PostgreSQL 13 概述

PostgreSQL已经走过了漫长的道路。30多年来的不断发展为我们提供了一个最卓越的开源产品。多年来，PostgreSQL已经变得越来越流行。 PostgreSQL 13提供了更多的功能，将给解决方案带来比以往更大的推动力。许多新功能为未来的进一步发展打开了大门，并将使开发者在未来几十年内实现最先进的技术。在这一章中，你将被介绍到这些新的功能，并将得到一个概览，了解哪些功能得到了改进、增加，甚至改变。

将涵盖以下主题。

- PostgreSQL 13的新功能是什么?
- SQL和开发人员相关的特性
- 备份、恢复和复制
- 性能相关的主题
- 存储器相关的主题

所有相关的特性都将被涵盖。当然，总是有更多的内容，还有数以千计的其他微小变化已经进入PostgreSQL 13。在本章中你将看到的是新版本的亮点。

1. PostgreSQL 13的新功能是什么?

PostgreSQL 13是一个重要的里程碑，许多大大小小的功能这次都进入了数据库核心。在本章中，你将被介绍到PostgreSQL世界中最重要的发展。 让我们开始吧，看看开发者们都想出了什么。

1.1 挖掘SQL和开发人员相关的话题

PostgreSQL 13提供了一些对开发者特别重要的新功能。

1.1.1 改进psql的命令行处理

在PostgreSQL 13中，已经提交了一个小的扩展，基本上可以帮助更好地跟踪psql中的会话状态。现在你可以看到一个事务是否在成功运行。让我们看一下一个简单的列表。

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

这个代码块有什么特别之处？需要注意的是，提示符已经改变。原因是%x已经被添加到PROMPT1和PROMPT2中。我们现在可以立即看到一个事务是否正在进行，是否处于困境，或者是否没有事务正在进行。这使得命令行处理变得更容易一些，而且希望能减少错误的发生。

1.1.2 改进pgbench

pgbench是最常用的PostgreSQL数据库基准测试工具之一。在旧版本中，pgbench创建了一些标准表。随着PostgreSQL 13的引入，默认的数据集现在可以被分区（开箱即用）。

下面是发生的情况。

```
% pgbench -i -s 100 --partitions=10
...
done in 19.70 s (drop tables 0.00 s, create tables 0.03 s, generate 7.34 s,
vacuum
10.24 s, primary keys 2.08 s).
test=# \d+
 List OF relations
 Schema |     Name |                TYPE | ...
--------+----------------------+-------------------+ ...
 public | pgbench_accounts     |  partitioned TABLE | ...
 public | pgbench_accounts_1   |  TABLE | ...
 public | pgbench_accounts_10 |  TABLE | ...
 public | pgbench_accounts_2   |  TABLE | ...
 public | pgbench_accounts_3   |  TABLE | ...
...
```

-- 分区告诉pgbench要创建多少个分区。现在可以更容易地对一个分区的表和一个非分区的默认数据集进行基准测试。

1.1.3 更轻松地生成随机 UUID

在旧版本中，我们必须加载一个扩展来处理UUIDs。许多用户不知道这些扩展实际上是存在的，或者由于某种原因不想启用它们。在PostgreSQL 13中，不需要任何扩展就可以轻松地生成随机UUID。下面是它的工作原理。

```
test=# SELECT gen_random_uuid();
 gen_random_uuid
--------------------------------------
 ca1754dc-b5d5-442b-a40b-9c6a9d51a9a1
(1 row)
```

1.1.4 更快地删除数据库

更快地删除数据库的能力是一个重要的功能，绝对是我最喜欢的新功能之一。问题是什么呢?

```
postgres=# DROP DATABASE test;
ERROR: database "test" is being accessed by other users
DETAIL: There is 1 other session using the database.
```

一个数据库只有在没有其他人连接到该数据库时才能被放弃。对于许多用户来说，这很难实现。通常情况下，新的连接不断出现，运行ALTER DATABASE来阻止新的连接并放弃现有的连接以杀死数据库，这实在是太麻烦了。 从终端用户的角度来看，WITH（强制）选项极大地简化了事情。

```
postgres=# DROP DATABASE test WITH (force);
DROP DATABASE
```

无论系统中发生了什么，数据库都会被丢弃，这使得这个过程更加可靠。

1.1.5 添加ALTER TABLE ... DROP EXPRESSION...

