了解事务和锁

现在我们已经介绍了PostgreSQL 13的介绍，我们想把注意力集中在下一个重要的话题上。锁定对于任何类型的数据库都是一个重要的概念。仅仅了解它是如何工作的还不够，还需要写出适当的或更好的应用程序--从性能的角度来看，它也是必不可少的。如果不正确地处理锁，你的应用程序不仅可能很慢，而且还可能出现非常意外的行为。在我看来，锁是性能的关键，对它有一个很好的概述肯定会有帮助。因此，了解锁和事务对管理员和开发人员都很重要。在本章中，你将了解到以下内容。

- 使用PostgreSQL事务
- 了解基本的锁
- 利用FOR SHARE和FOR UPDATE
- 了解事务隔离级别
- 观察死锁和类似问题
- 利用咨询锁
- 优化存储和管理清理

在本章结束时，你将能够理解并以最有效的方式利用PostgreSQL事务。你将看到，许多应用程序可以从性能的提高中受益。

1.使用PostgreSQL事务

PostgreSQL为你提供了高度先进的事务机制，为开发者和管理员提供了无数的功能。在这一节中，我们将看一下事务的基本概念。 要知道的第一件重要的事情是，在PostgreSQL中，所有的东西都是一个事务。如果你向服务器发送一个简单的查询，它已经是一个事务了。下面是一个例子。

```
test=# SELECT now(), now();
 now | now
-------------------------------+-------------------------------
 2020-08-13 11:03:17.741316+02 | 2020-08-13 11:03:17.741316+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。

请记住，now()函数将返回事务时间。 因此，SELECT语句将总是返回两个相同的时间戳。如果你想要真实的时间，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，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
URL: https://www.postgresql.org/docs/13/sql-begin.html
```

BEGIN语句将确保一条以上的命令被打包到一个事务中。下面是它的工作原理。

```
test=# BEGIN;
BEGIN
test=*# SELECT now();
 now
-------------------------------
 2020-08-13 11:04:15.379104+02
(1 row)
test=*# SELECT now();
 now
-------------------------------
 2020-08-13 11:04:15.379104+02
(1 row)
test=*# COMMIT;
COMMIT
```

这里重要的一点是，两个时间戳将是相同的。正如我们前面提到的，我们正在谈论事务时间。 为了结束事务，可以使用COMMIT

```
test=# \h COMMIT
Command: COMMIT
Description: commit the current transaction
Syntax:
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
URL: https://www.postgresql.org/docs/13/sql-commit.html
```

这里有一些语法元素。你可以只使用COMMIT，COMMIT WORK，或COMMIT TRANSACTION。这三个命令的含义都是一样的。如果这还不够，还有更多。

```
test=# \h END
Command: END
Description: commit the current transaction
Syntax:
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
URL: https://www.postgresql.org/docs/13/sql-end.html
```

END子句与COMMIT子句相同。

ROLLBACK是COMMIT的对应项。它不是成功地结束一个事务，而是简单地停止该事务，而不会让其他事务看到事情，如下面的代码所示。

```
test=# \h ROLLBACK
Command: ROLLBACK
Description: abort the current transaction
Syntax:
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
URL: https://www.postgresql.org/docs/13/sql-rollback.html
```

有些应用程序使用ABORT而不是ROLLBACK。其含义是一样的。 在PostgreSQL 12中，新的东西是链式事务的概念。 这一切的意义是什么呢？下面的列表显示了一个例子。

```
test=# SHOW transaction_read_only;
 transaction_read_only
-----------------------
 off
(1 row)
test=# BEGIN TRANSACTION READ ONLY ;
BEGIN
test=*# SELECT 1;
 ?column?
----------
 1
(1 row)
test=*# COMMIT AND CHAIN;
COMMIT
test=*# SHOW transaction_read_only;
 transaction_read_only
-----------------------
 on
(1 row)
test=*# SELECT 1;
 ?column?
----------
 1
(1 row)
test=*# COMMIT AND NO CHAIN;
COMMIT
test=# SHOW transaction_read_only;
 transaction_read_only
-----------------------
 off
(1 row)
test=# COMMIT;
WARNING: there is no transaction in progress
COMMIT
```

让我们一步一步地看这个例子：

1. 显示 transaction_read_only 设置的内容。它是关闭的，因为在默认情况下，我们处于读/写模式。
2. 使用BEGIN启动一个只读事务。这将自动调整 transaction_read_only 变量。
3. 使用AND CHAIN提交事务，然后PostgreSQL会自动启动一个新的事务，其属性与之前的事务相同。

在我们的例子中，我们也将处于只读模式，就像之前的事务一样。不需要显式地打开一个新的事务并再次设置什么值，这可以大大减少应用程序和服务器之间的往返次数。如果一个事务被正常提交（=NO CHAIN），该事务的只读属性将消失。

