迁移到 PostgreSQL

在第12章 "PostgreSQL的故障排除 "中，我们学习了如何处理与PostgreSQL故障排除有关的最常见的问题。重要的是要有一个系统的方法来追踪问题，这正是这里所提供的内容。

本书的最后一章是关于从其他数据库转移到PostgreSQL。你们中的许多人可能还在忍受商业数据库许可证费用带来的痛苦。我想给你们所有的人一条出路，告诉你们如何将数据从专有系统转移到PostgreSQL。转移到PostgreSQL不仅从财务角度来看是有意义的，而且如果你正在寻找更高级的功能和更多的灵活性，它也是有意义的。PostgreSQL有很多东西可以提供，在写这篇文章的时候，每天都在增加新的功能。这同样适用于可用于迁移到PostgreSQL的工具的数量。事情正在变得越来越好，而且开发人员一直在发布更多更好的工具。

本章将涵盖以下主题：

- 将SQL语句迁移到PostgreSQL

- 从Oracle迁移到PostgreSQL

在本章结束时，你应该能够将一个基本的数据库从其他系统转移到PostgreSQL。

1. 将SQL语句迁移到PostgreSQL

当从一个数据库转移到PostgreSQL时，看一看并弄清楚哪个数据库引擎提供哪种功能是有意义的。 移动数据和结构本身通常是相当容易的。然而，重写SQL可能就不容易了。因此，我决定包括一个部分，明确地关注SQL的各种高级功能以及它们在今天的数据库引擎中的可用性。

1.1 使用横向连接

在SQL中，横向连接基本上可以被看作是某种循环。这允许我们对一个连接进行参数化处理，并在LATERAL子句中多次执行所有的内容。下面是一个简单的例子。

```
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)
```

LATERAL子句将对x的每个实例进行调用。对终端用户来说，这基本上是某种循环。

1.1.1 支持横向连接

一个重要的SQL特性是横向连接。下面的列表显示了哪些引擎支持横向连接，哪些不支持。

- MariaDB：不支持
- MySQL：不支持
- PostgreSQL：从PostgreSQL 9.3开始支持
- SQLite：不支持
- Db2 LUW：从9.1版（2005年）开始支持
- Oracle：从12c开始支持
- Microsoft SQL Server：自2005年起支持，但使用不同的语法

1.2 使用分组集

如果我们想同时运行一个以上的聚合，分组集就非常有用。使用分组集可以加快聚合速度，因为我们不需要多次处理数据。

下面是一个例子。

```
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。也支持CUBE和GROUPING SETS子句。

1.2.1 支持分组集

分组集对于在一个查询中生成不止一个聚合是必不可少的。下面的列表显示了哪些引擎支持分组集，哪些不支持。

- MariaDB：从5.1开始只支持ROLLUP子句（不完全支持）
- MySQL：从5.0开始只支持ROLLUP子句（不完全支持）
- PostgreSQL：从PostgreSQL 9.5开始支持
- SQLite：不支持
- Db2 LUW：至少从1999年开始支持
- Oracle：从9iR1开始支持（2000年左右）
- Microsoft SQL Server：自2008年起支持

1.3 使用 WITH 子句——公用表表达式

普通表表达式是一种在SQL语句中执行东西的好方法，但只有一次。PostgreSQL 将执行所有的 WITH 子句，并允许我们在整个查询中使用这些结果

这是一个简化的例子：

```
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)
```

在这个例子中，WITH子句的公共表扩展（CTE）计算了由generate_series函数生成的时间序列的平均值。产生的x可以像表一样在查询中被使用。在我的例子中，x被使用了两次。

1.3.1 支持 WITH 子句

下面的列表显示了哪些引擎支持WITH子句，哪些不支持

- MariaDB：不支持
- MySQL：不支持
- PostgreSQL：从PostgreSQL 8.4开始支持
- SQLite：从 3.8.3 开始支持
- Db2 LUW：从8（2000）开始支持
- Oracle：从9iR2开始支持
- Microsoft SQL Server：自2005年起支持

> 请注意，在PostgreSQL中，CTE甚至可以支持写入（INSERT、UPDATE和DELETE条款）。据我所知，没有其他数据库能够真正做到这一点。

1.3.2 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）
- 在SQL中运行递归的一种方法

上一节介绍了CTE的简单形式。在下一节中，我们将介绍递归版本。

1.3.3 支持 WITH RECURSIVE 子句

下面的列表显示了哪些引擎支持WITH RECURSIVE子句，哪些不支持。

- MariaDB：不支持
- MySQL：不支持
- PostgreSQL：从PostgreSQL 8.4开始支持
- 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年起支持

1.4 使用 FILTER 子句

在查看SQL标准本身时，你会注意到FILTER子句从SQL（2003）开始就存在了。然而，实际上没有多少系统支持这个非常有用的语法元素。

下面是一个例子。