PostgreSQL有能力将一个表达式的输出物化。下面的列表显示了如何使用一个生成的列。

```
test=# CREATE TABLE t_test (
 a int,
 b int,
 c int GENERATED ALWAYS AS (a * b) STORED
);
CREATE TABLE
test=# INSERT INTO t_test (a, b) VALUES (10, 20);
INSERT 0 1
```

如您所见，表格中添加了一行:

```
test=# SELECT * FROM t_test;
 a | b | c
----+----+-----
 10 | 20 | 200
(1 row)
```

现在的问题是：我们怎样才能再次摆脱生成的表达式？ PostgreSQL 13有了答案。

```
test=# ALTER TABLE t_test ALTER COLUMN c DROP EXPRESSION ;
ALTER TABLE
```

c 现在是一个普通的列，就像所有其他列一样：

```
test=# \d t_test;
 Table "public.t_test"
 Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 a | integer | | |
 b | integer | | |
 c | integer | | |
```

现在你可以直接插入到c中，而不会出现错误。

1.2 利用性能的改进

PostgreSQL 13增加了一些性能改进。在这一节中，你将了解到一些更相关的改进，这些改进可以在实际生活中产生作用。当然，这里有成百上千的小改动，但本章的目的实际上是集中在能产生真正变化的最重要的方面。

1.2.1 B树索引去重

B-树是迄今为止PostgreSQL世界中最重要的索引结构。大多数的索引都是B-树。在PostgreSQL 13中，B-树得到了很大的改进。在旧版本中，相同的条目被单独存储在索引中。如果你有100万个hans的表项，索引实际上在数据结构中也会有100万个 "hans"的副本。在PostgreSQL 12中，典型的索引条目曾经是"（value, ctid）"。 这一点已经被改变了。现在PostgreSQL将更有效地处理重复的内容。

让我们创建一个例子，看看这是如何工作的。

```
test=# CREATE TABLE tab (a int, b int);
CREATE TABLE
test=# INSERT INTO tab SELECT id, 1 FROM generate_series(1, 5000000) AS id;
INSERT 0 5000000
```

在这种情况下，我们已经创建了500万行。a列有500万个不同的值，而b列包含500万个相同的条目。让我们创建两个索引，看看会发生什么。

```
test=# CREATE INDEX idx_a ON tab (a);
CREATE INDEX
test=# CREATE INDEX idx_b ON tab (b);
CREATE INDEX
```

表本身的大小在PostgreSQL 13中没有变化。我们将看到和以前一样的大小。

```
test=# \d+
 List of relations
 Schema | Name | Type | Owner | Persistence | Size | Description
--------+------+-------+-------+-------------+--------+-------------
 public | tab | table | hs | permanent | 173 MB |
(1 row)
```

然而，重要的是，idx_b索引的大小将与idx_a索引的大小不同。

```
test=# \di+
 List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
--------+-------+-------+-------+-------+-------------+--------+-------------
 public | idx_a | index | hs | tab | permanent | 107 MB |
 public | idx_b | index | hs | tab | permanent | 33 MB |
(2 rows)
```

正如你所看到的，PostgreSQL 13的效率要高得多，并且提供了一个更低的存储空间占用。较低的空间消耗将自动转化为较高的缓存命中率，因此，一般来说，性能更好。

> 如果你正在从旧版本迁移到PostgreSQL 13，考虑重新创建包含许多重复的索引，以确保你能利用这些改进。

1.3 增加增量排序的功能

PostgreSQL 13围绕索引和排序又有一个改进。 新版本提供了增量排序。这意味着什么呢？ 假设我们有两列：a和b。在我们的例子中，a已经被排序了，因为它有索引。如果我们有一个预排序的列表，只是想增加一些列呢？在这种情况下，增量排序就会发生。 为了说明这一点，让我们先放弃idx_b的索引。

```
test=# DROP INDEX idx_b;
DROP INDEX
```