1.1 处理事务中的错误

事务从开始到结束并不总是正确的。 事情可能因为某种原因而出错。然而，在PostgreSQL中，只有无错误的事务才能被提交。下面的列表显示了一个失败的事务，它由于一个除以0的错误而出错。

```
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=!# SELECT 1;
ERROR: current transaction is aborted, commands ignored until end of transaction
block
test=!# COMMIT;
ROLLBACK
```

请注意，除以零没有结果。

> 在任何适当的数据库中，类似这样的指令会立即出错，使语句失败。

需要指出的是，PostgreSQL会出错。在错误发生后，将不再接受任何指令，即使这些指令在语义和语法上是正确的。仍然有可能发出COMMIT。然而，PostgreSQL会回滚交易，因为这是当时唯一正确的做法。

1.2 使用保存点

在专业的应用程序中，要写出合理的长事务而不遇到一个错误是相当困难的。为了解决这个问题，用户可以利用称为SAVEPOINT的东西。顾名思义，保存点是事务中一个安全的地方，如果事情出了大错，应用程序可以返回。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，以确保应用程序能够始终返回到事务内部的这一点。正如你所看到的，这个保存点有一个名字，这个名字将在后面提到。

返回到名为a的保存点后，事务可以正常进行。代码已经跳回到了错误之前，所以一切都很正常。

一个事务内的保存点的数量实际上是无限的。我们已经看到客户在一次操作中拥有超过25万个保存点。PostgreSQL可以很容易地处理这个问题。

如果你想从一个事务内部删除一个保存点，有一个RELEASE SAVEPOINT命令。

```
test=# \h RELEASE
Command: RELEASE SAVEPOINT
Description: destroy a previously defined savepoint
Syntax:
RELEASE [ SAVEPOINT ] savepoint_name
URL: https://www.postgresql.org/docs/13/sql-release-savepoint.html
```

许多人问，如果你在事务结束后试图到达一个保存点，会发生什么？答案是，事务一结束，保存点的生命就会结束。换句话说，在交易完成后，没有办法返回到某一时间点。

1.3 事务性的DDLs

PostgreSQL有一个非常好的功能，不幸的是，这个功能在许多商业数据库系统中是不存在的。在PostgreSQL中，可以在一个事务块中运行DDL（改变数据结构的命令）。在一个典型的商业系统中，一个DDL将隐含地提交当前的事务。这在PostgreSQL中不会发生。

除了一些小的例外（DROP DATABASE、CREATE TABLESPACE、DROP TABLESPACE等等），PostgreSQL中的所有DDL都是事务性的，这是一个巨大的优势，对终端用户是一个真正的好处。

下面是一个例子。

```
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 | Collation | Nullable | Default
--------+--------+-----------+----------+---------
 id | bigint | | |
test=*# ROLLBACK;
ROLLBACK
test=# \d t_test
Did not find any relation named "t_test".
```

在这个例子中，一个表被创建和修改了，整个事务被中止了。正如你所看到的，没有隐含的COMMIT命令或任何其他奇怪的行为。PostgreSQL只是按照预期的方式行事。

如果你想部署软件，事务性DDL就特别重要。试想一下，运行一个内容管理系统（CMS）。如果一个新的版本发布了，你会想要升级。运行旧版本仍然是可以的；运行新版本也是可以的，但你真的不希望出现新旧混合的情况。因此，在一个事务中部署升级是非常有利的，因为它是一个原子性的升级操作。

> 为了促进良好的软件实践，我们可以将源码控制系统中几个单独编码的模块纳入一个部署事务。

2.了解基本的锁

在本节中，你将学习基本的锁定机制。其目的是了解锁的一般工作原理，以及如何正确地进行简单的应用。

为了向你展示事情是如何进行的，我们将创建一个简单的表。出于演示的目的，我将使用一个简单的INSERT命令向表中添加一条记录。

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

第一件重要的事情是，表可以被并发地读取。许多用户在同一时间读取相同的数据，不会互相阻塞。这使得PostgreSQL能够处理成千上万的用户而没有任何问题。

现在的问题是，如果读和写同时发生会怎样？这里有一个例子。让我们假设该表包含一条记录，其id=0。

Transaction 1	**Transaction 2**
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
User will see 1	SELECT * FROM t_test;
	User will see 0
COMMIT;	COMMIT;

有两个事务被打开。第一个将改变一行。然而，这并不是一个问题，因为第二个事务可以继续进行。它将返回UPDATE之前的旧行。这种行为被称为多版本并发控制（MVCC）。

> 一个事务只有在写事务在读事务启动之前已经提交的情况下才会看到数据。一个事务不能检查另一个活动连接所做的改变。一个事务只能看到那些已经被提交的变化

还有第二个重要的方面--许多商业或开源数据库仍然无法处理并发的读和写。在PostgreSQL中，这绝对不是一个问题--读和写可以并存。

> 写事务不会阻塞读事务。

在事务提交后，该表将包含1.如果两个人同时改变数据会发生什么？下面是一个例子。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
It will return 2	UPDATE t_test SET id = id + 1 RETURNING *;
	It will wait for transaction 1
COMMIT;	It will wait for transaction 1
	It will reread the row, find 2, set the value, and return 3
	COMMIT;