```
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)
```

如果一个条件不能在正常的WHERE子句中使用，因为有其他的聚合需要数据，那么FILTER子句就很有用。 在引入FILTER子句之前，可以通过更繁琐的语法形式来实现同样的目的。

```
SELECT sum(CASE WHEN .. THEN 1 ELSE 0 END) AS whatever FROM some_table;
```

1.4.1 支持 FILTER 子句

下面的列表显示了哪些引擎支持FILTER子句，哪些不支持。

- MariaDB：不支持
- MySQL：不支持
- PostgreSQL：从PostgreSQL 9.4开始支持
- SQLite：不支持
- Db2 LUW：不支持
- Oracle：不支持
- Microsoft SQL Server：不支持

1.5 使用窗口函数

本书中已经广泛地讨论了窗口和分析。因此，我们可以直接跳到SQL顺应性方面。

1.5.1 支持窗口和分析

下面的列表显示了哪些引擎支持Windows功能，哪些不支持。

- MariaDB：在最新版本中支持
- MySQL：在最新的版本中支持
- PostgreSQL：从PostgreSQL 8.4开始支持
- SQLite：不支持
- Db2 LUW：从7版开始支持
- Oracle：从8i版开始支持
- Microsoft SQL Server：自2005年起支持

其他一些数据库，如Hive、Impala、Spark和NuoDB，也支持分析。

1.6 使用有序集——WITHIN GROUP 子句

有序集合对PostgreSQL来说是相当新的。有序集和普通聚合的区别在于，在有序集的情况下，数据被送入聚合的方式确实有区别。假设你想在你的数据中找到一个趋势--数据的顺序是相关的。

下面是一个计算中位数的简单例子。

```
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)
```

只有在有排序输入的情况下才能确定中位数。

1.6.1 支持 WITHIN GROUP 子句

下面的列表显示了哪些引擎支持Windows功能，哪些不支持。

- MariaDB：不支持MySQL。不支持
- PostgreSQL：从PostgreSQL 9.4开始支持
- SQLite：不支持
- Db2 LUW：支持的
- Oracle：从9iR1版本开始支持的
- Microsoft SQL Server：支持，但查询必须使用窗口功能进行重塑

1.7 使用 TABLESAMPLE 子句

长期以来，表采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很多年的采样。然而，这种垄断已经被打破。从PostgreSQL 9.5开始，我们也有了解决采样问题的方法。

下面是它的工作原理。

```
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，创建一个包含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)
```

在这个例子中，同一个测试被执行了两次。每次都使用1%的随机样本。两次的平均值都很接近500万，所以从统计学的角度来看，结果是相当不错的。

1.7.1 支持 TABLESAMPLE 子句

下面的列表显示了哪些引擎支持TABLESAMPLE子句，哪些不支持。

- MariaDB：不支持
- MySQL：不支持
- PostgreSQL：从PostgreSQL 9.5开始支持
- SQLite：不支持
- Db2 LUW：从8.2版开始支持
- Oracle：从8版开始支持
- Microsoft SQL Server：自2005年起支持

1.8 使用限制/偏移

在SQL中限制一个结果是一个有点悲伤的故事。简而言之，每个数据库的做法都有些不同。尽管实际上有一个关于限制结果的SQL标准，但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)
```

如果你以前从未见过这种语法，不要担心。你绝对不是一个人

1.8.1 支持 FETCH FIRST 子句

下面的列表显示了哪些引擎支持FETCH FIRST子句，哪些不支持。

- MariaDB：从5.1开始支持（通常，使用limit/offset）
- MySQL：从3.19.3开始支持（通常，使用limit/offset）
- PostgreSQL：从PostgreSQL 8.4开始支持（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）。

