2025 年版 トランザクションID 関連の問題の傾向と対策

株式会社SRA OSS

会社紹介

株式会社SRA OSS

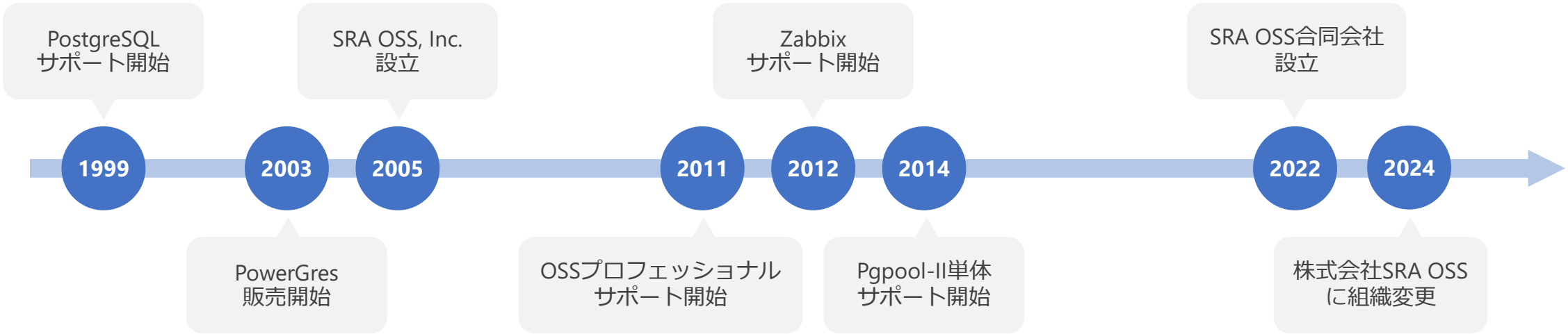
所 在 地: 東京都豊島区南池袋2-32-8
設 立 日: 2022年6月17日
株　　主: 株式会社SRA
　　株式会社NTTデータ
資 本 金: 7,000万円
社　　長: 稲葉 香理

事業内容

- オープンソースソフトウェア (OSS) 関連のサポート、製品開発・販売、構築・コンサル
- OSSの教育、開発、コミュニティ運営支援
- ソフトウェアの研究開発

顧　　問: 石井達夫

技術顧問:増永 良文 (お茶の水女子大学名誉教授)



自己紹介

三和　陽菜

- SRA OSS OSS事業本部 データベース技術グループ所属
- PostgreSQLのサポート、トレーニング講師を担当

鳥越　淳

- SRA OSS OSS事業本部 データベース技術グループ所属
- PostgreSQLのサポート、案件支援などに従事
- PostgreSQL Contributor。主にモニタリング周りの機能開発

目次

- トランザクションIDとそれに関連する問題
- マルチトランザクションIDとそれに関連する問題

トランザクションID
とそれに関連する問題

前提として.. 追記型アーキテクチャ

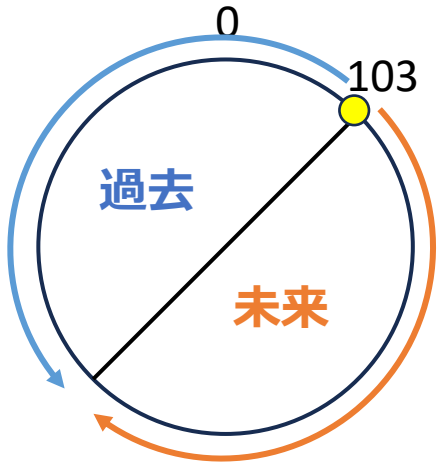
- PostgreSQLは追記型のMVCC(多版型同時実行制御)を採用しており、**1つの論理的なデータ**に対して**複数の物理的なバージョンの行**を持つ
  - トランザクション開始タイミングや分離レベルによって、見えるべきデータのバージョンが違うため、DELETE文などで削除が実行されても削除前のバージョンの行も残す
  - どのトランザクションからも参照されなくなった古い行は不要なので、VACUUMにより回収される
- トランザクションごとに、どのバージョンの行を見せるべきかの判断(可視性判断)が必要
- トランザクションIDはこの可視性判断に利用される