假设你想计算一个网站的点击次数。如果你运行前面的代码，不会有任何点击率丢失，因为PostgreSQL保证一个UPDATE语句在另一个之后执行。

> PostgreSQL只锁定受UPDATE影响的行。所以，如果你有1000条记录，理论上你可以在同一个表上运行1000个并发变化。

同样值得注意的是，你总是可以运行并发的读取。我们的两个写不会阻塞读。

2.1 避免典型错误和显式锁定

在我作为一个专业的PostgreSQL顾问（https://www.cybertec-postgresql.com）的生活中，我看到有几个错误经常被重复。如果说生活中存在常数，这些典型的错误绝对是一些永远不会改变的东西。

这是我最喜欢的：

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT max(id) FROM product;	SELECT max(id) FROM product;
User will see 17	User will see 17
User will decide to use 18	User will decide to use 18
INSERT INTO product ... VALUES (18, ...)	INSERT INTO product ... VALUES (18, ...)
COMMIT;	COMMIT;

在这种情况下，要么有一个重复的密钥违规，要么有两个相同的条目。这两个问题的变化都不那么吸引人。 解决这个问题的一个方法是使用显式表锁。下面的代码向我们展示了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:
 ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
 | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
URL: https://www.postgresql.org/docs/13/sql-lock.html
```

正如你所看到的，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。下面的列表显示了这些锁的作用。

- ACCESS SHARE：这种类型的锁被读取，只和ACCESS EXCLUSIVE冲突，ACCESS EXCLUSIVE是由DROP TABLE等设置的。实际上，这意味着如果一个表即将被丢弃，SELECT就不能启动。这也意味着DROP TABLE必须要等到读取事务完成后才能开始。
- ROW SHARE：PostgreSQL在SELECT FOR UPDATE/SELECT FOR SHARE的情况下使用这种锁。它与 EXCLUSIVE 和 ACCESS EXCLUSIVE 冲突。
- ROW EXCLUSIVE：这种锁由 INSERT, UPDATE, 和 DELETE 使用。它与SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE冲突。
- SHARE UPDATE EXCLUSIVE: 这种锁被CREATE INDEX CONCURRENTLY, ANALYZE, ALTER TABLE, VALIDATE, 和一些其他类型的ALTER TABLE，以及VACUUM（不是VACUUM FULL）使用。它与SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式相冲突。
- SHARE：当索引被创建时，SHARE锁将被设置。它与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE冲突。
- SHARE ROW EXCLUSIVE：这是由CREATE TRIGGER和某些形式的ALTER TABLE设置的，与ACCESS SHARE以外的所有东西冲突。
- EXCLUSIVE：这种类型的锁是迄今为止限制性最强的一种。它可以防止读和写。如果这个锁被一个事务占用了，其他人就不能对被影响的表进行读写。
- ACCESS EXCLUSIVE：这种锁可以防止并发的事务读和写。

考虑到PostgreSQL的锁定基础设施，我们之前概述的最大问题的一个解决方案是如下的。下面代码中的例子显示了如何锁定一个表。

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

请记住，这是一种相当讨厌的操作方式，因为在你的操作过程中，没有其他人可以读或写到表。 因此，应该不惜一切代价避免使用ACCESS EXCLUSIVE。

2.2 考虑替代解决方案

对于这个问题，有一个替代的解决方案。考虑一个例子，你被要求编写一个应用程序来生成发票号码。税务局可能要求你创建没有空白和无重复的发票号码。你将如何做到这一点？当然，一个解决方案是使用表锁。然而，你真的可以做得更好。下面是你可以做的，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，我们引入了一个名为t_watermark的表。它只包含一条记录。 首先将执行 WITH 命令。该行将被锁定和递增，并返回新的值。每次只有一个人可以做这个事情。CTE返回的值会在t_invoice表中使用。它被保证是唯一的。美中不足的是，在t_watermark表上只有一个简单的行锁，这导致发票表中没有读取被阻止。总的来说，这种方式更具可扩展性。

3.利用FOR SHARE和FOR UPDATE

有时，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，在数据库中进行一些修改。这是一个典型的SELECT FOR UPDATE的例子。

下面是一个例子，显示了SELECT经常以错误的方式执行。

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

这里的问题是，两个人可能会选择相同的未经处理的数据。对这些处理过的行所做的修改就会被覆盖掉。简而言之，将发生一个竞赛条件。

为了解决这个问题，开发人员可以利用SELECT FOR UPDATE。下面是它的使用方法。下面的例子将展示一个典型的场景。

```
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一样锁定记录。这意味着没有变化可以同时发生。所有的锁都会像往常一样在COMMIT时被释放。