我们只剩下一个索引（idx_a），它提供了一个按a排序的列表。下面的查询想要一个按a和b排序的结果。

```
test=# explain SELECT * FROM tab ORDER BY a, b;
 QUERY PLAN
-----------------------------------------------------------------------------
 Incremental Sort (cost=0.47..376979.43 rows=5000000 width=8)
 Sort Key: a, b
 Presorted Key: a
 -> Index Scan using idx_a on tab (cost=0.43..151979.43 rows=5000000 width=8)
(4 rows)
```

PostgreSQL将进行由idx_a索引提供的增量排序。在我的机器上的总运行时间大约是1.4秒（使用默认配置）。

```
test=# explain analyze SELECT * FROM tab ORDER BY a, b;
 QUERY PLAN
-------------------------------------------------------------------------------------
 Incremental Sort (cost=0.47..376979.43 rows=5000000 width=8)
 (actual time=0.167..1297.696 rows=5000000 loops=1)
 Sort Key: a, b
 Presorted Key: a
 Full-sort Groups: 156250 Sort Method: quicksort
 Average Memory: 26kB Peak Memory: 26kB
 -> Index Scan using idx_a on tab (cost=0.43..151979.43 rows=5000000 width=8)
 (actual time=0.069..773.260 rows=5000000 loops=1)
 Planning Time: 0.068 ms
 Execution Time: 1437.362 ms
(7 rows)
```

为了看看在PostgreSQL 12和之前会发生什么，我们可以暂时关闭增量排序。

```
test=# SET enable_incremental_sort TO off;
SET
test=# explain analyze SELECT * FROM tab ORDER BY a, b;
 QUERY PLAN
------------------------------------------------------------------------
 Sort (cost=765185.42..777685.42 rows=5000000 width=8)
 (actual time=1269.250..1705.754 rows=5000000 loops=1)
 Sort Key: a, b
 Sort Method: external merge Disk: 91200kB
 -> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=8)
 (actual time=0.032..311.777 rows=5000000 loops=1)
 Planning Time: 0.057 ms
 Execution Time: 1849.416 ms
(6 rows)
```

PostgreSQL将退回到旧的执行计划，这当然需要超过400毫秒的时间，这是相当可观的。但还有一点：内存占用率更低。旧的机制必须对整个数据集进行排序--新的机制几乎不需要任何内存可言。换句话说：增量排序不仅仅是一个性能问题。它还大大降低了内存消耗。

1.4 将 -j 8 添加到 reindexdb

reindexdb命令已经存在了很多年。它可以被用来轻松地重新索引整个数据库。在PostgreSQL 13中，reindexdb得到了扩展。它现在支持-j标志，这使得你可以一次使用超过一个CPU核心来重新索引一个数据库。

```
$ reindexdb --help
reindexdb reindexes a PostgreSQL database.
Usage:
 reindexdb [OPTION]... [DBNAME]
Options:
 -a, --all reindex all databases
 --concurrently reindex concurrently
 -d, --dbname=DBNAME database to reindex
 -e, --echo show the commands being sent to the server
 -i, --index=INDEX recreate specific index(es) only
 -j, --jobs=NUM use this many concurrent connections to reindex
...
```

如接下来的列表所示，性能差异可能是相当大的。

```
$ time reindexdb -j 8 database_name
real 0m6.789s
user 0m0.012s
sys 0m0.008s
$ time reindexdb database_name
real 0m24.137s
user 0m0.001s
sys 0m0.004s
```

在这种情况下，为了加快进程，使用了8个核心。请记住，只有当你有足够多的索引可以同时创建时，-j选项才会加速事情。如果你只有一个表和一个索引，就不会有任何改善。你有越多的大表，结果就会越好。