正如你所看到的，限制结果集是相当棘手的，当你把一个商业数据库移植到PostgreSQL时，你很可能会遇到一些专有的语法。

1.9 使用 OFFSET 子句

OFFSET子句类似于FETCH FIRST子句。它很容易使用，但它还没有被广泛采用。它不像FETCH FIRST子句那样糟糕，但它仍然倾向于成为一个问题。

1.9.1 支持 OFFSET 子句

下面的列表显示了哪些引擎支持OFFSET子句，哪些不支持。

- MariaDB：从5.1开始支持
- MySQL：自4.0.6起支持
- PostgreSQL：从PostgreSQL 6.5开始支持
- SQLite：自2.1.0版起支持
- Db2 LUW：从11.1版开始支持
- Oracle：从12c版开始支持
- Microsoft SQL Server：自2012年起支持

正如你所看到的，限制结果集是相当棘手的，当你把一个商业数据库移植到PostgreSQL时，很可能会遇到一些专有的语法。

1.10 使用时态表

一些数据库引擎提供了时间表来处理版本问题。不幸的是，在PostgreSQL中没有这种开箱即用的版本管理。所以，如果你是从Db2或Oracle迁移过来的，你需要做一些工作来将所需的功能移植到PostgreSQL上。基本上，在PostgreSQL方面改变一下代码并不是太难。然而，这确实需要一些人工干预--它不再是一个直接复制和粘贴的工作。

1.10.1 支持时态表

下面的列表显示了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版本开始支持
- Oracle：从12cR1版本开始支持
- Microsoft SQL Server：自2016年起支持

1.11 时间序列中的匹配模式

在写这篇文章的时候，最新的SQL标准（SQL 2016）提供了一个功能，旨在寻找时间序列中的匹配。到目前为止，只有Oracle在其最新版本的产品中实现了这个功能。

在这一点上，没有其他数据库厂商跟随他们并增加类似的功能。如果你想在PostgreSQL中模拟这种最先进的技术，你必须与窗口函数和子选择一起工作。 在Oracle中匹配时间序列模式是相当强大的；在PostgreSQL 中实现此目的的查询类型不止一种。

2.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特性。鉴于这些介绍，现在是时候特别看一下迁移Oracle数据库系统了。

这些天，由于Oracle新的许可和商业政策，从Oracle迁移到PostgreSQL已经变得非常流行。在世界范围内，人们正在远离Oracle而采用PostgreSQL。

2.1 使用 oracle_fdw 扩展移动数据

我首选的将用户从Oracle转移到PostgreSQL的方法之一是Laurenz Albe的oracle_fdw扩展（https://github.com/laurenz/oracle_fdw）。它是一个外来数据封装器（FDW），允许你将Oracle中的表 表示 为PostgreSQL中的表。oracle_fdw扩展是最复杂的FDW之一，它坚如磐石，有很好的文档，是免费和开源的。

安装oracle_fdw扩展需要你安装Oracle客户端库。幸运的是，已经有了可以开箱即用的RPM包（http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html）。 oracle_fdw扩展需要OCI驱动来与Oracle对话。除了现成的Oracle客户端驱动之外，还有一个oracle_fdw扩展本身的RPM包，它是由社区提供的。如果你没有使用基于RPM的系统，你可能不得不自己编译，这显然是可能的，但有点费力。

