Tuning Autovacuum in PostgreSQL and Autovacuum Internals

PostgreSQL 数据库的性能可能会受到死元组的影响，因为它们会继续占用空间并导致膨胀。我们在之前的一篇博文中介绍了真空和膨胀。不过，现在是时候看看 postgres 的 autovacuum，以及维护高性能 PostgreSQL 数据库所需的高性能应用程序的内部知识了。

什么是 autovacuum?

Autovacuum 是启动 PostgreSQL 时自动启动的后台实用程序进程之一。如您在下面的日志中所看到的，具有 pid 2862 的 postmaster（父 PostgreSQL 进程）已经使用 pid 2868 启动了 autovacuum 启动程序进程。要启动 autovacuum，必须将参数 autovacuum 设置为“on”。事实上，在生产系统中不应该将其设置为关闭，除非您 100%确定您正在做什么及其所造成的影响。

```
avi@percona:~$ps -eaf | egrep "/post|autovacuum"
postgres  2862     1  0 Jun17 pts/0    00:00:11 /usr/pgsql-10/bin/postgres -D /var/lib/pgsql/10/data
postgres  2868  2862  0 Jun17 ?        00:00:10 postgres: autovacuum launcher process
postgres 15427  4398  0 18:35 pts/1    00:00:00 grep -E --color=auto /post|autovacuum
```

为什么需要 autovacuum?

我们需要真空来移除死元组，以便表可以重新使用死元组所占用的空间，以便将来进行插入/更新。要了解更多关于死元组和膨胀。我们还需要对所更新表的统计信息进行分析，以便优化器可以为 SQL 语句选择最佳执行计划。是 postgres 中的 autovacuum，负责在表上执行真空和分析。

postgres 中还有一个名为 Stats Collector 的后台进程，用于跟踪使用情况和活动信息。autovacuum launcher 使用此过程收集的信息来确定 autovacuum 的候选表列表。PostgreSQL 标识需要 vacuum 或自动分析的表，但仅在启用 autovacuum 时。这可以确保 postgres 自我修复并阻止数据库发生更多的膨胀/碎片。

在 PostgreSQL 中启用 autovacuum 所需的参数有：

```
autovacuum = on    # ( ON by default )
track_counts = on # ( ON by default )
```

track_counts 由 stats 收集器使用。如果没有这个，autovacuum 就不能访问候选表

记录 autovacuum

最后，您可能需要记录 autovacuum 花费更多时间的表。在这种情况下，将参数 log_autovacuum_min_duration 设置为一个值（默认值为毫秒），以便运行时间超过此值的任何 autovacuum 都会记录到 PostgreSQL 日志文件中。这可能有助于适当地调整表级 autovacuum 设置。0 表示记录所有的 autovacuum, -1 表示不记录。

```
# Setting this parameter to 0 logs every autovacuum to the log file.
log_autovacuum_min_duration = '250ms' # Or 1s, 1min, 1h, 1d
```

这里是一个 autovacuum 的例子日志，并进行了分析：

```
< 2018-08-06 07:22:35.040 EDT > LOG: automatic vacuum of table "vactest.scott.employee": index scans: 0
pages: 0 removed, 1190 remain, 0 skipped due to pins, 0 skipped frozen
tuples: 110008 removed, 110008 remain, 0 are dead but not yet removable
```

```
buffer usage: 2402 hits, 2 misses, 0 dirtied
avg read rate: 0.057 MB/s, avg write rate: 0.000 MB/s
system usage: CPU 0.00s/0.02u sec elapsed 0.27 sec
< 2018-08-06 07:22:35.199 EDT > LOG: automatic analyze of table "vactest.scott.employee" system usage: CPU 0.00s/0.02u sec elapsed 0.15 sec
```

PostgreSQL 什么时候在表上做 autovacuum?

