

リリースされたばかり の新バージョン！ PostgreSQL 14 徹底解説

澤田 雅彦

DB TECH SHOWCASE 2021



自己紹介

澤田 雅彦  masahiko_sawada

Principal Engineer, Database Server

Major Contributor

Vacuum (Freeze map, Parallel Vacuum, Failsafe mode)
Replication



Agenda

- PostgreSQL 14
- 新機能紹介
 - 性能
 - 運用・管理 (DBA)
 - 開発



PostgreSQL 14



PostgreSQL 14が9/30にリリース！

- PostgreSQLは毎年メジャーバージョンをリリース
- 約200個の新機能や改善
- リリースノート（英語）公開済み
- 9.6が11月にEOL



前バージョン(PG13)からの移行時の注意点

- 27個の非互換があります（最近リリースの中では多め）
 - password_encryptionのデフォルトがmd5からscram-sha-256に変更
 - vacuum_cleanup_index_scale_factorが廃止
 - contrib/pg_standbyが廃止
- その他、演算子、関数などの微妙な挙動の変更があります

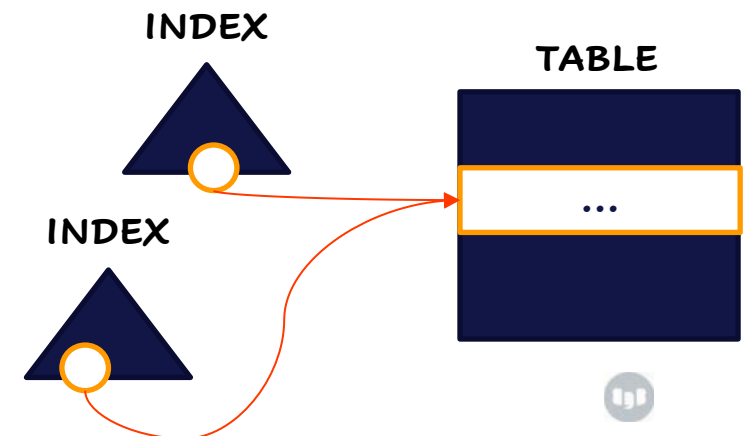


性能



Btree: Bottom-up Index Deletion

- インデックスタブルの削除は非同期
 - DELETEやUPDATE：なにもしない
 - その後のSELECT等：削除マークを付ける
 - Btreeインデックスのノードが満杯になったとき：削除マーク済みのインデックスタブルを削除する
- 「削除マークを付ける」 + 「削除する」の2ステップが異なるタイミングで必要
- Bottom-up Index Deletionでは、2つのステップをページ単位で一緒に行うことで、積極的にインデックスタブルを削除する
- ノードの分割を防ぐ = インデックスが大きくなるしない
- 通常のインデックスタブル削除 + Bottom-up Deletion + 重複削除の3つの機構でノード分割を避ける



Bottom-up Index Deletion

- 値が変更されていない列へのインデックスでのみ有効

INDEX 1	
100	(1, 10)

INDEX 2	
'A'	(1, 10)

TABLE			
(1,10)	100	'A'	...

Bottom-up Index Deletion

- 値が変更されていない列へのインデックスでのみ有効
- インデックス列を更新する場合は、新しいインデックスタプルも追加する

INDEX 1	
100	(1, 10)

INDEX 2	
'A'	(1, 10)

TABLE		
(1,10)	100	DELETED
(1,11)	100	'Z' ...

Bottom-up Index Deletion

- 値が変更されていない列へのインデックスでのみ有効
- インデックス列を更新する場合は、新しいインデックスタプルも追加する
- INDEX 1の値は変更されていないが新しいタプルが必要
 - これらのインデックスタプルは近く（同じページ内）に存在する

100	(1, 10)
100	(1, 11)

'A'	(1, 10)
'Z'	(1, 11)

(1,10)	100	DELETED
(1,11)	100	'Z' ...

Bottom-up Index Deletion

- 値が変更されていない列へのインデックスでのみ有効
- インデックス列を更新する場合は、新しいインデックスタプルも追加する
- INDEX 1の値は変更されていないが新しいタプルが必要
 - これらのインデックスタプルは近く（同じページ内）に存在する

100	(1, 10)
100	(1, 11)

'Z'	(1, 10)
'Z'	(1, 11)

(1,10)	100	DELETED
(1,11)	100	'Z' ...