1.5 允许哈希聚合溢出到磁盘

当运行简单的GROUP BY语句时，PostgreSQL基本上有两个选项来执行这些类型的查询。

- 组聚合
- 哈希聚合

当组的数量真的很大时，通常会使用组的聚合。 如果你按电话号码对数十亿人进行分组，那么组的数量就很高，PostgreSQL无法在内存中完成。组聚合是执行这种类型查询的首选方法。第二个选择是散列聚合。假设你按性别对数十亿人进行分组。结果组的数量会很小，因为根本就只有少量的性别。然而，如果规划者弄错了呢？如果规划器对组的数量产生了一个错误的估计呢？从PostgreSQL 13开始，如果一个哈希聚合所使用的内存超过work_mem，就有可能使哈希聚合的数据溢出到磁盘。 为了看看会发生什么，我将首先运行一个简单的GROUP BY语句。

```
test=# explain SELECT a, count(*) FROM tab GROUP BY 1;
 QUERY PLAN
------------------------------------------------------------------------
 GroupAggregate (cost=0.43..204852.43 rows=5000000 width=12)
 Group Key: a
 -> Index Only Scan using idx_a on tab (cost=0.43..129852.43 rows=5000000
width=4)
(3 rows)
```

PostgreSQL预计组的数量是500万行，并去做组的聚合。但是，如果我们让索引变得更昂贵（random_page_cost），并告诉PostgreSQL真的有足够的内存来处理这个操作呢？让我们来试试。

```
test=# SET random_page_cost TO 100;
SET
test=# SET work_mem TO '10 GB';
SET
test=# explain SELECT a, count(*) FROM tab GROUP BY 1;
 QUERY PLAN
------------------------------------------------------------------------
 HashAggregate (cost=97124.00..147124.00 rows=5000000 width=12)
 Group Key: a
 -> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=4)
(3 rows)
```

规划器将决定一个哈希集合，并在内存中执行，具体如下。

```
test=# explain analyze SELECT a, count(*) FROM tab GROUP BY 1;
 QUERY PLAN
------------------------------------------------------------------------
 HashAggregate (cost=97124.00..147124.00 rows=5000000 width=12)
 (actual time=2466.159..3647.708 rows=5000000 loops=1)
 Group Key: a
 Peak Memory Usage: 589841 kB
 -> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=4)
 (actual time=0.061..615.607 rows=5000000 loops=1)
 Planning Time: 0.065 ms
 Execution Time: 3868.609 ms
(6 rows)
```

正如你所看到的，系统使用了大约600MB的内存（在峰值消耗时）来运行该操作。但是，如果我们把work_mem设置为刚好低于这个值，会发生什么？让我们来了解一下。

```
test=# SET work_mem TO '500 MB';
SET
test=# explain analyze SELECT a, count(*) FROM tab GROUP BY 1;
 QUERY PLAN
------------------------------------------------------------------------
 HashAggregate (cost=97124.00..147124.00 rows=5000000 width=12)
 (actual time=2207.005..3778.903 rows=5000000 loops=1)
 Group Key: a
 Peak Memory Usage: 516145 kB Disk Usage: 24320 kB
 HashAgg Batches: 4
 -> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=4)
 (actual time=0.054..545.678 rows=5000000 loops=1)
 Planning Time: 0.092 ms
 Execution Time: 3970.435 ms
(6 rows)
```

500MB的work_mem是可用的，但我们的操作会用到稍微多一点。 因此，PostgreSQL将把这些额外的数据发送到磁盘。如果我们进一步减少内存，PostgreSQL将回落到一个群集。 然而，在估计稍有错误的情况下，这个新功能确实能起到帮助作用。