如果一个SELECT FOR UPDATE命令正在等待另一个SELECT FOR UPDATE命令，你将不得不等待，直到另一个命令完成（COMMIT或ROLLBACK）。如果第一个事务不想结束，不管什么原因，第二个事务有可能永远等待。为了避免这种情况，可以使用SELECT FOR UPDATE NOWAIT。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;	
Some processing	SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;
Some processing	ERROR: could not obtain lock on row in relation tab

如果NOWAIT对你来说不够灵活，可以考虑使用lock_timeout。它将包含你想在锁上等待的时间量。你可以在每个会话级别上设置它。

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

在这种情况下，该值被设置为5秒。

虽然SELECT基本上不做锁定，但SELECT FOR UPDATE却可以很苛刻。 试想一下下面的业务流程：我们想让一架有200个座位的飞机满员。许多人想同时预订座位。在这种情况下，可能会发生以下情况。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT ... FROM flight LIMIT 1 FOR UPDATE;	
Waiting for user input	SELECT ... FROM flight LIMIT 1 FOR UPDATE;
Waiting for user input	It has to wait

问题是，每次只能预订一个座位。有可能有200个座位，但每个人都必须等待第一个人。当第一个座位被封锁的时候，其他人就不能预订座位了，即使人们并不关心最后得到哪个座位。

SELECT FOR UPDATE SKIP LOCKED将解决这个问题。让我们先创建一些样本数据。

```
test=# CREATE TABLE t_flight AS
 SELECT * FROM generate_series(1, 200) AS id;
SELECT 200
```

现在，神奇的事情来了。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;	SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;
It will return 1 and 2	It will return 3 and 4

如果每个人都想获取两行数据，我们就可以同时为100个并发事务提供服务，而不必担心事务阻塞的问题。

> 请记住，等待是最慢的执行方式。如果一次只能有一个事务处于活动状态，那么如果你的真正问题是由一般的锁定和冲突事务引起的，那么购买更昂贵的服务器是没有意义的。

然而，还有一点。在某些情况下，FOR UPDATE会产生意想不到的后果。大多数人没有意识到FOR UPDATE会对外键产生影响的事实。让我们假设我们有两个表：一个用于存储货币，另一个用于存储账户。下面的代码显示了这方面的一个例子。

```
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：

Transaction 1	**Transaction 2**
BEGIN;	
SELECT * FROM t_account FOR UPDATE;	BEGIN;
Waiting for user to proceed	UPDATE t_currency SET id = id * 10;
Waiting for user to proceed	It will wait on transaction 1

虽然在账户上有一个SELECT FOR UPDATE命令，但货币表的UPDATE命令将被阻止。这是必要的，因为，否则就有可能完全破坏外键约束。因此，在一个相当复杂的数据结构中，你很容易在一个最不希望出现的区域（一些非常重要的查询表）出现争执。

除了FOR UPDATE之外，还有FOR SHARE、FOR NO KEY UPDATE和FOR KEY SHARE。 下面的列表描述了这些模式的实际含义。

- FOR NO KEY UPDATE：这个和FOR UPDATE很相似。然而，锁的作用较弱，因此，它可以与SELECT FOR SHARE共存。
- FOR SHARE：FOR UPDATE是非常强大的，它的工作假设是你肯定会改变行。FOR SHARE则不同，因为不止一个事务可以同时持有FOR SHARE锁。
- FOR KEY SHARE：它的行为类似于FOR SHARE，只是锁的作用比较弱。它将阻止FOR UPDATE，但不会阻止FOR NO KEY UPDATE。

这里最重要的是简单地尝试一下，观察一下会发生什么。改善锁定行为真的很重要，因为它可以极大地提高你的应用程序的可扩展性。

4.了解事务隔离级别

到现在为止，你已经看到了如何处理锁，以及一些基本的并发性。在本节中，你将学习事务隔离。对我来说，这是现代软件开发中最被忽视的话题之一。只有一小部分软件开发者真正意识到了这个问题，这反过来又导致了令人匪夷所思的错误。

下面是一个可能发生的例子。

Transaction 1	Transaction 2
BEGIN;	
SELECT sum(balance) FROM t_account;	
User will see 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	
User will see 400	
COMMIT;	

大多数用户实际上会期望第一笔事务总是返回300，而不管第二笔事务如何。然而，这并不正确。默认情况下，PostgreSQL运行在READ COMMITTED事务隔离模式下。这意味着事务内的每个语句都会得到一个新的数据快照，这个快照在整个查询过程中是不变的。

> 一个SQL语句将对同一快照进行操作，并在运行时忽略并发事务的变化。

如果你想避免这种情况，你可以使用 TRANSACTION ISOLATION LEVEL REPEATABLE READ。在这个事务隔离级别中，一个事务将在整个事务中使用同一个快照。下面是将发生的情况。

Transaction 1	Transaction 2
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
SELECT sum(balance) FROM t_account;	
User will see 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	SELECT sum(balance) FROM t_account;
User will see 300	User will see 400
COMMIT;	

正如我们所概述的，第一笔事务将冻结其数据的快照，并在整个事务过程中为我们提供恒定的结果。如果你想运行报告，这个功能尤其重要。报告的第一页和最后一页应该始终是一致的，并在相同的数据上操作。 因此，可重复读取是一致性报告的关键。 请注意，与隔离有关的错误不会总是即时出现。 有时，麻烦是在一个应用程序被转移到生产中多年后才被注意到的。

> 可重复读取并不比读取承诺的费用高。没有必要担心性能上的损失。对于正常的在线事务处理（OLTP），读提交有各种优势，因为可以更早地看到变化，而且发生意外错误的几率通常更低。

4.1 考虑序列化快照隔离事务

在读承诺和可重复读的基础上，PostgreSQL提供了可序列化的快照隔离（SSI）事务。所以，总的来说，PostgreSQL支持三个隔离级别。请注意，不支持 "已读未提交"（在一些商业数据库中仍然是默认的）：如果你试图启动一个 "已读未提交 "的事务，PostgreSQL会默默地映射到 "已读提交"。让我们回到可序列化的隔离级别。

> 如果你想了解更多关于这个隔离级别的信息，可以考虑查看https://wiki.postgresql.org/wiki/Serializable。

可序列化隔离背后的想法很简单；如果已知一个事务在只有一个用户时能正常工作，那么在选择这个隔离级别时，它在并发的情况下也会正常工作。然而，用户必须做好准备；事务可能会失败（通过设计）并出错。除此以外，还必须支付性能损失。

> 只有当你对数据库引擎内部发生的事情有相当的了解时，才考虑使用可序列化的隔离。

5.观察死锁和类似问题

死锁是一个重要的问题，可能发生在每个数据库中。 基本上，如果两个事务必须互相等待，就会发生死锁。

在本节中，你将看到这种情况如何发生。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：

Transaction 1	Transaction 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;	
Waiting on transaction 2	UPDATE t_deadlock SET id = id * 10 WHERE id = 1;
Waiting on transaction 2	Waiting on transaction 1
	Deadlock will be resolved after 1 second (deadlock_timeout)
COMMIT;	ROLLBACK;

一旦检测到死锁，将显示以下错误消息