１つの論理的なデータに対して、複数の行バージョンを持つ

	西暦	出来事
A0	794	平安京遷都
B0	1192	鎌倉幕府成立
B1	1185	鎌倉幕府成立

トランザクションIDについて

- トランザクションIDは、トランザクションに払い出される固有のID
- トランザクションIDは、基本的にトランザクションが開始して初めてデータの変更処理を行ったタイミングで払い出される
- 32bitの非負数であり、**約42億で循環**する
- 循環して使い続けるための仕組みがある
- 前後約20億トランザクションで分割され、自分より古い値を過去、自分より新しい値を未来として扱う



トランザクションID

- トランザクションIDは、通常変更処理の際に発行される

変更が無ければ、トランザクションIDは発行されない

```
=# BEGIN;
BEGIN
=*# SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
--------------------------------

(1 row)

=*# INSERT INTO test VALUES (1);
INSERT 0 1
=*# SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
--------------------------------
                            756
(1 row)
```

データが変更されると、トランザクションIDが発行

トランザクションID

- トランザクションIDは、メタデータとして各タプルに格納されている

```
=# CREATE EXTENSION pageinspect;
=# INSERT INTO test VALUES (1);
INSERT 0 1
=# SELECT t_xmin, t_xmax FROM
heap_page_items(get_raw_page('test',0));
 t_xmin | t_xmax
--------+--------
    762 |      0
(1 row)

=# UPDATE test SET id = 2;
UPDATE 1
=# SELECT t_xmin, t_xmax FROM
heap_page_items(get_raw_page('test',0));
 t_xmin | t_xmax
--------+--------
    762 |    763
    763 |      0
(2 rows)
```

xminにそのトランザクションのトランザクションIDが記録される

更新すると、新旧両方のタプルにトランザクションIDが入る

削除を実行したトランザクションIDがxmaxに記録される

可視性判断の例①

- トランザクションIDは、可視性判断に使用される

xmin	xmax	行データ
101		test1

txid = 101のトランザクション
まではコミット済み。

xmin	xmax	行データ
101		test1
103		test2

txid = 102, 103が開始する。
txid = 103がtest2をINSERTした。
コミットはまだされていない。

可視性判断の例②

- トランザクションIDは、可視性判断に使用される

xmin	xmax	行データ
101		test1
103		test2

102

txid=102にとって、test2の行は不可視。(txid=103は未コミットのため。)
test1のみ表示する。

```
=# SELECT * FROM t1;
   c
-------
 test1
(1 row)
```

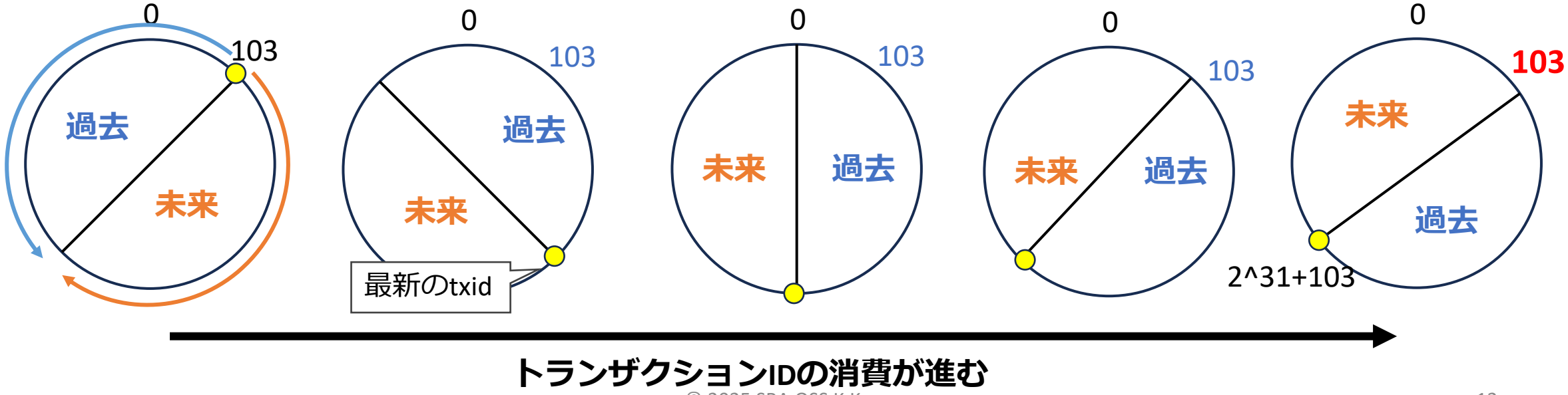
103

txid=103にとって、test2の行は可視。(自分が追加した行のため。)
test1, test2を表示する。

```
=# SELECT * FROM t1;
   c
-------
 test1
 test2
(2 rows)
```