1.6 加速 PL/pgSQL

PL/pgSQL是在PostgreSQL中运行存储过程的唯一最流行的语言。然而，PL/pgSQL的一些角落曾经相当缓慢。 PostgreSQL 13具有一些性能改进，以帮助使用PL/pgSQL的开发者运行更快的代码。下面是一个例子。

```
CREATE OR REPLACE FUNCTION slow_pi() RETURNS double precision AS $$
 DECLARE
 a double precision := 1;
 s double precision := 1;
 r double precision := 0;
 BEGIN
 FOR i IN 1 .. 10000000 LOOP
 r := r + s/a;
 a := a + 2;
 s := -s;
 END LOOP;
```

```
 RETURN 4 * r;
 END;
$$ LANGUAGE plpgsql;
SELECT slow_pi();
 slow_pi
--------------------
 3.1415925535898497
(1 row)
Time: 13060,650 ms (00:13,061) vs 2108,464 ms (00:02,108)
```

我们的函数是一种计算圆周率的超慢方法。有趣的是，循环被加速了，代码的执行速度比以前快了几倍。美中不足的是，代码本身并没有被改变--事情只是在默认情况下运行得更快。

1.7 并行化 VACUUM 操作

在每一个PostgreSQL的部署中都需要VACUUM。从13版开始，PostgreSQL可以并行地处理关系上的索引。其工作方式如下。

```
VACUUM (verbose ON, analyze ON, parallel 4) some_table;
```

简单地告诉VACUUM你可能想使用多少个核心，PostgreSQL将尝试使用最终用户指定的多少个核心。

1.8 允许跳过 WAL 进行全表写入

当wal_level被设置为minimal时（现在已经不是默认值了），你将能够享受一个真正有趣的性能改进：如果一个关系（通常是表或物化视图的同义词）大于wal_skip_threshole，PostgreSQL会尝试通过直接同步关系而不是通过事务日志机制来避免写入WAL。在旧版本中，只有COPY可以做到这一点。根据你的存储的属性，如果提交一个事务的速度慢到不可接受的程度，提高或降低这个值可能会有帮助。

请记住，如果你想使用复制，wal_level = minimal是不可能的。

1.9 其他性能改进

除了我们刚才看到的变化之外，还有一些性能上的增强。第一件值得一提的事情是能够使用ALTER STATISTICS ... 设置统计数据。PostgreSQL现在允许在同一个查询中使用多个扩展的统计数据，如果你的操作相当复杂，这可能相当有用。

为了告诉PostgreSQL你的存储系统可以处理多少I/O并发，在postgresql.conf中增加了maintain_io_concurrency参数。

```
test=# SHOW maintenance_io_concurrency;
 maintenance_io_concurrency
----------------------------
 10
(1 row)
```

这个参数允许管理任务以一种更巧妙的方式行事。

如果你碰巧对真正的大表进行操作，你可能会喜欢下一个功能：大型关系现在可以更快地被截断。如果你只在有限的时间内存储数据，这可能是非常有用的。

为了存储大字段（通常>1,996字节），PostgreSQL使用了一种通常被称为TOAST（超大属性存储技术）的技术。PostgreSQL改善了TOAST的解压性能，并允许更有效地检索TOAST字段中的领先字节。

传统上，你必须在Windows上使用TCP/IP连接。问题是，TCP/IP连接比简单的UNIX套接字更容易受到更高的延迟影响。现在Windows支持套接字连接，这有助于减少本地连接的延迟。

最后，LISTEN/NOTIFY也得到了改进，使之具有更好的性能。

1.10 让监控更强大

监控是每个成功的数据库操作的关键。PostgreSQL的这一领域也得到了改进。

1.11 其他系统视图

为了让管理员和DevOps工程师的生活更轻松，PostgreSQL社区在第13版中增加了一些易于使用的系统视图。

我最喜欢的一个新东西是一个系统视图，它允许我们跟踪pg_basebackup的进展。

```
test=# \d pg_stat_progress_basebackup
 View "pg_catalog.pg_stat_progress_basebackup"
 Column | Type | Collation | Nullable | Default
---------------------+---------+-----------+----------+---------
 pid | integer | | |
 phase | text | | |
 backup_total | bigint | | |
 backup_streamed | bigint | | |
 tablespaces_total | bigint | | |
 tablespaces_streamed | bigint | | |
```

