    ON (tab1.id = tab2.id);
```

基本上，这三个查询是相同的，`planner` 将以相同的方式对待。第一个查询由隐式连接组成。最后一个只包含显式连接。在内部，`planner` 将检查这些请求并相应的排序连接顺序，以获得最佳的执行时间。这里的问题是，PostgreSQL 将会默默计划出多少显式的连接次序？这正是您可以通过设置 `join_collapse_limit` 变量来告诉 `planner` 的。对于普通查询来说默认值相当不错。但是，如果您的查询包含非常多的连接，那么使用此设置可以大大减少计划时间。缩短计划时间对于保持良好的吞吐量至关重要。

要查看 `join_collapse_limit` 变量如何更改计划，我们将编写此简单查询：

```
test=# EXPLAIN WITH x AS
(  

    SELECT *
        FROM generate_series(1, 1000) AS id
)  

SELECT *
FROM x AS a
JOIN x AS b ON (a.id = b.id)
JOIN x AS c ON (b.id = c.id)
JOIN x AS d ON (c.id = d.id)
JOIN x AS e ON (d.id = e.id)
JOIN x AS f ON (e.id = f.id);
```

尝试使用不同的设置运行查询，并查看计划如何更改。遗憾的是，该计划因为太长了无法复制在此，因此无法在此部分中包含此内容。

启用和禁用优化设置

到目前为止，已经详细讨论了 `planner` 执行的最重要的优化。多年来 PostgreSQL 已经有了很大的改进。然而，事情发生可能南辕北辙，用户必须说服 `planner` 做正确的事情。

为了修改计划，PostgreSQL 提供了一些对计划产生重大影响的运行时变量。这个想法是让最终用户有机会使计划中的某些类型的节点比其他节点更昂贵。这在实践中意味着什么？这是一个简单的计划：

```
test=# explain SELECT *
FROM generate_series(1, 100) AS a,
generate_series(1, 100) AS b
```

```
WHERE a = b;
      QUERY PLAN
-----
Merge Join (cost=119.66..199.66 rows=5000 width=8)
  Merge Cond: (a.a = b.b)
    -> Sort (cost=59.83..62.33 rows=1000 width=4)
      Sort Key: a.a
      -> Function Scan on generate_series a
          (cost=0.00..10.00 rows=1000 width=4)
    -> Sort (cost=59.83..62.33 rows=1000 width=4)
      Sort Key: b.b
      -> Function Scan on generate_series b
          (cost=0.00..10.00 rows=1000 width=4)
(8 rows)
```

该计划显示 PostgreSQL 从函数中读取数据并对两个结果进行排序。然后，执行合并连接。

但是，如果合并连接不是运行查询的最快方法，该怎么办？在 PostgreSQL 中，没有办法像在 Oracle 中那样将计划者提示放入注释中。相反，您可以确保某些操作被认为是昂贵的。`SET enable_mergejoin TO off` 命令会使合并连接操作过于昂贵：

```
test=# SET enable_mergejoin TO off;
SET
test=# explain SELECT *
  FROM generate_series(1, 100) AS a,
       generate_series(1, 100) AS b
 WHERE a = b;
      QUERY PLAN
-----
Hash Join (cost=22.50..210.00 rows=5000 width=8)
  Hash Cond: (a.a = b.b)
    -> Function Scan on generate_series a
        (cost=0.00..10.00 rows=1000 width=4)
    -> Hash (cost=10.00..10.00 rows=1000 width=4)
      -> Function Scan on generate_series b
          (cost=0.00..10.00 rows=1000 width=4)
(5 rows)
```

由于合并连接太昂贵，PostgreSQL 决定尝试散列连接。正如您所看到的，成本略高，但是执行计划仍然生成了并且没有使用合并连接的方式。

如果散列连接关闭会发生什么？

```
test=# SET enable_hashjoin TO off;
SET
```

```
test=# explain SELECT *
  FROM generate_series(1, 100) AS a,
       generate_series(1, 100) AS b
 WHERE a = b;
                                         QUERY PLAN
-----
Nested Loop (cost=0.01..22510.01 rows=5000 width=8)
  Join Filter: (a.a = b.b)
    -> Function Scan on generate_series a
        (cost=0.00..10.00 rows=1000 width=4)
    -> Function Scan on generate_series b
        (cost=0.00..10.00 rows=1000 width=4)
(4 rows)
```

PostgreSQL 将再次尝试其他东西，并使用了嵌套循环。嵌套循环的成本已经非常惊人，但是 planner 已经没有其他办法了。

如果嵌套循环也关闭会发生什么？

```
test=# SET enable_nestloop TO off;
SET
test=# explain SELECT *
  FROM generate_series(1, 100) AS a,
       generate_series(1, 100) AS b
 WHERE a = b;
                                         QUERY PLAN
-----
Nested Loop (cost=10000000000.00..10000022510.00
  rows=5000 width=8)
  Join Filter: (a.a = b.b)
    -> Function Scan on generate_series a
        (cost=0.00..10.00 rows=1000 width=4)
    -> Function Scan on generate_series b
        (cost=0.00..10.00 rows=1000 width=4)
(4 rows)
```

PostgreSQL 仍将执行嵌套循环。这里的重要部分是 off 并不意味着关闭它，只是意味着将它视为一种非常昂贵的东西。这很重要，否则无法执行查询。

什么设置会影响 planner？以下选项开关可用：

- enable_bitmapscan = on
- enable_hashagg = on
- enable_hashjoin = on
- enable_indexscan = on
- enable_indexonlyscan = on

-
- enable_material = on
 - enable_mergejoin = on
 - enable_nestloop = on
 - enable_seqscan = on
 - enable_sort = on
 - enable_tidscan = on

虽然这些设置肯定是有益的，但让我们应该理解这些调整处理起来应该小心。它们应该仅用于加速单个查询，而不是在全局设置层面关闭它。关闭可能会很快对您的系统产生不利影响并破坏性能。因此，在更改这些参数之前要三思而后行。

了解遗传查询优化（GEQO）

执行计划规划过程的结果是实现卓越绩效的关键。如本章所示，生成执行计划远非微不足道，涉及各种复杂的计算。查询触及的表越多，计划就越复杂。表越多，planner的选择就越多。从逻辑上讲，规划的时间会增加。在某些时候，规划过程将花费很长时间，以至于经典的穷举法搜索已不再可行。最重要的是，无论如何，在规划期间所犯的偏差是如此之大，以至于找到理论上最好的计划并不一定会是运行时的最佳计划。

在这种情况下，遗传查询优化（GEQO）可以拯救上面的问题。什么是 GEQO？这个想法源于自然的灵感，类似于自然的进化过程。

PostgreSQL 将像旅行商问题一样处理这个问题，并将可能的连接编码为整型字符串。例如，4-1-3-2 表示：首先，连接 4 和 1，然后是 3，然后是 2.数字表示关系的 ID。

最初，遗传优化器将生成一组随机计划。然后检查这些计划。丢弃坏的，并根据好的基因生成新的。这样，可能会产生更好的计划。可以根据需要经常重复该过程。在一天结束时，我们留下的计划预计会比使用随机计划好很多。可以通过调整 geqo 变量来打开和关闭 GEQO，如以下行所示：

```
test=# SHOW geqo;
geqo
-----
on
(1 row)
test=# SET geqo TO off;
SET
```

默认情况下，如果语句超过某个复杂程度，则 geqo 变量将启动，该复杂程度由以下变量控制：

```
test=# SHOW geqo_threshold ;
geqo_threshold
-----
12
(1 row)
```

如果您的查询太大并开始达到此阈值，那么使用此设置来查看计划程序如何更改这些变量的确定计划当然是有意义的。

但是，作为一般规则，我建议尽可能避免使用 `GEQO`，并尝试通过使用 `joinCollapse_limit` 变量以某种方式修复连接顺序来解决问题。请注意，每个查询都是不同的，因此通过了解 `planner` 在不同情况下的行为方式，它确实有助于实验并获得更多经验。



TIP 如果你想看看真正疯狂的连接是什么，请考虑查看我在马德里的演讲
<http://de.slideshare.net/hansjurgenschonig/postgresql-joining-1-million-tables>。

分区数据

鉴于默认的 8 k 块大小，PostgreSQL 可以在一个表中存储多达 32 TB 的数据。如果使用 32 k 块编译 PostgreSQL，您甚至可以将 128 TB 数据放入一个表中。但是，诸如此类的大表使用上是不方便的，然而分区表可以使其处理更容易并且在某些情况下更快一些，这当然是有相当意义的。从版本 10.0 开始，PostgreSQL 提供了改进的分区功能，这将使最终用户更容易处理数据分区。

在本章中，将介绍旧的分区方法以及 PostgreSQL 11.0 中提供的新功能。

创建分区

首先，我们将仔细研究过时的数据分区方法。请记住，理解这种技术对于深入了解 PostgreSQL 在幕后的作用非常重要。

在深入研究分区的优点之前，我想展示如何创建分区。整个事情可以从我们使用以下命令创建的父表开始：

```
test=# CREATE TABLE t_data (id serial, t date, payload text);
CREATE TABLE
```

在此示例中，父表有三列。日期列将用于分区，但稍后我们将对此进行更多介绍。

现在父表已就绪，可以创建子表。这是它的工作方式：

```
test=# CREATE TABLE t_data_2016 () INHERITS (t_data);
CREATE TABLE
test=# \d t_data_2016
          Table "public.t_data_2016"
   Column | Type      | Modifiers
-----+-----+-----+
    id   | integer   | not null default
                  nextval('t_data_id_seq'::regclass)
      t   | date      |
```

```
payload | text      |
Inherits: t_data
```

该表名为 `t_data_2016`, 并继承自 `t_data`。这意味着不会向子表添加额外的列。如您所见, 继承意味着父表中的所有列都可以在子表中使用。另请注意, `id` 列将从父表继承序列, 以便所有子级可以共享相同的数值。

让我们创建更多的表:

```
test=# CREATE TABLE t_data_2015 () INHERITS (t_data);
CREATE TABLE
test=# CREATE TABLE t_data_2014 () INHERITS (t_data);
CREATE TABLE
```

到目前为止, 所有表结构都是相同的, 只是从父表继承。但是, 还有更多: 子表实际上可以有比父项更多的列。添加更多字段很简单:

```
test=# CREATE TABLE t_data_2013 (special text) INHERITS (t_data);
CREATE TABLE
```

在这种情况下, 添加了一个特殊列。它对父表没有影响; 它只是丰富了子表, 使他们能够持有更多的数据。

创建一些表后, 可以添加一行数据:

```
test=# INSERT INTO t_data_2015 (t, payload)
          VALUES ('2015-05-04', 'some data');
INSERT 0 1
```

现在最重要的是父表可用于查找子表中的所有数据:

```
test=# SELECT * FROM t_data;
      id |      t      | payload
-----+-----+-----
      1 | 2015-05-04 | some data
(1 row)
```

查询父表允许您以简单有效的方式访问父表下的所有内容。

要了解 PostgreSQL 如何进行分区, 请查看该计划:

```
test=# EXPLAIN SELECT * FROM t_data;
                                         QUERY PLAN
-----
Append (cost=0.00..84.10 rows=4411 width=40)
  -> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
  -> Seq Scan on t_data_2016
    (cost=0.00..22.00 rows=1200 width=40)
  -> Seq Scan on t_data_2015
```

```
(cost=0.00..22.00 rows=1200 width=40)
-> Seq Scan on t_data_2014
  (cost=0.00..22.00 rows=1200 width=40)
-> Seq Scan on t_data_2013
  (cost=0.00..18.10 rows=810 width=40)
(6 rows)
```

实际上，这个过程非常简单。 PostgreSQL 将简单地统一所有表并向我们显示我们正在查看的分区内部和下方所有表中的所有内容。 请注意，所有表都是独立的，只是通过系统目录逻辑连接。

应用表约束

如果过滤器应用于表格会发生什么？ 优化器将如何以最有效的方式决定执行此查询的内容？ 以下示例向我们展示了 PostgreSQL planner 的行为：

```
test=# EXPLAIN SELECT * FROM t_data WHERE t = '2016-01-04';
          QUERY PLAN
-----
Append (cost=0.00..95.12 rows=23 width=40)
-> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
  Filter: (t = '2016-01-04'::date)
-> Seq Scan on t_data_2016 (cost=0.00..25.00 rows=6 width=40)
  Filter: (t = '2016-01-04'::date)
-> Seq Scan on t_data_2015 (cost=0.00..25.00 rows=6 width=40)
  Filter: (t = '2016-01-04'::date)
-> Seq Scan on t_data_2014 (cost=0.00..25.00 rows=6 width=40)
  Filter: (t = '2016-01-04'::date)
-> Seq Scan on t_data_2013 (cost=0.00..20.12 rows=4 width=40)
  Filter: (t = '2016-01-04'::date)
(11 rows)
```

PostgreSQL 将数据过滤器应用于表结构中的所有分区。 它不知道表名是否以某种方式与表的内容相关。 对于数据库，名称只是名称，与您要查找的内容无关。 当然，这是有道理的，因为没有其他任何数学上的理由。

现在的观点是：我们如何教数据库 2016 表只包含 2016 数据，2015 表只包含 2015 数据，依此类推？ 表约束就是这样做的。 他们教 PostgreSQL 关于这些表的内容，因此允许 planner 做出比以前更聪明的决定。 该功能称为约束排除，有助于在许多情况下显著加快查询速度。

以下清单显示了如何创建表约束：

```
test=# ALTER TABLE t_data_2013
ADD CHECK (t < '2014-01-01');
ALTER TABLE
```

```
test=# ALTER TABLE t_data_2014
      ADD CHECK (t >= '2014-01-01' AND t < '2015-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2015
      ADD CHECK (t >= '2015-01-01' AND t < '2016-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2016
      ADD CHECK (t >= '2016-01-01' AND t < '2017-01-01');
ALTER TABLE
```

对于每个表，可以添加 CHECK 约束。



TIP 如果这些表中的所有数据都完全正确并且每一行都满足约束条件，PostgreSQL 将只创建约束。与 MySQL 相比，PostgreSQL 中的约束在任何情况下都被认真对待并受到尊重。

在 PostgreSQL 中，这些约束可以重叠 - 这不是被禁止的，在某些情况下可能有意义。但是，通常最好使用非重叠约束，因为 PostgreSQL 可以选择裁剪更多表。

以下是添加这些表约束后会发生的情况：

```
test=# EXPLAIN SELECT * FROM t_data WHERE t = '2016-01-04';
          QUERY PLAN
-----
Append (cost=0.00..25.00 rows=7 width=40)
  -> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
        Filter: (t = '2016-01-04'::date)
  -> Seq Scan on t_data_2016 (cost=0.00..25.00 rows=6 width=40)
        Filter: (t = '2016-01-04'::date)
(5 rows)
```

规划器将能够从查询中剔除许多表，并仅保留那些可能包含数据的表。查询可以从更短、更有效的执行计划中受益。特别是，如果这些表非常大，剔除它们可以大大提高速度。

修改继承的结构

偶尔，必须修改数据结构。ALTER TABLE 子句就是这样做的。这里的问题是，如何修改分区表。

基本上，您所要做的就是处理父表并添加或删除列。

PostgreSQL 会自动将这些更改传递到子表，并确保对所有关系进行更改，如下所示：

```
test=# ALTER TABLE t_data ADD COLUMN x int;
ALTER TABLE
test=# \d t_data_2016
```

Table "public.t_data_2016"

Column	Type	Modifiers
id	integer	not null default nextval('t_data_id_seq'::regclass)
t	date	
payload	text	
x	integer	

Check constraints:

```
"t_data_2016_t_check"
CHECK (t >= '2016-01-01'::date AND t < '2017-01-01'::date)
```

Inherits: t_data

如您所见，该列已添加到父表，并在此处自动添加到子表。

请注意，这适用于修改列等。索引则是一个完全不同的故事。在继承的结构中，每个表都必须单独编制索引。如果向父表添加索引，它将仅存在于父表中 - 它将不会部署在这些子表上。给所有这些表中的列创建索引是您的任务，PostgreSQL 不会为您做出这些决定。当然，这可以被视为新特性或限制。从好的方面来说，你可以说 PostgreSQL 为你提供了单独创建索引内容的所有灵活性，因此可能更有效。但是，人们也可能会争辩说，逐个部署所有这些索引的工作要多得多。

将表移入和移出分区结构

假设您有一个继承的数据结构。数据按日期进行分区，您希望向最终用户提供最近几年的数据。在某些时候，您可能希望从用户范围内删除一些数据而不实际触及它。您可能希望将数据移入归档中。

PostgreSQL 提供了一种简单的方法来实现这一点。首先，可以创建一个新的父表：

```
test=# CREATE TABLE t_history (LIKE t_data);
CREATE TABLE
```

LIKE 关键字允许您创建一个与 t_data 表具有完全相同结构的表。如果你忘记了 t_data 表实际上有哪些列，这可能会派上用场，因为它可以为你节省大量的工作。还可以包括索引，约束和默认值。

然后，可以将表从旧父表移开并放在新表下面。

下面是它的工作方式：

```
test=# ALTER TABLE t_data_2013 NO INHERIT t_data;
ALTER TABLE
test=# ALTER TABLE t_data_2013 INHERIT t_history;
ALTER TABLE
```

当然，整个过程可以在单个事务中完成，以确保操作的原子性。

清理数据

分区表的一个优点是能够快速清理数据。假设我们要删除一整年的数据；如果数据是相应的分区，一个简单的 DROP TABLE 子句可以完成这项工作：

```
test=# DROP TABLE t_data_2014;  
DROP TABLE
```

如您所见，丢弃子表很容易。但是父表会怎么样？因为有依赖的对象，因此 PostgreSQL 自然会出错，以确保不会发生任何意外情况：

```
test=# DROP TABLE t_data;  
ERROR: cannot drop table t_data because other objects depend on it  
DETAIL: default for table t_data_2013 column id depends on  
sequence t_data_id_seq  
table t_data_2016 depends on table t_data  
table t_data_2015 depends on table t_data  
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

DROP TABLE 子句将警告我们存在依赖对象并拒绝删除这些表。需要 CASCADE 子句来强制 PostgreSQL 实际删除这些对象以及父表。以下示例向我们展示了如何使用级联 DROP TABLE：

```
test=# DROP TABLE t_data CASCADE;  
NOTICE: drop cascades to 3 other objects  
DETAIL: drop cascades to default for table t_data_2013 column id drop  
cascades to table t_data_2016  
drop cascades to table t_data_2015  
DROP TABLE
```

了解 PostgreSQL 11.0 分区

PostgreSQL 10 和 11 中添加的许多内容将确保您在“旧版本”中看到的内容是自动化完成的。对于索引，插入时的元组数据路由等等尤其如此。但是，让我们以更有条理的方式审视这些事情。

多年来，PostgreSQL 社区一直致力于实现内置分区功能。最后，PostgreSQL 10.0 现在提供了第一个内核分区实现，本章将对此进行介绍。在 PostgreSQL 10 中，分区功能仍然非常繁琐，因此 PostgreSQL 11 中的许多内容都得到了改进，使想要使用这一重要功能的人们的生活变得更加轻松。

为了向您展示分区的工作原理，我编写了一个包含范围分区的简单示例，如下所示：

```
CREATE TABLE data (  
    payload integer
```

```
) PARTITION BY RANGE (payload);
CREATE TABLE negatives PARTITION
    OF data FOR VALUES FROM (MINVALUE) TO (0);
CREATE TABLE positives PARTITION
    OF data FOR VALUES FROM (0) TO (MAXVALUE);
```

在此示例中，一个分区将保留所有负值，而另一个分区将处理正值。在创建父表时，您只需指定要对数据进行分区的方式。



在 PostgreSQL 10.0 中，有范围分区和列表分区。PostgreSQL 11.0 可能会立即支持散列分区和 LIKE 操作。

一旦创建了父表，就可以继续创建分区了。为此，必须添加 PARTITION OF 子句。此时，仍有一些限制（截至 PostgreSQL 10）。最重要的一个是在元组（一行）不能从一个分区移动到另一个分区，例如：

```
UPDATE data SET payload = -10 WHERE id = 5
```

幸运的是，这个限制已被取消，PostgreSQL 11 数据能够从一个分区移动到另一个分区。但是，请记住，在分区之间移动数据可能不是好的想法。

下一个重要的方面与索引有关：在 PostgreSQL 10 中，每个表（每个分区）都必须单独编制索引。这在 PostgreSQL 11 中不再适用。让我们试一试，看看会发生什么：

```
test=# CREATE INDEX idx_payload ON data (payload);
CREATE INDEX
test=# \d positives
Table "public.positives"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
 payload | integer |  || |
 Partition of: data FOR VALUES FROM (0) TO (MAXVALUE)
Indexes:
"positives_payload_idx" btree (payload)
```

你可以在这里看到的是索引也被自动添加到子分区表中，这是 PostgreSQL 11 的一个非常重要的特性，并且已经得到那些将其应用程序移动迁移到 PostgreSQL 11 的用户的广泛认可。

另一个重要功能是能够创建默认分区。为了向您展示它是如何工作的，我们可以删除两个分区中的一个：

```
test=# DROP TABLE negatives;
DROP TABLE
```

然后，可以轻松创建数据表的默认分区：

```
test=# CREATE TABLE p_def PARTITION OF data DEFAULT;
```

CREATE TABLE

所有不适合任何地方的数据都将在此默认分区中结束，这可确保永远不会忘记创建正确的分区。 经验表明，随着时间的推移， 默认分区的存在使应用程序更加可靠。

以下清单显示了如何插入数据以及数据最终会在何处结束：

```
test=# INSERT INTO data VALUES (1), (-1);
INSERT 0 2
test=# SELECT * FROM data;
payload
-----
 1
 -1
(2 rows)
test=# SELECT * FROM positives;
payload
-----
 1
(1 row)
test=# SELECT * FROM p_def;
payload
-----
 -1
(1 row)
```

如您所见，查询父表将返回所有数据。 各个分区将保存难找分区规则定义的数据。

调整参数以获得良好的查询性能

编写好的查询语句是实现良好性能的第一步。 没有好的查询语句，你很可能遇到糟糕的表现。 因此，编写好的、智能的代码将为您提供最大的优势。一旦您的查询从逻辑和语义的角度进行了优化，良好的内存设置可以为您提供良好的查询加速。在本节中，我们将了解更多内存可以为您做什么。PostgreSQL 如何将它用于提升您的效益。同样，本节假设我们使用单核查询来使计划更具可读性。 要确保始终只有一个核心，请使用以下命令：

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
```

这里是一个简单的示例，演示了内存参数可以为您做什么：

```
test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test (name)
      SELECT 'hans' FROM generate_series(1, 100000);
INSERT 0 100000
```

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

将向表中添加包含 hans 的 100 万行。然后，加载包含 paul 的 100 万行。总而言之，将有 200 万个唯一 ID，但只有两个不同的名称。

现在让我们使用 PostgreSQL 的默认内存设置运行一个简单的查询：

```
test=# SELECT name, count(*) FROM t_test GROUP BY 1;
      name   | count
-----+-----
      hans   | 100000
      paul   | 100000
(2 rows)
```

将返回两行结果，但这不应该是个惊喜。这里重要的不是结果，而是 PostgreSQL 在幕后做的事情：

```
test=# EXPLAIN ANALYZE SELECT name, count(*)
      FROM t_test
      GROUP BY 1;
                                         QUERY PLAN
-----
HashAggregate (cost=4082.00..4082.01 rows=1 width=13)
(actual time=51.448..51.448 rows=2 loops=1)
  Group Key: name
-> Seq Scan on t_test
  (cost=0.00..3082.00 rows=200000 width=5)
  (actual time=0.007..14.150 rows=200000 loops=1)
Planning time: 0.032 ms
Execution time: 51.471 ms
(5 rows)
```

PostgreSQL 发现分组数据的量实际上非常小。因此，它创建一个哈希并为每个分组数据添加一个哈希条目并开始计数。由于分组的数量很少，哈希值非常小，PostgreSQL 可以通过递增每个分组的数量来执行快速计算。

如果我们按 ID 分组而不是按名称分组会怎样？分组的数量将猛增：

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;
                                         QUERY PLAN
-----
GroupAggregate (cost=23428.64..26928.64 rows=200000 width=12)
(actual time=97.128..154.205 rows=200000 loops=1)
  Group Key: id
-> Sort (cost=23428.64..23928.64 rows=200000 width=4)
```

```
(actual time=97.120..113.017 rows=200000 loops=1)
Sort Key: id
Sort Method: external sort Disk: 2736kB
-> Seq Scan on t_test
(cost=0.00..3082.00 rows=200000 width=4)
(actual time=0.017..19.469 rows=200000 loops=1)

Planning time: 0.128 ms
Execution time: 160.589 ms

(8 rows)
```

PostgreSQL 发现分组数据的量现在要大得多，并且很快就会改变其策略。问题是包含这么多条目的哈希不适合存在于内存中：

```
test=# SHOW work_mem ;
work_mem
-----
4MB
(1 row)
```

`work_mem` 变量控制 GROUP BY 子句使用的哈希的大小。由于条目太多，PostgreSQL 必须找到一种不需要将整个数据集保存在内存中的策略。解决方案是按 ID 对数据进行排序并对其进行分组。一旦数据被排序，PostgreSQL 就可以向下移动列表并形成一个接一个的组。如果第一类值被计算出，则这部分部分结果可以被读取且被换出。然后，可以处理下一组。一旦排序列表中的值在向下移动时发生变化，它将永远不再出现；因此，系统知道部分结果已准备就绪。

为了加快查询速度，可以动态设置更高的 `work_mem` 变量值（当然，全局的）：

```
test=# SET work_mem TO '1 GB';
SET
```

该计划现在将再次实现具有快速有效的哈希聚合：

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;
QUERY PLAN
-----
HashAggregate (cost=4082.00..6082.00 rows=200000 width=12)
(actual time=76.967..118.926 rows=200000 loops=1)
Group Key: id
-> Seq Scan on t_test
(cost=0.00..3082.00 rows=200000 width=4)
(actual time=0.008..13.570 rows=200000 loops=1)

Planning time: 0.073 ms
Execution time: 126.456 ms

(5 rows)
```

PostgreSQL 知道（或至少假设）数据将适合存储在内存并切换到更快的计划。如您所

见，执行时间较短。 查询的速度不会像 GROUP By name 列那样快，因为必须计算更多的哈希值，但在绝大多数情况下，您将能够看到一个好的可靠的益处。

加快排序操作

`work_mem` 变量不仅可以加快分组速度。 它还可以对简单的事物产生非常好的影响，例如排序，这是世界上每个数据库系统都掌握的重要机制。

以下查询显示使用默认设置 4 MB 的简单操作：

```
test=# SET work_mem TO default;
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
                                QUERY PLAN
-----
Sort (cost=24111.14..24611.14 rows=200000 width=9)
  (actual time=219.298..235.008 rows=200000 loops=1)
    Sort Key: name, id
    Sort Method: external sort Disk: 3712kB
      -> Seq Scan on t_test
        (cost=0.00..3082.00 rows=200000 width=9)
          (actual time=0.006..13.807 rows=200000 loops=1)

Planning time: 0.064 ms
Execution time: 241.375 ms
(6 rows)
```

PostgreSQL 需要 13.8 毫秒来读取数据，超过 200 毫秒来对数据进行排序。 由于可用内存量较少，因此必须使用临时文件执行排序。 外部排序 Disk 方法只需要少量 RAM，但是必须将中间数据发送到相对较慢的存储设备，这当然会导致吞吐量不足。

增加 `work_mem` 变量设置将使 PostgreSQL 使用更多内存进行排序：

```
test=# SET work_mem TO '1 GB';
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
                                QUERY PLAN
-----
Sort (cost=20691.64..21191.64 rows=200000 width=9)
  (actual time=36.481..47.899 rows=200000 loops=1)
    Sort Key: name, id
    Sort Method: quicksort Memory: 15520kB
      -> Seq Scan on t_test
        (cost=0.00..3082.00 rows=200000 width=9)
          (actual time=0.010..14.232 rows=200000 loops=1)

Planning time: 0.037 ms
Execution time: 55.520 ms
```

(6 rows)

由于现在有足够的内存，数据库将在内存中进行所有排序，从而大大加快了这一过程。排序现在仅需 33 毫秒，与我们之前的查询相比，这是七倍的改进。更多的内存将导致更快的排序，并将加快系统。

到目前为止，您已经看到了两种排序数据的机制：外部排序磁盘和快速排序内存。除了这两种机制之外，还有第三种算法，它是 top-N heapsort Memory。它可用于仅为您提供前 N 行：

```
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id LIMIT 10;
                                         QUERY PLAN
-----
Limit (cost=7403.93..7403.95 rows=10 width=9)
  (actual time=31.837..31.838 rows=10 loops=1)
    -> Sort (cost=7403.93..7903.93 rows=200000 width=9)
        (actual time=31.836..31.837 rows=10 loops=1)
          Sort Key: name, id
          Sort Method: top-N heapsort Memory: 25kB
        -> Seq Scan on t_test
            (cost=0.00..3082.00 rows=200000 width=9)
            (actual time=0.011..13.645 rows=200000 loops=1)

Planning time: 0.053 ms
Execution time: 31.856 ms
```

(7 rows)

该算法非常快速，整个查询将在 30 毫秒内完成。排序部分现在只有 18 毫秒，因此几乎与首先读取数据一样快。

请注意，每个操作都分配了 `work_mem` 变量。从理论上讲，查询需要多次执行 `work_mem` 变量。它不是一个全局性的设置 - 它实际上是每个操作。因此，您必须谨慎设置它。

我们需要记住的一件事是，有很多书声称在 OLTP 系统上将 `work_mem` 变量设置得太高可能会导致服务器内存不足。是的，如果 1000 个人同时排序 100 MB，这可能会导致内存不足故障。但是，您希望磁盘能够有能力处理吗？我对此表示怀疑。解决方案只能是重新思考你在做什么。同时排序 100 MB 1000 次并不是 OLTP 系统中应该发生的事情。考虑部署正确的索引，编写更好的查询，或者只是重新考虑您的需求。在任何情况下，经常同时排序这么多数据是一个坏主意 - 在这些事情停止你的应用程序之前请停止。

加快管理任务

有更多的操作实际上需要进行某种排序或内存分配。诸如 `CREATE INDEX` 子句之类的管理变量不依赖于 `work_mem` 变量，而是使用 `maintenance_work_mem` 变量。下面是它

的工作原理：

```
test=# SET maintenance_work_mem TO '1 MB';
SET
test=# timing
Timing is on.
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
Time: 104.268 ms
```

如您所见，在 200 万行上创建索引大约需要 100 毫秒，这非常慢。因此，`maintenance_work_mem` 变量可用于加速排序，这基本上是 `CREATE INDEX` 子句所需要的：

```
test=# SET maintenance_work_mem TO '1 GB';
SET
test=# CREATE INDEX idx_id2 ON t_test (id);
CREATE INDEX
Time: 46.774 ms
```

速度现在已经翻了一倍，因为排序已经得到了很大改善。

有更多的管理工作可以从更多的内存中受益。最突出的是 `VACUUM` 子句（清除索引）和 `ALTER TABLE` 子句。

`maintenance_work_mem` 变量的规则与 `work_mem` 变量的规则相同。该设置是按操作进行的，只需动态分配所需的内存。

在 PostgreSQL 11 中，数据库引擎增加了一个额外的功能：PostgreSQL 现在能够并行构建 btree 索引，这可以大大加快大型表的索引。负责配置并行性的参数如下：

```
test=# SHOW max_parallel_maintenance_workers;
max_parallel_maintenance_workers
-----
2
(1 row)
```

`max_parallel_maintenance_workers` 控制 `CREATE INDEX` 可以使用的最大工作进程数。对于每个并行操作，PostgreSQL 将根据表大小确定工作量。给大型表创建索引时，可以看到显著的改进。在 Cybertec，我做了一些广泛的测试，并在我的一篇博文中总结了我 的 发 现：：<https://www.cybertec-postgresql.com/en/postgresql-parallel-create-index-for-better-performance/>。

使用并行查询

从版本 9.6 开始，PostgreSQL 支持并行查询。随着时间的推移，对并行性的支持逐渐得到了改进，版本 11 为这一重要功能增加了更多功能。在本节中，我们将了解并行性如何工作以及可以采取哪些措施来加快速度。

在深入研究细节之前，有必要创建一些示例数据，如以下部分所示：

```
test=# CREATE TABLE t_parallel AS
  SELECT * FROM generate_series(1, 25000000) AS id;
SELECT 25000000
```

加载初始数据后，我们可以运行第一个并行查询。一个简单的计数将显示，一般的并行查询是什么样的：

```
test=# explain SELECT count(*) FROM t_parallel;
          QUERY PLAN
-----
-----
Finalize Aggregate (cost=258537.40..258537.41 rows=1 width=8)
-> Gather (cost=258537.19..258537.40 rows=2 width=8)
  Workers Planned: 2
    -> Partial Aggregate (cost=257537.19..257537.20 rows=1 width=8)
      -> Parallel Seq Scan on t_parallel (cost=0.00..228153.75
rows=11753375 width=0)
(5 rows)
```

让我们详细了解一下查询的执行计划。PostgreSQL第一次执行并行顺序扫描。这实现了PostgreSQL将使用多于1个CPU来处理表（逐块），它将创建部分聚合。收集节点的工作是收集数据并将其传递以进行最终聚合。因此，聚集节点是并行性工作的结束点。值得一提的是，并行性（目前）从未嵌套。收集节点中永远不会有嵌套有收集节点。在我的示例中，PostgreSQL决定使用了两个工作进程。这是为什么？

让我们考虑以下变量设置：

```
test=# SHOW max_parallel_workers_per_gather;
max_parallel_workers_per_gather
-----
2
(1 row)
```

`max_parallel_workers_per_gather`将收集节点下允许的工作进程数限制为两个。重要的是：如果表很小，它将永远不会使用并行性。表的大小必须至少为8MB，如如下配置设置所定义：

```
test=# SHOW min_parallel_table_scan_size;
min_parallel_table_scan_size
-----
8MB
(1 row)
```

并行规则现在是：表的大小必须增加三倍才能使PostgreSQL再添加一个工作进程。换句话说：要获得四名额外的工作进程，您需要至少81倍的数据。这是有道理的，因为

你的数据库的大小上升了 100 倍，存储系统通常不会快 100 倍。因此，有用核心的数量有限。

但是，我们的表相当大：

```
test=# \d+
List of relations
 Schema | Name      | Type  | Owner | Size   | Description
-----+-----+-----+-----+-----+
 public | t_parallel | table | hs    | 864 MB |
(1 row)
```

在我的示例中，`max_parallel_workers_per_gather` 限制了核心数。如果我们更改此设置，PostgreSQL 将决定使用更多核：

```
test=# SET max_parallel_workers_per_gather TO 10;
SET
test=# explain SELECT count(*) FROM t_parallel;
          QUERY PLAN
-----
-----
Finalize Aggregate (cost=174120.82..174120.83 rows=1 width=8)
 -> Gather (cost=174120.30..174120.81 rows=5 width=8)
   Workers Planned: 5
     -> Partial Aggregate (cost=173120.30..173120.31 rows=1 width=8)
     -> Parallel Seq Scan on t_parallel (cost=0.00..160620.24 rows=5000024
       width=0)
(5 rows)
```

在这种情况下，我们只得到 5 个工作进程（如预期的那样）。

但是，在某些情况下，您希望某个表使用的核心数量要高得多。想象一下 200 GB 的数据库，1 TB 或 RAM 或只有一个用户。此用户可以用尽所有 CPU 而不会伤害任何其他人。`ALTER TABLE` 可用于否决我们刚才讨论过的内容：

```
test=# ALTER TABLE t_parallel SET (parallel_workers = 9);
ALTER TABLE
```

如果要取代 x3 规则以确定所需 CPU 的数量，可以使用 `ALTER TABLE` 显式硬化 CPU 数量。注意，`max_parallel_workers_per_gather` 仍然有效并作为上限。

如果您查看该计划，您将看到实际将考虑的核心数量：

```
test=# explain SELECT count(*) FROM t_parallel;
          QUERY PLAN
-----
-----
Finalize Aggregate (cost=146343.32..146343.33 rows=1 width=8)
```

```
-> Gather (cost=146342.39..146343.30 rows=9 width=8)
    Workers Planned: 9
        -> Partial Aggregate (cost=145342.39..145342.40 rows=1 width=8)
            -> Parallel Seq Scan on t_parallel (cost=0.00..138397.91 rows=2777791
width=0)
(5 rows)
```

但是，这并不意味着实际使用这些核心：

```
test=# explain analyze SELECT count(*) FROM t_parallel;
                                         QUERY PLAN
-----
-----
Finalize Aggregate (cost=146343.32..146343.33 rows=1 width=8)
    (actual time=1209.441..1209.441 rows=1 loops=1)
        -> Gather (cost=146342.39..146343.30 rows=9 width=8)
            (actual time=1209.432..1209.772 rows=8 loops=1)
                Workers Planned: 9
                Workers Launched: 7
                -> Partial Aggregate (cost=145342.39..145342.40 rows=1 width=8)
                    (actual time=1200.437..1200.438 rows=1 loops=8)
                        -> Parallel Seq Scan on t_parallel (cost=0.00..138397.91 ...)
                            (actual time=0.038..817.430 rows=3125000
loops=8)
Planning Time: 0.091 ms
Execution Time: 1209.827 ms
(8 rows)
```

正如您所看到的那样，只有七个核心被启动 - 尽管计划了九个。 是什么原因？ 在这个例子中，还有两个变量发挥作用：

```
test=# SHOW max_worker_processes;
max_worker_processes
-----
8
(1 row)
test=# SHOW max_parallel_workers;
max_parallel_workers
-----
8
(1 row)
```

第一个过程告诉 PostgreSQL，通常有多少个工作进程可用。 max_parallel_workers 说，有多少工人可用于并行查询。 为什么有两个参数？ 后台进程不仅被并行查询基础结构使用 - 它们也可以用于其他目的，因此开发人员需要考虑使用这两个参数。

一般来说，我们 Cybertec (<https://www.cybertec-postgresql.com>) 倾向于将 `max_worker_processes` 设置为服务器中的 CPU 数量。使用更多通常似乎是没有益处的。

PostgreSQL 可以并行执行什么操作？

正如本节已经提到的，自 PostgreSQL 9.6 以来，对并行性的支持已逐渐得到改善。在每个版本中都添加了新内容。

以下操作可以并行完成：

- 并行顺序扫描 Parallel sequential scans
- 并行索引扫描（仅限 btree）Parallel index scans (btrees only)
- 并行位图堆栈扫描 Parallel bitmap heapscans
- 并行连接（所有类型的连接）Parallel joins (all types of joins)
- 并行创建索引 Parallel indexing

在 PostgreSQL 11 中，PostgreSQL 增加了对并行索引创建的支持。

正常排序操作尚未完全并行 - 到目前为止，只能并行创建并行 btree。要控制并行度，以下参数适用：

```
test=# SHOW max_parallel_maintenance_workers;
max_parallel_maintenance_workers
-----
2
(1 row)
```

并行规则与正常操作基本相同。

如果您想加快索引创建速度，请考虑查看我的一篇关于索引创建和性能的博客文章：

<https://www.cybertec-postgresql.com/en/postgresql-parallel-create-index-for-better-performance/>

并行性的实践

在向您介绍并行性的基础知识后，我们必须检查它在现实世界中的含义。我们来看看以下查询。

```
test=# explain SELECT * FROM t_parallel;
          QUERY PLAN
-----
 Seq Scan on t_parallel (cost=0.00..360621.20 rows=25000120 width=4)
(1 row)
```

为什么 PostgreSQL 不使用并行查询？该表足够大，工作进程 PostgreSQL 可用，为什么它不使用并行查询？答案是：进程间通信非常昂贵。如果 PostgreSQL 必须在进程之

间传递行，则查询实际上可能比单进程模式慢。 优化器使用成本参数来惩罚进程间通信。

```
#parallel_tuple_cost = 0.1
```

每次在进程之间移动元组时，将在计算中添加 0.1 分。要查看 PostgreSQL 如何强制执行并行查询，我已经包含了以下示例：

```
test=# SET force_parallel_mode TO on;
SET
test=# explain SELECT * FROM t_parallel;
QUERY PLAN
-----
-
Gather (cost=1000.00..2861633.20 rows=25000120 width=4)
  Workers Planned: 1
  Single Copy: true
    -> Seq Scan on t_parallel (cost=0.00..360621.20 rows=25000120 width=4)
(4 rows)
```

如您所见，成本高于单核模式。 在现实世界中，这是一个重要的问题，因为很多人都在想为什么 PostgreSQL 会使用单核模式。

在一个真实的例子中，同样重要的是要看到更多内核不会自动导致更快的速度。 要找到完美的核心数量，需要进行微妙的平衡。

介绍 JIT 编译

JIT 编译一直是 PostgreSQL 11 中的热门话题之一。这是一项重大工作，第一个结果看起来很有希望。但是，让我们从基础知识开始：什么是 JIT 编译？当您运行查询时，PostgreSQL 必须在运行时找出很多东西。当编译 PostgreSQL 本身时，它不知道您将在下一个运行哪种查询，因此必须为各种场景做好准备。

核心的是通常来说它可以做各种各样的东西。但是，当您在执行查询时，您只想尽快执行当前查询 - 而不是其他一些随机的东西。关键是：在运行时，您比在编译时知道更多关于您必须做的事情（=编译 PostgreSQL 时）。这正是重点：当启用 JIT 编译时，PostgreSQL 将检查您的查询，如果它足够耗时，将为您的查询创建高度优化的代码（Just in Time）。

配置 JIT

为了使用 JIT，必须在编译时添加它。可以使用以下配置选项：