```
psql: 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甚至好心地告诉我们哪一行引起了冲突。在我的例子中，万恶的根源是一个元组，（0，1）。你在这里看到的是ctid，它是表中某一行的唯一标识符。它告诉我们一个行在表中的物理位置。在这个例子中，它是第一块（0）中的第一行。

如果该行对你的事务仍是可见的，甚至有可能查询该行。 下面是它的工作原理。

```
test=# SELECT ctid, * FROM t_deadlock WHERE ctid = '(0, 1)';
ctid | id
-------+-----
(0,1) | 10
(1 row)
```

请记住，如果某条记录已经被删除或修改，这个查询可能不会返回。

然而，这并不是唯一可能导致死锁的情况，而是可能导致事务失败。事务也可能因为各种原因而不能被序列化。下面的例子显示了可能发生的情况。为了使这个例子有效，我假设你仍然有两行，id=1和id=2。

Transaction 1	**Transaction 2**
BEGIN ISOLATION LEVEL REPEATABLE READ;	
SELECT * FROM t_deadlock;	
Two rows will be returned	
	DELETE FROM t_deadlock;
SELECT * FROM t_deadlock;	
Two rows will be returned	
DELETE FROM t_deadlock;	
The transaction will error out	
ROLLBACK; - we cannot COMMIT anymore	

在这个例子中，有两个并发的事务在工作。只要第一个事务只是在选择数据，一切都很好，因为PostgreSQL可以很容易地保持静态数据的假象。但是如果第二个事务提交了一个DELETE命令，会发生什么？只要只有读，就没有问题。当第一个事务试图删除或修改此时已经死亡的数据时，麻烦就开始了。对于PostgreSQL来说，唯一的解决办法是由于我们的事务造成的冲突而出错。

```
test=# DELETE FROM t_deadlock;
psql: ERROR: could not serialize access due to concurrent update
```

实际上，这意味着终端用户必须准备好处理错误的事务。如果出了问题，正确编写的应用程序必须能够再次尝试

6.利用咨询锁

PostgreSQL有高效和复杂的交易机制，能够以真正精细和高效的方式处理锁。 几年前，人们想出了用这种代码来使应用程序相互同步的想法。因此，咨询锁就诞生了。

当使用咨询锁时，重要的是要提到，它们不会像普通锁那样在COMMIT时消失。因此，确保解锁是以一种完全可靠的方式正确完成的，这一点真的很重要。

如果你决定使用咨询锁，你真正锁定的是一个数字。所以，这不是关于行或数据，它真的只是一个数字。下面是它的工作原理。

Session 1	**Session 2**
BEGIN;	
SELECT pg_advisory_lock(15);	
	SELECT pg_advisory_lock(15);
	It has to wait
COMMIT;	It still has to wait
SELECT pg_advisory_unlock(15);	It is still waiting
	Lock is taken