如前所述，postgres 中的 autovacuum 指的是 autovacuum 和 analyze，而不仅仅是 autovacuum。根据下列数学方程式，autovacuum 或分析在工作台上运行。

计算有效表级 autovacuum 阈值的公式为：

```
Autovacuum VACUUM thresold for a table = autovacuum_vacuum_scale_factor * number of tuples + autovacuum_vacuum_threshold
postgres=# show autovacuum_vacuum_scale_factor;
 autovacuum_vacuum_scale_factor
--------------------------------
 0.2
(1 row)

postgres=#
postgres=# show autovacuum_vacuum_threshold;
 autovacuum_vacuum_threshold
-----------------------------
 50
(1 row)
```

根据上面的公式，很明显，如果由于更新和删除，表中的实际死元组数超过此有效阈值，则该表将成为 autovacuum 的候选表。

```
Autovacuum ANALYZE threshold for a table = autovacuum_analyze_scale_factor * number of tuples + autovacuum_analyze_threshold
postgres=# show autovacuum_analyze_scale_factor;
 autovacuum_analyze_scale_factor
---------------------------------
 0.1
(1 row)
postgres=# show autovacuum_analyze_threshold;
 autovacuum_analyze_threshold
------------------------------
 50
(1 row)
```

上面的公式表明，自上次分析以来插入/删除/更新总数超过此阈值的任何表都有资格进行 autovacuum 分析。

让我们详细了解这些参数。

- `autovacuum_vacuum_scale_factor` Or `autovacuum_analyze_scale_factor`：将添加到公

式中的表记录的分数。例如，值 0.2 等于表记录的 20%。

- `autovacuum_vacuum_threshold` Or `autovacuum_analyze_threshold`：触发 autovacuum 所需的过时记录或 dml 的最小数量。

让我们考虑一个表：percona.employee，它有 1000 条记录和以下 autovacuum 参数。

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

以上述数学公式为参考，

```
Table：percona.employee 成为 autovacuum Vacuum 的候选者，当下面的条件满足时：
Total number of Obsolete records = (0.2 * 1000) + 50 = 250
```

```
Table : percona.employee 成为 autovacuum ANALYZE 候选者，当下面的条件满足时：
Total number of Inserts/Deletes/Updates = (0.1 * 1000) + 50 = 150
```

在 PostgreSQL 中调整 Autovacuum

我们需要了解这些是全局设置。这些设置适用于实例中的所有数据库。这意味着，无论表大小，如果达到上述公式，表都有资格进行 autovacuum 或分析。

这是不是一个问题?

考虑一个有 10 条记录的表和一个有 100 万条记录的表。尽管拥有一百万条记录的表可能更频繁地参与事务，但对于只有十条记录的表，真空或分析自动运行的频率可能更高。

因此，PostgreSQL 允许您绕过全局设置去设置个别表级 autovacuum 设置。

```
ALTER TABLE scott.employee SET (autovacuum_vacuum_scale_factor = 0, autovacuum_vacuum_threshold = 100);
```

上面的设置在 scott.employee 表上运行 autovacuum vacuum，只要有超过 100 条过时的记录。

如何确定需要调整其 autovacuum setting 的表?

为了单独调整表的 autovacuum，必须知道一段时间内表上的插入/删除/更新数。您还可以查看 postgres 目录视图：pg_stat_user_table 以获取该信息。

```
percona=# SELECT n_tup_ins as "inserts",n_tup_upd as "updates",n_tup_del as "deletes", n_live_tup as "live_tuples", n_dead_tup as "dead_tuples"
FROM pg_stat_user_tables
WHERE schemaname = 'scott' and relname = 'employee';
 inserts | updates | deletes | live_tuples | dead_tuples
---------+---------+---------+-------------+-------------
      30 |      40 |       9 |          21 |          39
(1 row)
```

如上面日志中所述，在一定时间间隔内获取此数据的快照应有助于了解每个表上 DML 的频率。反过

来，这将有助于调整个别表的 autovacuum 设置。

一次可以运行多少个 autovacuum 过程

在可能包含多个数据库的实例/群集上，一次运行的 autovacuum 进程数不能超过 autovacuum_max_workers。Autovacuum launcher 后台进程为需要真空或分析的表启动工作进程。如果有四个数据库，并且 autovacuum_max_workers 设置为 3，则第四个数据库必须等到现有工作进程中的一个空闲。

在启动下一个 autovacuum 之前，它会等待 autovacuum_naptime，大多数版本的默认值是 1 分钟。如果您有三个数据库，则下一个 autovacuum 将等待 60/3 秒。因此，启动下一个 autovacuum 之前的等待时间总是（autovacuum_naptime/N），其中 N 是实例中数据库的总数。

增加 autovacuum_max_workers 本身是否会增加可以并行运行的 autovacuum 进程的数量？

不，这在接下来的几行中解释得更清楚。

真空 IO 是密集型的吗？

autovacuum 可以看作是一种清洁。如前所述，每个表有一个工作进程。Autovacuum 从磁盘读取表的 8KB（默认块大小）页，并修改/写入包含死元组的页。这涉及读写 IO。因此，这可能是一个 IO 密集型操作，在事务高峰时间，当一个具有许多死元组的大型表上运行 autovacuum 时。为了避免这个问题，我们设置了一些参数来最小化真空对 IO 的影响。