```
--with-llvm build with LLVM based JIT support
...
LLVM_CONFIG path to llvm-config command
```

一些 Linux 发行版附带了一个包含对 JIT 支持的额外包。如果您想使用 JIT，请确保已安装这些软件包。

一旦确定 JIT 可用，就可以使用以下配置参数来微调您的查询的 JIT 编译：

```
#jit = on                      # allow JIT compilation
#jit_provider = 'llvmjit'        # JIT implementation to use
#jit_above_cost = 100000          # perform JIT compilation if
available                         # and query more expensive, -1
disables                          # disables
#jit_optimize_above_cost = 500000 # optimize JITed functions if query
is                                # more expensive, -1 disables
#jit_inline_above_cost = 500000  # attempt to inline operators and
                                # functions if query is more
expensive,                      # -1 disables
```

`jit_above_cost` 表示仅在预期成本至少为 100.000 时才考虑 JIT。为什么这有关系？如果查询不够长，则编译的开销可能比潜在的增益高很多。因此，仅仅尝试优化。但是，还有两个参数：如果认为查询比 500.000 更昂贵，则会尝试进行深度优化。在这种情况下，函数调用将被内联并且所有这些。

此时 PostgreSQL 仅支持 LLVM 作为 JIT 后端。未来也可能会有额外的后端。目前，LLVM 做得非常好，涵盖了专业环境中使用的大多数环境。

运行查询

为了说明 JIT 如何工作，我们将编译一个简单的例子。让我们从创建一个包含大量数据的大表开始。请记住，JIT 编译仅在操作足够大时才有用。对于初学者来说，5000 万行就足够了。以下示例显示了如何填充表。

```
jit=# CREATE TABLE t_jit AS
      SELECT (random()*10000)::int AS x, (random()*100000)::int AS y,
             (random()*1000000)::int AS z
        FROM generate_series(1, 50000000) AS id;
SELECT 50000000
jit=# VACUUM ANALYZE t_jit;
VACUUM
```

在这种情况下，我们将使用随机函数来生成一些数据。为了展示 JIT 如何工作以及使执行计划更容易阅读，您可以转换并行查询。JIT 适用于并行查询，但执行计划往往会长：

```
jit=# SET max_parallel_workers_per_gather TO 0;
```

```

SET
jit=# SET jit TO off;
SET
jit=# explain (analyze, verbose) SELECT avg(z+y-pi()), avg(y-pi()),
max(x/pi())
      FROM   t_jit
      WHERE  ((y+z))>((y-x)*0.000001);

```

QUERY PLAN

```

-----
-----  

Aggregate (cost=1936901.68..1936901.69 rows=1 width=24)
(actual time=20617.425..20617.425 rows=1 loops=1)
Output: avg(((z + y))::double precision - '3.14159265358979'::double
precision),
avg((y)::double precision - '3.14159265358979'::double
precision),
max((x)::double precision / '3.14159265358979'::double
precision))
-> Seq Scan on public.t_jit  (cost=0.00..1520244.00 rows=16666307
width=12)
(actual time=0.061..15322.555 rows=50000000 loops=1)
Output: x, y, z
Filter: (((t_jit.y + t_jit.z))::numeric > (((t_jit.y -
t_jit.x))::numeric * 0.000001))
Planning Time: 0.078 ms
Execution Time: 20617.473 ms
(7 rows)

```

在这种情况下，查询花了 20 秒。



我做了一个 VACUUM 操作，以确保所有提示位等都被正确设置，以确保 jitted 查询和普通查询之间的公平比较。

让我们在启用 JIT 的情况下重复测试：

```

jit=# SET jit TO on;
SET
jit=# explain (analyze, verbose) SELECT avg(z+y-pi()), avg(y-pi()),
max(x/pi())
      FROM   t_jit
      WHERE  ((y+z))>((y-x)*0.000001);

```

QUERY PLAN

```

-----
-----  

Aggregate (cost=1936901.68..1936901.69 rows=1 width=24)

```

```
(actual time=17585.788..17585.789 rows=1 loops=1)
Output: avg(((z + y))::double precision - '3.14159265358979'::double
precision)),
          avg(((y)::double precision - '3.14159265358979'::double
precision)),
          max(((x)::double precision / '3.14159265358979'::double
precision))
-> Seq Scan on public.t_jit (cost=0.00..1520244.00 rows=16666307
width=12)
(actual time=81.991..13396.227 rows=50000000 loops=1)
Output: x, y, z
Filter: (((t_jit.y + t_jit.z))::numeric > (((t_jit.y -
t_jit.x))::numeric * 0.000001))
Planning Time: 0.135 ms
JIT:
  Functions: 5
  Options: Inlining true, Optimization true, Expressions true, Deforming
true
  Timing: Generation 2.942 ms, Inlining 15.717 ms, Optimization 40.806
ms, Emission 25.233 ms,
  Total 84.698 ms
Execution Time: 17588.851 ms
(11 rows)
```

在这种情况下，您可以看到查询比以前快 10% 以上，这已经很重要了。在某些情况下，收益可能更高。但是，请记住，重新编译代码也与一些额外的工作相关联，因此对任何类型的查询都没有意义。

小结

在本章中，讨论了许多查询优化。您已经了解了优化器以及各种内部优化，例如常量折叠，内联视图，联接等等。所有这些优化都有助于提高性能并有助于加快速度。

在介绍优化之后，第 7 章“编写存储过程”将介绍存储过程。您将看到 PostgreSQL 可用于处理用户定义代码的选项。

7. 编写存储过程

在第 6 章“优化查询以获得良好性能”中，我们学习了很多关于优化器以及系统中正在进行的优化的知识。本章将介绍存储过程以及如何高效，轻松地使用它们。您将了解存储过程的组成，可用的语言以及如何加快速度。最重要的是，您将了解 PL / pgSQL 的一些更高级的功能。

本章将介绍以下主题：

- 决定正确的语言
- 区分程序和功能
- 如何执行存储过程
- PL / pgSQL 的高级功能
- 创建扩展
- 优化以获得良好的性能
- 配置函数参数

到本章结束时，您将能够编写出色而有效的程序。

了解存储过程语言

在存储过程和函数方面，PostgreSQL 与其他数据库系统有很大不同。大多数数据库引擎强制您使用某种编程语言来编写服务器端代码。Microsoft SQL Server 提供 Transact-SQL，而 Oracle 鼓励您使用 PL / SQL。PostgreSQL 不会强迫您使用某种语言，但允许您决定您最了解的内容以及您最喜欢的内容。

PostgreSQL 如此灵活的原因实际上在历史意义上也很有趣。许多年前，最著名的 PostgreSQL 开发人员之一 Jan Wieck 早期曾写过无数补丁，他提出了使用 TCL 作为服务器端编程语言的想法。麻烦的是，没有人想要使用 TCL，没有人想在数据库引擎中拥有这些东西。该问题的解决方案是使语言界面如此灵活，基本上任何语言都可以轻松地与 PostgreSQL 集成。在这个例子中，CREATE LANGUAGE 子句诞生了。以下是 CREATE LANGUAGE 的语法：

```
test=# \h CREATE LANGUAGE
Command:    CREATE LANGUAGE
Description: Define a new procedural language
Syntax:
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ]
    LANGUAGE name HANDLER call_handler
    [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```

如今，许多不同的语言可用于编写函数和存储过程。

PostgreSQL 增加的灵活性确实得到了回报；我们现在可以从丰富的编程语言中进行选择。

PostgreSQL 究竟如何处理语言？ 如果我们看一下 CREATE LANGUAGE 子句的语法，我们将观察到几个关键字：

- **HANDLER:** 这个函数实际上是 PostgreSQL 和你想要使用的任何外部语言之间的粘合剂。 它负责将 PostgreSQL 数据结构映射到语言所需的任何内容，并有助于传递代码。
- **VALIDATOR:** 这是基础设施的警察。 如果可用，它将负责向最终用户提供可口语的语法错误。 许多语言都能够在实际执行代码之前解析代码。 PostgreSQL 可以使用它，并在创建函数时告诉您函数是否正确。 不幸的是，并非所有语言都可以这样做，因此在某些情况下，您仍然会在运行时出现问题。
- **INLINE:** 如果存在，PostgreSQL 将能够使用此处理函数运行匿名代码块。

了解基础知识 - 存储过程与功能

在深入研究存储过程的解剖之前，通常重要的是讨论函数和存储过程。 术语存储过程传统上用于实际谈论的是函数。 因此，我们必须了解函数和存储过程之间的区别。

函数是普通 SQL 语句的一部分，不允许启动或提交事务。 这是一个例子：

```
SELECT func(id) FROM large_table;
```

假设 func (id) 被称为 5000 万次。 如果使用名为 commit 的函数，究竟应该发生什么？ 简单地在查询中间结束事务并启动新事务是不可能的。 将违反事务完整性，一致性等的整个概念。

相反，存储过程能够控制事务甚至一个接一个地运行多个事务。 但是，您无法在 SELECT 语句中运行它。 相反，你必须调用 CALL。 以下清单显示了 CALL 命令的语法：

```
test=# \h CALL
Command: CALL
Description: invoke a procedure
Syntax:
CALL name ( [ argument ] [, ...] )
```

因此，函数和存储过程之间存在根本区别。 您将在互联网上找到的术语并不总是很清楚。 但是，您必须意识到这些重要的差异。 在 PostgreSQL 中，函数从一开始就一直存在。 但是，如本节所述，存储过程的概念是新的，并且仅在 PostgreSQL 11 中引入。 在本章中，我们将介绍函数和存储过程。

功能的解剖

在实际挖掘特定语言之前，我们将研究典型的函数的解剖结构。 出于演示目的，让我们看看这个两个数字相加的函数：

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
```

```
SELECT $1 + $2;
'LANGUAGE 'sql';
CREATE FUNCTION
```

首先要注意的是该函数是用 SQL 编写的。PostgreSQL 需要知道我们使用的是哪种语言，因此我们必须在定义中指定它。



请注意，函数的代码作为字符串 (') 传递给 PostgreSQL。这有点值得注意，因为它允许函数成为执行机制的黑盒子。

在其他数据库引擎中，函数的代码不是字符串，而是直接附加到语句。这个简单的抽象层为 PostgreSQL 函数管理器提供了所有的功能。

在字符串中，您基本上可以使用您选择的编程语言提供的所有内容。

在这个例子中，我们简单地将两个已传递给函数的数字相加。两个整数变量被使用。这里重要的部分是 PostgreSQL 为您提供函数重载。换句话说，`mysum (int, int)` 函数与 `mysum (int8, int8)` 函数不同。

PostgreSQL 将这些东西视为两个截然不同的功能。函数重载是一个很好的特性；但是，如果参数列表不时发生变化，则必须非常小心，不要意外部署太多函数。始终确保不再需要的函数已被删除。



`CREATE OR REPLACE FUNCTION` 子句不会更改参数列表。因此，只有在签名不变的情况下才能使用它。它会出错或只是生成一个新的函数。

我们来运行这个函数：

```
test=# SELECT mysum(10, 20);
mysum
-----
30
(1 row)
```

这里的結果是 30，这并不奇怪。在介绍函数之后，重点关注下一个主题：引用。

介绍\$引用分隔符

将代码作为字符串传递给 PostgreSQL 非常灵活。但是，使用单引号可能是个问题。在许多编程语言中，单引号经常出现。为了能够使用这些引号，人们在将字符串传递给 PostgreSQL 时必须将它们转义。多年来，这一直是标准程序。幸运的是，那些旧时代已经过去，并且可以使用新方法将代码传递给 PostgreSQL。其中一个是\$分隔符，如下所示：

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
$$
    SELECT $1 + $2;
$$ LANGUAGE 'sql';
CREATE FUNCTION
```

您可以简单地使用\$\$，而不是使用引号来开始和结束字符串。 目前，有两种语言赋予了\$\$的含义。 在 Perl 和 bash 脚本中，\$\$表示进程 ID。 为了克服这个小障碍，我们可以在几乎任何事情之前使用\$来开始和结束字符串。 以下示例显示了它的工作原理：

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int) RETURNS int AS
$body$
    SELECT $1 + $2;
$body$ LANGUAGE 'sql';
CREATE FUNCTION
```

所有这些灵活性使您能够真正克服一劳永逸的引用问题。 只要起始字符串和结束字符串匹配，就不会有任何问题。

利用匿名代码块

到目前为止，我们已经编写了最简单的存储过程，并且学会了执行代码。 但是，代码执行不仅仅是完整的功能。 除了函数之外，PostgreSQL 还允许使用匿名代码块。我们的想法是运行只需要一次的代码。这种代码执行对于处理管理任务特别有用。匿名代码块不带参数，也不会永久存储在数据库中，因为它们没有名称。

这是一个简单的示例，显示了一个匿名代码块：

```
test=# DO
$$
BEGIN
    RAISE NOTICE 'current time: %', now();
END;
$$ LANGUAGE 'plpgsql';
NOTICE: current time: 2016-12-12 15:25:50.678922+01
CONTEXT: PL/pgSQL function inline_code_block line 3 at RAISE
DO
```

在此示例中，代码仅发出消息并退出。 同样，代码块必须知道它使用哪种语言。 使用简单的\$\$引用符将字符串传递给 PostgreSQL。

使用函数和事务

如您所知，PostgreSQL 在用户端公开的所有内容都是一个事务。当然，如果您正在编写

函数，这同样适用。 函数始终是您所处事务的一部分。它不是自治的，就像一个操作符或任何其他操作一样。

这里是一个例子：

```
test=# SELECT now(), mysum(id, id) FROM generate_series(1, 3) AS id;
          now           | mysum
-----+-----
2017-10-12 15:54:32.287027+01 |    2
2017-10-12 15:54:32.287027+01 |    4
2017-10-12 15:54:32.287027+01 |    6
(3 rows)
```

所有三个函数调用都发生在同一个事务中。 这很重要，因为它意味着你不能在函数内部进行太多的事务流控制。当第二个函数调用提交时会发生什么？ 它无法正常工作。

但是，Oracle 有一种允许自治事务的机制。 这个想法是，即使交易回滚，仍可能需要一些部分并且应该保留。 一个典型的例子如下：

1. 启动一个函数来查找机密数据
2. 在文档中添加一条日志行，说明有人修改了这个重要的机密数据
3. 提交日志行但回滚更改
4. 保留尝试更改数据的信息

为了解决诸如此类的问题，可以使用自治事务。 我们的想法是能够独立地在主事务中提交事务。 在这种情况下，将在回滚更改时以日志表中的条目为准。

从 PostgreSQL 11.0 开始，未实现自治事务。 但是，已经有一些补丁可以实现此功能。 可以看出这些特性何时变成核心功能。

为了让您了解事情最有可能如何运作，这里有一个基于第一个补丁的代码片段：

```
...
AS
$$
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    FOR i IN 0..9  LOOP
        START TRANSACTION;
        INSERT INTO test1 VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
```

```
RETURN 42;
END;
$$;
...
```

这个例子的目的是表明我们可以即时决定是提交还是回滚自治事务。

了解各种存储过程语言

正如前面本章所述，PostgreSQL 使您能够以各种语言编写函数和存储过程。以下选项可用并随 PostgreSQL 核心一起提供：

- SQL
- PL/pgSQL
- PL/Perl and PL/PerlU
- PL/Python
- PL/Tcl and PL/TclU

SQL 是编写函数的明显选择，应尽可能使用，因为它为优化器提供了最大的自由度。但是，如果您想编写稍微复杂的代码，PL / pgSQL 可能是您选择的语言。

PL / pgSQL 提供流程控制等等。在本章中，将展示 PL / pgSQL 的一些更高级和少为人知的功能，但请记住，本章并不是关于 PL / pgSQL 的完整教程。

核心包含服务器端运行 Perl 功能的代码。基本上，这里的逻辑是一样的。代码将作为字符串传递并由 Perl 执行。请记住，PostgreSQL 不会谈论 Perl；它只是将代码传递给外部编程语言。

也许您已经注意到 Perl 和 TCL 有两种版本：“可信的”(PL / Perl 和 PL / TCL) 和“不可信的”(PL / PerlU 和 PL / TCLU)。可信和不可信语言之间的区别实际上是一个重要的区别。在 PostgreSQL 中，语言直接加载到数据库连接中。因此，该语言能够做很多令人讨厌的事情。为了摆脱安全问题，已经发明了可信任语言的概念。我们的想法是，受信任的语言仅限于语言的核心。无法执行以下操作：

- 包括库文件
- 打开网络套接字
- 执行任何类型的系统调用，包括打开文件等

Perl 提供了一种叫做污点模式的东西，用于在 PostgreSQL 中实现这个功能。如果即将发生安全违规，Perl 会自动将自己限制为受信任模式并出错。在不受信任的模式下，一切皆有可能，因此，只允许超级用户运行不受信任的代码。

如果要运行受信任的代码和不受信任的代码，则必须激活两种语言，plperl 和 plperlU(分别为 pltcl 和 pltclu)。

Python 目前仅作为不受信任的语言提供；因此，管理员在一般安全性方面必须非常小心，

因为在不受信任模式下运行的函数可以绕过 PostgreSQL 强制执行的所有安全机制。
请记住，Python 作为数据库连接的一部分运行，并不负责安全性。

让我们开始讨论本章最期待的主题。

介绍 PL / pgSQL

在本节中，您将了解 PL / pgSQL 的一些更高级的功能，这些功能对于编写正确且高效的代码非常重要。



请注意，这不是初学者对编程或 PL / pgSQL 的入门指引。

处理引用

数据库编程中最重要的事情之一是引用。如果您没有使用正确的引用，您肯定会遇到 SQL 注入问题并打开不可令人接受的安全漏洞。

什么是 SQL 注入？让我们考虑以下示例：

```
CREATE FUNCTION broken(text) RETURNS void AS
$$
DECLARE
    v_sql text;
BEGIN
    v_sql := 'SELECT schemaname
              FROM pg_tables
             WHERE tablename = "' || $1 || '"';
    RAISE NOTICE 'v_sql: %', v_sql;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

在此示例中，SQL 代码只是粘贴在一起，而不必担心安全性。我们在这里所做的就是使用 `||` 运算符以连接字符串。如果人们运行普通查询，这可以正常工作。请考虑以下示例显示一些破坏的代码：

```
SELECT broken('t_test');
```

但是，我们必须为尝试利用您的系统的人做好准备。请考虑以下示例：

```
SELECT broken(""); DROP TABLE t_test; ';
```

使用此参数运行该函数将显示问题。下一个清单显示了经典的 SQL 注入：

```
NOTICE: v_sql: SELECT schemaname FROM pg_tables
WHERE tablename = ""; DROP TABLE t_test; '
CONTEXT: PL/pgSQL function broken(text) line 6 at RAISE
```

```
broken
```

```
-----  
(1 row)
```

当您只想进行查找时删除表是不可以接受的事情。使应用程序的安全性取决于传递给语句的参数绝对是不可接受的。

为了避免 SQL 注入，PostgreSQL 提供了各种功能；这些功能应始终使用，以确保应用安全保持完整：

```
test=# SELECT quote_literal(E'o\"reilly'), quote_ident(E'o\"reilly');  
quote_literal | quote_ident  
-----+-----  
'o"reilly' | "o'reilly" (1 row)
```

`quote_literal` 函数将以一种不会发生任何坏事的方式转义字符串。它将在字符串周围添加所有引号，并在字符串中转义有问题的字符。因此，不再需要手动启动和结束字符串。

这里显示的第二个函数是 `quote_ident`。它可用于正确引用对象名称。请注意，使用双引号，这正是处理表名所需的。下一个示例显示了如何使用复杂名称：

```
test=# CREATE TABLE "Some stupid name" ("ID" int);  
CREATE TABLE  
test=# \d "Some stupid name" Table "public.Some stupid name"  
Column | Type | Modifiers  
-----+-----+  
ID | integer |
```

通常，PostgreSQL 中的所有表名都是小写的。但是，如果使用双引号，则对象名称可以包含大写字母。一般来说，使用这种技巧并不是一个好主意，因为你必须一直使用双引号，这可能有点不方便。

在对引用进行基本介绍之后，重要的是要看一下如何处理 `NULL` 值。下一个清单显示了 `quote_literal` 函数如何处理 `null`：

```
test=# SELECT quote_literal(NULL);  
quote_literal  
-----  
(1 row)
```

如果在 `NULL` 值上调用 `quote_literal` 函数，它将只返回 `NULL`。在这种情况下，没有必要关注字符串引用。

PostgreSQL 提供了更多的函数来显式处理 `NULL` 值：

```
test=# SELECT quote_nullable(123), quote_nullable(NULL);  
quote_nullable | quote_nullable
```

```
-----+-----  
'123' | NULL (1 row)
```

它不仅可以引用字符串和对象名称，还可以使用系统自带的 PL / pgSQL 来格式化和准备整个查询。这里的美妙之处在于您可以使用 `format` 函数将参数添加到语句中。下面是它的工作方式：

```
CREATE FUNCTION simple_format() RETURNS text AS  
$$  
DECLARE  
    v_string text;  
    v_result text;  
BEGIN  
    v_string := format('SELECT schemaname || '.' || tablename  
                      FROM pg_tables  
                      WHERE %I = $1  
                            AND %I = $2', 'schemaname', 'tablename');  
    EXECUTE v_string USING 'public', 't_test' INTO v_result;  
    RAISE NOTICE 'result: %', v_result;  
    RETURN v_string;  
END;  
$$ LANGUAGE 'plpgsql';
```

字段的名称将传递给 `format` 函数。最后，`EXECUTE` 语句的 `USING` 子句用于将参数添加到查询中，然后执行该参数。同样，这里的美妙之处在于不会发生 SQL 注入。

以下是调用 `simple_format` 函数时发生的情况：

```
test=# SELECT simple_format();  
NOTICE: result: public.t_test  
          simple_format  
-----  
          SELECT schemaname || '.' || tablename      +  
          FROM pg_tables      +  
          WHERE schemaname = $1+  
                AND tablename = $2  
(1 row)
```

如您所见，调试消息正确得显示了表，包括表的模式，并正确返回了查询。

管理范围

在处理引用和基本安全（SQL 注入）之后，我们将重点关注另一个重要主题：范围。

就像大多数流行的编程语言一样，PL / pgSQL 根据其上下文使用变量。变量在函数的

DECLARE 语句中定义。但是，PL / pgSQL 允许您嵌套 DECLARE 语句：

```
CREATE FUNCTION scope_test () RETURNS int AS
$$
DECLARE
    i int := 0;
BEGIN
    RAISE NOTICE 'i1: %', i;
    DECLARE
        i int;
    BEGIN
        RAISE NOTICE 'i2: %', i;
    END;
    RETURN i;
END;
$$ LANGUAGE 'plpgsql';
```

在 DECLARE 语句中，定义了变量 i 并为其分配了值。然后，显示 i。当然了输出会是 0。然后，第二个 DECLARE 语句启动。它包含另一个变量 i，未赋值。因此，该值将为 NULL。请注意，PostgreSQL 现在将显示内部 i。这是发生的事情：

```
test=# SELECT scope_test();
NOTICE:  i1: 0
NOTICE:  i2: <NULL>
scope_test
-----
0
(1 row)
```

正如预期的那样，调试消息将显示 0 和 NULL。



PostgreSQL 允许您使用各种技巧。但是，强烈建议您保持代码简单易读。

了解高级错误处理

对于编程语言，在每个程序和每个模块中，错误处理是一件重要的事情。预计一切都会出错，因此正确和专业地处理错误至关重要。在 PL / pgSQL 中，您可以使用 EXCEPTION 块来处理错误。我们的想法是，如果 BEGIN 块出错了，EXCEPTION 块将处理它并正确处理问题。就像许多其他语言（如 Java）一样，您可以对不同类型的错误做出反应并单独捕获它们。

在以下示例中，代码可能会遇到除零问题。目标是捕获此错误并做出相应的反应：

```
CREATE FUNCTION error_test1(int, int) RETURNS int AS
$$
BEGIN
```

```
RAISE NOTICE 'debug message: % / %', $1, $2;
BEGIN
    RETURN $1 / $2;
EXCEPTION
WHEN division_by_zero THEN
    RAISE NOTICE 'division by zero detected: %', sqlerrm;
WHEN others THEN
    RAISE NOTICE 'some other error: %', sqlerrm;
END;
RAISE NOTICE 'all errors handled';
RETURN 0;
END;
$$ LANGUAGE 'plpgsql';
```

BEGIN 块可以清楚地抛出错误，因为这里可能会产生除以零的情况。但是，EXCEPTION 块会捕获我们正在查找的错误，并且还会处理可能意外弹出的所有其他潜在问题。

从技术上讲，这与保存点或多或少相同，因此错误不会导致整个事务完全失败。只有导致错误的块才会进行小回滚。

通过检查 `sqlerrm` 变量，您还可以直接访问错误消息本身。我们运行如下代码：

```
test=# SELECT error_test1(9, 0);
NOTICE: debug message: 9 / 0
NOTICE: division by zero detected: division by zero
NOTICE: all errors handled
error_test1
-----
0
(1 row)
```

PostgreSQL 捕获异常并在 EXCEPTION 块中显示消息。很高兴能告诉我们错误所在。如果代码被破坏，这使得调试和修复代码变得更加容易。

在某些情况下，抛出自己定义的异常也是有意义的。正如人们所料，这很容易做到：

```
RAISE uniqueViolation
USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

除此之外，PostgreSQL 提供了许多预定义的错误代码和异常。以下页面包含这些错误消息的完整列表：<https://www.postgresql.org/docs/10/static/errcodes-appendix.html>。

使用 GET DIAGNOSTICS

许多过去使用过 Oracle 的人可能熟悉 GET DIAGNOSTICS 条款。GET DIAGNOSTICS 子句

背后的想法是允许用户查看系统中发生的情况。虽然对于习惯于现代代码的人来说，语法可能显得有点奇怪，但它仍然是一个有价值的工具，可以使您的应用程序执行得更好。

从我的观点来看，`GET DIAGNOSTICS` 子句有两个主要任务可用于：

- 检查行数
- 获取上下文信息并获取回溯

检查行数绝对是您在日常编程中需要的。如果要调试应用程序，则提取上下文信息将非常有用。

以下示例显示了如何在代码中使用`GET DIAGNOSTICS`子句：

```
CREATE FUNCTION get_diag() RETURNS int AS
$$
DECLARE
    rc    int;
    _sqlstate text;
    _message text;
    _context text;
BEGIN
    EXECUTE 'SELECT * FROM generate_series(1, 10)';
    GET DIAGNOSTICS rc = ROW_COUNT;
    RAISE NOTICE 'row count: %', rc;
    SELECT rc / 0;
EXCEPTION
    WHEN OTHERS THEN
        GET STACKED DIAGNOSTICS
            _sqlstate = returned_sqlstate,
            _message = message_text,
            _context = pg_exception_context;
        RAISE NOTICE 'sqlstate: %, message: %, context: [%]',
            _sqlstate,
            _message,
            replace(_context, E'n', ' <- ');
RETURN rc;
END;
$$ LANGUAGE 'plpgsql';
```

声明这些变量后的第一件事是执行 SQL 语句并向`GET DIAGNOSTICS`子句获取行计数，然后将其显示在调试消息中。然后，该功能强制 PL / pgSQL 出错。一旦发生这种情况，我们将使用`GET DIAGNOSTICS`子句从服务器中提取信息以显示它。

以下是调用`get_diag`函数时发生的情况：

```
test=# SELECT get_diag();
NOTICE:  row count: 10
```

```
CONTEXT: PL/pgSQL function get_diag() line 12 at RAISE
NOTICE: sqlstate: 22012,
message: division by zero,
context: [SQL statement "SELECT rc / 0"
<- PL/pgSQL function get_diag() line 14 at
SQL statement]
CONTEXT: PL/pgSQL function get_diag() line 22 at RAISE
get_diag
-----
10
(1 row)
```

如您所见，`GET DIAGNOSTICS` 子句为我们提供了有关系统中活动的非常详细的信息。

使用游标以块的形式获取数据

如果执行 SQL，数据库将计算结果并将其发送到您的应用程序。将整个结果集发送到客户端后，应用程序可以继续完成其工作。问题是这样的：如果结果集太大而不再适合内存，会发生什么？如果数据库返回 100 亿行怎么办？客户端应用程序通常不能一次处理这么多数据，实际上它不应该这样做。该问题的解决方案是游标。游标背后的思想是，只有在需要数据时（调用 `fetch` 时）才会生成数据。因此，应用程序可以在实际由数据库生成数据时开始使用数据。最重要的是，执行操作所需的内存要低得多。

说到 PL / pgSQL，游标也起着重要作用。无论何时循环结果集，`postgresql` 都会自动在内部使用游标。这样做的好处是，应用程序的内存消耗将大大减少，而且由于处理大量数据，几乎不会出现内存耗尽的情况。使用游标的方法有很多种。以下是函数内部游标的最简单示例：

```
CREATE OR REPLACE FUNCTION c(int)
RETURNS setof text AS
$$
DECLARE
v_rec record;
BEGIN
FOR v_rec IN SELECT tablename
    FROM pg_tables
    LIMIT $1
LOOP
    RETURN NEXT v_rec.tablename;
END LOOP;
RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

这段代码之所以有趣有两个原因。首先，它是一个集合返回函数（srf）。它产生一整列而不仅仅是一行。实现这一点的方法是使用变量集而不仅仅是数据类型。`return next` 子句将建立结果集，直到我们到达结尾。`return` 子句将告诉 `postgresql` 我们想要离开函数，结果已经完成。

第二个重要的问题是在游标上循环将自动创建一个内部游标。换言之，不必担心可能会耗尽内存。PostgreSQL 将优化查询，使其尽可能快地生成前 10% 的数据（由游标元组分组变量定义）。以下是查询将返回的内容：

```
test=# SELECT * FROM c(3);
c
-----
t_test
pg_statistic
pg_type
(3 rows)
```

在这个例子中，只有一个随机表列表。如果与你这边的结果不一样，这是意料之中的。

在我看来，您刚才看到的是在 PL / pgSQL 中使用隐式游标的最常见的方法。下面的示例显示了许多 Oracle 人员可能知道的较旧机制：

```
CREATE OR REPLACE FUNCTION d(int)
RETURNS setof text AS
$$
DECLARE
v_cur refcursor;
v_data text;
BEGIN
OPEN v_cur FOR
SELECT tablename
FROM pg_tables
LIMIT $1;
WHILE true LOOP
FETCH v_cur INTO v_data;
IF FOUND THEN
RETURN NEXT v_data;
ELSE
RETURN;
END IF;
END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

在此示例中，显式声明并打开游标。在内部，然后显式地获取循环数据并将其返回给调用者。基本上，查询完全相同。语法开发人员实际上更喜欢哪种语法，这只是一个

品味问题。

你还觉得你对游标知之甚少吗？还有更多；这里有第三个选择来做完全相同的事情：

```
CREATE OR REPLACE FUNCTION e(int)
RETURNS setof text AS
$$
DECLARE
v_cur CURSOR (param1 int) FOR
    SELECT tablename
    FROM pg_tables
    LIMIT param1;
v_data text;
BEGIN
OPEN v_cur ($1);
WHILE true LOOP
    FETCH v_cur INTO v_data;
    IF FOUND THEN
        RETURN NEXT v_data;
    ELSE
        RETURN;
    END IF;
END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

在这种情况下，光标会输入一个整数参数，该参数直接来自函数调用（\$ 1）。

有时，存储过程本身不会使用游标，但会返回以供以后使用。在这种情况下，您可以返回一个简单的使用 refcursor 作为返回值：

```
CREATE OR REPLACE FUNCTION cursor_test(c refcursor)
RETURNS refcursor AS
$$
BEGIN
OPEN c FOR SELECT *
    FROM generate_series(1, 10) AS id;
RETURN c;
END;
$$ LANGUAGE plpgsql;
```

这里的逻辑非常简单。游标的名称将传递给该函数。然后，打开并返回游标。这里的美妙之处在于游标后面的查询可以动态创建并动态编译。

应用程序可以从任何其他应用程序中获取游标。下面是它的工作方式：

```
test=# BEGIN;
```

```
BEGIN
test=# SELECT cursor_test('mytest');
cursor_test
-----
mytest
(1 row)
test=# FETCH NEXT FROM mytest;
id
-----
1
(1 row)
test=# FETCH NEXT FROM mytest;
id
-----
2
(1 row)
```

请注意，它仅在使用事务块时有效。

在本节中，我们了解到游标只会在消耗时生成数据。这适用于大多数查询。但是，这个例子有一个问题；无论何时使用 SRF，都必须实现整个结果。它不是即时创建的，而是立即创建的。原因是 SQL 必须能够重新扫描关系，这在普通表的情况下很容易实现。但是，对于函数，情况则不同。因此，SRF 总是被计算和具体化，使得此示例中的游标完全无用。换句话说，我们在编写函数时需要小心。在某些情况下，危险隐藏在漂亮的细节中。

使用复合类型

在大多数其他数据库系统中，存储过程仅用于原始数据类型，例如整数，数字，varchar 等。然而，PostgreSQL 却大不相同。我们可以使用所有可用的数据类型。这包括原始数据、复合数据和自定义数据类型。就数据类型而言，没有任何限制。为了充分发挥 PostgreSQL 的威力，复合类型非常重要，并且经常被互联网上的扩展使用。

以下示例显示如何将复合类型传递给函数以及如何在内部使用它。最后，将再次返回复合类型：

```
CREATE TYPE my_cool_type AS (s text, t text);
CREATE FUNCTION f(my_cool_type)
RETURNS my_cool_type AS
$$
DECLARE
v_row my_cool_type;
BEGIN
RAISE NOTICE 'schema: (%) / table: (%)'
```

```
, $1.s, $1.t;
SELECT schemaname, tablename
INTO v_row
FROM pg_tables
WHERE tablename = trim($1.t)
    AND schemaname = trim($1.s)
LIMIT 1;
RETURN v_row;
END;
$$ LANGUAGE 'plpgsql';
```

这里的主要问题是，您可以简单地使用`$1.field_name` 来访问复合类型。返回复合类型也不难。您只需动态地组装复合类型变量并返回它，就像任何其他数据类型一样。您甚至可以轻松地使用数组或更复杂的结构。

以下清单显示了 PostgreSQL 将返回的内容：

```
test=# SELECT (f).s, (f).t
  FROM f ('("public", "t_test")'::my_cool_type);
NOTICE:  schema: (public) / table: ( t_test )
          s      |   t
-----+-----
        public | t_test
(1 row)
```

在 PL / pgSQL 中编写触发器

如果您想对数据库中发生的某些事件做出反应，服务器端代码尤其流行。触发器允许您在表上发生 INSERT、UPDATE、DELETE 或 TRUNCATE 子句时调用函数。然后，触发器调用的函数可以修改表中更改的数据，或者只是执行一些必要的操作。

在 PostgreSQL 中，触发器多年来变得更加强大，现在提供了丰富的功能：

```
test=# \h CREATE TRIGGER
Command: CREATE TRIGGER
Description: define a new trigger
Syntax:
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event
[ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY
DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [
... ] ]
```

```
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( condition ) ]
EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:
INSERT
UPDATE [ OF column_name [, ... ] ]
DELETE
TRUNCATE
```

首先要注意的是，触发器总是为表或视图触发并调用函数。它有一个名称，可以在事件之前或之后发生。PostgreSQL 的美妙之处在于，在一个表上可以有无数个触发器。虽然这对于铁杆 PostgreSQL 用户来说并不奇怪，但我想指出的是，在世界各地仍在使用的许多昂贵的商业数据库引擎中，这是不可能的。

如果同一个表上有多个触发器，那么多年前在 PostgreSQL 7.3 中引入了以下规则：触发器按字母顺序触发。首先，所有那些 BEFORE 触发按字母顺序发生。然后，postgresql 执行触发的行操作，并按字母顺序在触发之后继续执行。换句话说，触发器的执行顺序是绝对确定的，并且触发器的数量基本上是无限的。

触发器可以在实际修改发生之前或之后修改数据。通常，这是验证数据和在违反某些自定义限制时出错的好方法。以下示例显示了在 INSERT 子句中触发并更改添加到表中的数据的触发器：

```
CREATE TABLE t_sensor (
    id serial,
    ts timestamp,
    temperature numeric
);
```

我们的表只存储了几个值。现在的目标是在插入行时立即调用函数：

```
CREATE OR REPLACE FUNCTION trig_func()
RETURNS trigger AS
$$
BEGIN
    IF NEW.temperature < -273
    THEN
        NEW.temperature := 0;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

如前所述，触发器将始终调用一个函数，这允许您很好地抽象代码。这里重要的是触发器功能必须返回一个触发器。要访问要插入的行，可以访问 NEW 变量。



TIP INSERT 和 UPDATE 触发器始终提供 NEW 变量。UPDATE 和 DELETE 将提供一个名为 OLD 的变量。这些变量包含您要修改的行。

在我的示例中，代码检查温度是否过低。如果是，则值不合适；它是动态调整的。为确保可以使用修改的行，只需返回 NEW。如果在此之后调用了第二个触发器，则下一个函数调用将已经看到修改的行。

在下一步中，可以使用 CREATE TRIGGER 命令创建触发器：

```
CREATE TRIGGER sensor_trig
  BEFORE INSERT ON t_sensor
  FOR EACH ROW
  EXECUTE PROCEDURE trig_func();
```

以下是触发器的作用：

```
test=# INSERT INTO t_sensor (ts, temperature)
      VALUES ('2017-05-04 14:43', -300) RETURNING *;
          id | ts           | temperature
-----+-----+-----
        1  | 2017-05-04 14:43:00 |    0
(1 row)

INSERT 0 1
```

如您所见，该值已正确调整。表格的内容显示温度为 0。

如果您正在使用触发器，您应该意识到触发器对自身了解很多。它可以访问几个变量，允许您编写更复杂的代码并实现更好的抽象。

让我们先放下触发器：

```
test=# DROP TRIGGER sensor_trig ON t_sensor;
DROP TRIGGER
```

然后，可以添加一个新函数：

```
CREATE OR REPLACE FUNCTION trig_demo()
  RETURNS trigger AS
$$
BEGIN
  RAISE NOTICE 'TG_NAME: %', TG_NAME;
  RAISE NOTICE 'TG_RELNAME: %', TG_RELNAME;
  RAISE NOTICE 'TG_TABLE_SCHEMA: %', TG_TABLE_SCHEMA;
  RAISE NOTICE 'TG_TABLE_NAME: %', TG_TABLE_NAME;
  RAISE NOTICE 'TG_WHEN: %', TG_WHEN;
  RAISE NOTICE 'TG_LEVEL: %', TG_LEVEL;
  RAISE NOTICE 'TG_OP: %', TG_OP;
```

```
RAISE NOTICE 'TG_NARGS: %', TG_NARGS;
-- RAISE NOTICE 'TG_ARGV: %', TG_NAME;
RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
CREATE TRIGGER sensor_trig
BEFORE INSERT ON t_sensor
FOR EACH ROW
EXECUTE PROCEDURE trig_demo();
```

此处使用的所有变量都是预定义的， 默认情况下可用。 我们所有的代码都显示它们，以便我们可以看到内容：

```
test=# INSERT INTO t_sensor (ts, temperature)
          VALUES ('2017-05-04 14:43', -300) RETURNING *;
NOTICE: TG_NAME: demo_trigger
NOTICE: TG_RELNAME: t_sensor
NOTICE: TG_TABLE_SCHEMA: public
NOTICE: TG_TABLE_NAME: t_sensor
NOTICE: TG_WHEN: BEFORE
NOTICE: TG_LEVEL: ROW
NOTICE: TG_OP: INSERT
NOTICE: TG_NARGS: 0
      id | ts           | temperature
-----+-----+
      2  | 2017-05-04 14:43:00 | -300
(1 row)
INSERT 0 1
```

我们在这里看到的是触发器知道它的名字， 它已被触发的表， 以及更多。 要对各种表应用类似的操作， 这些变量只需编写一个函数就可以避免重复代码。 然后， 这可以用于我们感兴趣的所有表。

到目前为止， 我们已经看到了简单的行级触发器， 每个语句触发一次。 但是， 随着 PostgreSQL 10.0 的推出， 还有一些新功能。 语句级触发器已经存在了一段时间。 但是， 无法访问触发器更改的数据。 这已在 PostgreSQL 10.0 中得到修复， 现在可以使用转换表， 其中包含所做的所有更改。

以下清单包含一个完整示例， 说明如何使用转换表：

```
CREATE OR REPLACE FUNCTION transition_trigger()
RETURNS TRIGGER AS $$

DECLARE
v_record record;
BEGIN
IF (TG_OP = 'INSERT') THEN
```

```
RAISE NOTICE 'new data: ';
FOR v_record IN SELECT * FROM new_table
LOOP
    RAISE NOTICE '%', v_record;
END LOOP;
ELSE
    RAISE NOTICE 'old data: ';
FOR v_record IN SELECT * FROM old_table
LOOP
    RAISE NOTICE '%', v_record;
END LOOP;
END IF;
RETURN NULL; -- result is ignored since this is an AFTER trigger
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER transition_test_trigger_ins
AFTER INSERT ON t_sensor
REFERENCING NEW TABLE AS new_table
FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();
CREATE TRIGGER transition_test_trigger_del
AFTER DELETE ON t_sensor
REFERENCING OLD TABLE AS old_table
FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();
```

在这种情况下，我们需要两个触发器定义，因为我们不能只将一切都压缩到一个定义中。在触发器功能内部，转换表易于使用：它可以像普通表一样访问。

让我们通过插入一些数据来测试触发器的代码：