トランザクションIDの周回

- トランザクションIDは約42億で循環する
- txid = 103が基準の時、約21億ずつ分割し、103より大きい値が「未来」、103より小さいほうの値が「過去」となる
- txidが進むと、どの値が未来/過去かの判定が変わっていく(しばらくはtxid=103は過去)
- 約21億進むと、今まで過去のIDとして扱われていたtxid=103が未来として扱われるようになってしまう＝**トランザクションID周回**

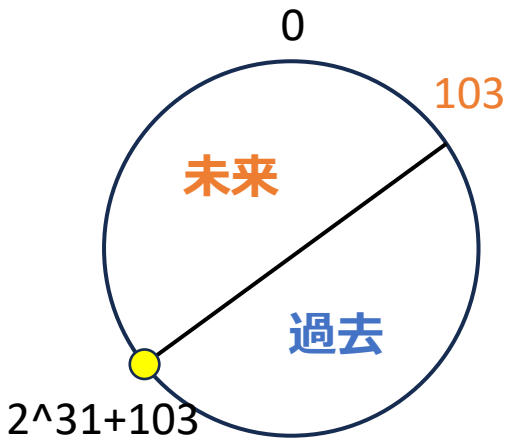


FREEZE①

- トランザクションIDが約21億を超えて発行できるよう、FREEZEという仕組みが備わっている
- トランザクションIDによって未来のタプルと判断される可能性があるのは、**そのタプルが持つxminより古いトランザクションが実行中の場合**
- あるタプルのxminより古いトランザクションがすべて終了しており、現在実行中のトランザクションがすべてそのxminより新しいトランザクションの場合、そのタプルは**現在存在しているどんなトランザクションより古いトランザクションによって作成されたことは明らか＝可視性判断の結果が可視となる**

FREEZE②

- 比較するトランザクションIDがどのような値であっても、それ**より過去に更新された行として扱うようフラグを立てる処理(FREEZE)**を行っている
  - かつては2という値に実際にxminを書き換えられていたが, 今はxminは書き換えず、別途フラグを立てている
  - FREEZEは、VACUUM時に合わせて行われる
- 問題なくFREEZEできていれば、周回問題は回避できる
  - トランザクションIDが一周したとしても、FREEZE済みのタプルは過去のものとして扱えばよい



xmin	xmax	t_infomask	行データ
101		XMIN_FROZEN	test1
103		XMIN_FROZEN	test2

FREEZE済みフラグがついているので、過去のトランザクションによって作成されたものだとわかる

FREEZEのタイミング

- vacuum_freeze_min_age(デフォルト5000万)
  - 行に記録されているxminがこの年代を超えたらFREEZEを行うようになる
- vacuum_freeze_table_age(デフォルト1.5億)
  - pg_class.relfrozenxidがこの年代に達すると、VACUUM時に積極的にFREEZEを行う
  - 通常は無効タプルが存在しているページのみをスキャンするところ、未凍結のタプルが存在するすべてのページをスキャンして未凍結のタプルをFREEZEする
- autovacuum_freeze_max_age(デフォルト2億)
  - pg_class.relfrozenidがこのパラメータで指定した年代に達したら、凍結状態でない行を含む可能性のあるテーブルに対し自動VACUUMを開始する
  - autovacuum = offでも発生
- vacuum_failsafe_age(デフォルト16億)
  - pg_class.relfrozenxidがこの年代に達すると、トランザクションID周回回避のために全力でVACUUMを行うようになる
    - コストベースの遅延を行わず、インデックスのバキュームなど緊急でない作業はスキップ
    - バッファアクセスストラテジは無効になり、VACUUMが共有バッファのすべてを自由に使用するようになる

周回問題を引き起こすケース

- FREEZEがうまくできていれば問題ないが、**FREEZEを阻害するような問題があると、周回問題が発生**する
- 実行中のトランザクション以後のトランザクションIDはFREEZEできないため、ロングトランザクションが典型的な阻害要因
- 他、使用していないレプリケーションスロットや、孤児となったプリペアドトランザクション(2相コミット)がFREEZEを阻害する
  - 論理レプリケーションではレプリケーションスロットが必ず作成されるため、論理レプリケーションが普及するにつれ、問題となるケースが増えている印象