一旦软件被安装，就可以很容易地启用。

```
test=# CREATE EXTENSION oracle_fdw;
```

CREATE EXTENSION子句将扩展加载到你想要的数据库中。 现在，可以创建一个服务器，并将用户映射到其在Oracle方面的对应方，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');
```

现在，是时候获取一些数据了。我的首选方法是使用IMPORT FOREIGN SCHEMA子句来导入数据定义。IMPORT FOREIGN SCHEMA子句将为远程模式中的每个表创建一个外表，并将数据暴露在Oracle一侧，然后可以很容易地读取这些数据

利用模式导入的最简单的方法是在PostgreSQL上创建单独的模式，这些模式只是容纳数据库的模式。然后，可以使用FDW轻松地将数据吸进PostgreSQL。本章的最后一节，迁移数据和模式，关于从MySQL的迁移，向你展示了一个如何用MySQL/MariaDB进行迁移的例子。 请记住，IMPORT FOREIGN SCHEMA子句是SQL/MED标准的一部分，因此这个过程与MySQL/MariaDB是一样的。这适用于几乎所有支持IMPORT FOREIGN SCHEMA子句的FDW。

虽然oracle_fdw扩展为我们做了大部分的工作，但看看数据类型是如何被映射的仍然是有意义的。Oracle和PostgreSQL没有提供完全相同的数据类型，所以有些映射是由oracle_fdw扩展或者我们手动完成的。下表提供了一个关于类型映射的概述。左边一列显示的是Oracle的类型，右边一列显示的是潜在的PostgreSQL的对应关系。

Oracle type	Possible PostgreSQL types
CHAR	char, varchar, text
NCHAR	char, varchar, text
VARCHAR	char, varchar, text
VARCHAR2	char, varchar, text, json
NVARCHAR2	char, varchar, text
CLOB	char, varchar, text, json
LONG	char, varchar, text
RAW	uuid, bytea
BLOB	bytea
BFILE	bytea (read-only)
LONG RAW	bytea
NUMBER	numeric, float4, float8, char, varchar, text
NUMBER(n,m) with m<=0	numeric, float4, float8, int2, int4, int8,boolean, char, varchar, text
FLOAT	numeric, float4, float8, char, varchar, text
BINARY_FLOAT	numeric, float4, float8, char, varchar, text
BINARY_DOUBLE	numeric, float4, float8, char, varchar, text
DATE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH TIME ZONE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH LOCAL TIME ZONE	date, timestamp, timestamptz, char, varchar, text
INTERVAL YEAR TO MONTH	interval, char, varchar, text
INTERVAL DAY TO SECOND	interval, char, varchar, text
MDSYS.SDO_GEOMETRY	geometry (see “PostGIS support” below)

如果你想使用几何图形，请确保你的数据库服务器上已经安装了PostGIS。

oracle_fdw扩展的缺点是，它不能迁移开箱即用的存储过程。存储过程是一个有点特殊的东西，需要一些人工干预

2.2 使用 ora_migrator 进行快速迁移

虽然oracle_fdw是一个好的开始，但我们可以做得更好。ora_migrator（https://www.cybertec-postgresql.com/en/ora_migrator-moving-from-oracle-to-postgresql-even-faster/, https://github.com/cybertec-postgresql/ora_migrator）是在oracle_fdw之上开发的，并以最有效的方式使用其所有功能。 它是如何工作的？一旦你从我们的GitHub页面安装了ora_migrator，你就可以通过使用以下命令来启用这个扩展。

```
CREATE EXTENSION ora_migrator;
```

一旦该模块被安装，看一看ora_migrator在做什么是有意义的。让我们运行一个示例调用并检查输出。