第一个事务将锁定15。第二个事务必须等待，直到这个数字再次被解锁。第二个事务甚至会在第一个事务提交后等待。这一点非常重要，因为你不能依靠事务的结束会很好地、奇迹般地为你解决事情。 如果你想解锁所有被锁定的数字，PostgreSQL提供了pg_advisory_unlock_all()函数来做这件事。

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

有时，你可能想看看你是否能得到一个锁，如果不可能的话就出错。为了达到这个目的，PostgreSQL提供了一些函数；要查看所有这些可用函数的列表，请在命令行输入

```
\df *try*advisory*
```

7.优化存储和管理清理

事务是PostgreSQL系统的一个组成部分。 然而，事务是有一个小小的代价的。正如我们在本章中已经表明的，有时，并发用户会得到不同的数据。不是每个人都会得到查询返回的相同数据。除此之外，DELETE和UPDATE不允许实际覆盖数据，因为ROLLBACK不起作用。如果你碰巧处于一个大型的DELETE操作中，你不能确定你是否能够COMMIT。

除此之外，在你执行DELETE操作时，数据仍然是可见的，有时甚至在你的修改早已完成后，数据仍然可见。 因此，这意味着清理工作必须以异步方式进行。事务不能清理自己的烂摊子，任何COMMIT/ROLLBACK都可能太早，无法处理死行。

解决这个问题的方法是VACUUM。下面的代码块为你提供了一个语法概述

```
test=# \h VACUUM
Command: VACUUM
Description: garbage-collect and optionally analyze a database
Syntax:
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ table_and_columns [, ...] ]
where option can be one of:
 FULL [ boolean ]
 FREEZE [ boolean ]
 VERBOSE [ boolean ]
 ANALYZE [ boolean ]
 DISABLE_PAGE_SKIPPING [ boolean ]
 SKIP_LOCKED [ boolean ]
 INDEX_CLEANUP [ boolean ]
 TRUNCATE [ boolean ]
 PARALLEL integer
and table_and_columns is:
 table_name [ ( column_name [, ...] ) ]
URL: https://www.postgresql.org/docs/13/sql-vacuum.html
```

VACUUM将访问所有可能包含修改的页面，并找到所有的死空间。然后，找到的自由空间被关系的自由空间图（FSM）所跟踪。

请注意，在大多数情况下，VACUUM不会缩减表的大小。相反，它将跟踪并找到现有存储文件内的自由空间。

> 在VACUUM之后，表通常会有相同的大小。如果一个表的末尾没有有效的行，文件大小会下降，尽管这种情况很少。这不是规则，而是例外。

这对终端用户意味着什么，将在本章的 "工作中观察VACUUM "小节中概述。

7.1 配置 VACUUM 和 autovacuum

在PostgreSQL项目的早期，人们不得不手动运行VACUUM。幸运的是，那些日子早就过去了。现在，管理员可以依靠一个叫做autovacuum的工具，它是PostgreSQL服务器基础设施的一部分。它自动处理清理工作并在后台工作。它每分钟唤醒一次（见postgresql.conf中autovacuum_naptime = 1），检查是否有工作要做。如果有工作，autovacuum将最多分叉三个工作进程（见postgresql.conf中的autovacuum_max_workers）。

主要的问题是，autovacuum什么时候触发工作进程的创建？

> 实际上，autovacuum进程本身并没有分叉进程。相反，它告诉主进程这样做。这样做是为了避免在发生故障时出现僵尸进程，并提高健壮性。

这个问题的答案同样可以在postgresql.conf中找到，如以下代码所示。

```
autovacuum_vacuum_threshold = 50
autovacuum_analyze_threshold = 50
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
autovacuum_vacuum_insert_threshold = 1000
```

autovacuum_vacuum_scale_factor命令告诉PostgreSQL，如果一个表有20%的数据被改变，就值得进行vacuuming。问题是，如果一个表只有一条记录，一个变化已经是100%了。为了清理一条记录而分出一个完整的过程是完全没有意义的。 因此，autovacuum_vacuuum_threshold说，我们需要20%，而且这个20%必须至少是50行。否则，VACUUM就不会启动。当涉及到创建优化器统计时，也使用了同样的机制。我们需要10%和至少50行来证明新的优化器统计数据的合理性。 理想情况下，autovacuum会在正常的VACUUM期间创建新的统计信息，以避免不必要的查表次数。

然而，还有更多。在过去，autovacuum不会被仅由INSERT语句组成的工作负载所触发，这可能是一个大问题。新增的autovacuum_vacuum_insert_threshold参数正是为了解决这种问题。现在，即使数据库中只有INSERT语句，PostgreSQL 13也会触发自动真空活动。