周回が近づいた場合の動作

- データベースの最も古いトランザクションIDが周回地点から4000万に達すると、WARNINGが発生
  - pg_databaseのdatfrozenxidにそのデータベースの中で最も古いトランザクションIDが記録されている
  - datfrozenxidはFREEZE実行時に更新される

```
WARNING:  database "mydb" must be vacuumed within 39985967 transactions
HINT:  To avoid XID assignment failures, execute a database-wide VACUUM in that database.
```

- 周回までのトランザクション数が300万未満になると、PostgreSQLは新しいトランザクションIDの割り当てを拒否、**更新処理実施不可**になる
  - すでに開始されているトランザクションは継続できる
  - 新しく開始したトランザクションでは、読み取りのみが可能となる

```
ERROR:  database is not accepting commands that assign new XIDs to avoid wraparound data loss in database "mydb"
HINT:  Execute a database-wide VACUUM in that database.
```

周回問題を発生させてみる①

- 周回問題を発生させるには、トランザクションIDを大量に消費する必要がある
  - pg_current_xact_id()関数は、実行すると現在トランザクションIDが割り当てられていない場合に、新しくトランザクションIDを割り当てる
  - pg_current_xact_id()を21億回実行してもよいが、実行に時間がかかる

- 今回は**xid_wraparoundモジュール**を使用
  - PostgreSQL 17から追加されたテスト用モジュール
  - ソースコードからビルドした場合に利用可能
  - トランザクションIDの発行を適度にスキップするため、**高速にトランザクションIDを進めることができる**

周回問題を発生させてみる②

- トランザクションを開始し、INSERTする

```
=# BEGIN;
BEGIN
=*# INSERT INTO test VALUES (1);
INSERT 0 1
=*# SELECT xmin, c FROM test;
 xmin | c
------+---
  759 | 1
(1 row)
```

周回問題を発生させてみる③

- 現在のデータベースの凍結済みトランザクションIDと、その年代を確認する

```
=# SELECT datname, datfrozenxid, age(datfrozenxid) FROM
pg_database;
  datname   | datfrozenxid | age
------------+--------------+-----
 postgres   |          748 |  23
 template1 |          748 |  23
 template0 |          748 |  23
(3 rows)
```

- age()について
  - 指定されたトランザクションIDと、現在のトランザクションID間のトランザクション数を返す関数
  - 周回問題が発生しないかを監視するには、このage()を使用すると便利

周回問題を発生させてみる④

- 別の接続で、トランザクションIDが残り4000万を切るまで進める
  - consume_xids()関数に進めるxid数を指定する

testテーブルのageが21億を超える

```
=# SELECT consume_xids('2107483647');

=# SELECT relname,relfrozenxid, age(relfrozenxid) FROM pg_class WHERE relname = 'test';
 relname | relfrozenxid |    age
---------+--------------+------------
 test    |          759 | 2107483660
(1 row)

=#  SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;
  datname  | datfrozenxid |    age
-----------+--------------+------------
 postgres  |          759 | 2107483660
 template1 |          759 | 2107483660
 template0 |          759 | 2107483660
(3 rows)
```

周回問題を発生させてみる⑤

- 使用可能なトランザクションID数が残り4000万を切ると、更新処理時にWARNINGメッセージが出るようになる
  - 更新処理自体には成功する

```
=# INSERT INTO test VALUES (2);
WARNING:  database "postgres" must be vacuumed within 39999987 transactions
HINT:  To avoid transaction ID assignment failures, execute a database-wide VACUUM in that
database.
You might also need to commit or roll back old prepared transactions, or drop stale replication
slots.
INSERT 0 1
```

周回問題を発生させてみる⑥

- 残り300万を切るようにさらにトランザクションIDを進める

```
=# SELECT consume_xids('37000000');

=# SELECT relname,relfrozenxid, age(relfrozenxid) FROM pg_class WHERE relname = 'test';
 relname | relfrozenxid |    age
---------+--------------+------------
 test    |          759 | 2144483647
(1 row)

=#  SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;
  datname  | datfrozenxid |    age
-----------+--------------+------------
 postgres  |          759 | 2144483647
 template1 |          759 | 2144483647
 template0 |          759 | 2144483647
(3 rows)
```