以下是用于调整 autovacuumIO 的参数：

autovacuum_vacuum_cost_limit : autovacuum 可达到的总成本限制（结合所有 autovacuum 作业）

autovacuum_vacuum_cost_delay : 当一个清理工作达到 autovacuum_vacuum_cost_limit 指定的成本限制时，autovacuum 将休眠数毫秒

vacuum_cost_page_hit : 读取已在共享缓冲区中且不需要磁盘读取的页的成本.

vacuum_cost_page_miss : 获取不在共享缓冲区中的页的成本.

vacuum_cost_page_dirty : 在每一页中发现死元组时写入该页的成本.

上面参数默认的值考虑如下：

autovacuum_vacuum_cost_limit = -1 (So, it defaults to vacuum_cost_limit) = 200

autovacuum_vacuum_cost_delay = 20ms

vacuum_cost_page_hit = 1

vacuum_cost_page_miss = 10

vacuum_cost_page_dirty = 20

考虑在 percona.employee 表上运行 autovacuum 清理。

让我们想象一下 1 秒后会发生什么。（1 秒=1000 毫秒）。

在读取延迟为 0 毫秒的最佳情况下，autovacuum 可以唤醒并进入睡眠 50 次（1000 毫秒/20 毫秒），因为唤醒之间的延迟需要 20 毫秒。

1 second = 1000 milliseconds = 50 * autovacuum_vacuum_cost_delay

由于在共享缓冲区中每次读取一个页面的相关成本是 1，因此在每个唤醒中可以读取 200 个页面（因为上面把总成本限制设置为 200），在 50 个唤醒中可以读取 50*200 个页面。

如果在共享缓冲区中找到了所有具有死元组的页，并且 autovacuum 代价延迟为 20 毫秒，则它可以在每一轮中读取：（（200/ vacuum_cost_page_hit）*8）KB，这需要等待 autovacuum 代价延迟时间量。

因此，考虑到块大小为 8192 字节，autovacuum 最多可以读取：50*200*8kb=78.13mb/s（如果在共享缓冲区中已经找到块）。

如果块不在共享缓冲区中，需要从磁盘提取，则 autovacuum 可以读取：50*（200/ vacuum_cost_page_miss）*8）KB=7.81 MB/秒。

我们在上面看到的所有信息都是用来读取 IO 的。

现在，为了从页/块中删除死元组，写操作的开销是：vacuum_cost_page_dirty，默认设置为 20。

一个 auto vacuum 每秒最多可以写/脏：50*（200/ vacuum_cost_page_dirty）*8）KB=3.9mb/秒。

通常，此成本平均分配给实例中运行的所有 autovacuum 过程的 autovacuum_max_workers 数。因此，增加 autovacuum_max_workers 可能会延迟当前运行的 autovacuum workers 的 autovacuum 执行。而增加 autovacuum_vacuum_cost_limit 可能会导致 IO 瓶颈。需要注意的一点是，可以通过设置单个表的存储参数来重写此行为，这样会忽略全局设置。

```
postgres=# alter table percona.employee set (autovacuum_vacuum_cost_limit = 500);
ALTER TABLE
postgres=# alter table percona.employee set (autovacuum_vacuum_cost_delay = 10);
ALTER TABLE
postgres=#
postgres=# \d+ percona.employee
Table "percona.employee"
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
id | integer | | | | plain | |
Options: autovacuum_vacuum_threshold=10000, autovacuum_vacuum_cost_limit=500,
autovacuum_vacuum_cost_delay=10
```

因此，在繁忙的 OLTP 数据库中，总是有一个策略，在低峰值窗口期间，在经常被 DMLs 命中的表上实现手动真空。在设置了相关的 autovacuum 设置后，手动运行真空作业时，可能会有尽可能多的并行真空作业。因此，建议在微调 autovacuum 设置的同时，执行预定的手动真空作业。

你也可能喜欢

ProxySQL 查询缓存可以很好地扩展，并帮助您的数据库实现显著的性能提升。但是，查询缓存并非没有其限制。请阅读我们的博客，以了解有关 ProxySQL 查询缓存、其配置、工作方式以及当前已知限制的更多信息。