Bottom-up Index Deletion

- 値が変更されていない列へのインデックスでのみ有効
- インデックス列を更新する場合は、新しいインデックスタプルも追加する
- INDEX 1の値は変更されていないが新しいタプルが必要
 - これらのインデックスタプルは近く（同じページ内）に存在する

INDEX 1	
100	(1, 10)
100	(1, 11)

INDEX 2	
'Z'	(1, 11)

TABLE		
(1,10)	100	DELETED
(1,11)	100	'Z' ...

Bottom-up Index Deletion

- 値が変更されていない列へのインデックスでのみ有効
- インデックス列を更新する場合は、新しいインデックスタプルも追加する
- INDEX 1の値は変更されていないが新しいタプルが必要
 - これらのインデックスタプルは近く（同じページ内）に存在する

INDEX 1	
100	(1, 11)

INDEX 2	
'Z'	(1, 11)

TABLE		
(1,10)	100	DELETED
(1,11)	100	'Z' ...

Bottom-up Index Deletion

- 値が変更されていない列へのインデックスでのみ有効
- インデックス列を更新する場合は、新しいインデックスタプルも追加する
- INDEX 1の値は変更されていないが新しいタプルが必要
 - これらのインデックスタプルは近く（同じページ内）に存在する

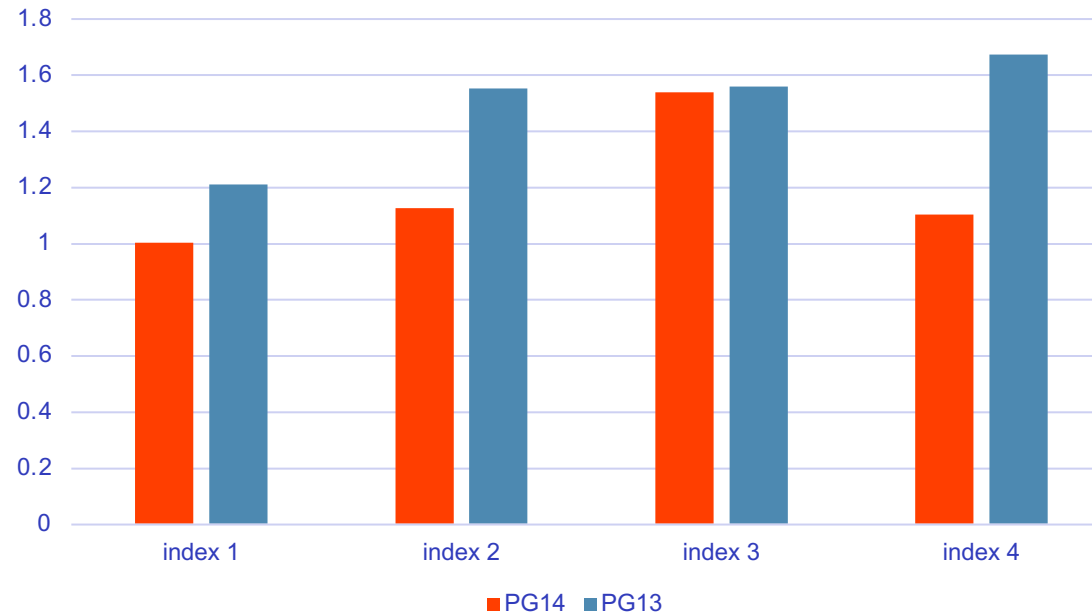
INDEX 1	
100	(1, 10) (1, 11)

INDEX 2	
'A'	(1, 10)
'Z'	(1, 11)

TABLE		
(1,10)	100	DELETED
(1,11)	100	'Z' ...