```
INSERT INTO t_sensor
SELECT  *, now(), random() * 20
FROM generate_series(1, 5);
DELETE FROM t_sensor;
```

在我的示例中，代码将仅为转换表中的每个条目发出 NOTICE：

```
NOTICE: new data:
NOTICE: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
NOTICE: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
NOTICE: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
NOTICE: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
NOTICE: (5,"2017-10-04 15:47:14.129151",10.9121138229966)
INSERT 0 5
NOTICE: old data:
NOTICE: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
NOTICE: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
```

```
NOTICE: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
NOTICE: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
NOTICE: (5,"2017-10-04 15:47:14.129151",10.9121138229966)
DELETE 5
```

请记住，将转换表用于数十亿行不一定是个好主意。 PostgreSQL 实际上是可扩展的，但在某些时候，有必要看到它也有性能影响。

在 PL / pgSQL 中编写存储过程

现在让我们继续了解如何编写存储程序。 在本节中，您将学习编写 PostgreSQL 11 中引入的实际存储过程。要创建过程，您必须使用 `CREATE PROCEDURE`。此命令的语法与 `CREATE FUNCTION` 非常相似。只有很小的差异：

```
test=# \h CREATE PROCEDURE
Command: CREATE PROCEDURE
Description: define a new procedure
Syntax:
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
[,...] ]
    { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    } ...
```

以下示例显示了一个存储过程，该过程运行两个事务。首先事务将提交，因此创建两个表。第二个语句存储过程将回滚：

```
test=# CREATE PROCEDURE test_proc()
        LANGUAGE plpgsql
AS $$

BEGIN
    CREATE TABLE a (aid int);
    CREATE TABLE b (bid int);
    COMMIT;
    CREATE TABLE c (cid int);
    ROLLBACK;

END;
$$;
CREATE PROCEDURE
```

如所观察到的，存储过程能够进行显式事务处理。存储过程背后的想法是能够运行批

处理作业和其他操作，这在函数中很难实现。

要运行该过程，必须使用 CALL，如以下示例所示：

```
test=# CALL test_proc();
CALL
```

前两个表由于提交已经创建。 由于存储过程内部的回滚，尚未创建第三个表。

```
test=# \d
List of relations
 Schema | Name | Type  | Owner
-----+-----+-----+
 public | a    | table | hs
 public | b    | table | hs
(2 rows)
```

存储过程是 PostgreSQL 11 中引入的最重要的功能之一，它将为软件开发的效率做出重大贡献。

介绍 PL / Perl

关于 PL / pgSQL 还有很多话要说。 然而，并非所有内容都可以在一本书中涵盖，现在是时候进入下一个程序语言了。PL / Perl 已被许多人采用作为字符串运算的理想语言。您可能知道，Perl 以其字符串处理功能而闻名，因此在这些年里仍然相当受欢迎。

要启用 PL / Perl，您有两种选择：

```
test=# create extension plperl;
CREATE EXTENSION
test=# create extension plperlu;
CREATE EXTENSION
```

您可以部署受信任或不受信任的 Perl。 如果您两者都想要，则必须启用这两种语言。

为了向您展示 PL / Perl 的工作原理，我实现了一个简单地解析电子邮件地址并返回 true 或 false 的函数。 下面是它的工作原理：

```
test=# CREATE OR REPLACE FUNCTION verify_email(text)
RETURNS boolean AS
$$
if  ($_[0] =~ /^[a-zA-Z.]+@[a-zA-Z.-]+\$/)
{
    return true;
}
return false;
$$ LANGUAGE 'plperl'; CREATE FUNCTION
```

文本参数传递给函数。在函数内部，可以使用\$ _ 访问所有这些输入参数。在此示例中，执行正则表达式并返回值。

可以像使用任何其他语言编写的任何其他过程一样调用该函数：

```
test=# SELECT verify_email('hs@cybertec.at');
verify_email
-----
T
(1 row)
test=# SELECT verify_email('totally wrong');
verify_email
-----
f
(1 row)
```

请记住，如果您位于受信任的功能中，则无法加载包等。

例如，如果要使用 w 命令查找单词，Perl 将在内部加载 utf8.pm，当然，这是不允许的。

使用 PL / Perl 进行数据类型抽象

如本章所述，PostgreSQL 中的函数非常通用，可以在许多不同的上下文中使用。如果要使用函数来提高数据质量，可以使用 CREATE DOMAIN 子句：

```
test=# \h CREATE DOMAIN
Command: CREATE DOMAIN
Description: define a new domain
Syntax:
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
    where constraint is:
        [ CONSTRAINT constraint_name ]
        { NOT NULL | NULL | CHECK (expression) }
```

在此示例中，PL / Perl 函数将用于创建名为 email 的域，该域又可用作数据类型。

以下清单显示了如何创建域：

```
test=# CREATE DOMAIN email AS text
          CHECK (verify_email(VALUE) = true);
CREATE DOMAIN
```

如前所述，域的功能与普通数据类型相同：

```
test=# CREATE TABLE t_email (id serial, data email);
```

CREATE TABLE

Perl 函数确保不会向数据库中插入任何违反我们检查的内容，如下例所示：

```
test=# INSERT INTO t_email (data)
          VALUES ('somewhere@example.com');

INSERT 0 1
test=# INSERT INTO  t_email (data)
          VALUES ('somewhere_wrong_example.com');
ERROR:  value  for domain email  violates check      constraint
"email_check"
```

Perl 可能是一个很好的字符串处理选项，但是，和往常一样，您必须决定是否将此代码直接放入数据库。

决定选择 PL / Perl 还是 PL / PerlU

到目前为止，Perl 代码没有打开任何与安全相关的问题，因为我们所做的只是正则表达式。这里的问题是，如果有人试图在 Perl 函数中做一些令人讨厌的事情怎么办？如前所述，PL / Perl 只会出错：

```
test=# CREATE OR REPLACE FUNCTION test_security()
RETURNS boolean AS
$$
use strict;
my $fp = open("/etc/password", "r");
return false;
$$ LANGUAGE 'plperl';
ERROR:  'open' trapped by operation mask  at line
CONTEXT:  compilation of PL/Perl function "test_security"
```

一旦您尝试创建该功能，PL / Perl 就会抱怨。将立即显示错误。

如果你真的想在 Perl 中运行不受信任的代码，你必须使用 PL / PerlU：

```
test=# CREATE OR REPLACE FUNCTION first_line()
RETURNS text AS
$$
open(my $fh, '<:encoding(UTF-8)', "/etc/passwd")
or elog(NOTICE, "Could not open  file  '$filename' $!");
my $row  = <$fh>;
close($fh);
return $row;
$$ LANGUAGE 'plperlu';
CREATE FUNCTION
```

程序保持不变。 它返回一个字符串。 但是，它可以做任何事情。唯一的区别是该功能被标记为 plperlu。

结果多少不足为奇：

```
test=# SELECT first_line();
first_line
-----
root:x:0:0:root:/root:/bin/bash+
(1 row)
```

使用 SPI 接口

偶尔，你的 Perl 程序必须做数据库工作。 请记住，该函数是数据库连接的一部分。 因此，实际创建数据库连接毫无意义。 为了与数据库通信，PostgreSQL 服务器基础结构提供了 SPI 接口，这是一个与数据库内部通信的 C 接口。 所有帮助您运行服务器端代码的过程语言都使用此接口向您公开功能。 PL / Perl 也是如此，在本节中，您将学习如何在 SPI 接口周围使用 Perl 包装器。

您可能需要做的最重要的事情是运行 SQL 并检索获取的行数。 `spi_exec_query` 函数正是为了实现这一点。传递给函数的第一个参数是查询。第二个参数包含实际要检索的行数。出于简单的原因，我决定把它们都拿来。下面的列表有一个示例：

```
test=# CREATE OR REPLACE FUNCTION spi_sample(int)
RETURNS void AS
$$
my $rv = spi_exec_query(" SELECT *
FROM generate_series(1, $_[0])", $_[0]
);
elog(NOTICE, "rows fetched: " . $rv->{processed});
elog(NOTICE, "status: " . $rv->{status});
return;
$$ LANGUAGE 'plperl';
```

SPI 将执行查询并显示行数。这里重要的是所有存储过程语言都提供了发送日志消息的方法。 在 PL / Perl 的情况下，此函数称为 `elog` 并采用两个参数。第一个定义消息的重要性（INFO, NOTICE, WARNING, ERROR 等），第二个参数包含实际消息。

```
test=# SELECT spi_sample(9);
NOTICE: rows fetched: 9
NOTICE: status: SPI_OK_SELECT
spi_sample
-----
(1 row)
```

使用 SPI 实现集合返回函数

在许多情况下，您不只是想执行一些 SQL 而忘记它。在大多数情况下，一个过程会循环遍历结果并对其进行处理。下面的示例将演示如何循环查询的输出。除此之外，我还决定对示例进行一些改进，使函数返回复合数据类型。在 Perl 中使用复合类型非常简单，因为您可以简单地将数据填充到哈希列表中并返回它。

`return_next` 函数将逐渐构建结果集，直到函数以简单的 `return` 语句终止。

此清单中的示例生成一个由随机值组成的表：

```
CREATE TYPE random_type AS (a float8, b float8);
CREATE OR REPLACE FUNCTION spi_srf_perl(int)
RETURNS setof random_type AS
$$
my $rv = spi_query("SELECT random() AS a,
                           random() AS b
                      FROM generate_series(1, $_[0])");
while (defined (my $row = spi_fetchrow($rv)))
{
    elog(NOTICE, "data: ".
          $row->{a} . " / " . $row->{b});
    return_next({a => $row->{a},
                 b => $row->{b}});
}
return;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

首先，执行 `spi_query` 函数并启动使用 `spi_fetchrow` 函数的循环。在循环内部，复合类型将被组装并填充到结果集中。

正如所料，该函数将返回一组随机值：

```
test=# SELECT * FROM spi_srf_perl(3);
NOTICE: data: 0.154673356097192 / 0.278830723837018
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.61588888947666 / 0.632620786316693
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.910436692181975 / 0.753427186980844
CONTEXT: PL/Perl function "spi_srf_perl"
a_col | b_col
-----+
0.154673356097192 | 0.278830723837018
0.61588888947666 | 0.632620786316693
```

0.910436692181975 | 0.753427186980844
(3 rows)

请记住，必须实现集合返回函数，以便将整个结果集存储在内存中。

在 PL / Perl 中转义和支持的函数

到目前为止，我们只使用了整数，因此 SQL 注入或特殊表名不是问题。基本上，可以使用以下功能：

- `quote_literal`: 这将返回字符串引号作为字符串文本
- `quote_nullable`: 这引用了一个字符串
- `quote_ident`: 它引用 SQL 标识符（对象名等）
- `decode_bytea`: 这将解码 PostgreSQL 字节数组字段
- `encode_bytea`: 这将对数据进行编码并将其转换为字节数组
- `encode_literal_array`: 这将编码一个文本数组
- `encode_typed_literal`: 这将 perl 变量转换为作为第二个参数传递的数据类型的值，并返回该值的字符串表示形式
- `encode_array_constructor`: 这将以数组构造函数格式将引用数组的内容作为字符串返回
- `looks_like_number`: 如果字符串看起来像数字，则返回 `true`
- `is_array_ref`: 如果是数组引用，则返回 `true`

这些功能始终可用，可以直接调用，无需包含任何库。

跨函数调用共享数据

有时，需要在调用之间共享数据。基础设施有办法做到这一点。在 Perl 中，可以使用哈希列表来存储所需的任何数据。请看以下示例：

```
CREATE FUNCTION perl_shared(text) RETURNS int AS
$$
if ( !defined ${$_[0]} )
{
    ${$_[0]} = 0;
}
else
{
    ${$_[0]}++;
}
return ${$_[0]};
$$ LANGUAGE 'plperl';
```

一旦我们发现传递给函数的键还没有，`$_[0]` 变量将初始化为 0。对于其他每一个调用，计数器中都会添加 1，这样我们就可以得到以下输出：

```
test=# SELECT perl_shared('some_key') FROM generate_series(1, 3);
perl_shared
-----
 0
 1
 2
(3 rows)
```

对于更复杂的语句，开发人员通常不知道函数的调用顺序。记住这一点很重要。在大多数情况下，不能依赖执行顺序。

用 Perl 编写触发器

PostgreSQL 核心附带的每一种存储过程语言都允许您用这种语言编写触发器。当然，Perl 也是如此。由于本章篇幅有限，我决定不包括用 Perl 编写的触发器示例，而是将您引向 PostgreSQL 官方文档：<https://www.postgresql.org/docs/10/static/plperl-triggers.html>。

基本上，在 Perl 中编写触发器与在 PL / pgSQL 中编写触发器没有什么不同。所有预定义变量都已到位，就返回值而言，规则适用于每种存储过程语言。

介绍 PL / Python

如果您不是 Perl 专家，那么 PL / Python 可能是适合您的。Python 长期以来一直是 PostgreSQL 基础架构的一部分，因此是一个可靠的、经过良好测试的实现。

在 PL / Python 方面，有一点需要记住：PL / Python 仅作为不受信任的语言提供。从安全的角度来看，时刻牢记这一点很重要。

要启用 PL / Python，可以从命令行运行以下行。test 是要与 PL / Python 一起使用的数据库的名称：

```
createlang plpythonu test
```

一旦启用了该语言，就可以编写代码了。

或者，您可以使用 CREATE LANGUAGE 子句。还要记住，为了使用服务器端语言，需要包含这些语言的 PostgreSQL 包（postgresql-plpython - \$ (VERSIONNUMBER) 等）。

编写简单的 PL / Python 代码

在本节中，您将学习编写简单的 Python 过程。这里讨论的例子很简单：如果你在奥地利开车去拜访一个客户，你可以每公里减去 42 欧分作为费用，以减少你的所得税。因此，该功能的作用是获取公里数并返回我们可以从税单中扣除的金额。下面是它的工

作原理：

```
CREATE OR REPLACE FUNCTION calculate_deduction(km float)
RETURNS numeric AS
$$
if    km <= 0:
    elog(ERROR, 'invalid number of kilometers')
else:
    return km * 0.42
$$ LANGUAGE 'plpythonu';
```

该函数确保只接受正值。 最后，计算并返回结果。 正如您所看到的，Python 函数传递给 PostgreSQL 的方式与 Perl 或 PL / pgSQL 没有什么不同。

使用 SPI 接口

与所有过程语言一样，PL / Python 允许您访问 SPI 接口。下面的示例演示如何将数字相加：

```
CREATE FUNCTION add_numbers(rows_desired integer)
RETURNS integer AS
$$
mysum = 0
cursor = plpy.cursor("SELECT * FROM
generate_series(1, %d) AS id" % (rows_desired))
while True:
    rows = cursor.fetch(rows_desired)
    if not rows:
        break
    for row in rows:
        mysum += row['id']
return mysum
$$ LANGUAGE 'plpythonu';
```

尝试此示例时，请确保对游标的调用实际上是一行。

Python 是关于缩进的，所以如果你的代码由一行或两行组成，它确实会有所不同。

一旦创建了游标，我们就可以循环并添加这些数字。 可以使用列名轻松引用这些行中的列。

调用该函数将返回所需的结果：

```
test=# SELECT add_numbers(10);
add_numbers
-----
```

(1 row)

如果要检查 SQL 语句的结果集，PL / Python 提供了各种函数来从结果中检索更多信息。同样，这些函数是 SPI 在 c 级别上所提供功能的包装器。

以下函数更密切地检查结果：

```
CREATE OR REPLACE FUNCTION result_diag(rows_desired integer)
RETURNS integer AS
$$
rv = plpy.execute("SELECT *
    FROM generate_series(1, %d) AS id" % (rows_desired))
plpy.notice(rv.nrows())
plpy.notice(rv.status())
plpy.notice(rv.colnames())
plpy.notice(rv.coltypes())
plpy.notice(rv.coltypmods())
plpy.notice(rv. str ())
return 0
$$ LANGUAGE 'plpythonu';
```

`nrows()` 函数将显示行数。`status()` 函数告诉我们一切是否正常。`colnames()` 函数返回列列表。`coltypes()` 函数返回结果集中数据类型的对象 ID。23 是内部整数，如下表所示：

```
test=# SELECT typname FROM pg_type WHERE oid = 23;
typname
-----
int4
(1 row)
```

然后是 `typmod`。比如 `varchar(20)`: 类型的配置部分就是 `typmod` 的全部内容。

最后，有一个函数将整个事物作为字符串返回以进行调试。调用该函数将返回以下结果：

```
test=# SELECT result_diag(3);
NOTICE:  3
NOTICE:  5
NOTICE:  ['id']
NOTICE:  [23]
NOTICE:  [-1]
NOTICE:  <PLyResult status=5 nrows=3 rows=[{'id': 1},
{'id': 2}, {'id': 3}]>
result_diag
-----
0
(1 row)
```

SPI 接口中还有许多功能可以帮助您执行 SQL。

处理错误

偶尔，您可能必须捕获错误。当然，这在 Python 中也是可能的。

以下示例显示了其工作原理：

```
CREATE OR REPLACE FUNCTION trial_error()
RETURNS text AS
$$
try:
    rv = plpy.execute("SELECT surely_a_syntax_error")
except plpy.SPIError:
    return "we caught the error" else:
else:
    return "all fine"
$$ LANGUAGE 'plpythonu';
```

您可以使用普通的 try / except 块并访问 plpy 来处理您想要捕获的错误。然后，该函数可以正常返回而不会破坏您的事务，如下所示：

```
test=# SELECT trial_error();
trial_error
-----
we caught the error
(1 row)
```

请记住，PL / Python 可以完全访问 PostgreSQL 的内部。因此，它还可以向您的存储过程公开各种错误。这里是一个例子：

```
except spiexceptions.DivisionByZero:
    return "found a division by zero"
except spiexceptions.UniqueViolation:
    return "found a unique violation"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
```

在 python 中捕捉错误非常简单，可以帮助防止函数失败。

改善函数

到目前为止，您已经了解了如何编写基本函数以及各种语言的触发器。

当然还支持更多语言。一些最突出的是 PL / R (R 是强大的统计软件包) 和 PL / v8 (基于 Google JavaScript 引擎)。但是，这些语言超出了本章的范围 (无论其用途如何)。

在本节中，我们将重点介绍如何提高函数的性能。 我们可以通过几种方式加快处理速度：

- 减少调用次数
- 使用执行计划缓存
- 给优化器提示

在本章中，将讨论所有三个主要方面。

减少函数调用次数

在许多情况下，性能很差的原因是因为函数的调用过于频繁。 在我个人看来，我不得不强调这一点：过于频繁地调用是导致性能不佳的主要原因。 创建函数时，可以选择三种类型的函数： volatile, stable 和 immutable。 这是一个例子：

```
test=# SELECT random(), random();
          random      |    random
-----+-----
 0.276252629235387 | 0.710661871358752
(1 row)

test=# SELECT now(), now();
          now      |    now
-----+-----
 2016-12-16 12:57:17.135751+01 | 2016-12-16 12:57:17.135751+01
(1 row)

test=# SELECT pi();
          pi
-----
 3.14159265358979
(1 row)
```

volatile 函数意味着该函数不能被优化。 必须一次又一次地执行。 volatile 函数也可能是不使用某个索引的原因。 默认情况下，每个函数都被认为是不稳定的。

一个稳定的函数总是在同一个事务中返回相同的数据。 它可以被优化，调用也可以被移除。 now () 函数是稳定函数的一个很好的例子；在同一事务中，它返回相同的数据。

不可变函数是黄金标准，因为它们大多数被允许优化，这是因为它们总是在给定相同输入时返回相同的结果。 作为优化函数的第一步，始终确保在函数定义的末尾添加 volatile, stable 和 immutable。

使用执行计划缓存

在 PostgreSQL 中，使用四个阶段执行查询：

1. 解析（Parser）：这会检查语法
2. 重写系统（Rewrite system）：这需要遵守规则

-
3. 优化器/规划器 (Optimizer/planner) : 这可以优化查询
 4. 执行者 (Executor) : 执行计划者提供的计划

如果查询很短，则与实际执行时间相比，前三个步骤相对耗时。因此，缓存执行计划是有意义的。PL / pgSQL 基本上可以在幕后自动执行所有计划缓存。你不必担心它。PL / Perl 和 PL / Python 将为您提供选择。

SPI 接口提供处理和运行准备好的查询的功能，因此程序员可以选择是否准备好查询。对于长查询，使用无准备的查询实际上是有意义的。短查询通常应该准备好减少内部开销。

为函数分配成本

从优化器的角度来看，函数基本上就像一个运算符。PostgreSQL 也会像对待一个标准的操作员一样对待成本。问题是这样的：使用 PostGIS 提供的一些功能，添加两个数字通常比使用海岸线相交便宜。问题是优化器不知道功能是便宜还是昂贵。

幸运的是，我们可以告诉优化器使功能更便宜或更昂贵：

```
test=# \h CREATE FUNCTION
Command: CREATE FUNCTION
Description: Define a new function
Syntax:
CREATE [ OR REPLACE ] FUNCTION
...
| COST  execution_cost
| ROWS  result_rows
...
```

COST 参数表示您的操作实际上比标准操作贵多少。它是 `cpu_operator_cost` 的乘数，而不是静态值。通常，默认值为 100，除非该函数已用 C 语言编写。

第二个参数是 ROWS 参数。默认情况下，PostgreSQL 假定一组返回函数将返回 1,000 行，因为系统无法精确计算出将有多少行。ROWS 参数允许开发人员告诉 PostgreSQL 有关预期的行数。

将函数用于各种用途

在 PostgreSQL 中，存储过程可以用于几乎所有内容。在本章中，您已经了解了 `CREATE DOMAIN` 子句等，但也可以创建自己的运算符，类型转换，甚至是排序规则。

在本节中，您将看到如何创建简单的类型转换以及如何使用它。要定义类型转换，请考虑查看 `CREATE CAST` 子句。该命令的语法显示在下一个清单中：

```
test=# \h CREATE CAST
```

Command: CREATE CAST

Description: define a new cast

Syntax:

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name [ (argument_type [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

使用这些东西非常简单。 您只需告诉 PostgreSQL 它应该调用哪个过程将任何类型转换为您想要的数据类型。

在标准的 PostgreSQL 中，您无法将 IP 地址转换为布尔值。因此，它就是一个很好的例子。首先，必须定义存储过程：

```
CREATE FUNCTION inet_to_boolean/inet)
RETURNS boolean AS
$$
BEGIN
    RETURN true;
END;
$$ LANGUAGE 'plpgsql';
```

为简单起见，它返回 `true`。但是，您可以使用任何语言的任何代码进行实际转换。

在下一步中，已经可以定义类型转换：

```
CREATE CAST (inet AS boolean)
WITH FUNCTION inet_to_boolean/inet) AS IMPLICIT;
```

首先要告诉 PostgreSQL 我们要将 `inet` 转换为 `boolean`。然后，列出了该函数，我们告诉 PostgreSQL 我们更喜欢隐式转换。

这只是一个简单的过程，我们可以测试该转换：

```
test=# SELECT '192.168.0.34'::inet::boolean;
bool
-----
t
(1 row)
```

基本上，也可以应用相同的逻辑来定义排序规则。同样，存储过程可用于执行必须执行的操作：

```
test=# \h CREATE COLLATION
Command: CREATE COLLATION
Description: define a new collation
Syntax:
CREATE COLLATION [ IF NOT EXISTS ] name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype, ]
    [ PROVIDER = provider, ]
    [ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
```

小结

在本章中，您学习了如何编写存储过程。经过理论介绍，我们的注意力集中在 PL / pgSQL 的一些选定功能上。除此之外，您还学习了如何使用 PL / Perl 和 PL / Python，它们只是 PostgreSQL 提供的两种重要语言。当然，还有更多的语言可供使用。但是，由于本书范围的限制，这些内容无法详细介绍。如果您想了解更多信息，请访问以下网站：
https://wiki.postgresql.org/wiki/PL_Matrix。

在第 8 章，管理 PostgreSQL 安全性，您将了解 PostgreSQL 安全性。您将学习如何管理用户和权限。最重要的是，您还将了解网络安全性。

QA

函数和存储过程有什么区别？

函数和存储过程一词经常混淆。但是，实际上存在差异：在一个存储过程中，您可以运行多个事务。因此，它不能是 SELECT 语句的一部分，必须使用 CALL 命令执行。与此相反，函数只有有限的事务控制，可以是 SELECT, INSERT, UPDATE 或 DELETE 语句的一部分。

受信任和不受信任的语言有什么区别？

受信任的语言限制了程序员执行系统调用功能的能力。因此，它们被认为是安全的。但是，不受信任的语言可以执行各种操作，因此仅供超级用户使用，以确保不会发生安全漏洞。

函数一般好的还是坏的？

这个问题有点难以回答。没有什么是好事还是坏事。这同样适用于函数。它们可以帮助您更快地实现目标，但有时如果过度使用则可能成为负担。与生活中的所有事物一样，应该以有用的方式使用函数。

PostgreSQL 中有哪些服务器端语言可用？

传统上，PostgreSQL 在服务器端功能方面非常灵活。该核心提供对 SQL, PL / pgSQL, PL / Perl, PL / TCL 和 PL / Python 的支持。但是，还有许多语言可以从各种来源安装，可以用来获得很大的好处。更受欢迎的是 PL / V8 (JavaScript) 和 PL / R。

什么是触发器？

触发器是在更改行时自动执行函数的方法。有各种类型的触发器：ROW LEVEL 触发器，为每一行触发，STATEMENT LEVEL 触发器，为语句触发，以及 EVENT TRIGGERS，可以为 DLL 命令触发。

哪些语言可用于编写函数？

基本上，函数可以用 SQL, PL / pgSQL, PL / Perl, PL / TCL 和 PL / Python 编写。如果需要，可以根据需要添加添加语言。这取决于您的用例，哪种语言是最好的，我们强烈建议您进行相关实验。

函数一般的好还是一般都不好？

这很难说。这实际上取决于你想要实现的目标。在我个人看来，将算法移动到数据所在的位置是一个非常好的主意。但是，有些人可能会对这个问题有不同的看法。我认为“什么都不适合”，所以通过观察你的情况和你想要达到的目标，找出最适合你的东西是有道理的。

哪种语言最快？

这真的取决于你想要做什么。我认为这个问题没有全面的答案，很可能永远不会有。您必须根据具体情况进行检查。

8. 管理 PostgreSQL 安全性

第 7 章“编写存储过程”是关于存储过程和编写服务器端代码的。在介绍了许多重要主题之后，现在是时候转向 PostgreSQL 安全性了。在这里，我们将学习如何保护服务器并配置权限。

本章将介绍以下主题：

- 配置网络访问
- 管理身份验证
- 处理用户和角色
- 配置数据库安全性
- 管理模式，表和列
- 行级安全性

到本章结束时，我们将能够以专业的方式配置 PostgreSQL 安全性。

管理网络安全

在讨论现实世界中的实际例子之前，我想简要介绍一下我们将要处理的安全性的各个层面。在处理安全问题时，记住这些级别是有意义的，以便有组织地处理与安全相关的问题。

这是我的思维模式：

- 地址绑定：`postgresql.conf` 文件中的 `listen_addresses`
- 基于主机的访问控制：`pg_hba.conf` 文件
- 实例级权限：用户，角色，数据库创建，登录和复制
- 数据库级权限：连接，创建模式等
- 架构级权限：使用架构并在架构内创建对象
- 表级权限：查询、插入、更新等
- 列级权限：允许或限制对列的访问
- 行级安全性：限制对行的访问

为了读取一个值，`postgresql` 必须确保我们在每个级别上都有足够的权限。整个权限链必须正确。

了解地址绑定和连接

在配置 PostgreSQL 服务器时，首先需要做的事情之一是定义远程访问。默认情况下，PostgreSQL 不接受远程连接。这里重要的是 PostgreSQL 甚至不拒绝连接，因为它根本不会监听端口。如果我们尝试连接，错误消息实际上将来自操作系统，因为 PostgreSQL 根本不关心。

假设有一个数据库服务器使用 192.168.0.123 上的默认配置，将发生以下情况：

```
iMac:~ hs$ telnet 192.168.0.123 5432  
Trying 192.168.0.123...  
telnet: connect to address 192.168.0.123: Connection refused  
telnet: Unable to connect to remote host
```

Telnet 尝试在端口 5432 上创建连接，但被远程框立即拒绝。从外部看，似乎 PostgreSQL 根本没有运行。

成功的关键可以在 postgresql.conf 文件中找到：

```
# - Connection Settings -  
# listen_addresses = 'localhost'  
    # what IP address(es) to listen on;  
    # comma-separated list of addresses;  
    # defaults to 'localhost'; use '*' for all  
    # (change requires restart)
```

listen_addresses 设置将告诉 PostgreSQL 要监听的地址。从技术上讲，这些地址是绑定地址。那到底是什么意思？假设我们的机器里有四张网卡。我们可以监听其中的三个 IP 地址。PostgreSQL 会考虑到这三张卡的请求，而不会监听第四张卡。端口完全关闭了。



TIP 我们必须将服务器的 IP 地址放入 listen_addresses 而不是客户端的 IP。

如果我们输入一个*号，PostgreSQL 将监听分配给您机器的每个 ip。



TIP 请记住，更改侦听地址需要重新启动 PostgreSQL 服务。它不能在没有重启的情况下即时更改。

但是，有更多与连接管理相关的设置非常重要。具体如下：

```
#port = 5432  
    # (change requires restart)  
max_connections = 100  
    # (change requires restart)  
# Note: Increasing max_connections costs ~400 bytes of  
# shared memory per  
# connection slot, plus lock space  
# (see max_locks_per_transaction).  
#superuser_reserved_connections = 3  
    # (change requires restart)  
#unix_socket_directories = '/tmp'  
    # comma-separated list of directories  
    # (change requires restart)  
#unix_socket_group = ""  
    # (change requires restart)
```

```
#unix_socket_permissions = 0777
    # begin with 0 to use octal notation
    # (change requires restart)
```

首先，PostgreSQL 监听一个 tcp 端口，默认值为 5432。请记住，PostgreSQL 只监听一个端口。每当有请求进入时，`postmaster` 将分叉并创建一个新进程来处理连接。默认情况下，最多允许 100 个正常连接。除此之外，还为超级用户保留了三个附加连接。这意味着我们可以有 97 个连接加上 3 个超级用户或者 100 个超级用户连接。



请注意，这些与连接相关的设置也需要重新启动。原因是静态内存分配给共享内存，而共享内存不能动态更改。

检查连接和性能

在咨询时，很多人问我，提高连接限制是否会对性能产生影响。答案是：由于上下文切换以及所有这些，总是会有一些开销。基本上，有多少个连接没有什么区别。但是，有区别的是打开的快照的数量。打开的快照越多，数据库端的开销就越大，换句话说，我们可以廉价地增加 `max_connections`。



TIP 如果您对某些真实数据感兴趣，请考虑查看我之前的一篇博文：https://www.cybertec-postgresql.com/max_connections-performance-impacts/。

生活在没有 TCP 连接的世界里

在某些情况下，我们可能不想使用网络。数据库通常只与本地应用程序进行对话。也许我们的 PostgreSQL 数据库已经随应用程序一起提供了，或者我们只是不想冒使用网络的风险：在这种情况下，您需要的是 unix 套接字。unix 套接字是一种无网络通信方式。您的应用程序可以通过 unix 套接字进行本地连接，而不向外部世界公开任何内容。

然而，我们需要的是一个目录。默认情况下，PostgreSQL 将使用 `/tmp` 目录。但是，如果每台计算机运行多个数据库服务器，则每个数据库服务器都需要一个单独的数据目录。

除了安全性之外，有很多原因说明不使用网络可能是个好主意。其中一个原因是性能。。使用 unix 套接字比通过环回设备（`127.0.0.1`）快得多。如果这听起来令人惊讶，不要担心，这是对许多人来说的。但是，如果您只运行非常小的查询，则不应低估实际网络连接的开销。

为了描述这意味着什么，我已经包含了一个简单的基准测试。

我们将创建一个 `script.sql` 文件。这是一个简单的脚本，只需创建一个随机数并选择它。这是最简单的说法。没有什么比仅仅取一个数字更简单的了。

那么，让我们在普通的笔记本电脑上运行这个简单的基准测试。为此，我们将编写一个名为 `script.sql` 的小东西。它将被基准测试使用：

```
[hs@linuxpc ~]$ cat /tmp/script.sql
SELECT 1
```

然后，我们可以简单地运行 `pgbench` 来反复执行 SQL。`-f` 选项允许将 SQL 的名称传递给脚本。`-c 10` 表示我们希望 10 个并发连接处于活动状态 5 秒 (`-T 5`)。该基准测试作为 `postgres` 用户运行，并且应该使用 `postgres` 数据库，默认情况下该数据库应该存在。请注意，以下示例适用于 RHEL 衍生产品。基于 Debian 的系统将使用不同的路径：

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql
-c 10 -T 5
-U postgres postgres 2>
/dev/null transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 871407
latency average = 0.057 ms
tps = 174278.158426 (including connections establishing)
tps = 174377.935625 (excluding connections establishing)
```

我们可以看到，没有主机名传递给 `pgbench`，因此该工具在本地连接到 Unix 套接字并尽可能快地运行脚本。在这款四核英特尔机型是哪个，该系统每秒可实现约 174,000 次交易。

如果添加 `-h localhost` 会发生什么？如下所示，性能将发生变化。

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql
-h localhost -c 10 -T 5
-U postgres postgres 2>
/dev/null transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 535251
latency average = 0.093 ms
tps = 107000.872598 (including connections establishing)
tps = 107046.943632 (excluding connections establishing)
```

吞吐量将下降，降到每秒 107000 次交易。差异显然与网络开销有关。



TIP 通过使用-j 选项（分配给 pgbench 的线程数），我们可以从系统中挤出更多的事务。但是，在我的情况下，它并没有改变基准的整体情况。在其他测试中，它确实是因为如果你没有提供足够的 CPU 能力，pgbench 可能是一个真正的瓶颈。

我们可以看到，网络不仅是一个安全问题，也是一个性能问题。

管理 pg_hba.conf

配置绑定地址后，我们可以进入下一个级别。 pg_hba.conf 文件将告诉 PostgreSQL 如何验证来自网络的人员。通常，pg_hba.conf 文件条目具有以下布局：

```
# local DATABASE USER METHOD [OPTIONS]
# host DATABASE USER ADDRESS METHOD [OPTIONS]
# hostssl DATABASE USER ADDRESS METHOD [OPTIONS]
# hostnossal DATABASE USER ADDRESS METHOD [OPTIONS]
```

可以将四种类型的规则放入 pg_hba.conf 文件中：

- **local:** 这可以用于配置本地 Unix 套接字连接。
- **host:** 这可用于 SSL 和非 SSL 连接。
- **hostssl:** 这仅适用于 SSL 连接。要使用此选项，必须将 SSL 编译到服务器中，如果我们使用 PostgreSQL 的预打包版本就是这种情况。除此之外，还必须在 postgresql.conf 文件中设置 ssl = on。这是服务器启动时的文件。
- **hostnossal:** 适用于非 SSL 连接。

可以将规则列表放入 pg_hba.conf 文件中。这是一个例子：

```
# TYPE DATABASE USER ADDRESS METHOD
# "local" is for Unix domain socket connections only
local all all trust
# IPv4 local connections:
host all all 127.0.0.1/32 trust
# IPv6 local connections:
host all all ::1/128 trust
```

你可以看到三个简单的规则。本地记录表明，所有数据库的本地 Unix 套接字的所有用户都是可信的。信任模式意味着不必向服务器发送密码，用户可以直接登录。另外两个规则说明，这同样适用于来自 localhost 127.0.0.1 和 ::1/128 (ipv6 地址) 的连接。

由于没有密码的连接肯定不是远程访问的最佳选择，PostgreSQL 提供了各种身份验证方法，可用于灵活配置 pg_hba.conf 文件。以下是可能的身份验证方法列表：

- **trust:** 这允许在不提供密码的情况下进行身份验证。必须在 PostgreSQL 端提供所需的用户。
- **reject:** 连接将被拒绝。
- **md5 和 password:** 可以使用密码创建连接。md5 表示密码是通过加密的线路

发送的。在密码的情况下，凭证以纯文本形式发送，这不应该在现代系统上出现。`md5` 不再被认为是安全的。你应该在 PostgreSQL 10 及更高版本中使用 `scram-sha-256`。

- `scram-sha-256`: 此设置是 `md5` 的后续设置，使用的哈希算法比以前的版本安全得多。
- `GSS 和 SSPI`: 这使用 GSSAPI 或 SSPI 身份验证。这仅适用于 TCP / IP 连接。这里的想法是允许单点登录。
- `ident`: 通过联系客户端的 `ident` 服务器并检查它是否与请求的数据库用户名匹配，获取客户端的操作系统用户名。
- `peer`: 假设我们在 Unix 上以 `abc` 身份登录。如果启用了 `peer`，我们只能以 `abc` 身份登录 PostgreSQL。如果我们尝试更改用户名，我们将被拒绝。好处是 ABC 不需要密码来进行身份验证。这里的想法是，只有数据库管理员才能登录到 Unix 系统上的数据库，而不是在同一台机器上只拥有密码或 Unix 帐户的其他人。这仅适用于本地连接。
- `PAM`: 这使用可插入的身份验证模块 (PAM)。如果您想使用 PostgreSQL 提供的即时身份验证方法，这一点尤其重要。要使用 PAM，请在 Linux 系统上创建一个名为 `/etc/pam.d/postgresql` 的文件，并将您计划使用的所需 PAM 模块放入配置文件中。使用 PAM，我们甚至可以针对不太常见的组件进行身份验证。但是，它也可以用于连接到 Active Directory 等。
- `LDAP`: 此配置允许您使用轻量级目录访问协议 (LDAP) 进行身份验证。请注意，PostgreSQL 只会要求 LDAP 进行身份验证；如果用户仅存在于 LDAP 但不存在于 PostgreSQL 端，则无法登录。另请注意，PostgreSQL 必须知道 LDAP 服务器的位置。所有这些信息都必须存储在 `pg_hba.conf` 文件中，如官方文档中所述：<https://www.postgresql.org/docs/10/static/auth-methods.html#AUTH-LDAP>。
- `RADIUS`: 远程身份验证拨入用户服务 (RADIUS) 是一种执行单点登录的方法。同样，使用配置选项传递参数。
- `cert`: 此身份验证方法使用 SSL 客户端证书执行身份验证，因此仅在使用 SSL 时才可以执行身份验证。这里的优点是不必发送密码。证书的 `CN` 属性将与请求的数据库用户名进行比较，如果匹配，则允许登录。映射可用于允许用户进行映射。

规则可以一个接一个地列出。这里重要的一点是顺序确实会产生影响，如下例所示：

```
host    all    all    192.168.1.0/24    scram-sha-256
host    all    all    192.168.1.54/32    reject
```

当 PostgreSQL 遍历 `pg_hba.conf` 文件时，它将使用第一个匹配的规则。所以，如果我们的请求来自 `192.168.1.54`，那么第一条规则在我们到达第二条规则之前总是匹配的。这意味着，如果密码和用户正确，`192.168.1.54` 将能够登录；因此，第二条规则毫无意义。

如果我们要排除 IP，请确保交换这两个规则。

处理 SSL

PostgreSQL 允许我们加密服务器和客户端之间的传输。加密是非常有益的，尤其是当我们在远距离通信时。SSL 提供了一种简单而安全的方法来确保没有人能够监听您的通信。

在本节中，我们将学习如何设置 SSL。

首先要做的是在服务器启动时在 `postgresql.conf` 文件中将 SSL 参数设置为 `on`。在下一步中，我们可以将 SSL 证书放入`$ PGDATA` 目录中。如果我们不希望证书位于其他目录中，请更改以下参数：

```
#ssl_cert_file = 'server.crt'      # (change requires restart)
#ssl_key_file = 'server.key'       # (change requires restart)
#ssl_ca_file = ''                  # (change requires restart)
#ssl_crl_file = ''                # (change requires restart)
```

如果我们要使用自签名证书，请执行以下步骤：

```
openssl req -new -text -out server.req
```

回答 OpenSSL 提出的问题。确保我们输入本地主机名作为通用名称。我们可以将密码留空。此调用将生成一个受密码保护的密钥；它不会接受长度少于四个字符的密码短语。

要删除密码（如果要自动启动服务器，则必须如此），请运行以下命令：

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

输入旧密码以解锁现有密钥。现在，执行此操作将证书转换为自签名证书，并将密钥和证书复制到服务器将查找的位置：

```
openssl req -x509 -in server.req -text
-key server.key -out server.crt
```

执行此操作后，请确保文件具有正确的权限集：

```
chmod og-rwx server.key
```

一旦将适当的规则放入 `pg_hba.conf` 文件中，我们就可以使用 SSL 连接到您的服务器。要验证我们确实使用 SSL，请考虑检查 `pg_stat_ssl` 函数。它将告诉我们每个连接以及它是否使用 SSL，它将提供有关加密的一些重要信息：

```
test=# \d pg_stat_sslView "pg_catalog.pg_stat_ssl"
Column          | Type   | Modifiers
-----+-----+-----
pid            | integer |
ssl             | boolean |
version        | text    |
```

cipher		text	
bits		integer	
compression		boolean	
clientdn		text	

如果进程的 ssl 字段包含 true; PostgreSQL 做了我们期望它做的事情:

```
postgres=# select * from pg_stat_ssl;
-[ RECORD 1 ] -----
pid          | 20075
ssl          | t
version      | TLSv1.2
cipher       | ECDHE-RSA-AES256-GCM-SHA384
bits         | 256
compression  | f
clientdn     |
```