7.2 深入研究与事务环绕相关的问题

在postgresql.conf中还有两个设置，对于真正利用PostgreSQL来说，理解它们是相当重要的。正如我们已经说过的，理解VACUUM是性能的关键。

```
autovacuum_freeze_max_age = 200000000
autovacuum_multixact_freeze_max_age = 400000000
```

要理解整个问题，重要的是要理解PostgreSQL如何处理并发。PostgreSQL的事务机制是基于对事务ID和事务所处状态的比较。

让我们看一个例子。如果我是事务ID 4711，如果你碰巧是4712，我不会看到你，因为你还在运行。如果我是交易ID 4711，但你是事务ID 3900，我将看到你。如果你的事务失败了，我可以安全地忽略所有由你失败的事务产生的行。

问题如下：事务ID是有限的，不是无限的。在某些时候，它们会开始缠绕在一起。在现实中，这意味着第5个事务可能实际上是在第8亿个事务之后。PostgreSQL如何知道哪个是第一个？它通过存储一个水印来实现。在某些时候，这些水印会被调整，而这正是VACUUM开始发挥作用的时候。通过运行VACUUM（或autovacuum），你可以确保水印的调整方式总是有足够的未来交易ID可以使用。

> 不是每个事务都会增加事务ID计数器。只要一个事务还在读，它就只有一个虚拟事务ID。这确保了事务ID不会被过快烧毁。

autovacuum_freeze_max_age命令定义了一个表的pg_class.relfrozenxid字段在强制进行VACUUM操作之前可以达到的最大事务数（age），以防止表内的事务ID缠绕。这个值是相当低的，因为它对堵塞的清理也有影响（堵塞或提交日志是一个数据结构，每个事务存储两个比特，这表明一个事务是否正在运行，中止，提交，或仍然在一个子事务中）。

autovacuum_multixact_freeze_max_age命令配置了表的pg_class.relminmxid字段在强制进行VACUUM操作之前可以达到的最大年龄，以防止表内的multixact ID缠绕。冻结元祖是一个重要的性能问题，在第6章 "优化查询以获得良好的性能 "中会有更多关于这个过程的内容，我们将讨论查询优化。

一般来说，在保持操作安全性的同时，试图减少VACUUM的负载是一个好主意。对大表进行VACUUM操作可能会很昂贵，因此关注这些设置是非常有意义的

7.3 关于 VACUUM FULL 的一句话

你也可以用VACUUM FULL来代替普通的VACUUM。然而，我真的想指出，VACUUM FULL实际上锁定了表并重写了整个关系。在一个小表的情况下，这可能不是一个问题。 但是，如果你的表很大，表锁可以在几分钟内杀死你 VACUUM FULL会阻止即将到来的写操作，因此，一些与你的数据库交谈的人可能会感觉到它实际上已经停机了。 因此，我们建议要非常谨慎。

> 为了摆脱VACUUM FULL，我推荐你查看pg_squeeze（http://www.cybertec.at/introducing-pg_squeeze-a-postgresql-extension-to-auto-rebuild-bloated-tables/），它可以重写一个表而不阻塞写入。

7.4 观察VACUUM的工作情况

现在，是时候看看VACUUM的操作了。我把这部分内容放在这里，因为我作为PostgreSQL顾问和支持者的实际工作（http://www.postgresql-support.com/）表明，大多数人对存储方面发生的事情只有非常模糊的认识。

再次强调这一点，在大多数情况下，VACUUM不会缩减你的表；空间通常不会返回到文件系统中。

下面是我的例子，它显示了如何用自定义的自动真空设置创建一个小表。该表充满了100000条记录。

```
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100000行的简单表。注意，可以关闭特定表的自动真空功能。通常情况下，这对大多数应用来说不是一个好主意。然而，在一个角落里，autovacuum_enabled = off是有意义的。只需考虑一个生命周期很短的表。如果开发者已经知道整个表将在几秒钟内被丢弃，那么清理元祖就没有意义了。在数据仓库中，如果你使用表作为暂存区域，就会出现这种情况。在这个例子中，VACUUM被关闭，以确保在后台没有任何事情发生。 你所看到的一切是由我触发的，而不是由某个进程触发的。

首先，考虑通过使用以下命令检查表的大小。

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

pg_relation_size命令返回一个表的大小，单位是字节。pg_size_pretty命令将把这个数字转变成人类可读的数字。

然后，表内的所有行将使用一个简单的UPDATE语句进行更新，如以下代码所示。

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

所发生的事情对理解PostgreSQL非常重要。数据库引擎必须复制所有的行。为什么呢？首先，我们不知道事务是否会成功，所以数据不能被覆盖。第二个重要的方面是，一个并发的事务可能还在看到旧版本的数据。 UPDATE操作会复制行。 从逻辑上讲，改变之后，表的大小会变大。