```
SELECT oracle_migrate(server => 'oracle', only_schemas => '{LAURENZ,SOCIAL}');
NOTICE: Creating staging schemas "ora_stage" and "pgsql_stage" ...
NOTICE: Creating Oracle metadata views in schema "ora_stage" ...
NOTICE: Copying definitions to PostgreSQL staging schema "pgsql_stage" ...
NOTICE: Creating schemas ...
NOTICE: Creating sequences ...
NOTICE: Creating foreign tables ...
NOTICE: Migrating table laurenz.log ...
...
NOTICE: Migrating table social.email ...
NOTICE: Migrating table laurenz.numbers ...
NOTICE: Creating UNIQUE and PRIMARY KEY constraints ...
WARNING: Error creating primary key or unique constraint on table
laurenz.badstring
DETAIL: relation "laurenz.badstring" does not exist:
WARNING: Error creating primary key or unique constraint on table laurenz.hasnul
DETAIL: relation "laurenz.hasnul" does not exist:
NOTICE: Creating FOREIGN KEY constraints ...
NOTICE: Creating CHECK constraints ...
NOTICE: Creating indexes ...
NOTICE: Setting column default values ...
NOTICE: Dropping staging schemas ...
NOTICE: Migration completed with 4 errors.
 oracle_migrate
----------------
 4
(1 row)
```

ora_migrator的工作方式如下。首先，它克隆了Oracle系统目录的一部分，并把这些数据放到PostgreSQL数据库的一个暂存模式中。然后，这些信息被转换，这样我们就可以在PostgreSQL上实际使用它来轻松地创建表、索引、视图等等。 在这个阶段，我们进行数据类型转换等等。

最后，数据被复制过来，索引、约束和类似的东西被应用。

你刚才看到的是最简单的情况。oracle_migrate只是一个封装函数，因此你也可以自己一步一步地调用需要的各个步骤。文档显示了在哪个级别可以做什么，你将很容易地以一种简单的方式迁移对象。

与其他一些工具相比，ora_migrator并不试图去做那些实际上不可能正确完成的事情。ora_migrator不触及的最重要的组件是程序。基本上不可能完全自动地将程序从Oracle程序转换为PostgreSQL程序。因此，我们不应该尝试去转换它们。 简而言之，迁移存储过程仍然是一个部分手工操作的过程。

ora_migrator正在稳步改进，并且从11.2版本开始，可以用于所有的Oracle版本。

2.3 CYBERTEC Migrator——“大男孩”的迁移

如果你正在寻找一个更全面的、具有24/7支持的商业解决方案，我们可以推荐你看一下CYBERTEC Migrator，它可以在我的网站（https://www.cybertec-postgresql.com/en/products/cybertec-migrator/）上找到。它带有内置的并行性、高级数据类型预测、零停机迁移、自动代码重写等功能。

在我们的测试中，我们已经看到了高达1.5GB/秒的传输速度，这是我目前所知的最快的实现。请查看我们的网站以了解更多。

2.4 使用 Ora2Pg 从 Oracle 迁移

早在FDW出现之前，人们就从Oracle迁移到了PostgreSQL。 长期以来，高额的许可证费用一直困扰着用户，因此，多年来，迁移到PostgreSQL是一件很自然的事情。

替代oracle_fdw扩展的是一个叫做Ora2Pg的东西，它已经存在了很多年，可以从https://github.com/darold/Ora2Pg。Ora2Pg是用Perl编写的，有一个长期的新版本传统。

Ora2Pg所提供的功能是惊人的。

- 迁移整个数据库模式，包括表、视图、序列和索引（唯一、主、外键和检查约束）。
- 迁移用户和组的权限。
- 迁移分区的表。
- 能够导出预定义的函数、触发器、程序、包和包体。
- 迁移全部或部分数据（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代码到PL/pgSQL代码的基本自动转换。注意，完全自动转换所有东西是不可能的。 然而，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数据库的迁移过程的复杂性。
- 从文件中对PL/SQL代码进行迁移成本评估。
- 能够生成用于Pentaho数据集成器（Kettle）的XML文件。
- 能够将Oracle定位器和空间几何图形导出到PostGIS。
- 能够将数据库链接导出为Oracle FDWs。
- 能够将同义词作为视图导出。
- 能够将一个目录作为外部表或外部文件扩展的目录导出。
- 能够在多个 PostgreSQL 连接上调度一个 SQL 命令列表。
- 能够为测试目的在Oracle和PostgreSQL数据库之间执行一个差异功能

使用Ora2Pg乍看之下很难。然而，它实际上比看起来容易得多。其基本概念如下。

```
/usr/local/bin/Ora2Pg -c /some_path/new_Ora2Pg.conf
```

Ora2Pg需要一个配置文件来运行。这个配置文件包含了处理这个过程所需要的所有信息。基本上，默认的配置文件已经非常不错了，对于大多数迁移来说，它是一个很好的起点。 在Ora2Pg语言中，一个迁移就是一个项目。

配置将驱动整个项目。当你运行它的时候，Ora2Pg会创建几个目录，里面有所有从Oracle提取的数据。

```
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.
```

正如你所看到的，可以直接执行的脚本被生成。 然后，生成的数据可以很好地导入PostgreSQL中。要准备好在这里和那里改变程序。并非所有的东西都能自动迁移，所以人工干预是必要的。

2.5 常见的陷阱

有一些非常基本的语法元素在Oracle中起作用，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隐患。当然，这个列表绝不是完整的，但它应该为你指出正确的方向。

在Oracle中，你可能会遇到下面的语句。