DBA 的价值和重要性并没有因为迁移到云而降低。虽然有些任务是自动化的，但工作的其他方面（如数据建模和数据安全）只会增长。我们的白皮书讨论了贵公司的 DBA 员工需要如何适应新的云数据库环境等。

实际案例

1、 创建实验用例：直接向表中快速插入 1000000 条数据

```
create table tbl_test (id int, info text, c_time timestamp);
insert into tbl_test select generate_series(1,100000),md5(random()::text),clock_timestamp();
```

2、 分析导致自动 analyze 的阈值：

根据计算公式为参考：

> Table tbl_test 成为 autovacuum ANALYZE 候选者，当下面的条件满足时：
> Total number of Inserts/Deletes/Updates = (0.1 * 100000) + 50 = 10050

3、 更新数据，达到触发 analyze 的阈值：

```
testdb=# update tbl_test set info=md5(random()::text) where id < 10050;
```

4、 查看分析情况，发现没有更新到 10050 也会分析，当时没有达到 autovacuum 条件：

```
testdb=# select * from pg_stat_all_tables where relname ='tbl_test';
-[ RECORD 1 ]-------+------------------------------
relid               | 164047
schemaname          | public
relname             | tbl_test
seq_scan            | 1
seq_tup_read        | 100000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 100000
n_tup_upd           | 10048
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 100000
n_dead_tup          | 10048
n_mod_since_analyze | 10048
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2020-04-21 06:33:27.451504-04
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1
```

5、 根据公式计算：

> 表：tbl_test 成为 autovacuum Vacuum 的候选者，当下面的条件满足时：

```
Total number of Obsolete records = (0.2 * 100000) + 50 = 20050
```

6、 因为前面已经更新了 10048，此次再更新 10000 行：

testdb=# update tbl_test set info=md5(random()::text) where id < 10001 ;

7、 查看统计信息，发现此时并没有触发 autovacuum。

8、 继续更新 2 行，以达到 20050 行：

testdb=# update tbl_test set info=md5(random()::text) where id < 3 ;

观察统计信息，还是没有发生 autovacuum。

9、 再次修改表记录：

update tbl_test set info=md5(random()::text) where id < 50;

10、观察统计信息，发现发生自动真空，同时日志中也有记录：

testdb=# select * from pg_stat_all_tables where relname ='tbl_test';

```
-[ RECORD 1 ]-------+------------------------------
relid               | 164047
schemaname          | public
relname             | tbl_test
seq_scan            | 8
seq_tup_read        | 799998
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 100000
n_tup_upd           | 20101  --总共更新的行数
n_tup_del           | 2
n_tup_hot_upd       | 12
n_live_tup          | 99998
n_dead_tup          | 0  --变成 0
n_mod_since_analyze | 55
last_vacuum         |
last_autovacuum     | 2020-04-21 09:07:04.395424-04
last_analyze        |
last_autoanalyze    | 2020-04-21 06:51:27.86126-04
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 2
```

--pg_log 下的日志记录，本案例设置 log_autovacuum_min_duration=0;

```
2020-04-21 09:07:04.395 EDT,,,9437,,5e9eeff8.24dd,1,,2020-04-21 09:07:04 EDT,6/5,0,LOG,00000,"automatic vacuum of table ""testdb.public.tbl_test"": index scans: 0
pages: 0 removed, 1123 remain, 0 skipped due to pins, 0 skipped frozen
tuples: 47 removed, 99998 remain, 0 are dead but not yet removable, oldest xmin: 1462
buffer usage: 2270 hits, 2 misses, 1124 dirtied
avg read rate: 0.047 MB/s, avg write rate: 26.244 MB/s
system usage: CPU: user: 0.00 s, system: 0.01 s, elapsed: 0.33 s",,,,,,,,"heap_vacuum_rel, vacuumlazy.c:440",""
```

11、查看表的使用块数没有变小，说明 autovacuum 做的不是 full 模式。使用 vacuum full tbl_test;发现块数变小，数据文件也缩小。

```
--清空前
testdb=# select relpages,reltuples from pg_class where relname='tbl_test';
 relpages | reltuples
----------+-----------
     1123 |     99998
(1 row)
--清空后
testdb=# select relpages,reltuples from pg_class where relname='tbl_test';
 relpages | reltuples
----------+-----------
      935 |     99998
(1 row)
```

12、再次测试 autovacuum，此次直接更新 20050 行，满足 autovacuum 触发的阈值：

```
testdb=# update tbl_test set info=md5(random()::text) where id < 20051;
UPDATE 20050
```