以前，很难看到备份进展到什么程度。ANALYZE的情况也是如此。在大表中，它可能需要相当长的时间，所以又增加了一个视图。

```
test=# \d pg_stat_progress_analyze
 View "pg_catalog.pg_stat_progress_analyze"
 Column | Type | Collation | Nullable | Default
-------------------------+---------+-----------+----------+---------
 pid | integer | | |
 datid | oid | | |
 datname | name | | |
 relid | oid | | |
 phase | text | | |
 sample_blks_total | bigint | | |
 sample_blks_scanned | bigint | | |
 ext_stats_total | bigint | | |
 ext_stats_computed | bigint | | |
 child_tables_total | bigint | | |
 child_tables_done | bigint | | |
 current_child_table_relid | oid | | |
```

pg_stat_progress_analyze告诉你关于抽样的情况，也为你提供了关于分区表的统计数据等等。

同样重要的是，pg_stat_replication中加入了字段。

```
test=# \d pg_stat_replication
 View "pg_catalog.pg_stat_replication"
 Column | Type | Collation | Nullable | Default
------------------+-------------+-----------+----------+---------
 pid | integer | | |
...
 spill_txns | bigint | | |
 spill_count | bigint | | |
 spill_bytes | bigint | | |
```

如果你使用逻辑解码，可能会发生一些事务不得不进入磁盘的情况，因为没有足够的RAM可用来处理要应用于复制的待定变化。pg_stat_replication现在可以跟踪这一信息。 第13版中还增加了一个视图，即pg_shmem_allocations。它允许用户跟踪分配的共享内存。

```
test=# \d pg_shmem_allocations
 View "pg_catalog.pg_shmem_allocations"
 Column | Type | Collation | Nullable | Default
---------------+--------+-----------+----------+---------
 name | text | | |
 off | bigint | | |
 size | bigint | | |
 allocated_size | bigint | | |
```

基本上，你可以看到什么东西分配了多少内存给什么东西。如果你正在处理许多不同的扩展和分配共享内存的事情（即不是本地内存，如work_mem等），这就特别有用。

最后，还有pg_stat_slru。现在的问题是：什么是SLRU？在PostgreSQL中，不仅仅是索引和表，还有更多的工作。在内部，PostgreSQL需要缓存一些东西，比如CLOG（Commit Log的缩写）、Multixacts和子事务数据。这就是SLRU缓存的作用。如果有许多不同的事务被访问，提交日志（不要与WAL（Write Ahead Log）混为一谈，它也被称为xlog）可能是一个主要的争用来源。

```
test=# \d pg_stat_slru
 View "pg_catalog.pg_stat_slru"
 Column | Type | Collation | Nullable | Default
-------------+--------------------------+-----------+----------+---------
 name | text | | |
 blks_zeroed | bigint | | |
 blks_hit | bigint | | |
 blks_read | bigint | | |
 blks_written | bigint | | |
 blks_exists | bigint | | |
 flushes | bigint | | |
 truncates | bigint | | |
 stats_reset | timestamp with time zone | | |
```

所有这些系统视图将使我们更容易看到PostgreSQL内部发生的事情。我们期望将来会有更多的视图加入。我们已经看到了几个针对PostgreSQL 14的补丁。Laurenz Albe（在Cybertec PostgreSQL International，我的公司）已经提出了一个补丁，帮助跟踪连接数和更多的东西。

2.总结

PostgreSQL 13提供了各种功能，包括性能改进，简化数据库处理，以及更多。更好的Btrees无疑是PostgreSQL的一个亮点。然而，像增量排序这样的东西也将有助于良好的性能，而额外的视图将有助于管理员有更好的可视性。我们可以期待在PostgreSQL 14中看到更多更酷的功能。特别是，围绕B树和其他性能问题的改进，以及更好的监控，将帮助最终用户更好、更有效、更容易地运行应用程序。

在下一章中，你将被介绍到事务和锁定，这对可扩展性和存储管理以及性能都很重要。