周回問題を発生させてみる⑦

- 残り300万を切ると、更新処理時にエラーが発生するようになる

```
postgres=# INSERT INTO test VALUES (3);
ERROR:  database is not accepting commands that assign new transaction IDs to avoid wraparound
data loss in database "postgres"
HINT:  Execute a database-wide VACUUM in that database.
You might also need to commit or roll back old prepared transactions, or drop stale replication
slots.
```

- ただし、この状態でも参照処理は可能(DBが停止するわけではない)

```
=# SELECT * FROM test;
 c
---
 2
(1 row)
```

発生させた場合の対処①

- FREEZEを阻害する要因を特定し、取り除く
  - ロングトランザクション

```
--調査方法
SELECT * FROM pg_stat_activity ORDER BY xact_start LIMIT 10;
--解消方法
SELECT pg_terminate_backend(《PID》);
```

  - 古いプリペアドトランザクション

```
--調査方法
SELECT * FROM pg_prepared_xacts ORDER BY prepared LIMIT 10;
--解消方法
ROLLBACK PREPARED '《transactionid》';
```

  - 使われていないレプリケーションスロット

```
--調査方法
SELECT * FROM pg_replication_slots;
--解消方法
SELECT pg_drop_replication_slot('《スロット名》');
```

発生させた場合の対処②

- FREEZEを阻害する要因を取り除いたら、**VACUUMを実行**する
  - データベース全体に対して行うのが簡単
  - 時間を短縮したいのであれば、relfrozenxidが最も古いテーブルに対して行ってもよい
- VACUUM FULLは失敗するため使用しない
- VACUUM FREEZEも使用しない
  - 復旧に必要な最小限の作業を超えた作業を行うため
- 古いバージョンではpostmasterを停止してシングルユーザモードでVACUUMを実行するように、という指示があったが、最近のバージョン(PG14~)ではこの手順は推奨されない
  - シングルユーザモードによるVACUUMは時間がかかる
  - VACUUMのモニタリングも難しい

モニタリング

- 周回問題が発生すると影響が大きいため、**モニタリングを実施し事前に検知・対処**することが重要
- pg_class, pg_databaseにこのトランザクションidより前は全て凍結済みであることを記録する列があるので、そちらを監視する

```
SELECT datname, age(datfrozenxid) FROM pg_database ORDER BY age DESC;
SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind IN ('r',
'm') ORDER BY age DESC;
```

- ageが増加傾向にある場合、FREEZEが阻害されている可能性あり
- vacuum_failsafe_ageを超えると、周回問題が近づいていて危険な状態

マルチトランザクションID
とそれに関連する問題

前提として..行ロックの話

- たとえば、ある行を更新する場合、他のトランザクションから更新されないようにロックが必要。このために利用されるのが行ロック
- 行ロックは大量に発生しうるため、テーブル単位のロックなどと同様に管理すると、管理対象が多くなりすぎるため不適切
- PostgreSQLでは、行ロックを取得する場合、**ロック対象行のタプルのヘッダにトランザクションIDを記録**して実現
- 行ロックが使われるケース例: 明示的なSELECT…FOR ~による共有業ロック、外部キー制約がある参照元テーブルの更新

pgbench -i --foreign-keysで外部制約付きで作成したテーブルへの操作

トランザクションIDを記録

```
(接続1)=# begin; insert into pgbench_accounts values (1000000, 1, 0, '');
(接続1)=# select txid_current_if_assigned();
 txid_current_if_assigned
--------------------------
                      792
(1 row)
(接続1)=# select xmax, * from pgbench_branches;
 xmax | bid | bbalance | filler
------+-----+----------+--------
  792 |   1 |        0 | [NULL]
(1 row)
```

前提として..行ロックの話

- pgrowlocksエクステションを利用すると、取得しているロックのモードやpidなど、行ロックの詳細な情報が確認できる:

```
=# create extension pgrowlocks;
=# select * from pgrowlocks('pgbench_branches');
   locked_row | locker | multi | xids  |       modes        |  pids
  ------------+--------+-------+-------+--------------------+---------
   (0,1)      |    792 | f     | {792} | {"For Key Share"} | {67334}
```

マルチトランザクションIDを作ってみる

- 典型的には、複数のトランザクションが同一行に対して行ロックを取得する際に利用される
- “タプルのヘッダにトランザクションIDを記録”している領域の**サイズは、1トランザクション分しかない**。このため複数トランザクションが同一行をロックする場合サイズが不足:

書けるIDは1つだけ!

```
=# select xmax, * from pgbench_branches;
  xmax | bid | bbalance | filler
 ------+-----+----------+--------
   792 |   1 |        0 | [NULL]
```

マルチトランザクションIDを作ってみる

- この問題を、**複数トランザクションとそのロック状態をまとめて新たに1つのIDを付与**して解決している。この１つにまとめたIDがマルチトランザクションID:

```
(接続1)=# begin; insert into pgbench_accounts values (1000000, 1, 0, ‘‘); --トランザクションID:775
(接続2)=# begin; insert into pgbench_accounts values (1000001, 1, 0, ‘‘); --トランザクションID:776
=# select xmax, * from pgbench_branches  ;
 xmax | bid | bbalance | filler
------+-----+----------+--------
    1 |   1 |        0 | [NULL]

=# select * from pgrowlocks('pgbench_branches');
 locked_row | locker | multi |   xids    |                modes                |     pids
------------+--------+-------+-----------+-------------------------------------+--------------
 (0,1)      |      1 | t     | {775,776} | {"For Key Share","For Key Share"} | {77054,77419}
```

マルチトランザクションID！

マルチトランザクションIDを作ってみる

- pgrowlocksがインストールされていない環境では本体同梱の**pg_get_multixact_members()**関数を使っても、マルチトランザクションの内容が確認できる。：

```
=#  select pg_get_multixact_members('1');
  pg_get_multixact_members
 -------------------------
  (775,keysh)
  (776,keysh)
```

マルチトランザクションIDを作ってみる

- 行ロックを取得する**トランザクションが増えると、新たにマルチトランザクションIDが払い出される**。既存のマルチトランザクションIDにトランザクションIDが追加されるわけではない点注意:

```
..前ページからの続き..
 (接続3)=# begin; insert into pgbench_accounts values (1000002, 1, 0, '');

 =#  select * from pgrowlocks('pgbench_branches');
  locked_row | locker | multi |     xids      |                  modes                  |       pids
 ------------+--------+-------+---------------+-----------------------------------------+-------------------
  (0,1)      |      2 | t     | {775,776,778} | {"For Key Share","For Key Share","For Key Share"} |
{77054,77419,77559}

 =# select pg_get_multixact_members('2');
  pg_get_multixact_members
 --------------------------
  (775,keysh)
  (776,keysh)
  (778,keysh)
```

新しいマルチトランザクションIDである2

マルチトランザクションID**ごとにmemberの数が違う**点も注意！

マルチトランザクションIDの特徴

- 32bit。これは普通のトランザクションIDと同じ
- 普通のトランザクションIDと同様に、周回予防のためのパラメータがある:
  - autovacuum_multixact_freeze_max_age (cf. autovacuum_freeze_max_age)
  - vacuum_multixact_freeze_table_age (cf. vacuum_freeze_table_age)
  - vacuum_multixact_failsafe_age (cf. vacuum_failsafe_age)
- 各マルチトランザクションIDを構成するトランザクションIDの情報は、$PGDATA/pg_multixact以下に保存
- キャッシュする仕組みもあり
- memberを保存する領域にも上限がある。次に払い出す上限を超過しそうになると当該トランザクションが実行できなくなる

マルチトランザクションIDの特徴

- 32bit。これは普通のトランザクションIDと同じ
- 普通のトランザクションIDと同様に、周回予防のためのパラメータがある:
  - autovacuum_multixact_freeze_max_age (cf. autovacuum_freeze_max_age)
  - vacuum_multixact_freeze_table_age (cf. vacuum_freeze_table_age)
  - vacuum_multixact_failsafe_age (cf. vacuum_failsafe_age)
- 各マルチトランザクションIDを構成するトランザクションIDの情報は、$PGDATA/pg_multixact以下に保存
- キャッシュする仕組みもあり
- memberを保存する領域にも上限がある。次に払い出す上限を超過しそうになると当該トランザクションが実行できなくなる

周回問題
(今回は割愛)