处理实例级安全性

到目前为止，我们已经配置了地址绑定，我们告诉 PostgreSQL 使用哪种 IP 范围进行身份验证。 到目前为止，配置完全与网络相关。

在下一步中，我们可以将注意力转移到实例级别的权限上。 最重要的是要知道 PostgreSQL 中的用户存在于实例级别。 如果我们创建一个用户，它不仅在一个数据库中可见；它可以被所有数据库看到。 用户可能只具有访问单个数据库的权限，但基本上用户是在实例级别创建的。

对于那些刚接触 PostgreSQL 的人来说，还有一件事要记住：用户和角色是一回事。 CREATE ROLE 和 CREATE USER 子句具有不同的默认值（字面上，唯一的区别是默认情况下角色没有得到 LOGIN 属性），但归根结底，用户和角色是相同的。 因此，CREATE ROLE 和 CREATE USER 子句支持完全相同的语法：

```
test=# \h CREATE USER
Command: CREATE USER
Description: define a new database role
Syntax:
CREATE USER name [ [ WITH ] option [ ... ] ]
where option can be:
    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCRAEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
```

```
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

让我们逐一讨论这些语法元素。我们首先看到的是，用户可以是超级用户，也可以是普通用户。如果某人被标记为 SUPERUSER，则不再有普通用户必须面对的任何限制。超级用户可以随意删除对象（数据库等）。

下一个重要的事情是它在实例级别上获取创建新数据库的权限。



请注意，当某人创建数据库时，该用户将自动成为数据库的所有者。

规则是这样的：创建者总是自动成为对象的所有者（除非另有说明，否则可以使用 CREATE DATABASE 子句）。美妙的是，对象所有者也可以再次丢弃对象。



CREATEROLE / NOCREATEROLE 子句定义是否允许某人创建新用户/角色。

下一个重要的是 INHERIT / NOINHERIT 条款。如果设置了 INHERIT 子句（这是默认值），则用户可以继承其他用户的权限。使用继承权限允许我们使用角色，这是抽象权限的好方法。例如，我们可以创建 bookkeeper 的角色，并使许多其他角色从 bookkeeper 继承。我们的想法是，即使我们有很多人从事会计工作，我们也只需告诉 PostgreSQL 一个 bookkeeper 可以做什么。

LOGIN / NOLOGIN 子句定义是否允许角色登录到实例。



请注意，LOGIN 子句不足以实际连接到数据库。为此，需要更多权限。

在这一点上，我们正试图将其放入实例中，这基本上是实例中所有数据库的入口。让我们回到我们的例子：bookkeeper 可能被标记为 NOLOGIN，因为我们希望人们使用他们的真实姓名登录。您的所有会计师（比如 Joe 和 Jane）可能会被标记为 LOGIN 子句，但可以继承 bookkeeper 角色的所有权限。这样的结构可以很容易地确保所有 bookkeeper 具有相同的权限，同时确保他们的个人活动在他们各自的身份下进行操作和记录。

如果我们计划使用流复制运行 PostgreSQL，我们可以作为超级用户执行所有事务日志流。但是，从安全角度来看，不建议这样做。作为保证，我们不必成为流式 xlog 的超级用户，PostgreSQL 允许我们为普通用户提供复制权限，然后可以用来进行流式传输。通常的做法是仅为了管理流媒体而创建一个特殊用户。

正如我们将在本章后面看到的，PostgreSQL 提供了一种称为行级安全性的功能。

我们的想法是，我们可以从用户范围中排除行。如果明确要求用户绕过 RLS，请将此值设置为 BYPASSRLS。NOBYPASSRLS 是默认值。

有时，限制用户允许的连接数是有意义的。

连接限制允许我们这样做。注意，总的来说，连接数不能超过 `postgresql.conf` 文件中定义的连接数 (`max_connections`)。但是，我们总是可以将某些用户限制为较低的值。

默认情况下，PostgreSQL 会将密码存储在加密的系统表中，这是一个很好的默认行为。但是，假设您正在参加培训课程，并且有十名学生参加，并且每个人都与您的数据库盒子相关联。您可以 100% 确定其中一个人会偶尔忘记他或她的密码。由于您的设置不是安全关键，您可能决定以纯文本格式存储密码，以便您可以轻松查找并将其提供给学生。如果您正在测试软件，此功能也可能会派上用场。

通常，我们已经知道有人会很快离开公司。`VALID UNTIL` 子句允许我们在他或她的帐户过期时自动锁定特定用户。

`IN ROLE` 子句列出一个或多个现有角色，新角色将立即作为新成员添加到其中。它有助于避免额外的手动步骤。代替 `IN ROLE` 的另一种方法是 `IN GROUP`。

`ROLE` 子句将定义自动添加为新角色成员的角色。

`ADMIN` 子句与 `ROLE` 子句相同，但添加了 `WITH ADMIN OPTION`。

最后，我们可以使用 `SYSID` 子句为用户设置一个特定的 ID（类似于 Unix 管理员在操作系统级别对用户名所做的操作）。

创建和修改用户

在这一理论介绍之后，是时候实际创建用户，并看看如何在实际示例中使用这些东西了：

```
test=# CREATE ROLE bookkeeper NOLOGIN;  
CREATE ROLE  
test=# CREATE ROLE joe LOGIN;  
CREATE ROLE  
test=# GRANT bookkeeper TO joe;  
GRANT ROLE
```

这里做的第一件事是创建一个名为 `bookkeeper` 的角色。

请注意，我们不希望人们以 `bookkeeper` 身份登录，因此该角色标记为 `NOLOGIN`。



TIP 另请注意，如果使用 `CREATE ROLE` 子句，`NOLOGIN` 是默认值。如果您更喜欢 `CREATE USER` 子句，则默认设置为 `LOGIN`。

然后，创建 joe 角色并将其标记为 LOGIN。 最后，bookkeeper 角色被分配给 joe 角色，这样他就可以完成 bookkeeper 实际上可以做的所有事情。

一旦用户到位，我们就可以测试到目前为止的情况：

```
[hs@zenbook ~]$ psql test -U bookkeeper  
psql: FATAL:  role "bookkeeper" is not permitted to log in
```

正如预期的那样，bookkeeper 角色不允许登录系统。 如果 joe 角色尝试登录会发生什么？

```
[hs@zenbook ~]$ psql test -U joe  
...  
test=>
```

这实际上将按预期工作。 但请注意，命令提示符已更改。
这只是 PostgreSQL 向您显示您未以超级用户身份登录的一种方式。

创建用户后，可能需要对其进行修改。 我们可能想要改变的一件事是密码。 在 PostgreSQL 中，允许用户更改自己的密码。 下面是它的工作原理：

```
test=> ALTER ROLE joe PASSWORD 'abc';  
ALTER ROLE  
test=> SELECT current_user;  
current_user  
-----  
joe  
(1 row)
```

ALTER ROLE 子句（或 ALTER USER）将允许我们更改在用户创建期间可以设置的大多数设置。 然而，管理用户还有更多的工作要做。 在许多情况下，我们希望为用户分配特殊参数。 ALTER USER 子句为我们提供了这样做的方法：

```
ALTER ROLE { role_specification | ALL }  
[ IN DATABASE database_name ]  
SET configuration_parameter { TO | = } { value | DEFAULT }  
ALTER ROLE { role_specification | ALL }  
[ IN DATABASE database_name ]  
SET configuration_parameter FROM CURRENT  
ALTER ROLE { role_specification | ALL }  
[ IN DATABASE database_name ] RESET configuration_parameter  
ALTER ROLE { role_specification | ALL }  
[ IN DATABASE database_name ] RESET ALL
```

语法相当简单和直接。 为了说明为什么这非常有用，我添加了一个真实的例子。 假设 Joe 碰巧住在毛里求斯岛上。 当他登录时，他希望在自己的时区，即使他的数据库服务器位于欧洲：

```
test=> ALTER ROLE joe SET TimeZone = 'UTC-4';
ALTER ROLE
test=> SELECT now();
-----
2017-01-09 20:36:48.571584+01
(1 row)
test=> q
[hs@zenbook ~]$ psql test -U joe
...
test=> SELECT now();
now
-----
2017-01-09 23:36:53.357845+04
(1 row)
```

ALTER ROLE 子句将修改用户。 Joe 一旦重新连接，就会为他设定时区。



TIP 时区不会立即更改。 您应该重新连接或使用 SET ... TO DEFAULT 子句。

这里重要的是，这也适用于某些内存参数，例如 `work_mem` 等，这些参数已在本书前面介绍过。

定义数据库级安全性

在实例级别配置用户之后，可以深入了解并查看在数据库级别可以做些什么。出现的第一个主要问题是：我们显式地允许 `joe` 登录到数据库实例，但是谁或什么允许 `joe` 实际连接到其中一个数据库？也许我们不希望 `joe` 访问您系统中的所有数据库。限制对某些数据库的访问正是我们在此级别上可以实现的。

对于数据库，可以使用 `GRANT` 子句设置以下权限：

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
| ALL [ PRIVILEGES ] }
ON DATABASE database_name [, ...]
TO role_specification [, ...] [ WITH GRANT OPTION ]
```

数据库级别有两个主要权限值得密切关注：

- `CREATE`: 这允许某人在数据库中创建一个模式。 请注意，`CREATE` 子句不允许创建表；它是关于模式的。 在 PostgreSQL 中，表位于模式中，因此您必须首先进入模式级别才能创建表。
- `CONNECT`: 这允许有人连接到数据库。

现在的问题是：没有人明确地为 `joe` 角色分配 `CONNECT` 权限，那么这些权限实际上来自哪里？ 答案是：有一个叫做 `public` 的东西，类似于 Unix 系统世界。 如果这个世界被允许做某些事，乔也是如此，因为他是公众的一部分。

最重要的是 `public` 不是一个角色，因为它可以被删除和重命名。我们可以简单地将其视为系统中每个人的等价物。

因此，为了确保不是每个人都可以随时连接到任何数据库，可能必须从 `public` 中撤销 `CONNECT`。为此，我们可以以超级用户身份进行连接并解决问题：

```
[hs@zenbook ~]$ psql test -U postgres  
...  
test=# REVOKE ALL ON DATABASE test FROM public;  
REVOKE  
test=# \q  
[hs@zenbook ~]$ psql test -U joe  
psql: FATAL: permission denied for database "test"  
DETAIL: User does not have CONNECT privilege.
```

我们可以看到，`joe` 角色不再允许连接。此时，只有超级用户才能访问 `test` 数据库。

一般来说，甚至在创建其他数据库之前撤销 `postgres` 数据库的权限也是一个好主意。这个概念背后的想法是，这些权限将不再存在于所有新创建的数据库中。如果有人需要访问某个数据库，则必须明确授予权限。权利不再自动存在。

如果我们想允许 `joe` 角色连接到 `test` 数据库，请以超级用户身份尝试以下行：

```
[hs@zenbook ~]$ psql test -U postgres  
...  
test=# GRANT CONNECT ON DATABASE test TO bookkeeper;  
GRANT  
test=# \q  
[hs@zenbook ~]$ psql test -U joe  
...  
test=>
```

基本上，这里有两种选择：

- 我们可以直接允许 `joe` 角色，这样只有 `joe` 角色才能连接。
- 或者，我们可以授予 `bookkeeper` 角色权限。记住，`joe` 角色将继承 `bookkeeper` 角色的所有权限，因此如果我们希望所有会计师都能够连接到数据库，那么将权限分配给 `bookkeeper` 角色似乎是一个很有吸引力的想法。

如果我们授予簿记员角色权限，则风险不大，因为首先不允许该角色登录到实例，因此它纯粹是权限的来源。

调整架构级权限

配置完数据库级别后，可以查看模式级别。

在实际查看模式之前，让我们运行一个小测试：

```
test=> CREATE DATABASE test;
ERROR: permission denied to create database test=>
CREATE USER xy;
ERROR: permission denied to create role test=>
CREATE SCHEMA sales;
ERROR: permission denied for database test
```

正如我们所看到的，Joe 今天过得很糟糕，基本上除了连接到数据库之外什么都不允许。

但是，有一个小例外，对许多人来说这是一个惊喜：

```
test=> CREATE TABLE t_broken (id int);
CREATE TABLE
test=> \d
      List   of   relations
 Schema |    Name    | Type  | Owner
-----+-----+-----+
 public | t_broken | table | joe
(1 rows)
```

默认情况下，允许 public 使用公共模式，该模式始终存在。如果我们对保护数据库非常感兴趣，请确保解决此问题。否则，普通用户可能会使用各种表格向您的公共架构发送垃圾邮件，整个设置可能会受到影响。还要记住，如果允许某人创建对象，则此人也是其所有者。所有权意味着创建者自动拥有所有权限，包括对象的销毁。

要从 public 权限中删除这些权限，请以超级用户身份运行以下行：

```
test=# REVOKE ALL ON SCHEMA public FROM public;
REVOKE
```

从现在开始，没有权限，任何人都无法将内容放入您的公共架构中。下一个清单证明了这一点：

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_data (id int);
ERROR: no schema has been selected to create in
LINE 1: CREATE TABLE t_data (id int);
```

我们可以看到，该命令将失败。这里重要的是将显示的错误消息；PostgreSQL 不知道将这些表放在何处。默认情况下，它会尝试将表放入以下架构之一：

```
test=> SHOW search_path ;
search_path
-----
 "$user", public
(1 row)
```

由于没有名为 joe 的模式，因此它不是一个选项，PostgreSQL 将尝试使用公共模式。由于没有权限，它会抱怨它不知道把表放在哪里。

如果表格明确加前缀，情况将立即改变：

```
test=> CREATE TABLE public.t_data (id int);
ERROR: permission denied for schema public
LINE 1: CREATE TABLE public.t_data (id int);
```

在这种情况下，我们将收到您期望的错误消息。PostgreSQL 拒绝访问公共模式。

现在的下一个逻辑问题是：可以在架构级别设置哪些权限，以便为 joe 角色提供更多功能：

```
GRANT {{ CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [, ...]
TO role_specification [, ...] [ WITH GRANT OPTION ]
```

CREATE 意味着有人可以将对象放入模式中。用法意味着允许某人输入架构。请注意，输入架构并不意味着可以实际使用架构中的某些内容；这些权限尚未定义。基本上，这只是意味着用户可以看到这个模式的系统目录。

要允许 joe 角色访问它先前创建的表，将需要以下行（以超级用户身份执行）：

```
test=# GRANT USAGE ON SCHEMA public TO bookkeeper;
GRANT
```

joe 角色现在能够按预期读取其表：

```
[hs@zenbook ~]$ psql test -U joe
test=> SELECT count(*) FROM t_broken;
          count
-----
           0
(1 row)
```

joe 角色还可以添加和修改行，因为它恰好是表的所有者。然而，尽管它已经可以做很多事情，但 joe 的角色还不是万能的。请考虑以下语句：

```
test=> ALTER TABLE t_broken RENAME TO t_useful;
ERROR: permission denied for schema public
```

让我们仔细看看实际的错误消息。如我们所见，消息抱怨的是架构上的权限，而不是表本身的权限（记住，joe 角色拥有表）。要解决这个问题，必须在模式上解决，而不是在表级别上。以超级用户身份运行以下行：

```
test=# GRANT CREATE ON SCHEMA public TO bookkeeper;
GRANT
```

joe 角色现在可以将其表的名称更改为更有用的名称：

```
[hs@zenbook ~]$ psql test -U joe
test=> ALTER TABLE t_broken RENAME TO t_useful;
ALTER TABLE
```

请记住，如果使用 DDL，这是必要的。在我作为 PostgreSQL 支持服务提供商的日常工作中，我已经看到了一些问题，这些都是一个问题。

使用表

在处理了地址绑定，网络身份验证，用户，数据库和架构之后，我们最终将它转到了表级别。以下代码段显示了可以为表设置的权限：

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE
          | REFERENCES | TRIGGER }
        [, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
      | ALL TABLES IN SCHEMA schema_name [, ...] }
TO role_specification [, ...] [ WITH GRANT OPTION ]
```

让我逐一解释这些权限：

- **SELECT:** 这允许您读取表数据。
- **INSERT:** 这允许您向表中添加行（这也包括副本等；它不仅仅是关于 **INSERT** 子句）。请注意，如果允许您插入，则不会自动允许您阅读。**select** 和 **insert** 子句需要能够读取您插入的数据。
- **UPDATE:** 这会修改表的内容。
- **DELETE:** 用于从表中删除行。
- **TRUNCATE:** 这允许您使用 **TRUNCATE** 子句。请注意，**DELETE** 和 **TRUNCATE** 子句是两个独立的权限，因为 **TRUNCATE** 子句将锁定表，而 **DELETE** 子句不会这样做（即使没有 **where** 条件也不会这样做）。
- **REFERENCES:** 这允许创建外键。必须在引用列和引用列上都具有此权限，否则密钥的创建将不起作用。
- **TRIGGER:** 这允许创建触发器。



GRANT 子句的优点是我们可以同时为模式中的所有表设置权限。

这大大简化了调整权限的过程。也可以使用 **WITH GRANT OPTION** 子句。这样做的目的是确保普通用户可以将权限传递给其他用户，这样做的好处是可以大大减少管理员的工作量。试想一下，一个提供数百个用户访问权限的系统；管理所有这些人可能要做很多工作，因此管理员可以指定管理数据子集的人员。

处理列级安全性

在某些情况下，并非所有人都能看到所有数据。想象一下银行。有些人可能会看到有关银行帐户的全部信息，而其他人可能仅限于数据的一部分。在现实世界中，可能有

人不被允许阅读余额栏，或者有人看不到人们贷款的利率。

另一个例子是，人们可以看到人们的个人资料，但不能看到他们的照片或其他私人信息。现在的问题是：如何使用列级安全性？

为了证明这一点，我们将向属于 joe 角色的现有表添加一列：

```
test=> ALTER TABLE t_useful ADD COLUMN name text;  
ALTER TABLE
```

该表现在由两列组成。 该示例的目标是确保用户只能看到其中一列：

```
test=> \d t_useful  
Table "public.t_useful"  
Column | Type | Modifiers  
-----+-----+  
id    | integer |  
name  | text   |
```

作为超级用户，让我们创建一个用户并让它访问包含我们表的模式：

```
test=# CREATE ROLE paul LOGIN;  
CREATE ROLE  
test=# GRANT CONNECT ON DATABASE test TO paul;  
GRANT  
test=# GRANT USAGE ON SCHEMA public TO paul;  
GRANT
```

不要忘记将连接权限授予新用户，因为在本章的前面，CONNECT 已从 public 中撤消。因此，明确的授权是绝对必要的，以确保我们甚至可以到表。

SELECT 权限可以赋予 paul 角色：

```
test=# GRANT SELECT (id) ON t_useful TO paul;  
GRANT
```

基本上，这已经足够了。 已经可以作为用户 paul 连接到数据库并读取列：

```
[hs@zenbook ~]$ psql test -U paul  
...  
test=> SELECT id FROM t_useful;  
id  
----  
(0 rows)
```

如果我们使用列级权限，请记住一件重要的事情；我们应该停止使用 SELECT *，因为它不再起作用：

```
test=> SELECT * FROM t_useful;  
ERROR: permission denied for relation t_useful
```

*仍然意味着所有列，但由于无法访问所有列，因此会立即出错。

配置默认权限

到目前为止，已经配置了很多东西。现在自然产生的麻烦是，如果将新表添加到系统会发生什么？逐个处理这些表并设置适当的权限可能会非常痛苦和风险。如果这些事情会自动发生，那不是很好吗？这正是 `ALTER DEFAULT PRIVILEGES` 子句所做的。这个想法是为用户提供一个选项，让 PostgreSQL 在对象出现后立即自动设置所需的权限。再也不会有人忘记设定这些权利了。

以下清单显示了语法规规范的第一部分：

```
postgres=# \h ALTER DEFAULT PRIVILEGES
Command: ALTER DEFAULT PRIVILEGES
Description: define default access privileges
Syntax:
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke
where abbreviated_grant_or_revoke is one of:
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
TRIGGER }
        [, ...] | ALL [ PRIVILEGES ] }
        ON TABLES
        TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
...
...
```

基本上，语法与 `GRANT` 子句类似，因此使用起来简单直观。为了向我们展示它的工作原理，我编写了一个简单的例子。我们的想法是，如果 `joe` 角色创建了一个表，该角色将自动使用它：

```
test=# ALTER DEFAULT PRIVILEGES FOR ROLE joe
      IN SCHEMA public GRANT ALL ON TABLES TO paul;
ALTER DEFAULT PRIVILEGES
```

让我们现在以 `joe` 角色连接并创建一个表：

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_user (id serial, name text, passwd text);
CREATE TABLE
```

作为 `paul` 角色连接将证明该表已分配给适当的权限集：

```
[hs@zenbook ~]$ psql test -U paul
...
...
```

```
test=> SELECT * FROM t_user;
      id | name | passwd
      ----+-----+
      (0 rows)
```

深入研究行级安全性 - RLS

到目前为止，表总是作为一个整体显示。当表包含 100 万行时，可以从中检索 100 万行。如果有人有权阅读一张桌，那就是整个表的内容。在许多情况下，这还不够。通常，不希望允许用户查看所有行。

考虑下面的现实世界的例子，其中一个会计正在为许多人做会计工作。包含税率的表格应该对每个人都是可见的，因为每个人都必须支付相同的税率。然而，当涉及到实际的交易时，会计师可能希望确保每个人都只能看到自己的交易。不应允许人员 A 查看人员 B 的数据。除此之外，一个部门的老板还可以查看他所在部门的所有数据。

行级安全性旨在实现这一目标，使您能够以快速简单的方式构建多租户系统。配置这些权限的方法是提出策略。CREATE POLICY 命令为我们提供了编写这些规则的方法：

```
test=# \h CREATE POLICY
Command: CREATE POLICY
Description: define a new row level security policy for a table
Syntax:
CREATE POLICY name ON table_name
  [ AS { PERMISSIVE | RESTRICTIVE } ]
  [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
  [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
  [ USING ( using_expression ) ]
  [ WITH CHECK ( check_expression ) ]
```

为了描述如何编写策略，让我们首先以超级用户身份登录并创建一个包含几个条目的表：

```
test=# CREATE TABLE t_person (gender text, name text);
```

```
CREATE TABLE
```

```
test=# INSERT INTO t_person
```

```
VALUES      ('male', 'joe'),
            ('male', 'paul'),
            ('female', 'sarah'),
            (NULL, 'R2- D2');
```

```
INSERT 0 4
```

然后授予 joe 角色访问权限：

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

到目前为止，一切都很正常，joe 角色将能够实际读取整个表，因为没有启用 RLS。但

是如果为表启用了 ROW LEVEL SECURITY 会发生什么:

```
test=# ALTER TABLE t_person ENABLE ROW LEVEL SECURITY;  
ALTER TABLE
```

有一个拒绝所有默认的策略，所以 joe 角色实际上会得到一个空表:

```
test=> SELECT * FROM t_person;  
gender | name  
-----+  
(0 rows)
```

实际上，默认策略很有意义，因为用户必须显式设置权限。

现在该表处于行级安全控制之下，可以使用超级用户编写策略:

```
test=# CREATE POLICY joe_pol_1  
ON t_person  
FOR SELECT TO joe  
USING (gender = 'male');  
CREATE POLICY
```

以 joe 角色登录并选择所有数据将只返回两行:

```
test=> SELECT * FROM t_person;  
gender | name  
-----+  
male   | joe  
male   | paul  
(2 rows)
```

让我们以更详细的方式检查我们刚刚创建的策略。 我们首先看到的是政策实际上有一个名称。 它还连接到表并允许某些操作（在本例中为 SELECT 子句）。 然后是 USING 子句。 它基本上定义了允许 joe 角色看到的内容。 因此，USING 子句是附加到每个查询的强制筛选器，用于仅选择用户应该看到的行。

还有一个重要的侧节点，如果不止一个策略，PostgreSQL 将使用 or 条件。 简而言之， 默认情况下，更多的策略将使您看到更多的数据。 在 PostgreSQL 9.6 中，总是这样。 但是，随着 PostgreSQL 10.0 的引入，用户可以选择条件应该是或和并连接:

PERMISSIVE | RESTRICTIVE

默认情况下，PostgreSQL 是 PERMISSIVE，因此 OR 连接正在工作。 如果我们决定使用 RESTRICTIVE，那么这些子句将与 AND 连接。

现在假设，由于某种原因，已经确定 joe 角色也被允许看到机器人。 实现目标有两种选择。 第一种选择是简单地使用 ALTER POLICY 子句来改变现有策略:

```
postgres=# \h ALTER POLICY  
Command: ALTER POLICY
```

Description: change the definition of a row level security policy

Syntax:

```
ALTER POLICY name ON table_name RENAME TO new_name  
ALTER POLICY name ON table_name  
[ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]  
[ USING ( using_expression ) ]  
[ WITH CHECK ( check_expression ) ]
```

第二个选项是创建第二个策略，如下一个示例所示：

```
test=# CREATE POLICY joe_pol_2  
ON t_person  
FOR SELECT TO joe  
USING (gender IS NULL);  
CREATE POLICY
```

其优点在于，除非使用限制性条件，否则这些策略只是使用前面所述的或条件连接起来的。因此，postgresql 现在将返回三行而不是两行：

```
test=> SELECT * FROM t_person;  
gender | name  
-----+-----  
male   | joe  
male   | paul  
      | R2-D2  
(3 rows)
```

R2-D2 角色现在也包含在结果中，因为它与第二个策略匹配。

为了描述 PostgreSQL 如何运行查询，我决定包含查询的执行计划：

```
test=> explain SELECT * FROM t_person;  
QUERY PLAN  
-----  
Seq Scan on t_person (cost=0.00..21.00 rows=9 width=64)  
  Filter: ((gender IS NULL) OR (gender = 'male'::text))  
(2 rows)
```

我们可以看到，USING 子句都已添加为查询的必需过滤器。我们可能已经在语法定义中注意到有两种类型的子句：

- **USING:** 此子句过滤已存在的行。这与 SELECT 和 UPDATE 子句相关，依此类推。
- **CHECK:** 此子句过滤即将创建的新行，因此它们与 INSERT 和 UPDATE 子句相关，依此类推。

如果我们尝试插入一行，会发生以下情况：

```
test=> INSERT INTO t_person VALUES ('male', 'kaarel');
```

```
ERROR: new row violates row-level security policy for table "t_person"
```

由于 INSERT 子句没有策略，因此该语句自然会出错。以下是允许插入的策略：

```
test=# CREATE POLICY joe_pol_3  
    ON t_person  
    FOR INSERT TO joe  
    WITH CHECK (gender IN ('male', 'female'));  
CREATE POLICY
```

允许 joe 角色将男性和女性添加到表中，如下面的清单所示：

```
test=> INSERT INTO t_person VALUES ('female', 'maria');  
INSERT 0 1
```

但是，还有一个问题；考虑以下示例：

```
test=> INSERT INTO t_person VALUES ('female', 'maria') RETURNING *;  
ERROR: new row violates row-level security policy for table "t_person"
```

记住，只有一个策略允许选择男性。这里的问题是，声明将返回一个女性，这是不允许的，因为 joe 角色是在一个只允许有男性的策略下。

只有对男人来说，RETURNING * 才真正起作用：

```
test=> INSERT INTO t_person VALUES ('male', 'max') RETURNING *;  
gender | name  
-----+-----  
male   | max  
(1 row)  
INSERT 0 1
```

如果我们不希望这种行为，我们必须编写一个实际包含正确的 USING 子句的策略。

检查权限

设置完所有权限后，有时需要知道谁拥有哪些权限。对于管理者来说，找出谁可以做什么是至关重要的。不幸的是，这个过程并不容易，需要一些知识。通常，我非常喜欢使用命令行。然而，在许可系统的情况下，使用图形用户界面来做事情确实是有意义的。

在我向您展示如何阅读 PostgreSQL 权限之前，让我们为 joe 角色分配权限，以便我们可以在下一步中检查它们：

```
test=# GRANT ALL ON t_person TO joe;  
GRANT
```

可以使用 psql 中的 z 命令检索有关权限的信息：

```
test=# \x  
Expanded display is on.
```

```

test=# \z t_person
Access privileges
-[ RECORD 1 ]-----+
-----
Schema          | public
Name            | t_person
Type            | table
Access privileges | postgres=arwdDxt/postgres
+
| joe=arwdDxt/postgres

Column privileges |
Policies        | joe_pol_1 (r):
+
| (u): (gender = 'male'::text)
+
| to: joe
+
| joe_pol_2 (r):
+
| (u): (gender IS NULL)
+
| to: joe
+
| joe_pol_3 (a):
+
| (c): (gender = ANY (ARRAY['male'::text,
'female'::text]))
+
| to: joe

```

这将返回所有这些策略以及有关访问权限的信息。

不幸的是，这些快捷方式很难阅读，我有一种感觉，它们并没有被管理员广泛理解。

在本例中，joe 角色从 PostgreSQL 获得了 arwdDxt。这些字符到底是什么意思？

- a: 这是 INSERT 追加子句
- r: 这是 SELECT 读取子句
- w: 这是 UPDATE 写入子句
- d: 这是 DELETE 删除子句
- D: 这用于 TRUNCATE 子句（当引入时，已经采用了 t）
- x: 这用于关联约束
- t: 这用于触发器

如果您不了解此代码，还有另一种方法可以使事物更具可读性。

考虑以下函数调用：

```

test=# SELECT * FROM aclexplode('{joe=arwdDxt/postgres}');
grantor | grantee  | privilege_type | is_grantable
-----+-----+-----+-----+
      10 | 18481   | INSERT       | f
      10 | 18481   | SELECT       | f
      10 | 18481   | UPDATE       | f
      10 | 18481   | DELETE       | f
      10 | 18481   | TRUNCATE     | f
      10 | 18481   | REFERENCES   | f

```

10	18481 TRIGGER	f
(7 rows)		

正如我们所看到的，权限集作为一个简单的表返回，这使生活变得非常简单。

重新分配对象并删除用户

分配权限和限制访问权限后，可能会发生用户被从系统中删除的情况。不出所料，执行此操作的命令是 DROP ROLE 和 DROP USER 命令：

```
test=# \h DROP ROLE
Command:  DROP ROLE
Description: remove a database role
Syntax:
DROP ROLE  [ IF EXISTS ] name  [, ...]
```

试一试吧。以下清单显示了它的工作原理：

```
test=# DROP ROLE joe;
ERROR:  role "joe" cannot be dropped because some objects depend on it
DETAIL:  target of policy joe_pol_3 on table t_person
target of policy joe_pol_2 on table t_person
target of policy joe_pol_1 on table t_person
privileges for table t_person
owner of table t_user
owner of sequence t_user_id_seq
owner of default privileges on new relations belonging to role joe in
schema public
owner of table t_useful
```

PostgreSQL 将发出错误消息，因为只有在一切都被带走后才能删除用户。这是有道理的：假设有人有一张表。PostgreSQL 应该如何处理该表？有人必须拥有它们。

要将表从一个用户重新分配给下一个用户，请考虑查看 REASSIGN 子句：

```
test=# \h REASSIGN
Command:  REASSIGN OWNED
Description: change the ownership of database objects owned by a database
role
Syntax:
REASSIGN OWNED  BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
    TO { new_role | CURRENT_USER | SESSION_USER }
```

语法也很简单，有助于简化移交过程。下面是一个例子：

```
test=# REASSIGN OWNED  BY joe TO postgres;
REASSIGN OWNED
```

所以，让我们再次尝试删除 joe 角色：

```
test=# DROP ROLE joe;
ERROR:  role "joe" cannot be dropped because some objects depend on it
DETAIL:  target of policy joe_pol_3 on table t_person target of policy
joe_pol_2 on table t_person
target of policy joe_pol_1 on table t_person privileges for table t_person
owner of default privileges on new relations belonging to role joe in
schema public
```

正如我们所看到的，问题清单已经大大减少。我们现在能做的就是一个接一个地解决所有这些问题，并放弃这个角色。我知道没有捷径。提高效率的唯一方法是确保尽可能少的权限分配给真实的用户。尽量把自己抽象成角色，而角色又可以被很多人使用。如果个人权限没有分配给真正的人，一般情况下事情会容易一些。

小结

数据库安全是一个广阔的领域，30页的章节很难涵盖 PostgreSQL 安全的所有方面。很多东西，比如 selinux、安全定义器/调用器等等，都没有被触及。然而，在这一章中，我们学习了作为 PostgreSQL 开发人员和 DBA 将要面对的最常见的事情。我们还学习了如何避免基本的陷阱，以及如何使我们的系统更安全。

在第 9 章“处理备份和恢复”中，我们将学习 PostgreSQL 流式复制和增量备份。本章还将介绍故障转移场景。

QA

如何配置 PostgreSQL 的网络访问？

配置网络访问有两个级别。在 postgresql.conf 中（listen_addresses），您可以配置绑定地址并打开远程连接。在 pg_hba.conf 中，您可以告诉 PostgreSQL 如何验证网络连接。根据请求来自的 IP 范围，可以应用不同的规则。

什么是用户，什么是角色？

基本上，用户和角色之间的区别是学术性的。创建角色时，默认值为 NOLOGIN，使用 CREATE USER 时不是这种情况。否则，可以认为角色和用户是相同的。

如何更改密码？

这很简单。您可以使用 ALTER USER 执行此任务，如以下示例所示：

```
test=# ALTER USER hs PASSWORD 'abc';
ALTER ROLE
```

请记住，密码不一定存储在 PostgreSQL 中。如果您使用 LDAP 身份验证或其他一些外部方法，则不会在 LDAP 端更改存储在 PostgreSQL 中的密码。

什么是行级安全性？

行级安全性是一项允许您限制用户访问表内容的功能。 这里是一个例子： 用户 `joe` 可能只允许看女性，而用户 `jane` 只允许看男性。 因此，行级安全性是一个强制过滤器，它应用于表以限制表的用户范围。

9. 处理备份和恢复

在第 8 章“管理 PostgreSQL 安全性”中，我们以最简单和最有益的方式了解了有关保护 PostgreSQL 的所有知识。本章中的主题是备份和恢复。执行备份应该是一项常规任务，每个管理员都应该关注这些重要的事情。幸运的是，PostgreSQL 提供了一种创建备份的简便方法。

因此，在本章中，我们将介绍以下主题：

- 运行 `pg_dump`
- 部分转储数据
- 备份恢复
- 利用并行性
- 保存全局数据

在本章的最后，我们将能够建立适当的备份机制。

执行简单转储

如果正在运行 PostgreSQL 安装程序，基本上有两种执行备份的主要方法：

- 逻辑转储（提取表示数据的 SQL 脚本）
- 交易日志传送

事务日志传送背后的思想是将对数据库所做的二进制更改存档。

大多数人声称事务日志传送是进行备份的唯一真正方法。

然而，在我看来，这未必是真的。

许多人依靠 `pg_dump` 来简单地提取数据的文本表示。有趣的是，`pg_dump` 也是最古老的创建备份的方法，并且自 PostgreSQL 项目的早期阶段就开始存在（事务日志传送很久以后就添加了）。每个 PostgreSQL 管理员迟早都会熟悉 `pg_dump`，因此了解它是如何工作的以及它的作用非常重要。

运行 `pg_dump`

我们要做的第一件事是创建一个简单的文本转储：

```
[hs@linuxpc ~]$ pg_dump test > /tmp/dump.sql
```

这是您可以想象的最简单的备份。基本上，`pg_dump` 登录到本地数据库实例，连接到数据库测试，并开始提取所有数据，然后将其发送到 `stdout` 并重定向到该文件。这里的美妙之处在于标准输出为您提供了 Unix 系统的所有灵活性。您可以使用管道轻松压缩数据或执行您想要执行的任何操作。

在某些情况下，您可能希望将 `pg_dump` 作为其他用户运行。所有 PostgreSQL 客户端程

序都支持一组一致的命令行参数来传递用户信息。 如果您只想设置用户，请使用-U 标志，如下所示：

```
[hs@linuxpc ~]$ pg_dump -U whatever_powerful_user test > /tmp/dump.sql
```

可以在所有 PostgreSQL 客户端程序中找到以下参数集：

```
...
Connection options:
-d, --dbname=DBNAME database to dump
-h, --host=HOSTNAME database server host or
          socket directory
-p, --port=PORT database server port number
-U, --username=NAME connect as specified database user
-w, --no-password never prompt for password
-W, --password force password prompt (should
          happen automatically)
--role=ROLENAME do SET ROLE before dump
...
...
```

您只需将您想要的信息传递给 `pg_dump`，如果您有足够的权限，PostgreSQL 将获取数据。这里重要的是看看程序是如何工作的。基本上，`pg_dump` 连接到数据库并打开一个大的可重复读取事务，它只读取所有数据。请记住，可重复读取可确保 PostgreSQL 创建一致的数据快照，这些快照在整个事务中不会发生变化。换句话说，转储始终是一致的 - 不会违反外键。输出是转储开始时的数据快照。一致性是这里的关键因素。它还意味着在转储运行时对数据所做的更改将不再进入备份。



TIP 转储只是读取所有内容 - 因此，没有单独的权限设置可以只转储某些内容。
只要你能阅读它，你就可以备份它。

另请注意，默认情况下备份采用文本格式。这意味着您可以安全地从 Solaris 系统中提取数据，并将其移至其他 CPU 架构。在二进制副本的情况下，这显然是不可能的，因为磁盘格式取决于您的 CPU 架构。

传递密码和连接信息

如果仔细查看上一节中显示的连接参数，您会注意到无法将密码传递给 `pg_dump`。您可以强制执行密码提示，但不能使用命令行选项将参数传递给 `pg_dump`。

原因很简单，因为密码可能会显示在进程表中，并且对其他人可见。现在的问题是，如果服务器上的 `pg_hba.conf` 强制执行密码，客户端程序如何提供密码？

有各种方法可以做到这一点。其中一些如下：

- 利用环境变量
- 利用 `.pgpass`
- 使用服务文件

在本节中，我们将了解所有三种方法。

使用环境变量

传递各种参数的一种方法是使用环境变量。如果信息没有显式地传递给 `pg_dump`，它将在预定义的环境变量中查找缺少的信息。所有潜在设置的列表可以在 <https://www.postgresql.org/docs/11/static/libpq-envvars.html> 上找到。

以下概述显示了备份通常需要的一些环境变量：

- `PGHOST`: 它告诉系统要连接的主机
- `PGPORT`: 它定义要使用的 TCP 端口
- `PGUSER`: 它告诉客户端程序有关所需用户的信息
- `PGPASSWORD`: 它包含要使用的密码
- `PGDATABASE`: 它是要连接的数据库的名称

这些环境的优点是密码不会显示在进程表中。但是，还有更多。请考虑以下示例：

```
psql -U ... -h ... -p ... -d ...
```

鉴于您是系统管理员，您是否真的想每天输入一段很长的代码，例如这几天？如果您反复使用同一个主机，只需设置这些环境变量并使用纯 SQL 连接即可。以下清单显示了如何连接：

```
[hs@linuxpc ~]$ export PGHOST=localhost  
[hs@linuxpc ~]$ export PGUSER=hs  
[hs@linuxpc ~]$ export PGPASSWORD=abc  
[hs@linuxpc ~]$ export PGPORT=5432  
[hs@linuxpc ~]$ export PGDATABASE=test  
[hs@linuxpc ~]$ psql  
psql (11.0)  
Type "help" for help.
```

如您所见，不再有命令行参数。只需输入 `psql` 就可以了。



TIP 所有基于标准 PostgreSQL C 语言客户端库（libpq）的应用程序都将理解这些环境变量，因此您不仅可以将它们用于 `psql` 和 `pg_dump`，还可以用于许多其他应用程序。

使用.pgpass

存储登录信息的一种非常常见的方法是使用 `.pgpass` 文件。这个想法很简单：将一个名为 `.pgpass` 的文件放入您的主目录并将您的登录信息放在那里。格式很简单：

```
hostname:port:database:username:password
```

一个例子如下：

192.168.0.45:5432:mydb:xy:abc

PostgreSQL 提供了一些不错的附加功能，其中大多数字段都可以包含*。这是一个例子：

***:*:*:xy:abc**

*表示在每个主机上，每个端口上，对于每个数据库，名为 xy 的用户将使用 abc 作为密码。要使 PostgreSQL 使用.pgpass 文件，请确保使用正确的文件权限：

chmod 0600 ~/.pgpass

此外，.pgpass 也可以在 Windows 系统上使用。在这种情况下，可以在%APPDATA%\postgresql\pgpass.conf 路径中找到该文件。

使用服务文件

但是，.pgpass 不是您可以使用的唯一文件。您还可以使用服务文件。下面是它的工作原理。如果要反复连接到同一台服务器，可以创建.pg_service.conf 文件。它将保存您需要的所有连接信息。

以下是.pg_service.conf 文件的示例：

```
Mac:~ hs$ cat .pg_service.conf
# a sample service
[hansservice]
host=localhost
port=5432
dbname=test
user=hs
password=abc
[paulservice]
host=192.168.0.45
port=5432
dbname=xyz
user=paul
password=cde
```

要连接其中一个服务，只需设置环境并连接：

iMac:~ hs\$ export PGSERVICE=hansservice

现在可以建立连接而无需将参数传递给 psql：