```
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返回到文件系统。

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

正如我们前面所说，在大多数情况下，VACUUM不会将空间返回到文件系统。相反，它将允许空间被重新使用。因此，该表根本不会缩减。

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty
----------------
 7080 kB
(1 row)
```

然而，下一个UPDATE不会使表增长，因为它将吃掉表内的自由空间。只有第二次UPDATE才会使表再次增长，因为所有的空间都没有了，所以需要额外的存储。

```
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)
```

如果我必须决定你在读完这本书后应该记住一件事，那就是这一点。了解存储是性能和一般管理的关键。

让我们再运行一些查询。

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

同样，尺寸也没有变化。让我们看看表格里面有什么。

```
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命令是一个行在磁盘上的物理位置。通过使用ORDER BY ctid DESC，你基本上会按照物理顺序向后读取表。 你为什么要关心这个问题呢？因为在表的末端有一些非常小的值和一些非常大的值。下面的代码显示了当数据被删除时，表的大小是如何变化的

```
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)
```

虽然只有2%的数据被删除，但表的大小却减少了三分之二。其原因是，如果VACUUM只发现表中某个位置之后的死行，它可以将空间返回到文件系统中。这是唯一的一种情况，在这种情况下，你会真正看到表的大小下降。当然，普通用户无法控制数据在磁盘上的物理位置。因此，除非所有的行都被删除，否则存储消耗很可能会保持一定程度的不变。

> 到底为什么在表的最后会有这么多的小值和大值？在表最初填充了100,000行后，最后一个区块并没有完全填满，所以第一个UPDATE会把最后一个区块填满变化。这就把表的末尾洗了一下。在这个精心制作的例子中，这就是表末尾奇怪布局的原因。

在现实世界的应用中，这一观点的影响怎么强调都不为过。不真正了解存储，就没有性能调整。

7.5 通过使用太旧的快照来限制事务

VACUUM做得很好，它将根据需要回收自由空间。然而，VACUUM什么时候才能真正清理出行并将其变成自由空间呢？规则是这样的：如果一个行不能再被任何人看到，它就可以被回收。在现实中，这意味着所有不再被看到的东西，即使是最古老的活动事务，都可以被认为是真正的死亡。

这也意味着，真正长的事务可以推迟相当长的时间来清理。逻辑上的后果是表的膨胀。表的增长将超出比例，性能将趋于下降。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一个事务的持续时间。Oracle管理员将熟悉快照太旧的错误。从PostgreSQL 9.6开始，这个错误信息也有了。然而，它更像是一个功能，而不是不良配置的意外副作用（在Oracle中它实际上是）。

为了限制快照的寿命，你可以利用PostgreSQL的配置文件postgresql.conf中的一个设置，其中有所有需要的配置参数。

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

如果这个变量被设置了，事务将在一定时间后失败。 请注意，这个设置是在实例层面上的，它不能在会话中设置。通过限制事务的年龄，疯狂的长事务的风险将大大降低。

7.6 使用更多的 VACUUM 特性

多年来，VACUUM一直在稳步改进。在本节中，你将了解到一些最近的改进。

在许多情况下，VACUUM可以跳过页面。当可见性地图显示一个区块对所有人都是可见的时候，这一点尤其真实。VACUUM也可能跳过一个被其他事务大量使用的页面。DISABLE_PAGE_SKIPPING禁止这种行为，并确保所有的页面在这次运行中被清理。

还有一种改进VACUUM的方法是使用SKIP_LOCKED：这里的想法是确保VACUUM不损害并发性。如果使用SKIP_LOCKED，VACUUM将自动跳过不能立即锁定的关系，从而避免冲突解决。这种功能在严重并发的情况下可能非常有用。

VACUUM的一个重要而有时被忽视的方面是需要清理索引。在VACUUM成功地处理了一个堆之后，索引被处理了。如果你想防止这种情况发生，你可以利用INDEX_CLEANUP。默认情况下，INDEX_CLEANUP是真的，但是根据你的工作负载，你可能会决定在一些罕见的情况下跳过索引清理。那么，那些罕见的情况是什么呢？为什么有人可能不想清理索引呢？答案很简单：如果你的数据库有可能很快因为事务缠绕而关闭，那么尽快运行VACUUM是有意义的。如果你在停机和某种推迟的清理之间有一个选择，你应该选择快速的VACUUM来保持你的数据库的活力。

8.总结

在本章中，你了解了事务、锁定及其逻辑含义，以及PostgreSQL事务机制在存储、并发和管理方面的一般架构。你看到了行是如何被锁定的，以及PostgreSQL中的一些功能。

在第3章 "使用索引 "中，你将了解到数据库工作中最重要的主题之一：索引。你还将了解PostgreSQL的查询优化器，以及各种类型的索引和它们的行为。

9.问题

- 事务的目的是什么？
- 在PostgreSQL中一个事务可以有多长？
- 什么是事务隔离？
- 我们应该避免表锁吗？
- 事务与VACUUM有什么关系？

这些问题的答案可以在GitHub仓库中找到（https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition）。