```
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，这种语句很容易解决。

接下来你可能会发现下面的情况。

```
SELECT sysdate FROM dual;
```

PostgreSQL既没有sysdate函数，也没有双重函数。 双重函数部分很容易解决，因为你可以简单地创建一个返回一行的VIEW函数。在Oracle中，双函数的工作原理如下。

```
SQL> desc dual
Name Null? Type
----------------------------------------- -------- -
DUMMY VARCHAR2(1)
SQL> select * from dual;
D
-
X
```

在PostgreSQL中，同样可以通过创建以下VIEW函数来实现。

```
CREATE VIEW dual AS SELECT 'X' AS dummy;
```

sysdate函数也很容易解决。它可以用clock_timestamp()函数代替。

另一个常见的问题是缺乏数据类型，如VARCHAR2，以及缺乏只有Oracle支持的特殊函数。解决这些问题的一个好办法是安装orafce扩展，它提供了大部分通常需要的东西，包括最常用的函数。当然，查看https://github.com/orafce/orafce以了解更多关于orafce扩展的信息是有意义的。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，orafce扩展有助于确保73%的Oracle SQL可以在PostgreSQL上执行而无需修改（由NTT完成）。 最常见的隐患之一是Oracle处理外层连接的方式。

请看下面的例子

```
SELECT employee_id, manager_id
 FROM employees
 WHERE employees.manager_id(+) = employees.employee_id;
```

这种语法不是由PostgreSQL提供的，将来也不会有。 因此，这个连接必须被改写成一个适当的外连接。

在本章中，你已经学到了一些关于如何从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是相当容易的。其原因是，Oracle可能很昂贵，而且时常有点麻烦。这同样适用于Informix。然而，Informix和Oracle都有一个重要的共同点：CHECK约束被正确地兑现，数据类型被正确地处理。一般来说，我们可以有把握地认为，这些商业系统中的数据基本上是正确的，没有违反数据完整性和常识的最基本规则。

我们的下一个候选人则不同。你所知道的关于商业数据库的许多事情在MySQL中并不正确。术语NOT NULL对MySQL没有什么意义（除非你明确使用严格模式）。在Oracle、Informix、Db2和我所知道的所有其他系统中，NOT NULL是一条在所有情况下都要遵守的法律。MySQL在默认情况下并不重视这些约束。(虽然，公平地说，这在最近的版本中已被改变。严格模式直到最近才被默认打开。然而，许多旧的数据库仍然使用旧的默认设置）。

在迁移的情况下，这引起了一些问题。你打算如何处理那些在技术上有问题的数据呢？如果你的NOT NULL列突然显示出无数的NULL条目，你打算如何处理？ MySQL并不只是在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0，所以事情可能变得非常糟糕。

3.处理 MySQL 和 MariaDB 中的数据

你可能可以想象，而且你可能已经注意到，当涉及到数据库时，我远非毫无偏见。然而，我并不想把这变成对MySQL/MariaDB的盲目抨击。我们的真正目标是看看为什么MySQL和MariaDB从长远来看会是如此痛苦。我有偏见是有原因的，我真的想指出为什么是这样。

我们将要看到的所有事情都是非常可怕的，对整个迁移过程有严重的影响。我已经指出，MySQL有些特殊，本节将试图证明我的观点。

同样，下面的例子假设我们使用的是没有开启严格模式的MySQL/MariaDB版本，本章最初写的时候就是这样（截至PostgreSQL 9.6）。从PostgreSQL 10.0开始，严格模式已经开启，所以我们在这里要读的大部分内容只适用于旧版本的MySQL/MariaDB。

让我们从创建一个简单的表开始。

```
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，这里没有什么特别之处。我们已经创建了一个由两列组成的表。第一列被明确地标记为NOT NULL。第二列应该包含一个数字值，它被限制在四位数。最后，我们添加了一个简单的行。你能看到一个即将爆炸的潜在地雷吗？很可能没有。然而，检查一下下面的列表

```
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，我们添加了一个四位数的数字，这本来就不应该起作用。然而，MariaDB却简单地改变了我的数据。当然，已经发出了警告，但这是不应该发生的，因为表的内容并不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.
```

表被创建了，就像以前一样，但与MariaDB/MySQL形成鲜明对比的是，PostgreSQL会出错，因为我们试图向表中插入一个明显不允许的值。如果数据库引擎不关心，那么明确定义我们想要的东西有什么意义呢？ 假设你中了彩票--你可能刚刚失去了几百万，因为系统已经决定了什么对你有利。

我一生都在与商业数据库作斗争，但我从未在任何昂贵的商业系统（Oracle、Db2、Microsoft SQL Server等）中看到过类似的事情。他们可能有自己的问题，但一般来说，数据就很好。

3.1 更改列定义

让我们看看如果要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)
```