```
iMac:~ hs$ psql
psql (11.0)
Type "help" for help.
test=#
```

或者，您可以使用以下内容：

```
psql service=hansservice
```

提取数据子集

到目前为止，我们已经看到了如何转储整个数据库。然而，这不是我们所希望的。在许多情况下，我们只想提取表或模式的子集。幸运的是，`pg_dump` 可以帮助我们做到这一点，同时还提供了许多开关：

- `a`: 它只转储数据而不转储数据结构
- `-s`: 它转储数据结构但跳过数据
- `-n`: 它只转储某个模式的数据
- `-N`: 它会转储所有内容但排除某些模式
- `-t`: 它只转储某些特定的表
- `-T`: 它转储除了某些表之外的所有内容（如果要排除日志记录表，这可能有意义等）

部分转储可能非常有用，以便大大加快速度。

处理各种格式

到目前为止，我们已经看到 `pg_dump` 可用于创建文本文件。这里的问题是文本文件只能完全重放。如果我们保存了整个数据库，我们只能重放整个数据库。在大多数情况下，这不是我们想要的。因此，PostgreSQL 具有提供更多功能的其他格式。

此时，支持四种格式：

```
-F, --format=c|d|t|p  output file  format (custom, directory, tar, plain  
text (default))
```

我们已经看过 `plain`，这只是普通文本。最重要的是，我们可以使用自定义格式。自定义格式背后的想法是拥有压缩转储，包括目录。以下是创建自定义格式转储的两种方法：

```
[hs@linuxpc ~]$ pg_dump -Fc test > /tmp/dump.fc  
[hs@linuxpc ~]$ pg_dump -Fc test -f /tmp/dump.fc
```

除了目录之外，压缩转储还有一个优点。它小得多。经验法则是，自定义格式转储比要备份的数据库实例小 90% 左右。当然，这在很大程度上取决于索引的数量，但是对于许多数据库应用程序，这种粗略的估计是正确的。

创建备份后，我们可以检查备份文件：

```
[hs@linuxpc ~]$ pg_restore --list /tmp/dump.fc  
;  
; Archive created at 2018-11-04 15:44:56 CET
```

```
; dbname: test
; TOC Entries: 18
; Compression: -1
; Dump Version: 1.12-0
; Format: CUSTOM
; Integer: 4 bytes
; Offset: 8 bytes
; Dumped from database version: 11.0
; Dumped by pg_dump version: 11.0
;
; Selected TOC Entries:
;
3103; 1262 16384 DATABASE - test hs
3; 2615 2200 SCHEMA - public hs
3104; 0 0 COMMENT - SCHEMA public hs
1; 3079 13350 EXTENSION – plpgsql
3105; 0 0 COMMENT - EXTENSION plpgsql
187; 1259 16391 TABLE public t_test hs
...
...
```

请注意，`pg_restore --list` 将返回备份的目录。

使用自定义格式是个好主意，因为备份的大小会缩小。但是，还有更多；`-Fd` 命令将以目录格式创建备份。现在，您将获得一个包含几个文件的目录，而不是单个文件：

```
[hs@linuxpc ~]$ mkdir /tmp/backup
[hs@linuxpc ~]$ pg_dump -Fd test -f /tmp/backup/
[hs@linuxpc ~]$ cd /tmp/backup/
[hs@linuxpc backup]$ ls -lh total  86M
-rw-rw-r--. 1 hs hs   85M Jan   4 15:54  3095.dat.gz
-rw-rw-r--. 1 hs hs  107 Jan   4 15:54  3096.dat.gz
-rw-rw-r--. 1 hs hs 740K Jan   4 15:54  3097.dat.gz
-rw-rw-r--. 1 hs hs   39 Jan   4 15:54  3098.dat.gz
-rw-rw-r--. 1 hs hs 4.3K Jan   4 15:54  toc.dat
```

目录格式的一个优点是我们可以使用多个核心来执行备份。对于普通格式或自定义格式，`pg_dump` 只使用一个进程。目录格式更改了该规则。下面的示例演示如何告诉 `pg_dump` 使用四个核心（作业）：

```
[hs@linuxpc backup]$ rm -rf *
[hs@linuxpc backup]$ pg_dump -Fd test -f /tmp/backup/ -j 4
```



数据库中的对象越多，潜在加速的可能性就越大。

备份重演

除非你试图重演它，否则备份是没有意义的。 幸运的是，这很容易做到。 如果您已创建纯文本备份，只需获取 SQL 文件并执行它。 以下示例显示了如何完成此操作：

```
psql your_db < your_file.sql
```

纯文本备份只是包含所有内容的文本文件。 我们总是可以简单地重放文本文件。

如果您已决定使用自定义格式或目录格式，则可以使用 `pg_restore` 重播备份。 另外，`pg_restore` 允许你做各种奇特的事情，比如只重放数据库的一部分等等。 但是，在大多数情况下，您只需重放整个数据库。 在这个例子中，我们将创建一个空数据库并重放自定义格式转储：

```
[hs@linuxpc backup]$ createdb new_db  
[hs@linuxpc backup]$ pg_restore -d new_db -j 4 /tmp/dump.fc
```

请注意，`pg_restore` 将数据添加到现有数据库。 如果您的数据库不为空，则 `pg_restore` 可能会出错，但会继续。

同样，`-j` 用于抛出多个进程。 在本例中，四个核心用于重播数据；但是，这仅在重播多个表时有效。



TIP 如果使用目录格式，则只需传递目录名称而不是文件名称即可。

就性能而言，如果您处理中小量数据，转储是一个很好的解决方案。有两个主要缺点：

- 我们将获得快照，因此自上次快照以来的所有内容都将丢失
- 与二进制副本相比，从头开始重建转储相对较慢，因为必须重建所有索引

我们将在第 10 章“理解备份和复制”中查看二进制备份。

处理全局数据

在前面的部分中，我们了解了 `pg_dump` 和 `pg_restore`，它们是创建备份时的两个重要程序。 问题是，`pg_dump` 创建数据库转储 - 它在数据库级别上工作。 如果我们要备份整个实例，我们必须使用 `pg_dumpall` 或单独转储所有数据库。 在深入研究之前，了解 `pg_dumpall` 的工作原理是有意义的：

```
pg_dumpall > /tmp/all.sql
```

让我们看看，`pg_dumpall` 将连接到另一个数据库，并将内容发送到标准输出，您可以使用 Unix 处理它。 请注意，`pg_dumpall` 可以像 `pg_dump` 一样使用。 但是，它有一些缺点。 它不支持自定义或目录格式，因此不提供多核支持。 这意味着我们将使用一个线程。

但是，`pg_dumpall` 还有更多内容。 请记住，用户存在于实例级别。

如果创建普通数据库转储，您将获得所有权限，但不会获得所有 CREATE USER 语句。这些全局变量不包含在普通转储中 - 它们只会被 pg_dumpall 提取。

如果我们只想要全局设置，我们可以使用-g 选项运行 pg_dumpall:

```
pg_dumpall -g > /tmp/globals.sql
```

在大多数情况下，您可能希望运行 pg_dumpall -g 以及自定义或目录格式转储来提取实例。一个简单的备份脚本可能如下所示：

```
#!/bin/sh
BACKUP_DIR=/tmp/
pg_dumpall -g > $BACKUP_DIR/globals.sql
for x in $(psql -c "SELECT datname FROM pg_database
WHERE datname NOT IN ('postgres', 'template0', 'template1')" postgres -
A -t)
do
pg_dump -Fc $x > $BACKUP_DIR/$x.fc done
```

它将首先转储全局变量，然后循环遍历数据库列表，以自定义格式逐个提取它们。

小结

在本章中，我们了解了一般创建备份和转储的过程。到目前为止，还没有涵盖二进制备份，但您已经能够从服务器中提取文本备份，以便您可以以最简单的方式保存和重放数据。

在第 10 章“了解备份和复制”中，您将了解事务日志传送，流复制和二进制备份。您还将学习如何使用 PostgreSQL 自带工具来复制实例。

QA

每个人都应该创建转储吗？

如果您的数据库相当小，转储肯定是有意义的。但是，如果数据库很大 (> XXX GB)，则转储可能不再可行，并且不同的方法可能有意义 (WAL 归档)。您还必须记住，转储仅提供数据快照 - 它不提供时间点恢复。因此，转储更多是一种额外的工具，而不是 WAL 存档的替代品。

为什么转储文件这么小？

压缩转储通常比您保存的 PostgreSQL 数据库快 10 倍左右。

原因是数据库必须存储索引的内容，而备份只包含定义。

这在空间消耗方面产生了巨大的差异。最重要的是，PostgreSQL 必须存储额外的数据，如元组标题等，这也需要空间。

是否还必须转储全局变量？

是的，这肯定是要的。

拥有`.pgpass` 文件是否安全？

是。如果权限已正确设置，则 `pgpass` 文件非常安全。请记住：必须备份其他数据库的计算机需要连接到目标系统的所有信息。以哪种格式存储数据是没有区别的。

10. 了解备份和复制

在第 9 章“处理备份和恢复”中，我们学到了很多关于备份和恢复的知识，这对管理至关重要。到目前为止，只涵盖了逻辑备份；我将在本章中改变这一点。

本章是关于 PostgreSQL 的事务日志，以及我们可以用它来改进我们的设置和使事情更安全。

在本章中，我们将介绍以下主题：

- 事务日志的作用以及需要的原因
- 执行时间点恢复
- 设置流复制
- 复制冲突
- 监控复制
- 同步与异步复制
- 了解时间线
- 逻辑复制
- 创建订阅和发布

在本章的最后，您将能够设置事务日志存档和复制。请记住，本章永远不会是复制的综合指南；这只是一个简短的介绍。完全覆盖复制需要大约 500 页。仅作比较，仅 Packt Publishing 的 PostgreSQL 复制就接近 400 页。

本章将以更紧凑的形式介绍最基本的内容。

了解事务日志

每一个现代数据库系统都提供了功能，以确保系统在发生故障或有人拔出插头时能够在崩溃中存活下来。对于文件系统和数据库系统都是这样的。

PostgreSQL 还提供了一种方法来确保崩溃不会损害数据的完整性或数据本身。保证如果断电，系统将始终能够重新启动并完成其工作。

提供这种安全性的方法是通过 Write Ahead Log（WAL）或 xlog 实现的。我们的想法是不直接写入数据文件，而是首先写入日志。为什么这很重要？想象一下，我们正在编写一些数据，如下所示：

```
INSERT INTO data ... VALUES ('12345678');
```

假设数据直接写入数据文件。如果操作中途失败，数据文件将被损坏。它可能包含半写的行、没有索引指针的列、缺少提交信息等等。由于硬件并不能真正保证大数据块的原子写入，因此必须找到一种方法使其更加健壮。通过写入日志而不是直接写入文件，可以解决此问题。



在 PostgreSQL 中，事务日志由记录组成。

单个写入可以包含所有具有校验和链接在一起的各种记录。单个事务可能包含 B 树，索引，存储管理器，提交记录等等。每种类型的对象都有自己的 WAL 条目，并确保对象可以在崩溃中幸存。如果发生崩溃，PostgreSQL 将根据事务日志启动并修复数据文件，以确保不会发生永久性损坏。

查看事务日志

在 PostgreSQL 中，除非在 `initdb` 上另有说明，否则 WAL 通常可以在数据目录的 `pg_wal` 目录中找到。在早期版本的 PostgreSQL 中，WAL 目录被称为 `pg_xlog`，但随着 PostgreSQL 10.0 的引入，该目录已经被重命名。

其原因是，人们经常会删除 `pg_xlog` 目录的内容，这当然会导致严重的问题和潜在的数据损坏。因此，社区采取了前所未有的步骤重命名 PostgreSQL 实例中的目录。希望能让这个名字足够吓人，没人敢再删除内容。

以下清单显示了 `pg_wal` 目录的外观如下：

```
[postgres@zenbook pg_wal]$ pwd  
/var/lib/pgsql/11/data/pg_wal  
[postgres@zenbook pg_wal]$ ls -l  
total 688132  
-rw----- 1 postgres postgres 16777216 Jan 19 07:58  
00000001000000000000000000CD  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04  
00000001000000000000000000CE  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04  
00000001000000000000000000CF  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04  
00000001000000000000000000D0  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04  
00000001000000000000000000D1  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04  
00000001000000000000000000D2
```

我们可以看到，事务日志是一个由 24 位数字组成的 16 MB 文件。编号是十六进制的。

我们可以看到，`CF` 之后是 `D0`。文件始终是固定大小。



需要注意的一点是，在 PostgreSQL 中，事务日志文件的数量与事务的大小无关。您可以拥有一小组事务日志文件，并且仍可轻松运行 TB 量级的事务。

传统上，WAL 通常由 16 MB 文件组成。但是，自从引入 PostgreSQL 以来，现在可以在 `initdb` 中设置 WAL 段的大小。在某些情况下，这可以加快速度。下面是它的工作原理。

以下示例向我们展示了如何将 WAL 文件大小更改为 32 MB:

```
initdb -D /pgdata --wal-segsize=32
```

了解检查点

正如我之前提到的，每个更改都以二进制格式写入 WAL（它不包含 SQL）。问题是这个数据库服务器不能永远写入 WAL，因为随着时间的推移会消耗越来越多的空间。因此，在某些时候，必须回收事务日志。这是由检查点完成的，该检查点在后台自动发生。

这个想法是，当写入数据时，它首先进入事务日志，然后将脏缓冲区放入共享缓冲区。那些脏缓冲区必须转到磁盘并由后台写进程或检查点写入数据文件。只要写入了到该点的所有脏缓冲区，就可以删除事务日志。



请不要手动删除事务日志文件。在崩溃的情况下，数据库服务器将无法再次启动，所需的磁盘空间量将在新事务进入时回收。千万不要手动触摸事务日志，PostgreSQL 会自己处理事务，在其中做事情确实有害。

优化事务日志

检查点自动发生并由服务器触发。但是，有一些配置设置可决定何时启动检查点：
postgresql.conf 文件中的以下参数负责处理检查点：

```
#checkpoint_timeout = 5min          # range 30s-1d  
#max_wal_size = 1GB  
#min_wal_size = 80MB
```

启动检查点有两个原因：

1. 我们可以耗尽时间，也可以耗尽空间。
2. 两个检查点之间的最长时间由 `checkpoint_timeout` 变量定义。

存储事务日志的空间量将在 `min_wal_size` 和 `max_wal_size` 变量之间变化。PostgreSQL 将自动触发检查点，使得真正需要的空间量将介于这两个数字之间。



`max_wal_size` 变量是一个软限制，PostgreSQL 可能（在重载情况下）暂时需要更多空间。换句话说，如果我们的事务日志在一个单独的磁盘上，那么确保有更多的空间来存储 wal 是有意义的。

如何在 PostgreSQL 9.6 和 10.0 中调整事务日志？在 9.6 中，对后台写进程和检查点机制进行了一些更改。在较旧的版本中，有一些用例从性能的角度来看，较小的检查点距离实际上是有意义的。在 9.6 及更高版本中，这已经发生了很大的变化，更宽的检查点距离基本上总是非常有利的，因为许多优化可以应用于数据库和操作系统级别以加快速度。最值得注意的优化是块在被写出来之前被排序，这大大减少了机械磁盘上的随机 I/O。

但还有更多。大检查点距离实际上会减少创建的 WAL 数量。是的，这是正确的 - 更大的检查点距离将导致更少的 WAL。

这样做的原因是简单的。每当一个块在检查点之后第一次触及时，必须将其完全发送到 WAL。如果更频繁地更改块，则只有更改才会将其更改为日志。较大的检查点距离基本上导致较少的整页写入，这反过来减少了创建的 WAL 的数量。差异可能非常大，可 以 点 在 我 <https://www.postgresql-support.com/checkpoint-distance-and-amount-of-wal/> 上的一篇博文中看到。

PostgreSQL 还允许我们配置检查点是否应该短而强烈，或者它们是否应该在更长的时间内展开。默认值为 0.5，这意味着检查点应该以进程在当前和下一个检查点之间完成的方式完成。以下列表显示了 `checkpoint_completion_target`:

```
#checkpoint_completion_target = 0.5
```

增加这个值基本上意味着检查点会被延长并且强度会降低。

在许多情况下，更高的值被证明有助于消除由密集的检查点引起的 I/O 峰值。

事务日志存档和恢复

在简要介绍了事务日志的一般情况之后，现在是时候关注事务日志归档的过程了。正如我们已经看到的，事务日志包含对存储系统所做的一系列二进制更改。所以，为什么不使用它来复制数据库实例，并做很多其他很酷的事情，比如归档等等？

配置归档

我们在本章中要做的第一件事是创建一个配置来执行标准的时间点恢复（PITR）。使用 PITR 而不是普通转储有几个优点：

- 我们将丢失更少的数据，因为我们可以将数据恢复到某个特定时间点，而不仅仅是固定备份点。
- 恢复将更快，因为索引不必从头创建。它们只是被复制过来并准备好使用。

配置 PITR 很容易。只需在 `postgresql.conf` 文件中进行一些更改：

```
wal_level = replica      # used to be "hot_standby" in older versions
max_wal_senders = 10     # at least 2, better at least 2
```

`wal_level` 变量表示服务器应该生成足够的事务日志以允许 PITR。如果 `wal_level` 变量设置为 `minimal`（这是 PostgreSQL 9.6 的默认值），则事务日志将只包含足够的信息来恢复单个节点 - 它不够丰富，无法处理复制。在 PostgreSQL 10.0 中，默认值已经正确，不再需要更改大多数设置。

`max_wal_senders` 变量将允许我们从服务器流式传输 WAL。它允许我们使用 `pg_basebackup` 创建初始备份而不是传统的基于文件的复制。这里的优点是

`pg_basebackup` 更容易使用。同样，10.0 中的默认值已经过某种改变，因此对于所有设置的 90%，不需要进行任何更改。

WAL 流背后的想法是将创建的事务日志复制到安全的存储位置。基本上，有两种运输方式：

- 使用 `pg_receivewal`（截止 9.6，这称为 `pg_recvexlog`）
- 使用文件系统作为归档的手段

在本节中，我们将介绍如何设置第二个选项。在正常操作期间，PostgreSQL 继续写入那些 WAL 文件。当 `postgresql.conf` 文件中的 `archive_mode = on` 时，PostgreSQL 将为每个文件调用 `archive_command` 变量。

配置可能如下所示。首先，可以创建存储这些事务日志文件的目录：

```
mkdir /archive  
chown postgres.postgres archive
```

可以在 `postgresql.conf` 文件中更改以下条目：

```
archive_mode = on  
archive_command = 'cp %p /archive/%f'
```

重启后将启用归档，但我们首先配置 `pg_hba.conf` 文件以将停机时间减少到绝对最小值。



请注意，我们可以将任何命令放入 `archive_command` 变量中。

许多人使用 `rsync`、`scp` 和其他方法将 `wal` 文件传输到安全的位置。如果我们的脚本返回 0，PostgreSQL 将假定该文件已存档。如果返回任何其他信息，PostgreSQL 将再次尝试存档该文件。这是必要的，因为数据库引擎必须确保没有文件丢失。要执行恢复过程，我们必须提供每个文件；不允许任何单个文件丢失。

配置 `pg_hba.conf` 文件

既然已经成功配置了 `postgresql.conf` 文件，就必须配置 `pg_hba.conf` 文件进行流式传输。请注意，只有在我们计划使用 `pg_basebackup` 时才需要这样做，`pg_basebackup` 是用于创建基本备份的最先进工具。

基本上，我们在 `pg_hba.conf` 文件中的选项与我们在第 8 章管理 PostgreSQL 安全性中看到的选项相同。要记住一个主要问题：

```
# Allow replication connections from localhost, by a user with the  
# replication privilege.  
local  replication  postgres                      trust  
host   replication  postgres  127.0.0.1/32  trust  
host   replication  postgres  ::1/128            trust
```

我们可以定义标准的 `pg_hba.conf` 文件规则。重要的是，第二栏说的是复制。常规规则

还不够，添加显式复制权限确实很重要。同时请记住，我们不必以超级用户的身份这样做。我们可以创建只允许执行登录和复制的特定用户。

同样，PostgreSQL 10 和更高版本已经按照我们在本节中概述的方式进行了配置。必须将现成的远程 IP 添加到 `pg_hba.conf` 时，本地复制才可以工作。

现在已正确配置 `pg_hba.conf` 文件，可以重新启动 PostgreSQL。

创建基本备份

在教 PostgreSQL 归档这些 `wal` 文件之后，是时候创建第一个备份了。我们的想法是有一个备份，并根据该备份重放 `wal` 文件，以达到任何时间点。

要创建初始备份，我们可以转到 `pg_basebackup`，这是一个用于执行备份的命令行工具。让我们调用 `pg_basebackup`，看看它是如何工作的：

```
pg_basebackup -D /some_target_dir  
-h localhost  
--checkpoint=fast  
--wal-method=stream
```

我们可以看到，我们将在这里使用四个参数：

- `-D`: 我们希望基本备份存放在哪里？PostgreSQL 需要一个空目录。在备份结束时，我们将看到服务器数据目录的副本。
- `-h`: 表示数据服务器（源库）的 IP 地址或名称。这是您要备份的服务器。
- `--checkpoint=fast`: 通常，`pg_basebackup` 会等待主服务器检查站。原因是重播进程必须从某个地方开始。检查点确保数据已写入某个点，这样 PostgreSQL 就可以安全地跳转到该点并启动重播进程。基本上，也可以在不使用 `--checkpoint=fast` 参数的情况下完成。但是，在这种情况下，`pg_basebackup` 开始复制数据可能需要一段时间。检查点最多可以间隔一个小时，这可能会不必要的延迟我们的备份。
- `--wal method=stream`: 默认情况下，`pg_basebackup` 连接到主服务器并开始复制文件。现在，请记住，这些文件在复制时会被修改。因此，到达备份的数据不一致。这种不一致性可以在恢复过程中使用 `wal` 进行修复。然而，备份本身并不一致。通过添加`--wal-method=stream` 参数，可以创建一个独立的备份；它可以直接启动，而无需重放事务日志。如果我们只想克隆一个实例而不使用 `pitri`，这是一个很好的方法。幸运的是，`-wal method=stream` 实际上已经是 `postgresql 10.0` 中的默认值。但是，在 9.6 或更早版本中，建议使用名为`-xlog method=stream` 的前身。

减少备份带宽

当 `pg_basebackup` 启动时，它会尝试尽快完成其工作。如果我们有一个良好的网络，`pg_basebackup` 肯定能够从远程服务器每秒获取数百兆字节。如果我们的服务器具有

弱 I / O 系统，则可能意味着 `pg_basebackup` 可以很容易的占用所有资源，最终用户可能会因为其 I/O 请求太慢而体验到糟糕的性能。

要控制最大传输速率，`pg_basebackup` 提供以下内容：

-r, --max-rate=RATE

maximum transfer rate to transfer data directory

(in kB/s, or use suffix "k" or "M")

创建基本备份时，请确保主服务器上的磁盘系统能够承受实际负载。因此，调整我们的传输速率是很有意义的。

映射表空间

通常，如果我们在目标系统上使用相同的文件系统布局，就可以直接调用 `pg_basebackup`。如果不是这样，`pg_basebackup` 允许您将主库的文件系统布局映射到所需的从库文件系统布局：

-T, --tablespace-mapping=OLDDIR=NEWDIR

relocate tablespace in OLDDIR to NEWDIR



如果您的系统很小，那么将所有内容保存在一个表空间中可能是个好主意。

如果 I / O 不是问题（也许是因为您只管理几千兆字节的数据），这是正确的。

使用不同的格式

`pg_basebackup` 可以创建各种格式。默认情况下，它会将数据放在空目录中。本质上，它将连接到源服务器并通过网络连接创建.tar 并将数据放入所需的目录中。

这种方法的问题在于 `pg_basebackup` 将创建许多文件，如果我们想将备份移动到外部备份解决方案（如 Tivoli 存储管理器或其他解决方案），则不适合。以下清单显示了 `pg_basebackup` 支持的有效输出格式：

-F, --format=p|t

output format (plain (default), tar)

要创建单个文件，我们可以使用 `-F = t` 选项。默认情况下，它将创建一个名为 `base.tar` 的文件，然后可以更轻松地进行管理。当然，缺点是我们必须在执行 PITR 之前再次解压缩文件。

测试事务日志存档

在我们深入了解实际的重放过程之前，实际检查归档以确保它正常工作并且正如预期的那样使用简单的“ls”，如下一个清单所示：

```
[hs@zenbook archive]$ ls -l
```

```
total 212996
-rw----- 1 hs hs 16777216 Jan 30 09:04 00000001000000000000000000000001
-rw----- 1 hs hs 16777216 Jan 30 09:04 00000001000000000000000000000002
-rw----- 1 hs hs 302 Jan 30 09:04
0000000100000000000000000002.00000028.backup
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000000000003
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000000000004
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000000000005
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000000000006
...
...
```

只要数据库中存在很多活动连接，就应将 WAL 文件发送到存档。

除了检查文件之外，以下视图也很有用：

```
test=# \d pg_stat_archiver
          View "pg_catalog.pg_stat_archiver"
  Column           |      Type      | Modifiers
-----+-----+-----+
archived_count    | bigint      |
last_archived_wal | text        |
last_archived_time| timestamp with time zone |
failed_count     | bigint      |
last_failed_wal  | text        |
last_failed_time | timestamp with time zone |
stats_reset      | timestamp with time zone |
```

`pg_stat_archiver` 系统视图对于确定归档是否以及何时因任何原因而停滞非常有用。它将告诉我们已归档的文件数 (`archived_count`)。我们还可以看到哪个文件是最后一个文件以及事件发生的时间。最后，`pg_stat_archiver` 系统视图可以告诉我们归档何时出错，这是至关重要的信息。不幸的是，错误代码或错误消息未显示在表中，但由于 `archive_command` 可以是任意命令，因此很容易记录。

归档中还有一件事要看。如前所述，务必确保这些文件已实际存档。但还有更多。当调用 `pg_basebackup` 命令行工具时，我们将在 `wal` 文件流中看到一个`.backup` 文件。它很小，只包含一些关于基本备份本身的信息--它纯粹是信息性的，回放过程不需要它。然而，它给了我们一些重要的线索。稍后开始回放事务日志时，可以删除所有早于`.backup` 文件的 `wal` 文件。在本例中，我们的备份文件名为 `000000010000000000000002.00000028.backup`。这意味着回放过程从文件 `0002`（位置 `28`）中的某个地方开始。这也意味着我们可以删除所有早于…`0002` 的文件。旧的 `wal` 文件将不再需要恢复。请记住，我们可以保留不止一个备份，所以我只指当前的备份。

既然存档正常工作了，我们可以把注意力转移到回放过程上了。

回放事务日志

让我们总结一下到目前为止的过程。我们已经调整了 `postgresql.conf` 文件 (`wal_level`、`max_wal_senders`、`archive_mode` 和 `archive_command`)，并且允许在 `pg_hba.conf` 文件中使用 `pg_basebackup` 命令。然后，重新启动数据库并成功生成基本备份。

请记住，只有在数据库完全运行时才能进行基本备份 - 只需短暂重启即可更改 `max_wal_sender` 和 `wal_level` 变量。

现在系统正常运行，我们可能会遇到我们想要恢复的崩溃。因此，我们可以执行 PITR 以尽可能多地恢复数据。我们要做的第一件事就是把基础备份放在所需的位置。



TIP 保存旧的数据库集群是个好主意。即使它坏了，我们的 PostgreSQL 支持公司也可能需要它来追踪崩溃的原因。你可以在以后删除它，一旦你有了一切，并再次运行。

鉴于前面的文件系统布局，我们可能希望执行以下操作：

```
cd /some_target_dir cp -Rv * /data
```

我们假设新的数据库服务器将位于 `/data` 目录中。在复制基本备份之前，请确保该目录为空。

在下一步中，可以创建名为 `recovery.conf` 的文件。它将包含有关重放过程的所有信息，例如 WAL 存档的位置，我们想要到达的时间等等。



TIP 在 PostgreSQL 10.0 中，`recovery.conf` 文件很可能不存在了。设置应该移动到 `postgresql.conf` 文件。在撰写本书时，并不完全清楚将会发生什么。

这是一个示例 `recovery.conf` 文件：

```
restore_command = 'cp /archive/%f %p'
recovery_target_time = '2019-04-05 15:43:12'
```

将 `recovery.conf` 文件放入 `$ PGDATA` 目录后，我们可以简单地启动我们的服务器。输出可能如下所示：

```
server starting
LOG: database system was interrupted; last known up
      at 2017-01-30 09:04:07 CET
LOG: starting point-in-time recovery to 2019-04-05 15:43:12+02
LOG: restored log file "000000010000000000000002" from archive
LOG: redo starts at 0/2000028
LOG: consistent recovery state reached at 0/20000F8
LOG: restored log file "000000010000000000000003" from archive
LOG: restored log file "000000010000000000000004" from archive
```

```
LOG: restored log file "000000010000000000000005" from archive
...
LOG: restored log file "00000001000000000000000E" from archive
cp: cannot stat '/archive/00000001000000000000000F':
        No such file or directory
LOG: redo done at 0/E7BF710
LOG: last completed transaction was at log time
        2017-01-30 09:20:47.249497+01
LOG: restored log file "00000001000000000000000E" from archive
cp: cannot stat '/archive/00000002.history': No such file or directory
LOG: selected new timeline ID: 2
cp: cannot stat '/archive/00000001.history': No such file or directory
LOG: archive recovery complete
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
LOG: autovacuum launcher started
```

当服务器启动时，需要查找几条消息以确保我们的恢复完美运行：第一个是达到一致的恢复状态。此消息表示 PostgreSQL 可以重放足够的事务日志，以使数据库恢复到使其可用的状态。

然后，PostgreSQL 将一个文件复制到另一个文件并重放它们。但是，请记住我们已经告诉 `recovery.conf` 文件将我们带到 2019 年。本文写于 2017 年，因此显然没有足够的 WAL 达到 2019。因此，PostgreSQL 会出错并告诉我们最后完成的事务。

当然，这只是一个展示，在现实世界的例子中，我们很可能会使用过去的日期，我们可以使用它来安全地恢复。但是，我想告诉你，将来使用日期是完全可行的 - 只要准备好接受错误发生的事情。

恢复完成后，`recovery.conf` 文件将重命名为 `recovery.done`，以便我们可以查看恢复期间的操作。我们的数据库服务器的所有进程都将启动并运行，我们将拥有一个可立即使用的数据库实例。

找到合适的时间戳

到目前为止，我们的进展是假设我们知道要恢复的时间戳，或者我们只是想重放整个事务日志以减少数据丢失。但是，如果我们不想重播所有内容呢？如果我们不知道该恢复到哪个时间点呢？在日常生活中，这实际上是一个非常常见的场景。我们的一个开发人员早上丢失了一些数据，我们应该让事情恢复正常。问题是：早上几点？一旦恢复结束，就无法轻松重新启动。一旦恢复完成，系统将升级，一旦升级，我们就不能继续重放 WAL。

但是，我们可以做的是暂停恢复而不进行升级，检查数据库中的内容，然后继续。

这样做很容易。我们必须确保的第一件事是在 `postgresql.conf` 文件中将 `hot_standby` 变量设置为 `on`。这将确保数据库在仍处于恢复模式时可读。然后，我们必须在开始重放过程之前调整 `recovery.conf` 文件：

```
recovery_target_action = 'pause'
```

有各种 `recovery_target_action` 设置。如果我们使用 `pause`（暂停），PostgreSQL 将在所需的时间暂停，让我们检查已经重播的内容。我们可以调整我们想要的时间，重新启动，然后再试一次。或者，我们可以将值设置为 `promote`（提升）或 `shutdown`（关闭）。

还有第二种方法可以暂停事务日志重播。基本上，它也可以在执行 PITR 时使用。但是，在大多数情况下，它与流复制一起使用。这是在 WAL 重放期间可以完成的任务：

```
postgres=# \x
Expanded display is on.
postgres=# \df *pause*
List of functions
-[ RECORD 1 ]-----+
Schema          | pg_catalog
Name            | pg_is_wal_replay_paused
Result data type | boolean
Argument data types |
Type            | normal
-[ RECORD 2 ]-----+
Schema          | pg_catalog
Name            | pg_wal_replay_pause
Result data type | void
Argument data types |
Type            | normal
postgres=# \df *resume*
List of functions
-[ RECORD 1 ]-----+
Schema          | pg_catalog
Name            | pg_wal_replay_resume
Result data type | void
Argument data types |
Type            | normal
```

我们可以调用 `SELECT pg_wal_replay_pause()`；命令停止 WAL 重放，直到我们调用 `SELECT pg_wal_replay_resume ()`；命令。

想法是弄清楚已经重播了多少 WAL 并在必要时继续。但是，请记住这一点：一旦服务器被提升，我们不能继续重播 WAL 而无需进一步的预防措施。

正如我们已经看到的，弄清楚我们需要恢复的程度可能相当棘手。因此，PostgreSQL 为

我们提供了一些帮助。考虑下面的现实世界的例子：在午夜，我们正在运行一个夜间存储过程，该过程在某些通常未知的点结束。目标是确切地恢复到夜间存储过程的结束时间点。问题在于：我们如何知道流程何时结束？在大多数情况下，这很难弄明白。那么，为什么不在事务日志中添加标记，如下所示：

```
postgres=# SELECT pg_create_restore_point('my_daily_process_ended');
 pg_create_restore_point
-----
 1F/E574A7B8
(1 row)
```

如果我们的进程在结束时立即调用此 SQL 语句，则可以在事务日志中使用此标签通过将以下指令添加到 recovery.conf 文件来精确恢复到此时间点：

```
recovery_target_name = 'my_daily_process_ended'
```

使用此设置而不是 recovery_target_time，重播过程将在晚上运行进程的结束时间点精准结束。

当然，我们也可以重放到某个事务 id。但是，在现实生活中，由于管理员很少知道确切的事务 id，这已经被证明是困难的，因此，这没有太大的实用价值。

清理事务日志存档

到目前为止，数据一直被写入到归档文件中，没有人注意再次清理归档文件以释放文件系统中的空间。PostgreSQL 无法为我们完成此项工作，因为它不知道我们是否要再次使用存档。因此，我们自己负责清理事务日志。当然，我们也可以使用备份工具，但重要的是要知道 postgresql 没有机会为我们进行清理。

假设我们想要清理不再需要的旧事务日志。也许我们希望保留几个基本备份并清除所有不再需要的事务日志来恢复其中一个备份。

在这种情况下，`pg_archivecleanup` 命令行工具正是我们所需要的。我们可以简单地将存档目录和备份文件的名称传递给 `pg_archivecleanup` 命令，它将确保从磁盘中删除文件。使用此工具可以让我们的生活更轻松，因为我们不必确定自己保留哪些事务日志文件。下面是它的工作原理：

```
[hs@asus ~]$ pg_archivecleanup --help
pg_archivecleanup removes older WAL files from PostgreSQL archives.

Usage:
  pg_archivecleanup [OPTION]... ARCHIVELOCATION OLDESTKEPTWALFILE

Options:
  -d generate debug output (verbose mode)
  -n dry run, show the names of the files that would be removed
  -V, --version output version information, then exit
  -x EXT clean up files if they have this extension
  -?, --help show this help, then exit
```

```
For use as archive_cleanup_command in recovery.conf when standby_mode = on:
archive_cleanup_command = 'pg_archivecleanup [OPTION]... ARCHIVELOCATION
%r'
e.g.
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archiverdir %r'
Or for use as a standalone archive cleaner:
e.g.
pg_archivecleanup /mnt/server/archiverdir
00000001000000000000000010.00000020.backup
```

该工具可以轻松使用，并且可在所有平台上使用。

设置异步复制

在看了事务日志归档和 PITR 之后，我们可以将注意力集中在今天 PostgreSQL 世界中使用最广泛的功能之一：流复制。流复制背后的想法很简单。在初始基本备份之后，从服务器可以连接到主服务器并实时获取事务日志并应用它。事务日志重播不再是单个操作，而是一个连续的过程，只要集群存在就持续运行。

执行基本设置

在本节中，我们将学习如何快速轻松地设置异步复制。目标是建立一个由两个节点组成的系统。

基本上，大多数工作已经完成了 WAL 归档。但是，为了便于理解，我们将查看设置流复制的整个过程，因为我们不能假设 WAL 传输确实已经根据需要进行了设置。

首先要做的是转到 `postgresql.conf` 文件并调整以下参数：

```
wal_level = replica
max_wal_senders = 10          # or whatever value >= 2
hot_standby = on              # already a sophistication
```



这些已经是 PostgreSQL 10.0 中的默认选项

正如我们之前所做的那样，必须调整 `wal_level` 变量，以确保 `postgresql` 生成足够的事务日志来支持 `slave`。然后，我们必须配置 `max_wal_senders` 变量。当从库启动并运行或创建基本备份时，`wal` 发送方进程将与客户端的 `wal` 接收方进程对话。`max_wal_senders` 设置允许 `postgresql` 创建足够的进程来服务这些客户端。



TIP 理论上，只要一个 `wal-sender` 进程就足够了。不过，这很不方便。使用 `--wal-method=stream` 参数的基本备份已经需要两个 `wal sender` 进程。如果要同时运行从库和执行基本备份，则已经有三个进程在使用中。因此，请确保允许 `postgresql` 创建足够的进程来防止无意义的重启。

然后是 `hot_standby` 变量。基本上，`master` 忽略 `hot_standby` 变量，并且不考虑它。它所做的就是让从库在 `wal` 回放时可读。那么，我们为什么在乎呢？请记住，`pg_basebackup` 命令将克隆整个服务器，包括其配置。这意味着，如果我们已经在主目录上设置了该值，那么当数据目录被克隆时，从目录将自动获取该值。

设置 `postgresql.conf` 文件后，我们可以将注意力转向 `pg_hba.conf` 文件：只需允许 `slave` 通过添加规则来执行复制。基本上，这些规则与我们已经看到的 PITR 相同。

然后，重启数据库服务器，就像您对 PITR 所做的那样。

然后，可以在从库上调用 `pg_basebackup` 命令。在我们这样做之前，请确保 `/target` 目录为空。如果我们使用 RPM 包部署，请确保关闭可能正在运行的实例并清空目录（例如，`/var/lib/pgsql/data`）：

```
pg_basebackup -D /target
  -h master.example.com
  --checkpoint=fast
  --wal-method=stream -R
```

只需将 `/target` 目录替换为所需的目标目录，并将 `master.example.com` 替换为主目录的 IP 或 DNS 名称。`--checkpoint=fast` 参数将触发即时检查点。然后，有一个`--wal-method = stream` 参数；它将打开两个流。一个将复制数据，另一个将获取在备份运行时创建的 WAL。

最后，还有`-R` 标志：

```
-R, --write-recovery-conf # write recovery.conf after backup
```

`-R` 标志是一个非常好的功能。`pg_basebackup` 命令能够自动创建从属配置。它会在 `recovery.conf` 文件中添加各种条目：

```
standby_mode = on primary_conninfo = '...'
```

第一个设置说 PostgreSQL 应该一直重播 WAL - 如果整个事务日志已被重放，它应该等待新的 WAL 到达。第二个设置将告诉 `postgresql` 主服务器在哪里。这是一个普通的数据库连接。



TIP 从服务器还可以连接到其他从服务器来传输事务日志。只需从从属服务器创建基本备份就可以实现级联复制。所以，`master` 实际上是指这个上下文中的源服务器。

运行 `pg_basebackup` 命令后，可以启动服务。我们应该检查的第一件事是主库是否显示了 `WAL sender` 进程：

```
[hs@linuxpc ~]$ ps ax | grep sender
17873 ? Ss 0:00 postgres: wal sender process
          ah ::1(57596) streaming 1F/E9000060
```

如果是这样，从库也将携带 WAL receiver 进程：

```
17872 ? Ss 0:00 postgres: wal receiver process
streaming 1F/E9000060
```

如果有这些进程，我们就已经走上了正确的轨道，而且复制正在按预期工作。双方现在都在交谈，WAL 从主库传输到从库。

提高安全性

到目前为止，我们已经看到数据以超级用户的身份流传输。但是，允许超级用户从远程访问并不是一个好主意。幸运的是，`postgresql` 允许我们创建一个只允许使用事务日志流但不能执行其他操作的用户。

创建仅用于流式传输的用户很简单：

```
test=# CREATE USER repl LOGIN REPLICATION;
CREATE ROLE
```

通过为用户分配复制权限，可以将其仅用于流式传输 - 禁止其他所有内容。

强烈建议不要使用超级用户帐户设置流媒体。只需将 `recovery.conf` 文件属主更改为新创建的用户。不公开超级用户帐户将极大地提高安全性，就像给复制用户一个密码一样。

暂停和恢复复制

一旦设置了流复制，它就可以完美运行而无需太多的管理员干预。但是，在某些情况下，停止复制并在稍后恢复复制可能是有意义的。为什么有人想这样做？

考虑以下用例：您负责主/从设置，它运行一些垃圾 CMS（内容管理系统）或一些可疑的论坛软件。假设您想将应用程序从糟糕的 cms 1.0 更新到糟糕的 cms 2.0。一些更改将在数据库中执行，这些更改将立即复制到从属数据库。如果升级过程出错了怎么办？由于流式传输，错误将立即复制到所有从节点。

为避免即时复制，我们可以暂停复制并根据需要恢复。在我们的 CMS 更新的情况下，我们可以简单地做以下事情：

1. 停止复制。
2. 在主服务器上执行应用程序更新。
3. 检查我们的应用是否仍然有效。如果是，请恢复复制。如果没有，则故障转移到仍具有旧数据的副本从库。

使用这种机制，我们可以保护我们的数据，因为我们可以返回到问题之前的数据。在本章的后面，我们将学习如何促使从库成为新的主库服务器。

现在的主要问题是：我们如何停止复制？下面是它的工作原理。在备用数据库上执行

以下行：