13、过一分钟后观察统计信息，发现再次马上就发生 autovacuum：

```
testdb=# select * from pg_stat_all_tables where relname ='tbl_test';
-[ RECORD 1 ]-------+------------------------------
relid               | 164061
schemaname          | public
relname             | tbl_test
seq_scan            | 1
seq_tup_read        | 100000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 100000
n_tup_upd           | 20050
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 100000
n_dead_tup          | 0  --没有清空前为 20050，达到触发机制
n_mod_since_analyze | 0
last_vacuum         |
last_autovacuum     | 2020-04-21 10:44:06.717406-04
last_analyze        |
last_autoanalyze    | 2020-04-21 10:44:06.774302-04
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 1
```

14、Update 再操作：

```
testdb=#  update tbl_test set info=md5(random()::text) where id < 10050;
UPDATE 10049
```

结果发现这一次 autovacuum 和 analyze 都做了。

再更新一次：

```
testdb=#  update tbl_test set info=md5(random()::text) where id < 10051;
```

没有触发 autovacuum。

继续再更新一次：

```
testdb=#  update tbl_test set info=md5(random()::text) where id < 10001;
UPDATE 10000
```

没有触发。

继续操作，此次为删除操作：

```
testdb=# delete from tbl_test where id < 55;
DELETE 54
    testdb=# select * from pg_stat_all_tables where relname ='tbl_test';
    -[ RECORD 1 ]-------+------------------------------
    relid               | 164074
    schemaname          | public
    relname             | tbl_test
    seq_scan            | 4
    seq_tup_read        | 400000
    idx_scan            |
    idx_tup_fetch       |
    n_tup_ins           | 100000
    n_tup_upd           | 30099
    n_tup_del           | 54
    n_tup_hot_upd       | 59
    n_live_tup          | 99946
    n_dead_tup          | 20049  - 没有达到 20050 所以没有触发 autovacuum
    n_mod_since_analyze | 54
    last_vacuum         |
    last_autovacuum     | 2020-04-21 10:52:06.879799-04
    last_analyze        |
    last_autoanalyze    | 2020-04-21 11:02:07.19891-04
    vacuum_count        | 0
    autovacuum_count    | 1
    analyze_count       | 0
    autoanalyze_count   | 2
```

为了满足条件，再删除一次：

```
testdb=# delete from tbl_test where id < 60;
DELETE 5
```

此时观察统计信息，发现 n_dead_tup 达到了 20054，此时触发 autovacuum，由此可见此指标非常关键，直接影响 autovacuum 的触发。还有，处于 obsolete 状态的行主要是因为 update 和 delete 操作造成的，insert 操作是不属于此类型。

```
testdb=# select * from pg_stat_all_tables where relname ='tbl_test';
-[ RECORD 1 ]-------+------------------------------
relid               | 164074
schemaname          | public
relname             | tbl_test
seq_scan            | 5
seq_tup_read        | 499946
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 100000
n_tup_upd           | 30099
n_tup_del           | 59
n_tup_hot_upd       | 59
n_live_tup          | 99941
n_dead_tup          | 20054  --达到触发条件
n_mod_since_analyze | 59
last_vacuum         |
last_autovacuum     | 2020-04-21 10:52:06.879799-04
last_analyze        |
last_autoanalyze    | 2020-04-21 11:02:07.19891-04
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 2
```

```
testdb=# select * from pg_stat_all_tables where relname ='tbl_test';
-[ RECORD 1 ]-------+------------------------------
relid               | 164074
schemaname          | public
relname             | tbl_test
seq_scan            | 5
seq_tup_read        | 499946
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 100000
n_tup_upd           | 30099
n_tup_del           | 59
```

```
n_tup_hot_upd       | 59
n_live_tup          | 84927
n_dead_tup          | 0  --清理完成后归 0
n_mod_since_analyze | 59
last_vacuum         |
last_autovacuum     | 2020-04-21 11:13:07.50089-04
last_analyze        |
last_autoanalyze    | 2020-04-21 11:02:07.19891-04
vacuum_count        | 0
autovacuum_count    | 2
analyze_count       | 0
autoanalyze_count   | 2
```

由此可以看出 autovacuum 操作的触发条件基本上按照上面的算法执行，但是有时可能也会绝对的遵守，可能还有其它触发的因素，情况比较复杂。

而 analyze 的触发比较稳定，基本上一达到条件就触发。