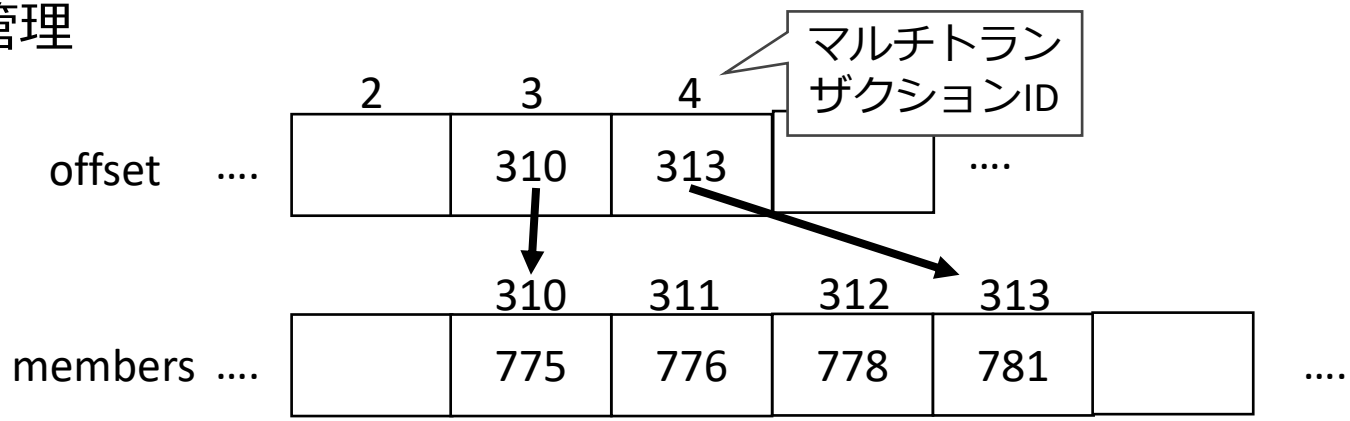
キャッシュ問題

member space枯渇問題

キャッシュ問題

問題

- pg_multixact/以下のストレージに都度アクセスするのは性能上の問題になりうる。そこで共有メモリ上にキャッシュを保存している
  - 具体的にはSLRUと呼ばれるLeast Recently Usedで管理されるバッファを利用
  - マルチトランザクションはメンバ数が可変なので、memberのほかoffsetもSLRUで管理



- **PostgreSQL 17まではこのキャッシュのサイズが固定**だったため、ワークロードによってはキャッシュの入れ替えが頻発すると、性能低下することがあった

キャッシュ問題

モニタリング

- **pg_stat_slruビュー**のmultixact_offset、multixact_memberのblks_hit, blks_readで**キャッシュヒット状況を確認**
- 待機イベント: MultiXactOffsetControlLock, MultiXactMemberControlLockの増加

対処

- PostgreSQL 17では、offset、memberのキャッシュサイズを指定するGUCとして、**multixact_offset_buffers, multixact_member_buffers**が追加されたので、こちらをチューニング
- なお、単純にキャッシュサイズを増やすだけでは逆に性能低下するケースもあったため、キャッシュを小分けにしてそれぞれ独立してロック管理する改善も同時に導入されている

member space枯渇問題

問題

- **memberを保存する領域にも上限**がある。上限を超過しそうになると当該トランザクションが失敗する:

```
ERROR: multixact "members" limit exceeded
  This command would create a multixact with %u members, but the remaining space is only enough for %u member
```

- マルチトランザクションの数に余裕があっても発生する
- 1つ行に対する共有ロックを複数のバックエンドが取得する場合、使用する**member数は、二次関数的に増加**する点に注意

バックエンド数	マルチトランザクションID数	総member数
2	1	2
3	2	3 + 2
4	3	4 + 3 + 2
N	n-1	n(n+1)/2 - 1

member space枯渇問題

モニタリング

- v18時点でmember spaceの消費状況を直接確認する手段は提供されていない
- コミュニティにて議論中
  [Proposal] Expose internal MultiXact member count function for efficient monitoring

対処

- 不要なマルチトランザクションIDを利用しているトランザクションがあれば終了させる
- VACUUMを実行。vacuum_multixact_freeze_min_ageとvacuum_multixact_freeze_table_ageを小さく設定してFREEZE対象を増やすとよい

まとめ

- トランザクションID周回問題は、発生すると更新処理が実施できなくなる、サービス停止につながる重大な問題
- 自動VACUUMに任せておけば問題が起こらずに済むことも多いが、ロングトランザクション・レプリケーションスロットの削除忘れなどには注意が必要。また更新トランザクション数が多い場合も注意が必要
- マルチトランザクションIDでは、周回問題に加え、キャッシュ問題・member space枯渇問題という問題もある
- それぞれモニタリングの実施と対処手順を事前に把握しておきましょう