```
test=# SELECT pg_wal_replay_pause();
```

此行将停止复制。请注意，事务日志仍将从主服务器流向从服务器 - 仅停止重播过程。您的数据仍然受到保护，因为它始终存在于从库服务器上。如果服务器崩溃，则不会丢失任何数据。

请记住，必须在从库上停止重播过程。否则，PostgreSQL 会抛出一个错误：

ERROR: recovery is not in progress

HINT: Recovery control functions can only be executed during recovery.

一旦要恢复复制，在从库上运行以下命令行：

```
SELECT pg_wal_replay_resume();
```

PostgreSQL 将再次开始重播 WAL。

检查复制以确保可用性

每个管理员的核心任务之一是确保复制始终保持正常运行。如果复制已关闭，则在主服务器崩溃时可能会丢失数据。因此，密切关注复制是绝对必要的。

幸运的是，PostgreSQL 提供了系统视图，这使我们可以深入了解正在发生的事情。其中一个视图是 pg_stat_replication：

```
\d pg_stat_replication
```

View "pg_catalog.pg_stat_replication"

Column	Type	Collation	Nullable
Default			
pid	integer		
usesysid	oid		
username	name		
application_name	text		
client_addr	inet		
client_hostname	text		
client_port	integer		
backend_start	timestamp with time zone		
backend_xmin	xid		
state	text		
sent_lsn	pg_lsn		
write_lsn	pg_lsn		
flush_lsn	pg_lsn		
replay_lsn	pg_lsn		
write_lag	interval		

<code>flush_lag</code>	<code>interval</code>			
<code>replay_lag</code>	<code>interval</code>			
<code>sync_priority</code>	<code>integer</code>			
<code>sync_state</code>	<code>text</code>			

`pg_stat_replication` 视图将包含有关发送者的信息。我不想在这里使用 `master` 这个词，因为从库可以连接到其他从库。可以构建服务器树。对于服务器树，主服务器将仅具有与其直接连接的从服务器的信息。

我们将在此视图中看到的第一件事是 `WAL sender` 进程的进程 ID。它可以帮助我们识别出现问题的过程。通常情况并非如此。然后，我们将看到从库用于连接到其发送服务器的用户名。`client_*` 字段将指示从库的位置。我们将能够从这些字段中提取网络信息。`backend_start` 字段显示从属服务器何时开始从我们的服务器流式传输。

然后，有一个神奇的 `backend_xmin` 字段。假设您正在运行主/从设置。可以告诉从机向主机报告其事务 ID。这背后的想法是延迟主服务器上的清理，这样数据就不会从从服务器上运行的事务中获取。

`state` 字段通知我们服务器的状态。如果我们的系统很好，该字段将包含 `streaming`。否则，需要进行更仔细的检查。

接下来的四个字段非常重要。`sent_lsn` 字段（以前称为 `sent_location` 字段）表示 `WAL` 已经到达另一侧的数量，这意味着它们已被 `WAL receiver` 接受。我们可以用它来计算已经有多少数据已经到达了从库。然后，有 `write_lsn` 字段，以前是 `write_location` 字段。一旦接受了 `WAL`，它就会被传递给操作系统。`write_lsn` 字段将告诉我们 `WAL` 的位置已经安全地进入操作系统。`flush_lsn` 字段（以前称为 `flush_location` 字段）将知道数据库已将多少 `WAL` 刷新到磁盘。

最后，还有 `replay_lsn`，以前是 `replay_location` 字段。`WAL` 已经进入从库磁盘的事实并不意味着 PostgreSQL 已经被重放或者最终用户可以看到。假设复制暂停。数据仍将流向备用数据库。但是，它将在以后应用。`replay_lsn` 字段将告诉我们有多少数据已经可见。

在 PostgreSQL 10.0 中，`pg_stat_replication` 中添加了更多字段；`*_lag` 字段表示从库的延迟，并提供了一种查看从库落后多远的方便方法。



这些字段的间隔不同，因此我们可以直接看到时间差。

最后，PostgreSQL 告诉我们复制是同步还是异步。

如果我们还在使用 postgresql 9.6，我们可能会发现用字节计算发送服务器和接收服务器之间的差异是有用的。`*_lag` 字段在 9.6 版本中还没有这样做，因此在字节上有差异是非常有益的。其工作原理如下：

```
SELECT client_addr, pg_current_wal_location() - sent_location AS diff
```

```
FROM pg_stat_replication;
```

在主库上运行时，`pg_current_wal_location()` 函数返回当前事务日志位置。PostgreSQL 9.6 有一个用于事务日志位置的特殊数据类型，名为 `pg_lsn`。它有两个操作符，用于从主库的 `wal` 位置减去从库的 `wal` 位置。因此，此处概述的视图以字节为单位返回两台服务器之间的差异（复制延迟）。



请注意，此语句仅适用于 PostgreSQL 10。此函数曾在旧版本中称为 `pg_current_xlog_location()`。

虽然 `pg_stat_replication` 系统视图包含有关发送方的信息，但 `pg_stat_wal_receiver` 系统视图将在接收方向我们提供类似的信息：

```
test=# \d pg_stat_wal_receiver
          View "pg_catalog.pg_stat_wal_receiver"
   Column    | Type   | Collation | Nullable |
Default
```

Column	Type	Collation	Nullable
Default			
pid	integer		
status	text		
receive_start_lsn	pg_lsn		
receive_start_tli	integer		
received_lsn	pg_lsn		
received_tli	integer		
last_msg_send_time	timestamp with time zone		
last_msg_receipt_time	timestamp with time zone		
latest_end_lsn	pg_lsn		
latest_end_time	timestamp with time zone		
slot_name	text		
sender_host	text		
sender_port	integer		
conninfo	text		

在 `wal receiver` 进程的进程 id 之后，PostgreSQL 将提供进程的状态。然后，`receive_start_lsn` 字段将告诉您 WAL 接收进程启动时的事务日志位置，而 `receive_start_tli` 字段将通知我们 WAL 接收进程启动时使用的时间线。

`received_lsn` 字段包含有关 WAL 位置的信息，该位置已被接收并刷新到磁盘。然后，我们获得了有关时间的信息，以及有关事务槽和连接的信息。

通常，许多人发现读取 `pg_stat_replication` 系统视图比 `pg_stat_wal_receiver` 视图更容易，并且大多数工具都是围绕 `pg_stat_replication` 视图构建的。

执行故障转移并了解时间线

一旦创建了主/从设置，它通常可以在很长一段时间内完美运行。但是，一切都可能失败，因此了解如何使用备份系统替换故障服务器非常重要。

PostgreSQL 使故障转移和升级变得容易。基本上，我们所要做的就是使用 `pg_ctl` 参数告诉从库副本自我提升：

`pg_ctl -D data_dir promote`

从服务器将断开与主服务器的连接，并立即执行提升。记住，在提升时，从库可能已经支持数千个只读连接。PostgreSQL 的一个很好的特性是，在提升执行期间，所有打开的连接都将转换为读/写连接，甚至不需要重新连接。

服务器提升时，PostgreSQL 将增加时间线：如果您设置了一个全新的服务器，它将位于时间线 1 中。如果从服务器克隆了一个从属服务器，它将与主服务器在同一时间线上。所以，这两个框都在时间轴 1 中。如果从库升级为独立主机，它将转到时间线 2。

时间线对 PITR 尤其重要。假设我们在午夜前后建立一个基本备份。上午 12 点，从库被提升了。下午 3 点，宕机了，我们想恢复到下午 2 点。我们将重放在基本备份之后创建的事务日志，并跟随我们所需服务器的 WAL 流，因为这两个节点在上午 12:00 开始分离。

时间轴更改也将显示在事务日志文件的名称中。以下是时间轴 1 中的 WAL 文件示例：

`000000010000000000000000F5`

如果时间轴切换为 2，则新文件名如下：

`000000020000000000000000F5`

如您所见，来自不同时间轴的 WAL 文件理论上可以存在于同一个归档目录中。

管理冲突

到目前为止，我们已经学到很多关于复制的知识。在下一步中，重要的是要看一下复制冲突。出现的主要问题是：如何在一开始就发生冲突？

请考虑以下示例：

主库	从库
----	----

	BEGIN;
	SELECT ... FROM tab WHERE ...
	... running ...
DROP TABLE tab;	... conflict happens ...
	... transaction is allowed to continue for 30 seconds ...
	... conflict is resolved or ends before timeout ...

这里的问题是主服务器不知道从服务器上产生了一个事务。因此，在读取事务结束之前，`drop table` 命令不会阻塞。如果这两个事务发生在同一个节点上，当然会是这样。不过，我们现在看的是两台服务器。`DROP TABLE` 命令将正常执行，同时删除磁盘上这些数据文件的请求将通过事务日志到达从库。从库没有问题：如果从磁盘上删除表，`SELECT` 语句就会终止，如果从库在应用 WAL 之前等待 `SELECT` 语句，它可能会无可救药地落在后面。

理想的解决方案是可以使用配置变量控制的折衷方案：

```
max_standby_streaming_delay = 30s
# max delay before canceling queries
# when reading streaming WAL;
```

我们的想法是在解决冲突之前等待 30 秒，方法是终止对从库的查询。根据我们的应用程序，我们可能希望将此变量更改为或多或少的激进设置。请注意，30 秒用于整个复制流，而不是单个查询。可能是因为某些其他查询已经等待了一段时间，因此单个查询很早就被杀死了。

虽然 `DROP TABLE` 命令显然可以造成冲突，但有些操作不太明显。这是一个例子：

```
BEGIN;
...
DELETE FROM tab WHERE id < 10000;
COMMIT;
...
VACUUM tab;
```

让我们再次假设在从站上发生了长时间运行的 `SELECT` 子句。`DELETE` 子句显然不是这里的问题，因为它只将行标记为已删除 - 它实际上并未将其删除。提交也不是问题，因为它只是将事务标记为已完成。在物理上，行仍在那里。

当 `VACUUM` 之类的操作启动时，问题就开始了。它将销毁磁盘上的行。当然，这些变化将使它到达 WAL，并最终到达从库，这就麻烦了。

为了防止标准 OLTP 工作负载引起的典型问题，PostgreSQL 开发团队引入了一个配置变量：

```
hot_standby_feedback = off
  # send info from standby to prevent
  # query conflicts
```

如果启用此设置，则从服务器将定期向主服务器发送最旧的事务 ID。VACUUM 将知道系统中的某个地方正在进行一个较旧的事务，并将清理时间推迟到稍后的时间点，这时可以安全地清理这些行。事实上，`hot_standby_feedback` 参数在主机上的作用与长事务相同。

如我们所见，`hot_standby_feedback` 参数在默认情况下是关闭的。为什么会这样？好吧，有一个很好的理由：如果它关闭了，从库不会对主库产生真正的影响。事务日志流不消耗大量的 cpu 资源，使得流复制成本低、效率高。但是，如果一个从机（甚至可能不在我们的控制下）让事务打开太长时间，我们的主机可能会因为延迟清理而出现表膨胀。在默认设置中，这比减少冲突更不可取。

使 `hot_standby_feedback = on` 通常会避免 99% 的所有与 OLTP 相关的冲突，如果您的事务需要的时间超过几毫秒，这一点尤为重要。

使复制更可靠

在这一章中，我们已经看到设置复制很容易，不需要太多的努力。然而，总是有一些可能会造成操作上的挑战。其中一个关键问题就是事务日志保留。

请考虑以下情形：

1. 获取基础备份
2. 备份后，一小时没有任何事情发生
3. 从库启动

记住，主库并不太在乎从库的存在。因此，从属服务器启动所需的事务日志可能不再存在于主服务器上，因为它可能已被检查点删除。问题是需要重新同步才能启动从库。对于 TB 级的数据库来说，这显然是一个问题。

此问题的一个潜在解决方案是使用 `wal_keep_segments` 设置：

```
wal_keep_segments = 0          # in logfile segments, 16MB each; 0 disables
```

默认情况下，PostgreSQL 会保留足够的事务日志以应对意外崩溃，但不会更多。使用 `wal_keep_segments` 设置，我们可以告诉服务器保留更多数据，以便从库可以赶上，即使它落后了。

必须记住，服务器不仅因为太慢或太忙而落后，在许多情况下，还因为网络太慢而发生延迟。假设您正在一个 1TB 的表上创建一个索引：PostgreSQL 将对数据进行排序，当实际构建索引时，它也将被发送到事务日志。想象一下，当数百兆的 WAL 通过一条可能只能处理 1 千兆字节字节左右的线路发送时会发生什么。丢失数千兆字节的数据可能是这种情况的结果，并且会在几秒钟内发生。因此，调整 `wal_keep_segments` 设置不应关

注典型延迟，而应关注管理员可容忍的最大延迟（可能是一些安全边际）。

为 `wal_keep_segments` 设置投入相当高的值很有意义，我建议确保总是有足够的数据。

解决事务日志耗尽问题的另一种解决方案是复制事务槽，本章稍后将对此进行介绍。

升级到同步复制

到目前为止，异步复制已经有了相当详细的介绍。但是，异步复制意味着允许在主服务器上提交之后在从服务器上进行提交。如果主服务器崩溃，即使正在进行复制，尚未到达从服务器的数据也可能丢失。

同步复制是为了解决这个问题如果 `postgresql` 使用同步复制，提交必须由至少一个副本刷新到磁盘才能在主服务器上进行。因此，同步复制基本上大大降低了数据丢失的可能性。

在 PostgreSQL 中，配置同步复制很容易。只有两件事要做：

- 调整 master 上 `postgresql.conf` 文件中的 `synchronous_standby_names` 设置
- 将 `application_name` 设置添加到从库中 `recovery.conf` 文件中的 `primary_conninfo` 参数

让我们开始使用 master 上的 `postgresql.conf` 文件：

```
synchronous_standby_names = "
    # standby servers that provide sync rep
    # number of sync standbys and comma-separated
    # list of application_name
    # from standby(s); '*' = all
```

如果我们输入 “*”，则所有节点都将被视为同步候选节点。然而，在现实场景中，只列出几个节点的可能性更大。下面是一个例子：

```
synchronous_standby_names = 'slave1, slave2, slave3'
```

现在，我们必须更改 `recovery.conf` 文件并添加 `application_name`：

```
primary_conninfo = '... application_name=slave2'
```

从库现在将作为 `slave2` 连接到主服务器。主服务器将检查它的配置，并找出 `slave2` 是列表中第一个可行的从服务器。因此，PostgreSQL 将确保只有当从服务器确认事务存在时，主服务器上的提交才会成功。

现在假设 `slave2` 由于某种原因关闭：`postgresql` 将尝试将另外两个节点中的一个转换为同步备用节点。问题是：如果没有其他服务器怎么办？

在这种情况下，如果事务应该是同步的，PostgreSQL 将永远等待提交。是的，这是真的。除非至少有两个可用节点可用，否则 PostgreSQL 不会继续提交。请记住，我们

已经要求 PostgreSQL 在至少两个节点上存储数据 - 如果我们无法在任何给定的时间点提供足够的主机，那就是我们的错。实际上，这意味着最好使用至少三个节点（一个主节点和两个从节点）实现同步复制，因为总有一个主机丢失的可能性。

谈到主机故障，有一点需要注意 - 如果同步伙伴在提交进入时死亡，PostgreSQL 将等待它返回。或者，同步提交可以与其他潜在的同步伙伴一起发生。最终用户可能甚至没有注意到同步伙伴发生了变化。

在某些情况下，仅在两个节点上存储数据可能还不够：我们可能希望进一步提高安全性并将数据存储在更多节点上。为此，我们可以在 PostgreSQL 9.6 或更高版本中使用以下语法：

```
synchronous_standby_names =
    '4(slave1, slave2, slave3, slave4, slave5, slave6)'
```

在这种情况下，在主库确认提交之前，数据应该在六个节点中的四个节点上存在。

当然，这带有价格标签 - 请记住，如果我们添加越来越多的同步从库，速度将会下降。没有免费午餐这样的东西。PostgreSQL 提供了几种方法来控制性能开销，我们将在下一节中讨论。

在 PostgreSQL 10.0 中，添加了更多功能：

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

引入了 ANY 和 FIRST 关键字。FIRST 允许您设置服务器的优先级，而 ANY 在提交同步事务时为 PostgreSQL 提供了更多的灵活性。

调整耐用性

在本章中，我们已经看到数据可以同步或异步复制。但是，这不是一个全球性的事情。为了确保良好的性能，PostgreSQL 允许我们以非常灵活的方式配置。可以同步或异步复制所有内容，但在许多情况下，我们可能希望以更细粒度的方式执行操作。这恰好是需要 `synchronous_commit` 设置的时候。

假设已配置了同步复制，`recovery.conf` 文件中的 `application_name` 设置以及 `postgresql.conf` 文件中的 `synchronous_standby_names` 设置，则 `synchronous_commit` 设置将提供以下选项：

- `off`: 这基本上是一个异步复制。WAL 不会立即刷新到主机上的磁盘，主库不会等待从库将所有内容写入磁盘。如果主服务器出现故障，某些数据可能会丢失（最多三次 - `wal_writer_delay`）
- `local`: 事务日志在 `master` 上提交时刷新到磁盘。但是，主服务器不会等待从服务器（异步复制）。
- `remote_write`: `remote_write` 设置已经使 PostgreSQL 同步复制。但是，只有主

服务器将数据保存到磁盘。对于从库，将数据发送到操作系统就足够了。我们的想法是不要等待第二次磁盘刷新以加快速度。两个存储系统几乎不可能在同一时间崩溃。因此，数据丢失的风险接近于零。

- **on**: 在这种情况下，如果主服务器和从服务器已成功将事务刷新到磁盘，则事务是可以的。除非数据安全地存储在两台服务器上（或更多，具体取决于配置），否则应用程序将不会收到提交。
- **remote_apply**: 虽然 **on** 确保数据安全地存储在两个节点上，但它并不保证我们可以立即简单地实现负载平衡。数据在磁盘上刷新的事实并不能确保用户已经可以看到数据。例如，如果发生冲突，从服务器将停止事务重播，但是，在冲突期间，事务日志仍会发送到从服务器并刷新到磁盘。简而言之，即使最终用户还看不到数据，也可能发生数据在从库上刷新的情况。**remote_apply** 选项修复了此问题。它确保数据必须在副本上可见，以便可以在从库设备上安全地执行下一个读取请求，从库设备可以看到对主库设备所做的更改并将其公开给最终用户。当然，**remote_apply** 选项是复制数据的最慢方式，因为它要求从服务器已经将数据公开给最终用户。

在 PostgreSQL 中，**synchronous_commit** 参数不是全局值。它可以在各种级别进行调整，就像许多其他设置一样。我们可能想要做如下的事情：

```
test=# ALTER DATABASE test SET synchronous_commit TO off;  
ALTER DATABASE
```

有时，只有一个数据库应该以某种方式复制。如果我们作为特定用户连接，也可以同步复制。最后但并非最不重要的是，也可以告诉单个事务如何提交。通过动态调整 **synchronous_commit** 参数，甚至可以在每个事务级别上控制事物。

例如，请考虑以下两种情况：

- 写入一个日志表，我们可能希望在其中使用异步提交，因为我们希望尽快
- 在我们希望安全的地方存储信用卡支付，因此同步交易可能是我们想要的

我们可以看到，完全相同的数据库可能有不同的要求，具体取决于修改的数据。因此，在事务级别更改参数设置非常有用，有助于提高速度。

利用复制槽

在介绍了同步复制和动态可调的持久性之后，我想专注于一个称为复制槽的功能。

复制槽的目的是什么？让我们考虑以下示例：有一个主服务器和一个从服务器。在主服务器上，执行大型事务，并且网络连接速度不够快，无法及时发送所有数据。在某些时候，主服务器会删除其事务日志（检查点）。如果从库设备太远，则需要重新同步。正如我们已经看到的，**wal_keep_segments** 设置可用于降低复制失败的风险。问题是：**wal_keep_segments** 设置的最佳值是多少？当然，越多越好，但最好的是多少？

复制槽将为我们解决这个问题：如果我们使用的是复制槽，主服务器只能在事务日志被所有副本使用后循环使用它。这里的优点是，从库永远不会落后到需要重新同步的程度。

问题是，假设我们在没有告诉主库的情况下关闭了从库。 主服务器将永久保留事务日志，主服务器上的磁盘最终将填满，从而导致不必要的停机时间。

为了降低主服务器的这种风险，复制插槽只能与适当的监视和警报一起使用。只需关注可能导致问题或不再使用的开放复制插槽。

在 PostgreSQL 中，有两种类型的复制槽：

- 物理复制槽
- 逻辑复制槽

物理复制槽可用于标准流复制。他们将确保数据不会过早回收。逻辑复制槽也可以做同样的事情。但是，它们用于逻辑解码。逻辑解码背后的想法是让用户有机会附加到事务日志并使用插件对其进行解码。因此，逻辑事务槽是数据库实例的某种 `tail -f`。它允许用户提取对数据库的更改，从而提取对事务日志的更改-以任何格式和任何目的。在许多情况下，逻辑复制槽用于逻辑复制。

处理物理复制槽

要使用复制槽，必须对 `postgresql.conf` 文件进行更改：

```
wal_level = logical  
max_replication_slots = 5      # or whatever number is needed
```

使用物理槽，逻辑尚不必要 - 复制就足够了。但是，对于逻辑插槽，我们需要更高的 `wal_level` 设置。然后，如果我们使用 PostgreSQL 9.6 或更低版本，则必须更改 `max_replication_slots` 设置。PostgreSQL 10.0 已经有了改进的默认设置。基本上，只需输入一个符合我们目的的数字。我的建议是添加一些备用插槽，以便我们可以轻松地连接更多的用户，而无需重新启动服务器。

重启后，可以创建插槽：

```
test=# \x  
Expanded display is on.  
postgres=# \df *create*physical*slot*  
List of functions  
-[ RECORD 1 ]-----  
Schema          | pg_catalog  
Name           | pg_create_physical_replication_slot  
Result data type | record  
Argument data types | slot_name name,  
                      | immediately_reserve boolean DEFAULT false,  
                      | temporary boolean DEFAULT false,  
                      | OUT slot_name name,  
                      | OUT lsn pg_lsn  
Type           | func
```

`pg_create_physical_replication_slot` 函数用于帮助我们创建插槽。可以使用以下两个参数之一调用它：如果只传递了插槽名称，则第一次使用时插槽将处于活动状态。如果将 `true` 作为第二个参数传递，则插槽将立即开始保存事务日志：

```
test=# SELECT * FROM pg_create_physical_replication_slot('some_slot_name',
true);
slot_name      | lsn
-----+-----
some_slot_name | 0/EF8AD1D8
(1 row)
```

要查看主服务器上哪些插槽处于活动状态，请考虑运行以下 SQL 语句：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-----+
slot_name      | some_slot_name
plugin         |
slot_type      | physical
datoid         |
database       |
temporary      | f
active          | f
active_pid     |
xmin           |
catalog_xmin   |
restart_lsn    | 0/1653398
confirmed_flush_lsn |
```

该视图将告诉我们很多关于插槽的信息。它包含有关正在使用的插槽类型，事务日志位置等信息。

要使用插槽，我们所要做的就是将它添加到 `recovery.conf` 文件中：

```
primary_slot_name = 'some_slot_name'
```

重新启动流后，将直接使用插槽并保护复制。如果我们不再需要我们的插槽，我们可以轻松删除它：

```
test=# \df *drop*slot*
List of functions
-[ RECORD 1 ]-----+
Schema          | pg_catalog
Name            | pg_drop_replication_slot
Result data type | void
Argument data types | name
```

Type	normal
------	--------

当一个插槽被丢弃时，逻辑插槽和物理插槽之间就没有区别了。只需将插槽名称传递给函数并执行即可。



TIP 删除时不允许任何人使用该插槽。否则，PostgreSQL 会因为充分的理由而出错。

处理逻辑复制槽

逻辑复制槽对逻辑复制至关重要。由于本章中的空间限制，很遗憾无法涵盖逻辑复制的所有方面。但是，我想概述一些基本概念，这些概念对于逻辑解码和逻辑复制是必不可少的。

如果我们想要创建一个复制槽，这是它的工作原理。这里需要的功能有两个参数：第一个将定义复制槽的名称，而第二个带有用于解码事务日志的插件。它将确定 PostgreSQL 用于返回数据的格式：

```
test=# SELECT *
  FROM pg_create_logical_replication_slot('logical_slot',
'test_decoding');
slot_name      | lsn
-----+-----
logical_slot  | 0/EF8AD4B0
(1 row)
```

我们可以使用与之前相同的命令检查插槽是否存在。要检查插槽的真正功能，可以创建一个小测试：

```
test=# CREATE TABLE t_demo (id int, name text, payload text);
CREATE TABLE
test=# BEGIN;
BEGIN
test=# INSERT INTO t_demo
VALUES (1, 'hans', 'some data');
INSERT 0 1
test=# INSERT INTO t_demo VALUES (2, 'paul', 'some more data');
INSERT 0 1
test=# COMMIT;
COMMIT
test=# INSERT INTO t_demo VALUES (3, 'joe', 'less data');
INSERT 0 1
```

请注意，执行了两个事务。现在可以从插槽中提取对这些事务所做的更改：

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
pg_logical_slot_get_changes
```

```
-----  
(0/EF8AF5B0,606546,"BEGIN 606546")  
(0/EF8CCCA0,606546,"COMMIT 606546")  
(0/EF8CCCD8,606547,"BEGIN 606547")  
(0/EF8CCCD8,606547,"table public.t_demo: INSERT: id[integer]:1  
    name[text]:'hans' payload[text]:'some data'")  
(0/EF8CCD60,606547,"table public.t_demo: INSERT: id[integer]:2  
    name[text]:'paul' payload[text]:'some more data'")  
(0/EF8CCDE0,606547,"COMMIT 606547")  
(0/EF8CCE18,606548,"BEGIN 606548")  
(0/EF8CCE18,606548,"table public.t_demo: INSERT: id[integer]:3  
    name[text]:'joe' payload[text]:'less data'")  
(0/EF8CCE98,606548,"COMMIT 606548")  
(9 rows)
```

这里使用的格式取决于我们之前选择的输出插件。 PostgreSQL 有各种输出插件，比如 wal2json。



如果使用默认值，则逻辑流将包含实数值而不仅仅是函数。 逻辑流具有最终在基础表中的数据。

另外，请记住，插槽在使用后不会再返回数据：

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);  
pg_logical_slot_get_changes  
-----  
(0 rows)
```

因此，第二次调用的结果集为空。 如果我们想要重复获取数据，PostgreSQL 提供了 pg_logical_slot_peek_changes 函数。 它的工作原理与 pg_logical_slot_get_changes 函数类似，但确保数据仍可在插槽中使用。

当然，使用普通 SQL 并不是使用事务日志的唯一方法。 还有一个名为 pg_recvlogical 的命令行工具。 它可以与在整个数据库实例上执行 tail -f 进行比较，并实时接收数据流。

让我们开始 pg_recvlogical 命令：

```
[hs@zenbook ~]$ pg_recvlogical -S logical_slot -P test_decoding  
-d test -U postgres --start -f -
```

在这种情况下，该工具连接到测试数据库并使用来自 logical_slot 的数据。 -f 表示将流发送到 stdout。 让我们删除一些数据：

```
test=# DELETE FROM t_demo WHERE id < random()*10;  
DELETE 3
```

更改将使其进入事务日志。 但是，默认情况下，数据库只关心删除后表的状态。 它知道

必须触摸哪些块等等，但它不知道以前是什么：

```
BEGIN 606549
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
COMMIT 606549
```

因此，输出是毫无意义的。要解决这个问题，需要采取以下措施：

```
test=# ALTER TABLE t_demo REPLICA IDENTITY FULL;
ALTER TABLE
```

如果使用数据重新填充表并再次删除，则事务日志流将如下所示：

```
BEGIN 606558
table public.t_demo: DELETE: id[integer]:1 name[text]:'hans'
    payload[text]:'some data'
table public.t_demo: DELETE: id[integer]:2 name[text]:'paul'
    payload[text]:'some more data'
table public.t_demo: DELETE: id[integer]:3 name[text]:'joe'
    payload[text]:'less data'
COMMIT 606558
```

现在，所有的变化都在。

逻辑插槽的用例

复制插槽有多种使用情形。最简单的用例如图所示。数据可以以所需的格式从服务器获取，并用于审核、调试或简单地监视数据库实例。

当然，下一个合乎逻辑的步骤是获取此更改流并将其用于复制。

像双向复制（BDR）这样的解决方案完全基于逻辑解码，因为二进制级别的更改不适用于多主复制。

最后，需要在不停机的情况下进行升级。请记住，二进制事务日志流不能用于在不同版本的 PostgreSQL 之间进行复制。因此，PostgreSQL 的未来版本将支持一个名为 pglogical 的工具，该工具有助于在不停机的情况下进行升级。

使用 CREATE PUBLICATION 和 CREATE SUBSCRIPTION

对于 10.0 版本，PostgreSQL 社区创建了两个新命令：CREATE PUBLICATION 和 CREATE SUBSCRIPTION。这些可用于逻辑复制，这意味着您现在可以选择性地复制数据并实现接近零的停机升级。到目前为止，已完全覆盖二进制复制和事务日志复制。但是，有时候，我们可能不想复制整个数据库实例 - 复制一两个表可能就足够了。这正是逻辑复制正确使用的时候。

在开始之前，首先要做的是在 postgresql.conf 中将 wal_level 更改为 logical 并重启：

wal_level = logical

然后，我们可以创建一个简单的表：

```
test=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

第二个数据库中也必须存在相同的表结构才能使其工作。 PostgreSQL 不会自动为我们创建这些表：

```
test=# CREATE DATABASE repl;
CREATE DATABASE
```

创建数据库后，可以添加相同的表：

```
repl=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

这里的目标是将测试数据库中 t_test 表的内容发布到其他地方。 在这种情况下，它将简单地复制到同一实例上的数据库中。 要发布这些更改，PostgreSQL 提供了 CREATE PUBLICATION 命令：

```
test=# \h CREATE PUBLICATION
Command: CREATE PUBLICATION
Description: define a new publication
Syntax:
  CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
    | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

语法实际上非常简单。 我们需要的只是一个名称和系统应该复制的所有表的列表：

```
test=# CREATE PUBLICATION pub1 FOR TABLE t_test;
CREATE PUBLICATION
```

在下一步中，可以创建订阅。 再次，语法非常简单，非常简单：

```
test=# \h CREATE SUBSCRIPTION
Command: CREATE SUBSCRIPTION
Description: define a new subscription
Syntax:
  CREATE SUBSCRIPTION subscription_name
    CONNECTION 'conninfo'
    PUBLICATION publication_name [, ...]
    [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

基本上，直接创建订阅绝对没有问题。 但是，如果我们在从测试数据库到 repl 数据库

在同一实例中玩这个游戏，则需要手动创建正在使用的复制槽。否则，CREATE SUBSCRIPTION 将永远不会完成：

```
test=# SELECT pg_create_logical_replication_slot('sub1', 'pgoutput');
pg_create_logical_replication_slot
-----
(sub1,0/27E2B2D0)
(1 row)
```

在这种情况下，在 master 数据库上创建的插槽名称称为 sub1。然后，我们需要连接到目标数据库并运行以下命令：

```
repl=# CREATE SUBSCRIPTION sub1
  CONNECTION 'host=localhost dbname=test user=postgres'
  PUBLICATION pub1
  WITH (create_slot = false);
CREATE SUBSCRIPTION
```

当然，我们必须调整我们的数据库连接参数。然后，PostgreSQL 将同步数据，我们将完成。



请注意，仅使用 `create_slot = false`，因为测试在同一数据库服务器实例中运行。如果我们碰巧使用不同的数据库，则无需手动创建插槽，也不需要 `create_slot = false`。

小结

在本章中，我们了解了 PostgreSQL 复制的最重要功能，例如流复制和复制冲突。然后我们了解了 PITR 以及复制槽。请注意，关于复制的书籍永远不会完整，除非它跨越大约 400 页左右，但我们已经了解了每个管理员应该知道的最重要的事情。

在第 11 章，决定有用的扩展，将是关于 PostgreSQL 的有用扩展。我们将了解业界广泛采用的扩展，并提供更多功能。

QA

逻辑复制的目的是什么？

如果您使用的是二进制复制，主版本和从版本必须使用相同的 PostgreSQL 主版本。换言之：不能使用流式处理使用事务日志流从 PostgreSQL10 升级到 PostgreSQL11。逻辑复制有助于弥补这一差距。除此之外，逻辑复制还可以帮助有选择地将数据复制到各种系统。

同步复制的性能影响是什么？

同步复制将比二进制复制慢。一般来说，短事务对性能的影响要比长事务大得多。由于性能的下降很大程度上取决于网络延迟，因此很难精确地说出一个数字。网络越慢-同

步复制越慢。

为什么不总是使用同步复制？

仅在绝对有必要减少对速度和可用性的影响时才尝试使用同步复制。在大多数情况下，异步复制完全正常，并且完全符合大多数应用程序的要求。谨慎使用同步复制。

11. 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、事务日志传送和逻辑解码。在研究了大部分与管理相关的主题之后，现在的目标是针对更广泛的主题。在 PostgreSQL 世界中，很多事情都是通过扩展来完成的。扩展的优点是可以添加功能，而不会使 PostgreSQL 核心膨胀。人们可以从有时相互竞争的扩展中进行选择，找到最适合自己的扩展。其理念是保持核心纤细，相对容易维护，并为未来做好准备。

在本章中，将讨论 PostgreSQL 的一些最广泛的扩展。但是，在深入研究这个问题之前，我想说明这一章只列出了我认为有用的扩展列表。现在有太多的模块，不可能以合理的方式涵盖所有模块。这些东西每天都会发布，有时甚至很难让专业人士了解那里的一切。我们发言时正在发布新的扩展，看看 PGXN (<https://pgxn.org/>) 可能是一个好主意，它包含了各种 PostgreSQL 扩展。

在本章中，我们将介绍以下主题：

- 扩展如何工作
- 精选的 contrib 模块
- 快速浏览与 GIS 相关的模块
- 其他有用的扩展

请注意，仅涵盖最重要的扩展。

了解扩展如何工作

在深入研究现有的扩展之前，最好先看看扩展是如何工作的。了解扩展的内部工作原理会非常有益。

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 ]
        [ FROM old_version ]
        [ CASCADE ]
```

如果要部署扩展，只需调用 CREATE EXTENSION 命令即可。它将检查扩展并将其加载到您的数据库中。请注意，扩展将加载到某个数据库中，而不是加载到整个数据库实例中。

如果我们正在加载扩展，我们可以决定我们想要使用的模式。可以重新定位许多扩展，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，我们不想部署最新版本的扩展，因为客户端可能正在运行过时的软件。在这种情况下，可以方便地在系统上部署任何可用的版本。

FROM old_version 子句需要更多关注。在过去，PostgreSQL 不支持扩展，所以很多未打包的代码仍然存在。此选项使 CREATE EXTENSION 子句运行一个替代安装脚本，该脚本将现有对象吸收到扩展中，而不是创建新对象。确保 SCHEMA 子句指定包含这些预先存在的对象的模式。只有在有旧模块的情况下才能使用它。

最后，还有 CASCADE 子句。某些扩展依赖于其他扩展。CASCADE 选项也将自动部署这些软件包。这是一个例子：

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

地球距离模块实现了大圆距离计算。你可能知道，地球上两点之间的最短距离不是直线；相反，飞行员必须不断调整他/她的航线，以找到从一个点飞到另一个点的最快路线。事实是，地球距离扩展取决于立方体扩展，它允许您在球体上执行操作。

要自动部署依赖项，可以使用 CASCADE 子句，如上所述：

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

在这种情况下，将部署所有依赖的扩展。

检查可用的扩展名

PostgreSQL 提供了各种视图来确定系统上的扩展以及实际部署的扩展。其中一个视图是 pg_available_extensions：

```
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 ]
[ FROM old_version ]
[ CASCADE ]
```

它包含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。为了方便最终用户，还提供了一个描述，告诉我们有关扩展的更多信息。

以下清单包含从 pg_available_extensions 获取的两行：

```
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

```

如您所见，earthdistance 和 plpgsql 扩展都在我的数据库中启用。默认情况下，plpgsql 扩展名已经存在，并且添加了 earthdistance。此视图的优点在于您可以快速了解已安装的内容和可安装的内容。

但是，在某些情况下，扩展程序不仅仅适用于一个版本。要了解有关版本控制的更多信息，请考虑查看以下视图：

```

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

```

这里提供了一些更详细的信息，如下面的清单所示：

```

test=# SELECT * FROM pg_available_extension_versions LIMIT 1;
-[ RECORD 1 ]-----+
name | earthdistance
version | 1.1
installed | t
superuser | t
relocatable | t
schema |
requires | {cube}
comment | calculate great-circle distances on the surface of the Earth

```

PostgreSQL 还将告诉您扩展是否可以重新定位，它部署在哪个模式中，以及需要哪些其他扩展。然后是描述扩展的注释，如前所示。

有人可能会想，PostgreSQL 在哪里找到有关系统扩展的所有信息？假设您已经从官方 PostgreSQL RPM 存储库部署了 PostgreSQL 10.0，`/usr/pgsql-11/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--unpacked--1.0.sql
...
...
```

`citext`（不区分大小写的文本）扩展名的默认版本是 1.4，因此有一个名为 `citext` 的文件 - 1.3.sql。除此之外，还有一些文件用于从一个版本移动到另一个版本（1.0→1.1, 1.1→1.2，依此类推）。

然后，有`.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
```

此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。请注意，这些内容将显示在我们刚才讨论的系统视图中。访问这些视图时，PostgreSQL 将转到该目录并读取所有`.control` 文件。然后，有默认版本和二进制文件的路径。

如果要从 RPM 安装一个典型的扩展，那么目录将是`$libdir`，它位于 PostgreSQL 二进制目录中。但是，如果您已经编写了自己的商业扩展，那么它很可能位于其他地方。

最后一个设置将告诉 PostgreSQL 扩展是否可以驻留在任何模式中，或者它是否必须位于固定的预定义模式中。

最后是解压文件。以下是摘录：

```
...
```

```
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);
...

```

解包文件将把现有代码转换为扩展名。因此，整合数据库中的现有内容非常重要。

使用 contrib 模块

现在我们已经看了扩展的理论介绍，现在是时候看看一些最重要的扩展了。在本节中，您将了解作为 postgresql contrib 模块的一部分提供给您的模块。在安装 postgresql 时，我建议您始终安装这些 contrib 模块，因为它们包含一些重要的扩展，这些扩展确实可以让您的生活更轻松。

在下一节中，您将了解一些我认为最有趣和最有用的方法，这些方法有各种原因(调试、性能调整等)。

使用 adminpack 模块

adminpack 模块背后的思想是为管理员提供一种无需 ssh 访问即可访问文件系统的方法。这个包包含两个函数来实现这一点。

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 模块最有趣的特性之一是能够检查日志文件。pg_logdir_ls 函数检查日志目录并返回日志文件列表：

```
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'
```

这里最重要的是 log filename 参数必须根据 adminspack 模块的需要进行调整。如果您碰巧运行从 postgresql 存储库下载的 rpm，则 log_filename 参数定义为 postgresql-%a，在本例中，必须更改该参数以避免错误。

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
-----+-----
2017-03-03 16:32:58 | pg_log/postgresql-2017-03-03_163258.log
```

(1 row)

还可以确定磁盘上文件的大小。 这里是一个例子：

```
test=# SELECT b, pg_catalog.pg_file_length(b)
  FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
          b           | pg_file_length
-----+
pg_log/postgresql-2017-03-03_163258.log | 1525
(1 row)
```

除了这些功能外，模块还提供了更多功能：

```
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)
```

您可以读取，写入，重命名或删除文件。



当然，这些功能只能由超级用户调用。

应用 bloom 过滤器

从 PostgreSQL 9.6 开始，就可以使用扩展动态添加索引类型。新的 CREATE ACCESS METHOD 命令以及一些附加功能使动态创建全功能和事务日志索引类型成为可能。

bloom 扩展为 PostgreSQL 用户提供了 bloom 过滤器，这些过滤器有助于尽快有效地减少数据量。布隆过滤器背后的想法是计算位掩码并将位掩码与查询进行比较。布隆过滤器可能会产生一些误报，但仍会显着减少数据量。

当表由数百列和数百万行组成时，它尤其有用。使用 b-tree 索引数百个列是不可能的，因此布隆过滤器是一个很好的选择，因为它允许一次索引所有内容。

为了理解其工作原理，我们将安装扩展：

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

在以下步骤中，将创建包含各种列的表：

```
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
```

为了简化这些操作，这些列具有默认值，以便可以使用如下简单的 SELECT 子句轻松添加数据：

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

该查询向表中添加了 100 万行。现在，表可以添加索引：

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

请注意，索引一次包含九列。与 b 树相比，这些列的顺序并没有真正的区别。



注意，我们刚刚创建的表大约是 65MB，没有索引。

该索引又为存储空间增加了 15 MB：

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

布隆过滤器的美妙之处在于可以查找列的任意组合：

```
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，你所看到的一切让人感觉很特别。一个自然产生的问题是：为什么不总是使用 bloom 过滤器？原因很简单，数据库必须读取整个 bloom 过滤器才能使用它。例如，对于 b-tree，这是不必要的。

将来，最有可能添加更多的索引类型，以确保 PostgreSQL 可以覆盖更多的用例。

如果您想了解有关布隆过滤器的更多信息，请参阅 <https://www.cybertec-postgresql.com/en/trying-out-postgres-bloom-indexes/>，阅读我们的博文。

部署 btree_gist 和 btree_gin

可以添加更多与索引相关功能。在 PostgreSQL 中，有一个操作符类的概念，这已在第 3 章“使用索引”中讨论过。

contrib 模块提供了两个扩展（即 btree-gist 和 btree-gin），用于向 GIST 和 GIN 索引添加 b-tree 功能。为什么这么有用？GIST 索引提供了各种 b 树不支持的特性。其中一个特性是执行 k 近邻（KNN）搜索的能力。

为什么这有关系？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，可能很难提出界限，例如，如果有人正在寻找价格约为 70 欧元的产品。KNN 可能会出手相救。这里是一个例子：

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

在下一步中，添加了一些简单数据：

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

现在，可以添加扩展了：

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

向列中添加 GIST 索引很容易。只需使用 USING GIST 子句。注意，在整数列上添加 GIST 索引只在扩展名存在时有效。否则，PostgreSQL 将报告没有合适的运算符类：

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

CREATE INDEX

部署索引后，可以按距离排序：

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

如您所见，第一行数据是完全匹配的。接下来的匹配已经不那么精确，而且越来越糟。查询将始终返回固定数量的行。

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)
```