正如你所看到的，这些数据又被修改了。它一开始就不应该在那里，而且又被重新修改了一遍。记住，你可能又损失了钱，或者其他一些好的资产，因为MySQL试图变得很聪明。

这就是PostgreSQL中发生的情况。

```
test=# INSERT INTO data VALUES (1, 34.5678);
INSERT 0 1
test=# SELECT * FROM data;
 id | data
----+-------
 1 | 34.57
(1 row)
```

现在，让我们更改列定义：

```
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会出错，它不允许我们对我们的数据做讨厌的事情。在任何重要的数据库中，预计也会发生同样的情况。这个规则很简单。PostgreSQL和其他系统不会允许我们破坏我们的数据。

然而，PostgreSQL允许你做一件事。

```
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该如何行动。在这个例子中，我们明确地告诉PostgreSQL将该列的内容除以10。开发人员可以明确地提供应用于数据的规则。PostgreSQL不会试图变得聪明，这是有原因的。

```
test=# SELECT * FROM data;
 id | data
----+------
 1 | 3.46
(1 row)
```

数据完全符合预期。

3.2 处理空值

我们不想把这变成一个为什么MariaDB不好的章节，但我想在这里补充一个最后的例子，我认为这个例子是非常重要的。

```
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)
```

很明显，MySQL和MariaDB认为空和零是一回事。让我试着用一个简单的比喻来解释这里的问题：如果你知道你的钱包是空的，这和不知道你有多少钱是不同的。在我写这几行字的时候，我不知道我带了多少钱（空=未知），但我100%肯定它比零要多得多（我很肯定地知道在从机场回家的路上有足够的钱给我心爱的汽车加油，如果你的口袋里什么都没有，这是很难做到的）。

这里有一些更可怕的消息。

```
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确实记得这个列应该是NOT NULL的；但是，它只是再次修改了你的数据。

3.3 期待问题

主要的问题是，我们可能在向PostgreSQL移动数据时遇到麻烦。试想一下，你想移动一些数据，而在PostgreSQL那边有一个NOT NULL的约束。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（有充分的理由），但如果你明确禁止它，它也可能不接受NULL值（有充分的理由）。在这种情况下，你必须做的是以某种方式修复数据，以确保它遵守你的数据模型的规则，这些规则的存在是有原因的。你不应该对此掉以轻心，因为你可能不得不改变那些实际上首先是错误的数据。

3.4 迁移数据和模式

现在我已经解释了为什么迁移到PostgreSQL是一个好主意，并概述了一些最重要的问题，现在是我解释我们最终摆脱MySQL/MariaDB的一些可能选项的时候了。

3.4.1 使用 pg_chameleon

从MySQL/MariaDB迁移到PostgreSQL的一个方法是使用Federico Campoli的工具pg_chameleon，它可以从GitHub（https://github.com/the4thdoctor/pg_chameleon）免费下载。它被明确地设计为将数据复制到PostgreSQL，并为我们做了很多工作，如转换模式。 基本上，该工具执行了以下四个步骤。

1. pg_chameleon工具从MySQL读取模式和数据，并在PostgreSQL中创建一个模式。
2. 它在PostgreSQL中存储MySQL的主连接信息。
3. 它在PostgreSQL中创建主键和索引。
4. 它从MySQL/MariaDB复制到PostgreSQL。

pg_chameleon工具提供对DDL的基本支持，如CREATE、DROP、ALTER TABLE和DROP PRIMARY KEY。然而，由于MySQL/MariaDB的特性，它并不支持所有的DDL。相反，它涵盖了最重要的功能。

然而，pg_chameleon还有更多的功能。我已经广泛地指出，数据并不总是它应该是或被期望是的样子。pg_chameleon处理这个问题的方法是丢弃垃圾数据并将其存储在一个叫做sch_chameleon.t_discarded_rows的表中。当然，这并不是一个完美的解决方案，但考虑到相当低质量的输入，这是我想到的唯一合理的解决方案。我们的想法是让开发人员决定如何处理所有被破坏的行。pg_chameleon真的没有办法决定如何处理被别人破坏的东西。

最近，已经进行了大量的开发工作，并在该工具中进行了大量的工作。因此，强烈建议你查看GitHub页面并阅读所有的文档。在写这本书的时候，功能和错误的修复正在增加。鉴于本章的范围有限，这里不可能全面覆盖。

> 存储过程、触发器等需要特殊处理，只能手动处理。pg_chameleon工具不能自动处理这些东西。

3.4.2 使用FDWs

如果我们想从MySQL/MariaDB转移到PostgreSQL，有不止一种方法可以成功。使用FDWs是pg_chameleon的一种替代方法，它提供了一种快速获取模式以及数据的方法，并将其导入PostgreSQL中。连接MySQL和PostgreSQL的能力已经存在了相当长的时间，因此FDWs绝对是一个可以利用的领域，对你有利。

基本上，mysql_fdw扩展的工作方式就像外面的其他FDW一样。 与其他不太知名的FDW相比，mysql_fdw扩展实际上相当强大，提供了以下功能。

- 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远程端执行，以获得更好的性能）
- 列下推（只从远程端获取需要的列；旧版本用于获取所有列，这导致更多的网络流量）
- 远程端的准备语句

使用mysql_fdw扩展的方法是利用IMPORT FOREIGN SCHEMA语句，它允许你将数据转移到PostgreSQL上。 幸运的是，在Unix系统上这是相当容易做到的。让我们来看看详细的步骤。

1.我们要做的第一件事是从GitHub上下载代码。