正如您所看到的，PostgreSQL 直接进行索引扫描，从而显著加快了查询速度。

在 PostgreSQL 的未来版本中，b-tree 很可能也支持 KNN 搜索。添加此功能的补丁已添加到开发邮件列表中。也许它最终会成为核心内容。将 KNN 作为 b 树特征可能最终导致标准数据类型上的 GiST 索引更少。

Dblink - 考虑逐步淘汰

使用数据库链接的愿望已经存在很多年了。然而，大约在本世纪初，PostgreSQL 外部数据包装器甚至还没有出现，传统的数据库链接实现也看不到。大约在这个时候，一位来自加利福尼亚的 PostgreSQL 开发人员（joe conway）率先在 PostgreSQL 中引入了 dblink

的概念，开始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，但它已不再是最先进的。

因此，建议从 `dblink` 转移到更现代的 `SQL / MED` 实现（这是一种定义外部数据可以集成在关系数据库中的方式的规范）。`postgres_fdw` 扩展是在 `SQL / MED` 之上构建的，它提供的不仅仅是数据库连接，因为它允许您连接到基本上的任何数据源。

使用 `file_fdw` 获取文件

在某些情况下，从磁盘读取文件并将其作为表公开给 PostgreSQL 是有意义的。这正是使用 `file_fdw` 扩展可以实现的功能。我们的想法是拥有一个模块，允许您从磁盘读取数据并使用 SQL 进行查询。

安装模块按预期工作：

```
CREATE EXTENSION file_fdw;
```

在下面的步骤中，我们创建一个虚拟服务器：

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

`file_server` 基于 `file_fdw` 扩展名外部数据包装器，它告诉 PostgreSQL 如何访问该文件。

要将文件公开为表，可以使用以下命令：

```
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
':');
```

在此示例中，将显示`/etc/passwd` 文件。必须列出所有字段，并且必须相应地映射数据类型。所有其他重要信息都使用选项传递给模块。在这个例子中，PostgreSQL 必须知道文件的类型（文本），名称和文件的路径，以及分隔符。也可以告诉 PostgreSQL 是否有标题。如果设置为 `true`，则将跳过第一行而不重要。如果您碰巧加载 CSV 文件，则跳过标题尤其重要。

创建表后，可以读取数据：

```
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
```

在查看执行计划时，您会看到 PostgreSQL 使用通常称为“外部扫描”的东西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大小等等。由于我们谈论的是执行计划生成器（planner），有一点值得一提。PostgreSQL 甚至可以获取该文件的统计信息。规划器检查文件大小并为文件分配与相同大小的普通 PostgreSQL 表相同的成本。

使用 pageinspect 检查存储

如果您遇到存储损坏或可能与表中的坏块相关的其他存储相关问题，则 pageinspect 扩展可能是您要查找的模块。我们将从创建扩展开始，如下一个示例所示：

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

pageinspect 背后的想法是为您提供一个模块，允许您在二进制级别上检查表。

使用模块时，最重要的是获取块：

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

该函数将返回单个块。在前面的示例中，它是参数传递中 pg_class 对象的第一个块，当然 pg_class 它是一个系统表。当然用户可以也选择任何其他表。

接下来，您可以提取页头：

```
test=# \x
Expanded display is on.

test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
-[ RECORD 1 ]-----
lsn          | 1/35CAE5B8
checksum     | 0
flags        | 1
lower        | 240
upper        | 1288
special      | 8192
pagesize    | 8192
version      | 4
prune_xid   | 606562
```

它已经包含了很多关于这个页面的信息。如果您想了解更多信息，可以调用 heap-page-items 函数，该函数将对页进行剖分并返回每个元组一行记录：

```
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;
```

要读取数据，我们必须熟悉 PostgreSQL 的磁盘存储格式。否则，数据可能看起来很模糊。

`pageinspect` 为所有访问方法（表，索引等）提供函数，并允许详细分析存储。

使用 pg_buffercache 调查缓存

在简要介绍 `pageinspect` 扩展之后，我们将把重点转向 `pg_buffercache` 扩展，它允许您深入了解 I/O 缓存的内容：

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

`pg_buffercache` 扩展为您提供了包含几个字段的视图:

```
test=# \d pg_buffercache
View "public.pg_buffercache"
 Column           | Type      | Modifiers
-----+-----+
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；它标识缓冲区。 然后是 `relfilenode` 字段，它指向磁盘上的文件。 如果我们想查找文件所属的表，我们可以查看 `pg_class` 模块，该模块还包含一个名为 `relfilenode` 的字段。然后，有 `reldatabase` 和 `reltablespace` 字段。请注意，所有

字段都定义为 `oid` 类型，因此要以更有用的方式提取数据，必须将系统表连接在一起。

`relforknumber` 字段告诉我们缓存了表的哪一部分。它可以是堆，可用空间映射或其他组件，例如可见性映射。在未来，肯定会有更多类型的关系分叉。

下一个字段 `relblocknumber` 告诉我们哪个块已被缓存。最后，有一个 `isdirty` 标志，表示块是否已被修改，还有使用计数，和 `pin` 住块的后端会话数。

如果您想了解 `pg_buffercache` 扩展，请务必添加其他信息。要确定哪个数据库使用最多缓存，以下查询可能会有所帮助：

```
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，必须连接 `pg_database` 扩展。正如我们所看到的，`oid` 是连接列，对于刚接触 PostgreSQL 的人来说可能并不明显。

有时，我们可能想知道数据库中我们连接的哪些块被缓存：

```
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，我们过滤了当前数据库并和 `pg_class` 表进行连接，该表包含了数据库对象列表。`relkind` 列特别值得注意：`r` 表示表（关系），`i` 表示索引。这告诉了我们正在查看哪个对象。

使用 pgcrypto 加密数据

`pgcrypto` 是整个 `contrib` 模块部分中最强大的模块之一。它最初由一个 Skype 系统管理员编写，提供无数的功能来加密和解密数据。

它提供对称和非对称加密的功能。由于有大量可用功能，因此建议您查看 <https://www.postgresql.org/docs/current/static/pgcrypto.html> 上的文档页面。

由于本章的范围有限，不可能深入研究 `pgcrypto` 模块的所有细节。

使用 pg_prewarm 预热缓存

当 PostgreSQL 正常运行时，它会尝试缓存重要数据。`shared_buffers` 变量很重要，因为它定义了 PostgreSQL 管理的缓存的大小。现在的问题是：如果重新启动数据库服务器，PostgreSQL 管理的缓存将丢失。也许操作系统仍然会有一些数据来减少对磁盘等待的影响，但在很多情况下，这还不够。这个问题的解决方案叫做 `pg_prewarm` 扩展：

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

此扩展部署了一个函数，允许我们在需要时显式预热缓存：

```
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` 扩展的最简单和最常用的方法是让它缓存整个对象：

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

请注意，如果表太大而不适合缓存，则只有部分表数据会保留在缓存中，这在大多数情况下都很好。

该函数返回函数调用处理的 8 kb 的块数。

如果您不想缓存对象的所有块，还可以选择表中的特定范围。在下面的示例中，我们可以看到块 10 到 30 在缓存中：

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

很明显，缓存了 21 个块。

使用 `pg_stat_statements` 检查性能

`pg_stat` 语句是 `contrib` 模块中最重要的模块。它应该一直处于启用状态，并且能够提供更好的性能数据。如果没有 `pg_statements` 模块，很难跟踪性能问题。

由于其重要性，本书前面已讨论过 `pg_stat_statements`。

使用 pgstattuple 检查存储

有时，postgresql 中的表可能会出现比例增长的情况。对于增长过快的表，技术语是表膨胀。现在出现的问题是：哪些表膨胀了，有多少膨胀了？pgstattuple 扩展将帮助回答这些问题：

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

如前所述，模块部署了两个功能。在 pgstattuple 扩展的情况下，这些函数返回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结果可读：

```
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
```

在本例中，用于测试的表似乎处于非常好的状态；该表的大小为 3.6mb，不包含任何死行。自由空间也有限。如果由于表膨胀而减慢了对表的访问，那么死行的数量和可用空间的数量就不成比例地增长了。一些空闲空间和少数死行是正常的，但是，如果表增长得太快，以至于它主要由死行和空闲空间组成，则需要采取果断的行动来再次控制这种情况。

pgstattuple 扩展还提供了检查索引的功能：

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

pgstatindex 函数返回有关要检查的索引的大量信息：

```
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

我们的指针相当密集（89%）。这是个好兆头。索引的默认填充因子设置为 90%，因此接近 90% 的值表示索引非常好。

有时，您不想检查一个表，而是检查所有表或架构中的所有表。如何才能做到这一点？通常，要处理的对象列表在 `from` 子句中。但是，在我的示例中，函数已经在 `from` 子句中，如何使 PostgreSQL 循环遍历表列表？答案是 `lateral` 连接。

请记住，`pgstattuple` 必须读取整个对象。如果我们的数据库很大，那么处理可能需要很长时间。因此，存储我们刚刚看到的查询结果是个好主意，这样我们就可以彻底检查它，而无需一次又一次地重新运行查询。

使用 `pg_trgm` 进行模糊搜索

`pg_trgm` 是一个允许您执行模糊搜索的模块。该模块已在第 3 章“使用索引”中讨论过。

使用 `postgres_fdw` 连接到远程服务器

数据并不总是在一个位置。通常情况下，数据分布在整个基础设施中，并且可能是驻留在不同位置的数据，必须进行集成。

该问题的解决方案是外部数据包装器，由 SQL / MED 标准定义。

在本节中，将讨论 `postgres_fdw` 扩展。它是一个允许您从 PostgreSQL 数据源动态获取数据的模块。您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' [, ... ] ) ]
```

幸运的是，`CREATE FOREIGN DATA WRAPPER` 命令隐藏在扩展中；它可以使用常规过程轻松安装，如下所示：

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

在下面的步骤中，必须定义虚拟服务器。它将指向另一个主机并告诉 PostgreSQL 在哪里获取数据。在数据的末尾，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。用户信息将在稍后添加。服务器将只包含主机、端口等。下面是 CREATE SERVER 的语法：

```
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' [, ... ] ) ]
```

要了解其工作原理，我们将在同一主机上创建第二个数据库并创建服务器：

```
[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    | t_customer   | table  | hs
(2 rows)
```

现在，应该将服务器添加到标准测试数据库中：

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

请注意，所有重要信息都存储在 OPTIONS 子句中。这有点重要，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。

一旦定义了服务器，就可以映射用户了。如果我们从一台服务器连接到另一台服务器，那么在这两个位置上可能没有相同的用户。因此，外部数据包装器要求用户定义实际的用户映射：

```
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 FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

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，所有重要信息都隐藏在 `OPTIONS` 子句中。根据外部数据包装器的类型，选项列表将再次不同。注意，我们必须在这里使用正确的用户数据，这将对我们的设置起作用。在这种情况下，我们只需要使用本地用户。

基础架构到位后，我们可以创建外部表。 创建外表的语法与创建普通本地表的方法非常相似。 必须列出所有列，包括其数据类型：

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

列出的所有列都与普通 `CREATE TABLE` 子句的情况类似。 特别之处在于外部表指向远程端的表。 必须在 `OPTIONS` 子句中指定模式的名称和表的名称。

创建后，可以使用该表：

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

要检查 PostgreSQL 内部执行的操作，最好使用 `analyze` 参数运行 `EXPLAIN` 子句。 它将揭示有关服务器中实际发生情况的一些信息：

```
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。外部数据包装器将向另一侧发送查询并获取尽可能少的数据，因为在远程端执行尽可能多的限制以确保不必在本地处理太多数据。过滤条件、连接甚至聚合都可以远程执行（从 PostgreSQL 10.0 开始）。

尽管 CREATE FOREIGN TABLE 子句肯定是一个很好的使用方法，但是一遍又一遍地列出所有这些列会非常麻烦。

此问题的解决方案称为 IMPORT 子句。这使我们可以快速轻松地将整个模式导入到本地数据库中，还可以创建外部表：

```
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' [, ...] ) ]
```

IMPORT 允许我们轻松链接大型表。它还可以降低拼写错误和错误的几率，因为所有信息都是直接从远程数据源获取的。

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，先前在公共模式中创建的所有表都直接链接。我们可以看到，所有远程表现在都可用：

```
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)

处理错误和拼写错误

创建外部表并不是很难 - 但是有时会发生错误，或者可能只是更改了使用过的密码。为了解决这些问题，PostgreSQL 提供了两个命令：ALTER SERVER 和 ALTER USER MAPPING。

ALTER SERVER 允许您修改服务器：

```
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
```

我们可以使用此命令添加和删除特定服务器的选项，如果我们忘记某些内容，这是一件好事。

要修改用户信息，我们也可以更改用户映射：

```
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'] [, ... ] )
```

SQL/MED 接口得到了定期改进，在编写本书时，已经添加了一些特性。在未来，更多的优化将使它成为核心功能，使 SQL/MED 接口成为提高可扩展性的好选择。

其他有用的扩展

到目前为止我们描述的扩展都是 PostgreSQL contrib 包的一部分，它是 PostgreSQL 源代码的一部分。但是，我们看到的软件包并不是 PostgreSQL 社区中唯一可用的软件包。还有更多的软件包可以让我们做各种各样的事情。

不幸的是，这一章太短了，无法深入了解目前的一切。模块数量与日俱增，不可能全部覆盖。因此，我只想指出我认为最重要的。

PostGIS (<http://postgis.net/>) 是开源世界中的地理信息系统 (GIS) 数据库界面。它已

在全球范围内采用，是关系型开源数据库领域的事实标准之一。这是一个专业且极其强大的解决方案。

如果您正在寻找地理空间路由，那么 `pgRouting` 就是您可能正在寻找的。它提供了各种算法来查找位置之间的最佳连接，并在 PostgreSQL 之上工作。

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。还有更多的外国数据包装器。其中最著名和最专业的是 `oracle_fdw` 扩展。它允许您与 Oracle 集成并通过连接获取数据，这可以通过 `postgres_fdw` 扩展来完成。

在某些情况下，您可能还有兴趣使用 `pg_crash` (https://github.com/cybertec-postgresql/pg_crash) 测试基础架构的稳定性。这样做的目的是让一个模块不断地崩溃您的数据库。`pg_crash` 是测试和调试连接池的绝佳选择，它具有应用程序重新连接到故障数据库的能力。`pg_crash` 将定期运行 `amok` 并终止数据库会话或损坏内存。它是长期测试的理想选择。

小结

在本章中，我们了解了 PostgreSQL 标准发行版附带的一些最有前途的模块。这些模块非常多样化，提供从数据库连接到不区分大小写的文本和检查服务器的模块等各种功能。

既然我们已经讨论了扩展，在下一章中，我们将把注意力转移到迁移上。我们将学习如何以最简单的方式迁移到 PostgreSQL。

12. PostgreSQL 故障排除

在第 11 章“决定有用扩展”中，我们了解了一些广泛采用的有用扩展，可以为您的部署提供真正的支持。在此之后，您将了解 PostgreSQL 故障排除。我们的想法是为您提供系统的方法来检查和修复您的系统。

在本章中，将涵盖以下主题：

- 接近一个未知的数据库
- 获得简要的概述
- 确定关键瓶颈
- 处理存储损坏
- 检查破坏的复制

记住，很多事情都可能出错，所以专业地监视数据库是很重要的。

接近未知数据库

如果您碰巧管理一个大型系统，您可能不知道该系统实际上在做什么。管理数以百计的系统意味着你不知道每一个系统发生了什么。

在排除故障时，最重要的事情可以归结为一个词：数据。如果没有足够的数据，就没有办法解决问题。因此，故障排除的第一步是始终设置一个监视工具，例如 pgwatch2 (<https://www.cybertec-postgresql.com/en/products/pgwatch2/>)，它可以让您深入了解数据库服务器。

一旦报告告诉您一个值得检查的情况，这意味着以一种有组织的方式接近系统已经被证明是有用的。

检查 pg_stat_activity

建议的第一件事是检查 pg_stat_statements。回答以下的问题：

- 您系统上当前正在执行多少个并发查询？
- 您是否一直在查询列中看到类似的查询类型？
- 你看到已经运行了很长时间的查询吗？
- 有没有未授予的锁？
- 你看到来自可疑主机的连接吗？

应该首先检查 pg_stat_activity 视图，因为它可以让我们了解系统上正在发生的事情。当然，图形化的监控应该给你一个系统的第一印象。然而，归根结底，它实际上归结为服务器上实际运行的查询。因此，对 pg_stat_activity 提供的系统进行一个良好的概述对于跟踪问题非常重要。

为了方便您，我编写了一些查询，我认为这些查询可以尽快发现问题。

查询 pg_stat_activity

以下查询显示当前正在数据库上执行的查询数：

```
test=# SELECT datname,
             count(*) AS open,
             count(*) FILTER (WHERE state = 'active') AS active,
             count(*) FILTER (WHERE state = 'idle') AS idle,
             count(*) FILTER (WHERE state = 'idle in transaction')
                   AS idle_in_trans
      FROM pg_stat_activity
     WHERE backend_type = 'client backend'
   GROUP BY ROLLUP(1);
    datname | open | active | idle | idle_in_trans
-----+-----+-----+-----+
  test   | 2    | 1    | 0    | 1
        | 2    | 1    | 0    | 1
(2 rows)
```

为了在同一屏幕上显示尽可能多的信息，使用了部分聚合。我们可以在查询中看到 active、idle 和 idle-in-transaction。如果我们在查询中看到大量 idle-in-transaction，那么深入研究这些事务保持打开的时间肯定很重要：

```
test=# SELECT pid, xact_start, now() - xact_start AS duration
      FROM pg_stat_activity
     WHERE state LIKE '%transaction%'
   ORDER BY 3 DESC;
    pid    | xact_start                         | duration
-----+-----+
  19758 | 2017-11-26 20:27:08.168554+01 | 22:12:10.194363
(1 row)
```

列出的事务已经执行了 22 个多小时。现在的主要问题是：一个事务怎么能打开那么长时间？在大多数应用程序中，一个花费如此长时间的事务是高度可疑的，并且具有潜在的高度危险性。危险从何而来？正如我们在本书前面了解到的，VACUUM 命令只能在任何事务都看不到它的情况下清理死行。现在，如果一个事务持续打开数小时甚至数天，VACUUM 命令将无法产生有用的结果，这将导致表膨胀。

因此，强烈建议确保在长事务太长的情况下监视或终止长事务。从 9.6 版开始，PostgreSQL 有一个叫做 snapshot too old（快照太旧）的特性，它允许我们在快照出现太长时间时终止长事务。

检查是否存在长时间运行的查询也是一个好主意：

```
test=# SELECT
```

```

now() - query_start AS duration, datname, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY 1 DESC;
duration      | datname | query
-----+-----+
00:00:38.814526 | dev      | SELECT pg_sleep(10000);
00:00:00        | test     | SELECT now() - query_start AS duration,
                                              datname, query
                                              FROM pg_stat_activity
                                              WHERE state = 'active'
                                              ORDER BY 1 DESC;
(2 rows)

```

在这种情况下，将执行所有活动查询，语句将计算每个查询已处于活动状态的时间。通常，我们会在顶部看到类似的查询，这可以为我们提供一些有关系统上发生的事情的有价值的线索。

处理 Hibernate 语句

许多 ORMs，比如 hibernate，都会生成非常长的 sql 语句。问题在于：`pg_stat_activity` 只会在系统视图中存储查询的前 1024 个字节。其余部分被截断。对于由 ORM（如 hibernate）生成的长查询，查询在感兴趣的部分（from 子句等）实际开始之前被切断。

该问题的解决方案是在 `postgresql.conf` 文件中设置配置参数：

```

test=# SHOW track_activity_query_size;
track_activity_query_size
-----
1024
(1 row)

```

如果我们将此参数增加到相当高的值（可能是 32,768）并重新启动 PostgreSQL，那么我们将能够查看更长的查询并能够更容易地检测问题。

弄清楚查询的来源

检查 `pg_stat_activity` 时，有一些字段会告诉我们查询的来源：

```

client_addr      | inet      |
client_hostname | text      |
client_port      | integer   |

```

这些字段将包含 IP 地址和主机名（如果已配置）。但是，如果每个应用程序都从同一 IP 发送请求，因为所有应用程序都位于同一个应用服务器上，会发生什么情况？我们很难

看到哪个应用程序生成了某个查询。

此问题的解决方案是要求开发人员设置 `application_name` 变量：

```
test=# SHOW application_name ;
application_name
-----
psql
(1 row)

test=# SET application_name TO 'some_name';
SET
test=# SHOW application_name ;
application_name
-----
some_name
(1 row)
```

如果人们是合作的，那么 `application_name` 变量将显示在系统视图中，并且可以更容易地查看查询的来源。`application_name` 变量也可以设置为连接字符串的一部分。

检查慢查询

检查 `pg_stat_activity` 后，查看缓慢，耗时的查询是有意义的。基本上，有两种方法可以解决这个问题：

- 在日志中查找单个慢查询
- 寻找花费太多时间的查询类型

查找单个慢查询是性能调优的经典方法。通过将 `log_min_duration_statement` 变量设置为所需的阈值，PostgreSQL 将开始为超过此阈值的每个查询写入日志行。默认情况下，慢查询日志已关闭：

```
test=# SHOW log_min_duration_statement;
log_min_duration_statement
-----
-1
(1 row)
```

然而，将这个变量设置为一个合理的好值是完全有意义的。当然，根据您的工作量，所需的时间可能会有所不同。

在许多情况下，所需的值可能因数据库而异。因此，也可以以更细粒度的方式使用变量：

```
test=# ALTER DATABASE test SET log_min_duration_statement TO 10000;
ALTER DATABASE
```

如果您的数据库面临不同的工作负载，则仅为某个数据库设置参数非常有意义。

在使用慢查询日志时，必须考虑一个重要因素：许多较小的查询可能比少数慢速运行的查询导致更多的负载。当然，注意单个慢速查询总是有意义的，但有时这些查询不是问题所在。请考虑以下示例：在您的系统上，执行 100 万个查询，每个查询花费 500 毫秒，同时执行一些分析查询，每个查询运行几毫秒。显然，真正的问题永远不会出现在慢查询日志中，而每次数据导出、每次索引创建和每次大容量加载（在大多数情况下无论如何都无法避免）都会在日志中被记录并指向错误的优化方向。

因此，我个人的建议是使用慢速查询日志，但要谨慎使用。不过，最重要的是，要知道我们真正衡量的是什么。

在我看来，更好的方法是更深入地使用 `pg_statements` 视图。它将提供聚合信息，而不仅仅是关于单个查询的信息。本书前面已经讨论过 `pg_statements` 视图，这个模块的重要性不必过分强调。

检查某个查询

有时，会发现查询速度慢，但我们仍然不知道到底发生了什么。当然，下一步是检查查询的执行计划，看看会发生什么。在计划中确定那些导致糟糕运行时间的关键操作相当简单。尝试使用以下清单：

- 试着看看计划中时间开始飙升的地方
- 检查缺失的索引（性能不佳的主要原因之一）
- 使用 `EXPLAIN` 子句（`buffer` 为 `true`, `analyze` 为 `true` 等）以查看查询是否使用了太多缓冲区
- 打开 `track_io_timing` 参数以确定是否存在 I/O 或 CPU 问题（明确检查是否存在随机 I/O）
- 寻找错误的估计值并尝试修复它们
- 查找过于频繁执行的存储过程
- 试着弄清楚其中一些是否可以标记为 `STABLE` 或 `IMMUTABLE`，只要这是可能的

请注意，`pg_stat_statements` 不考虑语句解析时间，因此如果您的查询非常长（查询字符串），`pg_stat_statements` 可能会有些误导。

用 `perf` 深入分析

在大多数情况下，通过这个小检查表可以帮助您以非常快速和有效的方式跟踪大多数问题。然而，即使是从数据库引擎中提取的信息有时也不够。

`perf` 工具是 Linux 的分析工具，可让您直接查看哪些 C 函数会导致系统出现问题。通常，默认情况下不会安装 `perf`，因此建议安装它。要在服务器上使用 `perf`，只需登录到根目录并运行以下命令：

`perf top`

屏幕每隔几秒就会刷新一次，你将有机会看到现场的情况。以下列表显示了标准的只

读基准可能如下所示：

Samples: 164K of event 'cycles:ppp', Event count (approx.): 109789128766

Overhead Shared Object	Symbol
3.10% postgres	[.] AllocSetAlloc
1.99% postgres	[.] SearchCatCache
1.51% postgres	[.] base_yyparse
1.42% postgres	[.] hash_search_with_hash_value
1.27% libc-2.22.so	[.] vfprintf
1.13% libc-2.22.so	[.] _int_malloc
0.87% postgres	[.] palloc
0.74% postgres	[.] MemoryContextAllocZeroAligned
0.66% libc-2.22.so	[.] __strcmp_sse2_unaligned
0.66% [kernel]	[k] _raw_spin_lock_irqsave
0.66% postgres	[.] _bt_compare
0.63% [kernel]	[k] __fget_light
0.62% libc-2.22.so	[.] strlen

您可以看到我们的示例中没有单个函数占用太多的 CPU 时间，这告诉我们系统很好。

但是，情况可能并非总是如此。有一个问题 - 一个很常见的问题 - 称为自旋锁争用。这是什么？ PostgreSQL 核心使用自旋锁 (<https://en.wikipedia.org/wiki/Spinlock>) 来同步缓冲区访问等内容。自旋锁是现代 CPU 提供的一项功能，用于避免操作系统与小操作（如递增数字）的交互。这是一件好事，但在一些非常特殊的情况下，自旋锁可能会发疯。这是一件好事，但在一些非常特殊的情况下，自旋锁可能会变得疯狂。

如果您面临自旋锁争用，症状如下：

- CPU 负载真的很高
- 令人难以置信的低吞吐量（通常需要几毫秒的查询需要几秒钟）
- I/O 通常很低，因为 CPU 忙于锁交易

在许多情况下，自旋锁争用是突然发生的。你的系统很好，突然间，负载增加，吞吐量下降，就像石头下落一样。perf top 命令将显示大部分时间都花在一个名为 s_lock 的 c 函数中。如果是这种情况，则应尝试执行以下操作：

```
huge_pages = try          # on, off, or try
```

将 huge_pages 从 try 更改为 off。在操作系统级别完全关闭大页面是个好主意。一般来说，似乎某些内核比其他内核更容易产生这些问题。Red Hat 2.6.32 系列似乎特别糟糕（注意我在这里使用了似乎这个词）。

如果您使用 PostGIS，perf 工具也很有趣。如果列表中的顶级函数都是与 GIS 相关的（一些底层库），那么您知道问题很可能不是来自于错误的 PostgreSQL 调整，而是与花费时间完成的昂贵操作有关。

检查日志

如果您的系统出现问题，那么检查日志以查看发生的情况是有意义的。重要的是：并非所有日志条目都是相同的。PostgreSQL 有一个日志条目层次结构，范围从 DEBUG 消息到 PANIC。

对于管理员，以下三个错误级别非常重要：

- ERROR
- FATAL
- PANIC

ERROR 用于语法错误、权限相关问题等问题。日志将始终包含错误消息。关键因素是某类错误出现的频率是多少？产生数百万个语法错误肯定不是运行数据库服务器的理想策略。

FATAL 比 ERROR 更可怕；您将看到诸如无法为共享内存名称或意外的 walreceiver 状态分配内存的消息。换句话说，这些错误消息已经非常可怕，并会告诉您事情出了问题。

最后，有 PANIC。如果你发现这种消息，你知道某些事情确实是非常错误的。PANIC 的经典示例是锁定表已损坏或创建了太多信号量。这些将导致数据库宕机。

检查丢失的索引

一旦我们完成了前三个步骤，就必须从总体上看一下性能。正如我在这本书中一直说的，索引缺失是导致性能极差的主要原因。因此，每当我们面对一个缓慢的系统时，建议我们检查丢失的索引并部署所需的内容。

通常，客户会要求我们优化 RAID 级别、优化内核或其他一些花哨的东西。实际上，这些复杂的请求往往归结为一些索引的缺少，根据我的判断，花一些额外的时间检查所有需要的索引是否都存在总是有意义的。检查丢失的索引既不困难，也不耗时，因此无论您面临什么样的性能问题，都应该始终进行此项检查。

这是我最喜欢的查询，以获得索引可能丢失的位置：

```
SELECT schemaname, relname, seq_scan, seq_tup_read,  
       idx_scan, seq_tup_read / seq_scan AS avg  
FROM   pg_stat_user_tables  
WHERE  seq_scan > 0  
ORDER BY seq_tup_read DESC  
LIMIT 20;
```

尝试找到经常扫描的大表（高平均值）。这些表通常会排在最前面。

检查内存和 I/O

一旦我们找到缺失的索引，我们就可以检查内存和 I/O。要弄清楚发生了什么，激活 `track_io_timing` 是有意义的。如果打开，PostgreSQL 将收集有关磁盘等待的信息并将其呈现给您。

通常，客户提出的主要问题是：如果我们添加更多磁盘，它会更快吗？可以猜测会发生什么，但一般来说，衡量是更好、更有用的策略。启用 `track_io_timing` 将帮助您收集数据以真正解决这个问题。

PostgreSQL 以各种方式公开磁盘等待。检查事物的一种方法是查看 `pg_stat_database`:

Column	Type	Modifiers
<code>datid</code>	<code>oid</code>	
<code>datname</code>	<code>name</code>	
...		
<code>conflicts</code>	<code>bigint</code>	
<code>temp_files</code>	<code>bigint</code>	
<code>temp_bytes</code>	<code>bigint</code>	
...		
<code>blk_read_time</code>	<code>double precision</code>	
<code>blk_write_time</code>	<code>double precision</code>	

请注意，最后有两个字段：`blk_read_time` 和 `blk_write_time`。

他们会告诉我们 PostgreSQL 等待操作系统响应的时间。请注意，我们并不是真正测量磁盘等待时间，而是测量操作系统需要返回数据的时间。

如果操作系统产生缓存命中，则此时间将相当低。如果操作系统必须执行非常糟糕的随机 I/O，我们将看到一个块甚至可能需要几毫秒。

在许多情况下，当 `temp_files` 和 `temp_bytes` 显示高数值时，会出现高 `blk_read_time` 和 `blk_write_time`。此外，在许多情况下，这指向一个糟糕的 `work_mem` 或糟糕的 `maintenance_work_mem` 设置。记住这一点：如果 PostgreSQL 无法在内存中执行任务，则必须将其溢出到磁盘中。您可以使用 `temp_files` 操作来检测它。只要有 `temp_files`，就有可能出现令人讨厌的磁盘等待。

虽然每个数据库级别上的全局视图是有意义的，但它并不能产生关于真正问题根源的深入信息。通常，只有少数查询会导致性能不佳。找出这些问题的方法是使用 `pg_stat_statements`:

Column	Type
<code>query</code>	<code>text</code>

Column	Type	Modifiers
...		
query	text	
calls	bigint	
total_time	double precision	
...		
temp_blk_reads	bigint	
temp_blk_written	bigint	
blk_read_time	double precision	
blk_write_time	double precision	

您将能够看到，在每个查询的基础上，是否有磁盘等待。重要的部分是 `blk_time` 与 `total_time` 的组合。这个比率才是最重要的，一般来说，显示超过 30% 磁盘等待的查询可以被视为严重的 I/O 限制。

一旦我们检查完 PostgreSQL 系统表，就可以检查 Linux 上 `vmstat` 命令告诉我们的内容。或者，我们可以使用 `iostat` 命令：

```
[hs@zenbook ~]$ vmstat 2
procs -----memory----- --swap-- -----io---- -system-
cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
0 0 367088 199488 96 2320388 0 2 83 96 106 156 16 6 78 0 0
0 0 367088 198140 96 2320504 0 0 0 10 595 2624 3 1 96 0 0
0 0 367088 191448 96 2320964 0 0 0 8 920 2957 8 2 90 0 0
```

在数据库工作时，我们应该把注意力集中在三个列：`bi`, `bo` 和 `wa`。`bi` 字段告诉我们读取的块数；1,000 相当于 1 MB /秒。`bo` 列是关于数据块写入的。它告诉我们写入磁盘的数据量。在某种程度上，`bi` 和 `bo` 是原始吞吐量。我不认为一个数字是有害的。问题是高 `wa` 值。`bi` 和 `bo` 字段的低值与高 `wa` 值相结合，告诉我们潜在的磁盘瓶颈，这很可能与系统上发生的大量随机 I/O 有关。`wa` 值越高，查询越慢，因为您正在等待磁盘响应。



良好的原始吞吐量是一件好事；它也可以指出一个问题。如果在 OLTP 系统上需要高吞吐量，它可以告诉你没有足够的 RAM 来缓存内容或索引丢失，PostgreSQL 必须读取太多数据。请记住，事物是相互关联的，数据不应被视为孤立的。

了解值得注意的错误情况

在阅读了基本的指导原则以找出您将在数据库中面临的最常见问题之后，接下来的部分将讨论 PostgreSQL 世界中发生的一些最常见的错误场景。

面对 clog 损坏

PostgreSQL 有一个叫做提交日志的东西（现在称为 pg_xact；它正式名称为 pg_clog）。它跟踪系统上每个事务的状态，并帮助 PostgreSQL 确定是否可以查看行。通常，交易可以处于四种状态：

```
#define TRANSACTION_STATUS_IN_PROGRESS      0x00  
#define TRANSACTION_STATUS_COMMITTED        0x01  
#define TRANSACTION_STATUS_ABORTED          0x02  
#define TRANSACTION_STATUS_SUB_COMMITTED    0x03
```

clog 在 PostgreSQL 数据库实例（pg_xact）中有一个单独的目录。

在过去，人们已经报告了一些叫做 clog 损坏的东西，这可能是由磁盘错误或者多年来修复过的 PostgreSQL 中的错误引起的。损坏的提交日志是一件非常糟糕的事情，因为我们所有的数据都在那里，但是 PostgreSQL 不知道这些东西是否有效。这一领域的损坏无异于灾难。

管理员如何判断提交日志是否已损坏？这是我们通常看到的：

ERROR: could not access status of transaction 118831

如果 PostgreSQL 无法访问事务的状态，那么麻烦是肯定的。主要问题是 - 如何解决这个问题？说实话，没有办法真正解决问题 - 我们只能尝试尽可能多地抢救数据。

如前所述，提交日志每个事务保留两个 bits。这意味着我们每个字节有四个事务，每个块有 32,768 个事务。一旦我们弄清楚它是哪个块，我们就可以伪造事务日志：

```
dd if=/dev/zero of=<data directory location>/pg_clog/0001  
bs=256K count=1
```

我们可以使用 dd 来伪造事务日志，并将提交状态设置为所需的值。核心问题是到底应该使用哪个事务状态？答案是，任何状态实际上都是错误的，因为我们真的不知道这些事务是如何结束的。

然而，通常情况下，最好将它们设置为 committed 以减少数据丢失。在决定哪种方法的破坏性较小时，它实际上取决于我们的工作负载和数据。

当我们必须这样做时，你应该制造尽可能少的 clog。请记住，我们实际上是在伪造提交状态，这对数据库引擎来说不是一件好事。

一旦我们完成了伪造 clog，我们应该尽可能快地创建一个备份，并从头开始重新创建数据库实例。我们正在使用的系统不再值得信赖，所以我们应该尽可能快地尝试提取数据。记住这一点：我们即将提取的数据可能是矛盾和错误的，因此我们将确保对我们能够从数据库服务器中提取的任何内容进行一些质量检查。

理解检查点消息

了解检查点消息检查点对于数据完整性和性能至关重要。检查点间隔越远，性能通常越好。在 PostgreSQL 中，默认配置通常相当保守，因此检查点相对较快。如果数据库核心中的大量数据同时发生更改，可能是 PostgreSQL 要告诉我们，它认为检查点过于频繁。日志文件将显示以下条目：

LOG: checkpoints are occurring too frequently (2 seconds apart)
LOG: checkpoints are occurring too frequently (3 seconds apart)

在由于转储/还原或其他大型操作而导致的大量写入过程中，PostgreSQL 可能会注意到配置参数太低。会向日志文件发送一条消息，告诉我们确切的信息。

如果我们看到这种消息，出于性能原因，强烈建议通过显著增加 `max_wal_size` 参数来增加检查点距离（在旧版本中，该设置称为 `checkpoint_segments`）。在 PostgreSQL 的最新版本中，默认配置已经比以前好多了。但是，过于频繁地写入数据仍然很容易发生。

当我们看到关于检查点的消息时，我们必须记住一件事。检查点太频繁一点也不危险，只是碰巧会导致糟糕的性能。写的速度比可能的要慢得多，但我们的数据没有危险。充分增加两个检查点之间的距离将使错误消失，同时加快数据库实例的速度。

管理损坏的数据页面

PostgreSQL 是一个非常稳定的数据库系统。它尽可能地保护数据，并且多年来证明了它的价值。但是，PostgreSQL 依赖于硬件和正常工作的文件系统。如果存储中断，PostgreSQL 也会中断 - 除了添加从库以提高故障安全性之外，我们对此无能为力。

偶尔，文件系统或磁盘会出现故障。但在许多情况下，整个事情不会向极端发展；只是几个数据块因为某种原因被破坏。最近，我们在虚拟环境中看到了这种情况。默认情况下，某些虚拟机不会刷新数据到磁盘，这意味着 PostgreSQL 不能依赖于写入磁盘的内容。这种行为可能导致难以预测的随机问题。

当无法再读取块时，您可能会遇到以下错误消息：

"could not read block %u in file "%s": %m"

您即将运行的查询将出错并停止工作。

幸运的是，PostgreSQL 有办法处理这些事情：

```
test=# SET zero_damaged_pages TO on;
SET
test=# SHOW zero_damaged_pages;
zero_damaged_pages
-----
on
```

(1 row)

`zero_damaged_pages` 变量是一个配置变量，它允许我们处理损坏的页面。PostgreSQL 不会抛出错误，而是获取块并用零填充它。

请注意，这肯定会导致数据丢失。但请记住，无论如何数据都已被破坏或丢失，因此这只是处理由存储系统中发生的不良事件引起的损坏的一种方法。

我建议大家小心处理 `zero_damaged_pages` 变量，当您调用它时，请注意您在做什么。

草率的管理连接

在 PostgreSQL 中，每个数据库连接都是一个独立的进程。所有这些进程都使用共享内存进行同步（技术上，在大多数情况下，它是映射内存，但在本例中，这没有区别）。这个共享内存包含 I/O 缓存、活动数据库连接列表、锁和使系统正常工作的更重要的东西。

当连接关闭时，它将从共享内存中删除所有相关条目，并使系统保持正常状态。但是，当数据库连接由于任何原因而崩溃时会发生什么情况？

`postmaster`（主进程）将检测到其中一个子进程丢失。然后，将终止所有其他连接并初始化前滚过程。为什么这是必要的？当进程崩溃时，很可能会发生共享内存区域被进程编辑的情况。换句话说，崩溃进程可能会使共享内存处于损坏状态。因此，在损坏可以在系统内传播之前，`postmaster` 会做出反应，把所有连接赶出去。清理所有内存，每个人都必须重新连接。

从最终用户的角度来看，这感觉像是 PostgreSQL 崩溃并重启了，事实并非如此。由于进程无法对自身崩溃（分段错误）或某些其他信号作出反应，因此清除所有内容对于保护数据是绝对必要的。

如果在数据库连接上使用 `kill -9` 命令，也会发生同样的情况。连接无法捕获信号（-9 无法通过定义捕获），因此 `postmaster` 必须再次作出反应。

对抗表膨胀

在处理 PostgreSQL 时，表膨胀是最重要的问题之一。当我们面对糟糕的性能时，找出是否有对象需要比预期更多的空间总是一个好主意。

我们怎样才能弄清楚那些表产生了膨胀？看看 `pg_stat_user_tables` 视图：

```
test=# \d pg_stat_user_tables
          View "pg_catalog.pg_stat_user_tables"
   Column      | Type       | Modifiers
-----+-----+-----
```

relid	oid	
schemaname	name	
relname	name	
...		
n_live_tup	bigint	
n_dead_tup	bigint	

n_live_tup 和 **n_dead_tup** 字段给我们一个关于正在发生的事情的印象。我们也可以使用 `pgstattuple`, 如前一章所述。

如果有严重的表膨胀, 我们该怎么办? 第一个选项是运行 `VACUUM FULL` 命令。问题是 `VACUUM FULL` 子句需要一个表锁。对于大型表, 这可能是一个真正的问题, 因为用户在重组表时无法对其进行写入。



TIP 如果您使用至少 PostgreSQL 9.6 的版本, 则可以使用名为 `pg_squeeze` 的工具。

它 在 幕 后 重 组 表 而 不 会 产 生 阻 塞
(https://www.cybertec-postgresql.com/en/products/pg_squeeze/)。如果你正在重组一个非常大的表。

小结

在这一章中, 我们学习了系统地处理数据库系统, 并发现人们在使用 PostgreSQL 时面临的最常见问题。我们学习了一些重要的系统表, 以及其他一些可以决定我们成败的重要因素。

在本书的最后一章中, 我们将重点关注迁移到 PostgreSQL。如果您使用的是 Oracle 或其他数据库系统, 则可能需要查看 PostgreSQL。在第 13 章, *迁移到 PostgreSQL*, 将告诉你如何做到这一点。

QA

为什么数据库不自我管理?

您要记住的是用户总是比数据库引擎本身知道的更多。除此之外, 用户/管理员还可以访问许多有关操作系统、硬件、用户模式等的外部信息。数据库引擎无法决定用户的查询是否没有意义—它不知道查询的目的。因此, 管理员和开发人员总是比数据库更具优势, 因此是必要的(并且很可能总是如此)。

PostgreSQL 经常面临崩溃吗?

不, 我的公司正在为数千家公司提供服务。我们很少看到数据库损坏的情况 - 如果存在损坏, 通常是由硬件问题引起的。

PostgreSQL 需要持续不断的关注吗?

通常不会, 除非您以次优的方式使用数据库。一般来说, PostgreSQL 自己做很多事情,

自动处理很多事情，比如 VACUUM 等等。

Javanchueng

13. 迁移到 PostgreSQL

在第 12 章, *PostgreSQL 故障排除* 中, 我们了解了如何处理与 PostgreSQL 故障排除相关的最常见问题。重要的是采用系统的方法来追踪问题, 这正是这里提供的。

本书的最后一章是关于从其他数据库迁移到 PostgreSQL。许多读者可能仍然遭受商业数据库许可证成本带来的痛苦。我想给所有这些用户提供一条出路, 向您展示如何将数据从某个专有系统迁移到 PostgreSQL。从财务角度来看, 迁移到 PostgreSQL 不仅有意义, 而且如果您正在寻找更高级的功能和更大的灵活性, 这也是有意义的。PostgreSQL 提供了很多功能, 在撰写本书时, 每天都会添加新功能。这同样适用于可移植到 PostgreSQL 的工具数量。事情变得越来越好, 开发人员一直在发布更多更好的工具。

本章将介绍以下主题:

- 将 SQL 语句迁移到 PostgreSQL
- 从 Oracle 迁移到 PostgreSQL
- 将 MySQL 移植到 PostgreSQL

到本章结束时, 您应该基本能够将数据库从其他系统移动到 PostgreSQL。

将 SQL 语句迁移到 PostgreSQL

从数据库迁移到 PostgreSQL 时, 有必要查看哪个数据库引擎提供哪种功能。移动数据和结构本身通常相当容易。但是, 重写 SQL 可能不是。因此, 我决定包含一个部分, 该部分显式地关注 SQL 的各种高级特性及其在当今数据库引擎中的可用性。

使用 `lateral` 连接

在 SQL 中, `lateral` 连接基本上可以看作某种循环。这允许我们参数化连接并多次执行 `LATERAL` 子句中的所有内容。这里是一个简单的例子:

```
test=# SELECT *
  FROM generate_series(1, 4) AS x,
       LATERAL (SELECT array_agg(y)
                  FROM generate_series(1, x) AS y
                 ) AS z;
      x   | array_agg
-----+
      1   | {1}
      2   | {1,2}
      3   | {1,2,3}
      4   | {1,2,3,4}
(4 rows)
```

将为每个 `x` 调用 `LATERAL` 子句。 对最终用户来说，它基本上是某种循环。

支持 `LATERAL`

一个重要的 SQL 功能是 `lateral` 连接。 以下列表显示哪些引擎支持，哪些不支持：

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 自 PostgreSQL 9.3 以来一直支持
- SQLite: 不支持
- Db2 LUW: 从版本 9.1 (2005) 开始支持
- Oracle: 从 12c 开始支持
- Microsoft SQL Server: 自 2005 年以来一直受支持，但使用的语法不同

使用分组集

如果要同时运行多个聚合，分组集非常有用。 使用分组集可以加快聚合，因为我们不必多次处理数据。

这里是一个例子：

```
test=# SELECT x % 2, array_agg(x)
      FROM generate_series(1, 4) AS x
      GROUP BY ROLLUP (1);
?column? | array_agg
-----+-----
 0 | {2,4}
 1 | {1,3}
    | {2,4,1,3}
(3 rows)
```

PostgreSQL 提供的不仅仅是 `ROLLUP` 子句。 还支持 `CUBE` 和 `GROUPING SETS` 子句。

支持分组集

分组集对于在单个查询中生成多于一个聚合至关重要。 以下列表显示哪些引擎支持，哪些不支持：

- MariaDB: 自 5.1 以来仅支持 `ROLLUP` 子句（不完整支持）
- MySQL: 自 5.0 以来仅支持 `ROLLUP` 子句（不完整支持）
- PostgreSQL: 自 PostgreSQL 9.5 起支持
- SQLite: 不支持
- Db2 LUW: 至少从 1999 年开始支持
- Oracle: 从 9iR1 开始支持（大约 2000 年）

-
- Microsoft SQL Server: 自 2008 年以来一直受支持

使用 WITH 子句 - 公用表表达式

公用表表达式是在 SQL 语句中执行内容的好方法，但只能执行一次。 PostgreSQL 将执行所有 WITH 子句，并允许我们在整个查询中使用结果。

这里是一个简单的例子：

```
test=# WITH x AS (SELECT avg(id)
                   FROM generate_series(1, 10) AS id)
              SELECT *, y - (SELECT avg FROM x) AS diff
                FROM generate_series(1, 10) AS y
               WHERE y > (SELECT avg FROM x);
y   | diff
----+-----
 6 | 0.5000000000000000
 7 | 1.5000000000000000
 8 | 2.5000000000000000
 9 | 3.5000000000000000
10 | 4.5000000000000000
(5 rows)
```

在此示例中，WITH 子句的公用表达式扩展 (CTE) 计算 generate_series 函数生成的时间系列的平均值。 生成的 x 可以像查询表一样使用。 在我的例子中，x 被使用了两次。

支持 WITH 子句

以下列表显示哪些引擎支持 WITH 子句，哪些引擎不支持：

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 自 PostgreSQL 8.4 以来一直支持
- SQLite: 从 3.8.3 开始支持
- Db2 LUW: 自 8 年 (2000 年) 起支持
- Oracle: 从 9iR2 开始支持
- Microsoft SQL Server: 自 2005 年以来一直受支持



请注意，在 PostgreSQL 中，CTE 甚至可以支持写入 (INSERT, UPDATE 和 DELETE 子句)。 没有我知道的其他数据库可以实际做到这一点。

使用 WITH RECURSIVE 子句

WITH 子句有两种形式:

- 标准 CTE, 如上一节所示 (使用 WITH 子句)
- 一种在 SQL 中运行递归的方法

上一节已经介绍了 CTE 的简单形式。 在下一节中, 将介绍递归的版本。

支持 WITH RECURSIVE 子句

以下列表显示哪些引擎支持 WITH RECURSIVE 子句, 哪些引擎不支持:

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 自 PostgreSQL 8.4 以来一直支持
- SQLite: 从 3.8.3 开始支持
- Db2 LUW: 从 7 (2000) 开始支持
- Oracle: 从 11gR2 开始支持 (在 Oracle 中, 通常使用 CONNECT BY 子句而不是 WITH RECURSIVE 子句更常见)
- Microsoft SQL Server: 自 2005 年以来一直受支持

使用 FILTER 子句

在查看 SQL 标准本身时, 您会注意到自 SQL (2003) 以来 FILTER 子句已经存在。但是, 实际上没有多少系统支持这种非常有用的语言元素。

这里是一个例子:

```
test=# SELECT count(*),
           count(*) FILTER (WHERE id < 5),
           count(*) FILTER (WHERE id > 2)
      FROM generate_series(1, 10) AS id;
count | count | count
-----+-----+
     10 |    4 |    8
(1 row)
```

如果由于其他聚合需要数据而无法在普通 where 子句内使用条件, 则 filter 子句非常有用。

在引入 FILTER 子句之前, 使用更麻烦的语法可以实现相同的目的:

```
SELECT sum(CASE WHEN .. THEN 1 ELSE 0 END) AS whatever FROM some_table;
```

支持 FILTER 子句

以下列表显示哪些引擎支持 FILTER 子句，哪些引擎不支持：

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 自 PostgreSQL 9.4 以来一直支持
- SQLite: 不支持
- Db2 LUW: 不支持
- Oracle: 不支持
- Microsoft SQL Server: 不支持

使用窗口函数

窗口和分析函数已在本书中进行了广泛讨论。因此，我们可以直接进入 SQL 合规性。

支持窗口和分析

以下列表显示哪些引擎支持窗口功能，哪些不支持：

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 自 PostgreSQL 8.4 以来一直支持
- SQLite: 不支持
- Db2 LUW: 从版本 7 开始支持
- Oracle: 从版本 8i 开始支持
- Microsoft SQL Server: 自 2005 年以来一直受支持



其他一些数据库（如 Hive, Impala, Spark 和 NuoDB）也支持分析。

使用有序集 - WITHIN GROUP 子句

有序集对 PostgreSQL 来说是相当新的。有序集和普通集合之间的区别在于，在有序集的情况下，数据馈送到集合的方式确实有所不同。假设您要在数据中找到一个趋势 - 数据的顺序是相关的。

这里是计算中值的简单示例：

```
test=# SELECT id % 2,
           percentile_disc(0.5) WITHIN GROUP (ORDER BY id)
      FROM generate_series(1, 123) AS id
     GROUP BY 1;
?column? | percentile_disc
```

```
-----+-----  
0 | 62  
1 | 61  
(2 rows)
```

只有在有分类输入时才能确定中位数。

支持 WITHIN GROUP 子句

以下列表显示哪些引擎支持窗口函数功能，哪些不支持：

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 自 PostgreSQL 9.4 以来一直支持
- SQLite: 不支持
- Db2 LUW: 不支持
- Oracle: 从版本 9iR1 开始支持
- Microsoft SQL Server: 支持，但必须使用窗口函数重新构建查询

使用 TABLESAMPLE 子句

长期以来，表抽样一直是商业数据库供应商的真正优势。传统的数据库系统已经提供了多年的样本。然而，垄断已经被打破。从 PostgreSQL 9.5 开始，我们也有了一个解决采样问题的方法。

以下是它的工作原理：

```
test=# CREATE TABLE t_test (id int);  
CREATE TABLE  
test=# INSERT INTO t_test  
    SELECT * FROM generate_series(1, 1000000);  
INSERT 0 1000000
```

首先，创建包含 100 万行的表。然后，可以执行测试：

```
test=# SELECT count(*), avg(id)  
      FROM t_test TABLESAMPLE BERNOULLI (1);  
count | avg  
-----+-----  
 9802 | 502453.220873291165  
(1 row)  
test=# SELECT count(*), avg(id)  
      FROM t_test TABLESAMPLE BERNOULLI (1);  
count | avg  
-----+-----  
10082 | 497514.321959928586
```

(1 row)

在此示例中，相同的测试执行两次。 在每种情况下使用 1% 的随机样品。 两个平均值都非常接近 500 万，所以从统计的角度来看，结果非常好。

支持 TABLESAMPLE 子句

以下列表显示哪些引擎支持 TABLESAMPLE 子句，哪些引擎不支持：

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 自 PostgreSQL 9.5 起支持
- SQLite: 不支持
- Db2 LUW: 从 8.2 版开始支持
- Oracle: 从版本 8 开始支持
- Microsoft SQL Server: 自 2005 年以来一直受支持

使用限制/偏移

限制 SQL 中的查询结果是一个有点悲伤的故事。 简而言之，每个数据库的工作方式都有所不同。 虽然实际上有一个限制结果的 SQL 标准，但并不是每个人都完全支持事情应该是这样的。 限制数据的正确方法是实际使用以下语法：

```
test=# SELECT * FROM t_test FETCH FIRST 3 ROWS ONLY;
      id
      ---
      1
      2
      3
(3 rows)
```

如果您以前从未见过这种语法，请不要担心。 你绝对不是一个人。

支持 FETCH FIRST 子句

以下列表显示哪些引擎支持 FETCH FIRST 子句，哪些引擎不支持：

- MariaDB: 自 5.1 起支持（通常使用 limit/offset）
- MySQL: 自 3.19.3 起支持（通常使用 limit / offset）
- PostgreSQL: 自 PostgreSQL 8.4 起支持（通常使用 limit / offset）
- SQLite: 从 2.1.0 版开始支持
- Db2 LUW: 从版本 7 开始支持
- Oracle: 从版本 12c 开始支持（使用带有 row_num 函数的子选择）
- Microsoft SQL Server: 自 2012 年起支持（传统上使用 top-N）

正如您所看到的，限制结果集非常棘手，当您将商业数据库移植到 PostgreSQL 时，您很可能会遇到一些专有语法。

使用 OFFSET

OFFSET 子句类似于 `FETCH FIRST` 子句。它易于使用，但尚未被广泛采用。它没有 `FETCH FIRST` 子句那么糟糕，但它仍然是一个问题。

支持 OFFSET 子句

以下列表显示哪些引擎支持 `OFFSET` 子句，哪些引擎不支持：

- MariaDB: 从 5.1 开始支持
- MySQL: 从 4.0.6 开始支持
- PostgreSQL: 自 PostgreSQL 6.5 以来一直支持
- SQLite: 从 2.1.0 版开始支持
- Db2 LUW: 从版本 11.1 开始支持
- Oracle: 从 12c 版开始支持
- Microsoft SQL Server: 自 2012 年起支持

正如您所看到的，限制结果集非常棘手，当您将商业数据库移植到 PostgreSQL 时，您很可能会遇到一些专有语法。

使用时态表

一些数据库引擎提供了时态表来处理版本控制。遗憾的是，PostgreSQL 中没有开箱即用的版本控制。因此，如果您要从 Db2 或 Oracle 迁移，那么您需要做一些工作才能将所需的功能移植到 PostgreSQL。基本上，在 PostgreSQL 端稍微改变代码并不难。但是，它确实需要一些人工干预 - 它不再是一个直接的复制粘贴的东西。

支持时态表

以下列表显示哪些引擎支持时态表，哪些不支持：

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 不支持
- SQLite: 不支持
- Db2 LUW: 从 10.1 版开始支持
- Oracle: 从版本 12cR1 开始支持
- Microsoft SQL Server: 自 2016 年起支持

时间序列中的匹配模式

我所知道的最新的 SQL 标准（SQL 2016）提供了一个在时间序列中查找匹配项的特性。到目前为止，只有 Oracle 在其最新版本的产品中实现了此功能。

目前，还没有其他数据库供应商遵循它们并添加了类似的功能。如果要在 PostgreSQL 中对这种最先进的技术进行建模，则必须使用窗口功能和子查询。在 Oracle 中匹配时间序列模式是非常强大的；要在 PostgreSQL 中实现这一点的查询不会只有一种类型。

从 Oracle 迁移到 PostgreSQL

到目前为止，我们已经看到了如何在 PostgreSQL 中移植或使用最重要的高级 SQL 功能。鉴于此介绍，现在是时候看一下特别是迁移 Oracle 数据库系统了。

目前，由于 Oracle 的新许可和业务政策，从 Oracle 迁移到 PostgreSQL 已经变得非常流行。在全球范围内，人们正在远离 Oracle 并采用 PostgreSQL。

使用 `oracle_fdw` 扩展移动数据

我最喜欢的一种人们从 Oracle 转移到 PostgreSQL 的首选方法之一是 Laurenz Albe 的 `oracle_fdw` 扩展 (https://github.com/laurenz/oracle_fdw)。它是一个外部数据包装器 (FDW)，允许您将 Oracle 中的表表示为 PostgreSQL 中的表。`oracle_fdw` 扩展是最复杂的 FDW 之一，它是一个稳定的、有良好文档记录的、免费的、开放源代码的扩展。

安装 `oracle_fdw` 扩展需要您安装 Oracle 客户端库。幸运的是，已有 RPM 软件包可以直接使用 (<http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html>)。`oracle_fdw` 扩展需要 OCI 驱动程序与 Oracle 通信。除了现成的 Oracle 客户端驱动程序之外，还有一个用于 `oracle_fdw` 扩展本身的 RPM 包，由社区提供。如果您没有使用基于 RPM 的系统，您可能需要自己编译，这显然是可能的，但需要更多的劳动。

安装软件后，可以轻松启用它：

```
test=# CREATE EXTENSION oracle_fdw;
```

`CREATE EXTENSION` 子句将扩展加载到所需的数据库中。下一步，可以创建服务器，并将用户映射到 Oracle 端的对应服务器，如下所示：

```
test=# CREATE SERVER oraserver FOREIGN DATA WRAPPER oracle_fdw
        OPTIONS (dbserver '//dbserver.example.com/ORADB');
test=# CREATE USER MAPPING FOR postgres SERVER oradb
        OPTIONS (user 'orauser', password 'orapass');
```

然后，是时候获取一些数据了。我首选的方法是使用 `IMPORT FOREIGN SCHEMA` 子句导入数据定义。`IMPORT FOREIGN SCHEMA` 子句将为远程模式中的每个表创建一个外表，

并在 Oracle 端公开数据，然后可以轻松读取。

使用模式导入的最简单方法是在 PostgreSQL 上创建单独的模式，PostgreSQL 只保存数据库模式。然后，可以使用 FDW 轻松地将数据吸引到 PostgreSQL 中。本书的最后一部分，关于从 MySQL 迁移，向您展示了如何使用 MySQL / MariaDB 完成此操作的示例。请记住，IMPORT FOREIGN SCHEMA 子句是 SQL / MED 标准的一部分，因此该过程与 MySQL / MariaDB 相同。这几乎适用于支持 IMPORT FOREIGN SCHEMA 条款的每个 FDW。

虽然 oracle_fdw 扩展为我们完成了大部分工作，但是查看数据类型是如何映射的仍然是有意义的。Oracle 和 PostgreSQL 不提供完全相同的数据类型，因此某些映射要么由 Oracle 的 FDW 扩展完成，要么由我们手动完成。以下列表概述了类型的映射方式。左侧列显示 Oracle 类型，右侧列显示潜在的 PostgreSQL 对应项：

Oracle types	PostgreSQL types
CHAR	char, varchar, and text
NCHAR	char, varchar, and text
VARCHAR	char, varchar, and text
VARCHAR2	char, varchar, and text
NVARCHAR2	char, varchar, and text
CLOB	char, varchar, and text
LONG	char, varchar, and text
RAW	uuid and bytea
BLOB	bytes
BFILE	bytea (read-only)
LONG RAW	bytea
NUMBER	numeric, float4, float8, char, varchar, and text
NUMBER(n,m) with m<=0	numeric, float4, float8, int2, int4, int8, boolean, char, varchar, and text
FLOAT	numeric, float4, float8, char, varchar, and text
BINARY_FLOAT	numeric, float4, float8, char, varchar, and text
BINARY_DOUBLE	numeric, float4, float8, char, varchar, and text
DATE	date, timestamp, timestamptz, char, varchar, and text
TIMESTAMP	date, timestamp, timestamptz, char, varchar, and text
TIMESTAMP WITH TIME ZONE	date, timestamp, timestamptz, char, varchar, and text
TIMESTAMP WITH LOCAL TIME ZONE	date, timestamp, timestamptz, char, varchar, and text
INTERVAL YEAR TO MONTH	interval, char, varchar, and text
INTERVAL DAY TO SECOND	interval, char, varchar, and text
MDSYS.SDO_GEOGRAPHY	geometry

如果要使用几何，请确保在数据库服务器上安装了 PostGIS。

`oracle_fdw` 扩展的缺点是它不能立即迁移过程。 存储过程有点特殊，需要一些人工干预。

使用 `ora2pg` 从 Oracle 迁移

在 FDW 存在之前很久，人们就从 Oracle 迁移到 PostgreSQL。 高昂的许可证费用长期困扰着人们，因此转向 PostgreSQL 多年来一直是很自然的事情。

`oracle_fdw` 扩展的替代品是 `ora2pg`，它已存在多年，可以从 <https://github.com/darold/ora2pg> 免费下载。 `Ora2pg` 是用 Perl 编写的，并且有很长的新版本传统。

`ora2pg` 提供的功能令人惊叹：

- 迁移完整数据库模式，包括表，视图，序列和索引（唯一，主键，外键和检查约束）。
- 迁移用户和组的权限。
- 迁移分区表。
- 能够导出预定义的函数，触发器，过程，包和包体。
- 迁移完整或部分数据（使用 WHERE 子句）。
- 完全支持 Oracle BLOB 对象作为 PostgreSQL bytea。
- 能够将 Oracle 视图导出为 PostgreSQL 表。
- 能够导出 Oracle 用户定义的类型。
- PL / SQL 代码基本自动转换为 PL / pgSQL 代码。 请注意，无法完全自动转换所有内容。 但是，很多东西都可以自动转换。
- 能够将 Oracle 表导出为 FDW 表。
- 能够导出物化视图。
- 能够显示有关 Oracle 数据库内容的详细报告。
- 评估 Oracle 数据库迁移过程的复杂性。
- 从文件中迁移成本评估 PL / SQL 代码。
- 能够生成与 Pentaho 数据集成器（Kettle）一起使用的 XML 文件。
- 能够将 Oracle 定位器和空间几何导出到 PostGIS 中。
- 能够将数据库链接导出为 Oracle FDW。
- 能够将同义词导出为视图。
- 能够将目录导出为外部表或 `external_file` 扩展名的目录。
- 能够通过多个 PostgreSQL 连接发送 SQL 命令列表
- 能够在 Oracle 和 PostgreSQL 数据库之间执行差异以进行测试。

乍一看，使用 `ora2pg` 很难看。 然而，它实际上比看起来容易得多。

基本概念如下：

```
/usr/local/bin/ora2pg -c /some_path/new_ora2pg.conf
```

`ora2pg` 需要一个配置文件来运行。 配置文件包含处理进程所需的所有信息。 基本上，默认配置文件已经非常好了，它是大多数迁移的良好起点。 在 `ora2pg` 语言中，迁移是一个项目。

配置将驱动整个项目。运行时，ora2pg 将创建两个目录，其中包含从 Oracle 提取的所有数据：

```
ora2pg --project_base /app/migration/ --init_project test_project
Creating project test_project.
/app/migration/test_project/
    schema/
        dblinks/
        directories/
        functions/
        grants/
        mviews/
        packages/
        partitions/
        procedures/
        sequences/
        synonyms/
        tables/
        tablespaces/
        triggers/
        types/
        views/
    sources/
        functions/
        mviews/
        packages/
        partitions/
        procedures/
        triggers/
        types/
        views/
    data/
    config/
    reports/
Generating generic configuration file
Creating script export_schema.sh to automate all exports.
Creating script import_all.sh to automate all imports.
```

如您所见，生成的脚本可以直接执行。结果数据可以很好地导入 PostgreSQL。随时准备改变程序。并不是所有的东西都可以自动迁移，所以需要人工干预。

常见的陷阱

有一些非常基本的语法元素在 Oracle 中有效，但在 PostgreSQL 中可能不起作用。本节

列出了一些需要考虑的最重要的陷阱。当然，这个列表目前还不完整，但它应该会给你指明正确的方向。

在 Oracle 中，您可能会发现以下语句：

```
DELETE mytable;
```

在 PostgreSQL 中，这个语句是错误的，因为 PostgreSQL 要求你在 DELETE 语句中使用 FROM 子句。好消息是这种语法很容易解决。

接下来你可能会发现以下内容：

```
SELECT sysdate FROM dual;
```

PostgreSQL 既没有 sysdate 功能，也没有 dual 功能。dual 功能易于修复，因为您可以简单地创建返回一行的 VIEW。在 Oracle 中，dual 功能的工作原理如下：

```
SQL> desc dual
```

Name	Null?	Type
DUMMY	VARCHAR2(1)	

```
SQL> select * from dual;
```

D

-

X

在 PostgreSQL 中，通过创建以下 VIEW 可以实现同样的目的：

```
CREATE VIEW dual AS SELECT 'X' AS dummy;
```

sysdate 函数也很容易修复。它可以用 clock_timestamp() 函数替换。

另一个常见的问题是缺少 VARCHAR2 之类的数据类型，以及缺少仅由 Oracle 支持的特殊函数。解决这些问题的一个好方法是安装 orafce 扩展，它提供了大多数通常需要的东西，其中包括最常用的函数。访问 <https://github.com/orafce/orafce> 了解更多关于 orafce 扩展的信息当然是有意义的。它已经存在了很多年，是一个坚实的软件。

最近的一项研究表明，如果有 orafce 扩展（由 NTT 完成），orafce 扩展有助于确保 73% 的所有 Oracle SQL 都可以在 PostgreSQL 上执行而无需修改。

最常见的陷阱之一是 Oracle 处理外连接的方式。请考虑以下示例：

```
SELECT employee_id, manager_id
  FROM employees
 WHERE employees.manager_id(+) = employees.employee_id;
```

PostgreSQL 不提供这种语法，也不会提供。因此，必须将连接重写为适当的外连接。+ 是 Oracle 特有的，必须删除。

ora_migrator - 快速从 Oracle 迁移到 PostgreSQL

作为一名 PostgreSQL 顾问，我有很多机会从 Oracle 迁移到 PostgreSQL。然而，有时我发现很难找到适合我的工具链。总的来说，我从来没有找到一个完美的工具来快速迁移简单的数据库，而不是试图迁移硬的东西（并打破它的同时）。解决方法是编写自己的工具。我和我的团队提出了一个基于 `oracle_fdw` 的解决方案，该解决方案已经具备了预测数据类型的所有基础设施。PostgreSQL 方面可以快速高效地加载数据。最重要的是，`oracle_fdw` 不仅可以连接表，还可以将远程查询转换为 PostgreSQL 表。因此，`oracle_fdw` 是迁移工具的一个非常好的基础。

ora_migrator 如何工作？

现在自然产生的问题是：ora_migrator 到底是如何工作的？让我们来看看。ora_migrator 做的第一件事就是连接到 Oracle 系统目录并将数据复制到 PostgreSQL 端的架构中。这是成功的关键，因为我们不必解析所有 Oracle SQL 并尝试转换它。通过直接访问系统表，我们可以自己编译 PostgreSQL，这样做更容易，也更健壮。`oracle_fdw` 具有执行此操作的所有机制，因此我们可以轻松地构建它。

然后，将 Oracle 内容克隆到 PostgreSQL 模式中。这里的看法是您可能想要更改表的布局和数据类型。`oracle_fdw` 在预测 PostgreSQL 上的数据结构方面做得很好，但它无法确定真正需要什么。因此，在将表真正编译到 PostgreSQL 之前以及最终迁移数据之前，有必要进行此中间步骤。

我们的网站详细介绍了 ora_migrator 的工作原理：https://www.cybertec-postgresql.com/en/ora_migrator-moving-from-oracle-to-postgresql-even-faster/。



请注意，代码是开源的，可以免费使用。

从 MySQL 或 MariaDB 迁移到 PostgreSQL

在本章中，您已经学习了一些有关如何从 Oracle 等数据库迁移到 PostgreSQL 的宝贵经验。将 MySQL 和 MariaDB 数据库系统迁移到 PostgreSQL 相当容易。这样做的原因是 Oracle 可能会很昂贵并且有时很麻烦。这同样适用于 Informix。但是，Informix 和 Oracle 都有一个重要的共同点：正确遵守 `CHECK` 约束并正确处理数据类型。一般来说，我们可以安全地假设这些商业数据库系统中的数据是正确的，并且不违反数据完整性和常识的最基本规则。

我们下一个候选人是不同的。你知道很多关于商业数据库的东西在 MySQL 中并不正确。术语 `NOT NULL` 对 MySQL 没有多大意义（除非你明确使用严格模式）。在 Oracle、Informix、Db2 以及我所知道的所有其他系统中，`NOT NULL` 是一种在任何情况下都遵守的法律。默认情况下，MySQL 并没有认真对待这些约束（不过，公平地说，这在最近的版本中已

经改变了。严格模式直到最近才默认打开。但是，许多旧数据库仍然使用旧的默认设置)。在迁移的情况下，这会导致一些问题。对于技术上错误的数据，你打算做什么？如果您的 NOT NULL 列突然显示无数 NULL 条目，您将如何处理？MySQL 不只是在 NOT NULL 列中插入 NULL 值。它将根据数据类型插入一个空字符串或零。因此，事情可能变得非常讨厌。

处理 MySQL 和 MariaDB 中的数据

正如您可能想象的那样，正如您可能已经注意到的那样，当谈到数据库时，我远非没有偏见。但是，我不想把它变成盲目的 MySQL / MariaDB 抨击。我们的真正目标是了解为什么从长远来看 MySQL 和 MariaDB 会如此痛苦。我有偏见是有原因的，我真的想指出为什么会这样。我们将要看到的所有事情都是非常可怕的，并且对整个迁移过程产生了严重影响。我已经指出 MySQL 有点特殊，本节将尝试证明我的观点。

同样，以下示例假设我们使用的是没有严格模式的 MySQL / MariaDB 版本，这是本章最初编写时的情况（从 PostgreSQL 9.6 开始）。从 PostgreSQL 10.0 开始，严格模式已经开启，因此我们在这里阅读的大部分内容仅适用于旧版本的 MySQL / MariaDB。
让我们开始创建一个简单的表：

```
MariaDB [test]> CREATE TABLE data (
    id integer NOT NULL,
    data numeric(4, 2)
);
Query OK, 0 rows affected (0.02 sec)

MariaDB [test]> INSERT INTO data VALUES (1, 1234.5678);
Query OK, 1 row affected, 1 warning (0.01 sec)
```

到目前为止，没有什么特别的。我们已经创建了一个由两列组成的表。第一列显式标记为非空。第二列应该包含一个数字值，该值限制为四位数字。最后，我们添加了一个简单的行。你能看到一个潜在的地雷即将爆炸吗？很可能不是。但是，请检查以下列表：

```
MariaDB [test]> SELECT * FROM data;
+---+-----+
| id | data  |
+---+-----+
| 1  | 99.99 |
+---+-----+
1 row in set (0.00 sec)
```

如果我没记错的话，我们已经添加了一个四位数字，这首先不应该有用。但是，MariaDB 只是改变了我的数据。当然了还发出警告，但这不应该发生，因为表的内容不能反映我们实际插入的内容。

让我们尝试在 PostgreSQL 中做同样的事情：

```
test=# CREATE TABLE data
()
```

```
    id integer NOT NULL,
    data numeric(4, 2)
);
CREATE TABLE
test=# INSERT INTO data VALUES (1, 1234.5678);
ERROR: numeric field overflow
DETAIL: A field with precision 4, scale 2 must round to an absolute value
less than 10^2.
```

与前面一样，该表是创建的，但与 MariaDB / MySQL 形成鲜明对比的是，PostgreSQL 将出错，因为我们试图在表中插入一个显然不允许的值。如果数据库引擎不关心，那么清楚地定义我们想要什么有什么意义？假设你中了彩票，你可能会损失几百万，因为系统已经决定了什么对你有好处。

我一直在与商业数据库作斗争，但在任何昂贵的商业系统（Oracle、DB2、Microsoft SQL Server 等）中，我都从未见过类似的事情，它们可能有自己的问题，但数据总体上还不错。

更改列定义

让我们看看如果你想修改表定义会发生什么：

```
MariaDB [test]> ALTER TABLE data MODIFY data numeric(3, 2);
Query OK, 1 row affected, 1 warning (0.06 sec)
Records: 1 Duplicates: 0 Warnings: 1
```

你应该在这里看到一个问题：

```
MariaDB [test]> SELECT * FROM data;
+---+-----+
| id | data |
+---+-----+
| 1  | 9.99 |
+---+-----+
1 row in set (0.00 sec)
```

如您所见，数据再次被修改。它本来就不应该出现在那里，现在又被重新改变了。记住，你可能又丢了钱，或者其他一些好资产，因为 MySQL 试图变得聪明。

这里是 PostgreSQL 中发生的事情：

```
test=# INSERT INTO data VALUES (1, 34.5678);
INSERT 0 1
test=# SELECT * FROM data;
 id  | data
-----+
```

```
1 | 34.57  
(1 row)
```

我们现在改变列定义:

```
test=# ALTER TABLE data ALTER COLUMN data  
        TYPE numeric(3, 2);  
ERROR: numeric field overflow  
DETAIL: A field with precision 3, scale 2 must round to  
an absolute value less than 10^1.
```

同样, PostgreSQL 也会出错, 它不允许我们对数据做坏事。在任何重要的数据库中都会发生同样的情况。规则很简单: PostgreSQL 和其他人不允许我们破坏我们的数据。

但是, PostgreSQL 允许您做一件事:

```
test=# ALTER TABLE data  
        ALTER COLUMN data  
        TYPE numeric(3, 2)  
        USING (data / 10);  
ALTER TABLE
```

我们可以明确地告诉系统如何表现。在这种情况下, 我们明确告诉 PostgreSQL 将列的内容除以 10。开发人员可以明确地提供应用于数据的规则。PostgreSQL 不会尝试变得聪明, 它有充分理由这样做:

```
test=# SELECT * FROM data;  
id | data  
---+---  
1  | 3.46  
(1 row)
```

数据完全符合预期。

处理空值

我们不想把这变成一个为什么 MariaDB 是一个糟糕的章节, 但我想在这里添加一个最后的例子, 我认为这个例子非常重要:

```
MariaDB [test]> UPDATE data SET id = NULL WHERE id = 1;  
Query OK, 1 row affected, 1 warning (0.01 sec)  
Rows matched: 1 Changed: 1 Warnings: 1
```

id 列被显式标记为 NOT NULL:

```
MariaDB [test]> SELECT * FROM data;  
+----+----+  
| id | data |
```

```
+---+-----+
| 0 | 9.99 |
+---+-----+
1 row in set (0.00 sec)
```

显然, MySQL 和 MariaDB 认为 null 和 zero 是一回事。让我试着用一个简单的类比来解释这个问题: 如果你知道你的钱包是空的, 那就不知到你有多少钱了。当我写这些文字时, 我不知道我有多少钱 (null = 未知), 但我百分百肯定它超过零 (我肯定知道有足够的钱给我加油 从机场回家的路上心爱的车, 如果口袋里什么都没有, 这很难做到)。

这是更可怕的消息:

```
MariaDB [test]> DESCRIBE data;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| id | int(11) | NO | | NULL || |
| data | decimal(3,2) | YES | | NULL || |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

MariaDB 确实记得该列应该是非 NULL, 但它仍然再次修改您的数据。

期待问题

主要问题是可能无法将数据移动到 PostgreSQL。想象一下, 你想移动一些数据, 并且 PostgreSQL 方面有一个 NOT NULL 约束。我们知道 MySQL 并不关心:

```
MariaDB [test]> SELECT
    CAST('2014-02-99 10:00:00' AS datetime) AS x,
    CAST('2014-02-09 10:00:00' AS datetime) AS y;
+-----+
| x | y |
+-----+
| NULL | 2014-02-09 10:00:00 |
+-----+
1 row in set, 1 warning (0.00 sec)
```

PostgreSQL 肯定会拒绝 2 月 99 日 (这是有充分理由的), 但如果你明确禁止它, 它可能也不会接受 NULL 值 (这是有充分理由的)。在这种情况下, 您需要做的是以某种方式修复数据, 以确保它符合数据模型的规则, 这些规则由于某种原因而存在。你不应该掉以轻心, 因为你可能不得不改变数据, 这实际上是错误的。

迁移数据和架构

在试图解释为什么迁移到 PostgreSQL 是个好主意之后，在概述了一些最重要的问题之后，现在是时候解释一下我们最终摆脱 MySQL / MariaDB 的一些可能的选择。

使用 pg_chameleon

从 MySQL / MariaDB 迁移到 PostgreSQL 的一种方法是使用 Federico Campoli 的工具，名为 pg_chameleon，可以从 GitHub 免费下载 (https://github.com/the4thdoctor/pg_chameleon)。它已被明确设计为将数据复制到 PostgreSQL 并做了很多工作，例如为我们转换模式。

基本上，该工具执行以下四个步骤：

1. pg_chameleon 从 MySQL 读取模式和数据，并在 PostgreSQL 中创建模式。
2. 它将 MySQL 的主连接信息存储在 PostgreSQL 中。
3. 它在 PostgreSQL 中创建主键和索引。
4. 它从 MySQL / MariaDB 复制到 PostgreSQL。

pg_chameleon 工具为 DDL 提供基本支持，例如 CREATE, DROP, ALTER TABLE 和 DROP PRIMARY KEY。但是，由于 MySQL / MariaDB 的特性，它不支持所有 DDL。相反，它涵盖了最重要的功能。

然而，pg_chameleon 还有更多。我已经广泛地说过，数据并不总是它应该或预期的方式。pg_chameleon 解决问题的方法是丢弃垃圾数据并将其存储在名为 sch_chameleon.t_discarded_rows 的表中。当然，这不是一个完美的解决方案，但鉴于相当低质量的输入，它是我脑海中唯一明智的解决方案。我们的想法是让开发人员决定如何处理所有损坏的行。pg_chameleon 确实无法决定如何处理被其他人破坏的东西。

最近，开发了很多工具，做了很多工作。

因此，建议您查看 [github](#) 页面并阅读所有文档。正如我们所说，正在添加功能和错误修复。鉴于本章的范围有限，这里不可能完全覆盖。



TIP 存储过程，触发器等需要特殊处理，并且只能手动处理。pg_chameleon 工具无法自动处理这些内容。

使用 FDW

如果我们想从 MySQL / MariaDB 迁移到 PostgreSQL，那么成功的方法不止一种。FDW 的使用是 pg_chameleon 的替代方法，它提供了一种快速获取模式和数据并将其导入 PostgreSQL 的方法。连接 MySQL 和 PostgreSQL 的能力已经存在了很长一段时间，因此 FDW 绝对是一个可以利用的领域。

基本上，`mysql_fdw` 扩展就像其他任何 FDW 一样工作。与其他鲜为人知的 FDW 相比，`mysql_fdw` 扩展实际上非常强大，并提供以下功能：

- 写入 MySQL / MariaDB
- 连接池
- WHERE 子句下推（这意味着应用于表的过滤器实际上可以在远程执行以获得更好的性能）
- 列下推（仅需要从远程端获取所需的列；旧版本用于获取所有列，这会导致更多的网络流量）
- 远程节点预编译语句

使用 `mysql_fdw` 扩展的方法是使用 `IMPORT FOREIGN SCHEMA` 语句，该语句允许将数据移动到 PostgreSQL。幸运的是，这在 Unix 系统上相当容易。让我们详细了解所需的步骤。

我们要做的第一件事是从 GitHub 下载代码：

```
git clone https://github.com/EnterpriseDB/mysql_fdw.git
```

然后，运行以下命令来编译 FDW。请注意，系统上的路径可能有所不同。在本章中，我假设 MySQL 和 PostgreSQL 都位于 `/usr/local` 目录下，在您的系统上可能不是这样：

```
$ export PATH=/usr/local/pgsql/bin/:$PATH  
$ export PATH=/usr/local/mysql/bin/:$PATH  
$ make USE_PGXS=1  
$ make USE_PGXS=1 install
```

编译代码后，可以将 FDW 添加到我们的数据库中：

```
CREATE EXTENSION mysql_fdw;
```

下一步是创建我们要迁移的服务器：

```
CREATE SERVER migrate_me_server  
FOREIGN DATA WRAPPER mysql_fdw  
OPTIONS (host 'host.example.com', port '3306');
```

创建服务器后，我们可以创建所需的用户映射：

```
CREATE USER MAPPING FOR postgres  
SERVER migrate_me_server  
OPTIONS (username 'joe', password 'public');
```

最后，是时候进行真正的迁移了。首先要做的是导入架构。我建议首先为链接表创建一个特殊的模式：

```
CREATE SCHEMA migration_schema;
```

运行 `IMPORT FOREIGN SCHEMA` 语句时，我们可以将此模式用作将存储所有数据库链接的目标模式。优点是我们可以在迁移后方便地删除它。

完成 IMPORT FOREIGN SCHEMA 语句后，我们就可以创建真实表了。最简单的方法是使用 CREATE TABLE 子句提供的 LIKE 关键字。它允许我们复制表的结构并创建一个真实的本地 PostgreSQL 表。幸运的是，如果您正在克隆的表只是一个 FDW，这也可以。这是一个例子：

```
CREATE TABLE t_customer  
(LIKE migration_schema.t_customer);
```

然后，我们可以处理数据：

```
INSERT INTO t_customer  
SELECT * FROM migration_schema.t_customer
```

这实际上是我们可以更正数据、消除块行或对数据进行一些处理的地方。考虑到数据的低质量来源，在第一次移动数据后应用约束等是很有用的。可能不那么痛苦。

导入数据后，我们就可以部署所有约束，索引等。在这一点上，你实际上会开始看到一些令人讨厌的惊喜，因为正如我之前所说，你不能指望数据坚如磐石。通常，在 MySQL 的情况下，迁移可能非常困难。

小结

在本章中，我们学习了如何迁移到 PostgreSQL。迁移是一个重要的话题，越来越多的人正在使用 PostgreSQL。

PostgreSQL 11 具有许多新功能，例如改进的内置分区等等。在未来，我们将在 PostgreSQL 的所有领域看到更多的发展，特别是允许用户更大规模地扩展和更快地运行查询。我们还没有看到我们的未来。