```
git clone https://github.com/EnterpriseDB/mysql_fdw.git
```

2.然后，运行以下命令来编译FDW。注意，在你的系统上，路径可能有所不同。在本章中，我假设MySQL和PostgreSQL都在/usr/local目录下，在你的系统中可能不是这样。

```
$ export PATH=/usr/local/pgsql/bin/:$PATH
$ export PATH=/usr/local/mysql/bin/:$PATH
$ make USE_PGXS=1
$ make USE_PGXS=1 install
```

3.一旦代码被编译，FDW就可以被添加到我们的数据库中。

```
CREATE EXTENSION mysql_fdw;
```

4.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');
```

5.一旦服务器被创建，我们就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');
```

6.最后，是时候进行真正的迁移了。要做的第一件事是导入模式。我建议先为链接表创建一个特殊的模式。

```
CREATE SCHEMA migration_schema;
```

7.当运行IMPORT FOREIGN SCHEMA语句时，我们可以使用这个模式作为目标模式，所有的数据库链接都将存储在这里。这样做的好处是，我们可以在迁移后方便地删除它。

8.一旦我们完成了IMPORT FOREIGN SCHEMA语句，我们就可以创建真正的表。最简单的方法是使用CREATE TABLE子句所提供的LIKE关键字。它允许我们复制一个表的结构并创建一个真正的、本地的PostgreSQL表。幸运的是，如果你要克隆的表只是一个FDW，这也是可行的。这里有一个例子。

```
CREATE TABLE t_customer
 (LIKE migration_schema.t_customer);
```

9.然后，我们可以对数据进行处理。

```
INSERT INTO t_customer
 SELECT * FROM migration_schema.t_customer
```

这实际上是我们可以纠正数据，消除大块的行，或者对数据做一些处理的地方。考虑到数据的低质量来源，在第一次移动数据后，应用约束条件等可能会很有用。这可能会让人不那么痛苦。

一旦数据被导入，我们就准备部署所有的约束、索引等等。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。一般来说，在MySQL的情况下，迁移可能是相当困难的。

4.总结

在这一章中，我们了解了如何将SQL语句迁移到PostgreSQL中，并且学会了将一个基本的数据库从其他系统迁移到PostgreSQL中。迁移是一个重要的话题，每天都有越来越多的人在采用PostgreSQL。

PostgreSQL 12有很多新的功能，比如改进的内置分区，以及很多其他的功能。在未来，我们将看到PostgreSQL所有领域的更多发展，特别是那些允许用户扩展更多，运行查询更快。我们还没有看到未来会有什么变化。