Bottom-up Index Deletion

- 4つの列を持つテーブルにデータを挿入
- インデックスは全ての列に付与
- 1列(index 3)のみデータ更新を30万Tx繰り返し、他のインデックスの肥大化状況を見る
- テーブルは約4GB



更新前のインデックスサイズ
index 1: 2.2GB
index 2: 712MB
index 3: 745MB
index 4: 704kB



TOASTの圧縮方式にLZ4が指定可能

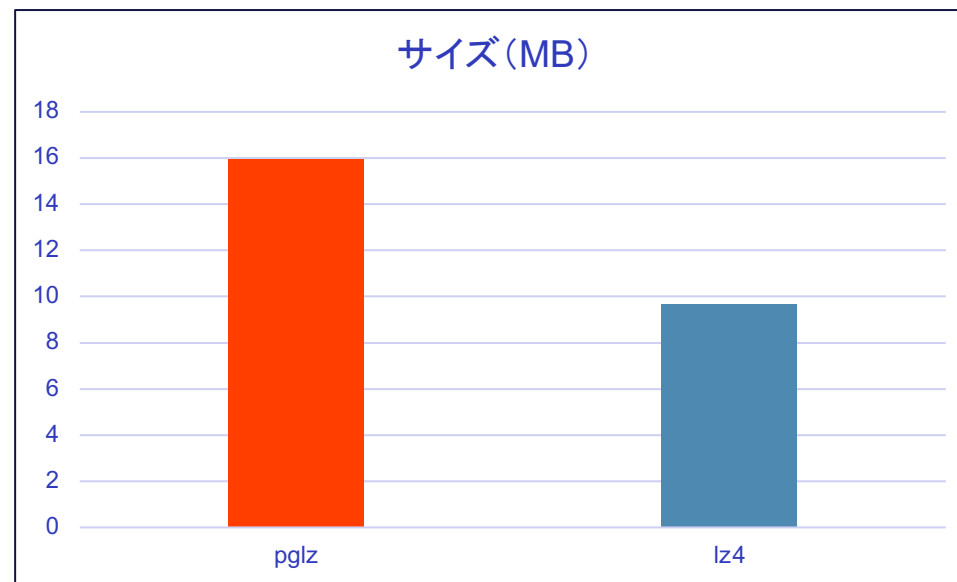
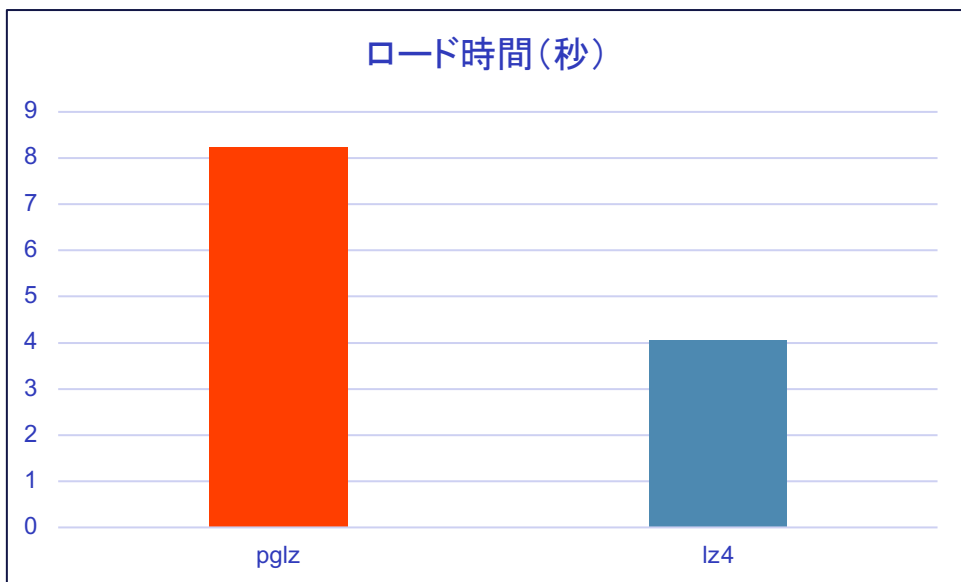
- TOAST(The Oversized-Attribute Storage Technique)
 - タプルサイズが大きい場合、一部の列をTOASTテーブルに格納する
- 圧縮方法がpglz、lz4から選べるようになった
- lz4を選ぶためには--with-lz4を指定する必要がある
- 圧縮方法は列毎に指定可能
- デフォルトの圧縮方法は、*default_toast_compression*で指定可能（デフォルトはpglz）



TOASTの圧縮方式にLZ4が指定可能

テキストデータの圧縮効果を計測

- 各列は'1234567890'を1000回を繰り返した値
- 全100,000件
- 平均列サイズは10kB

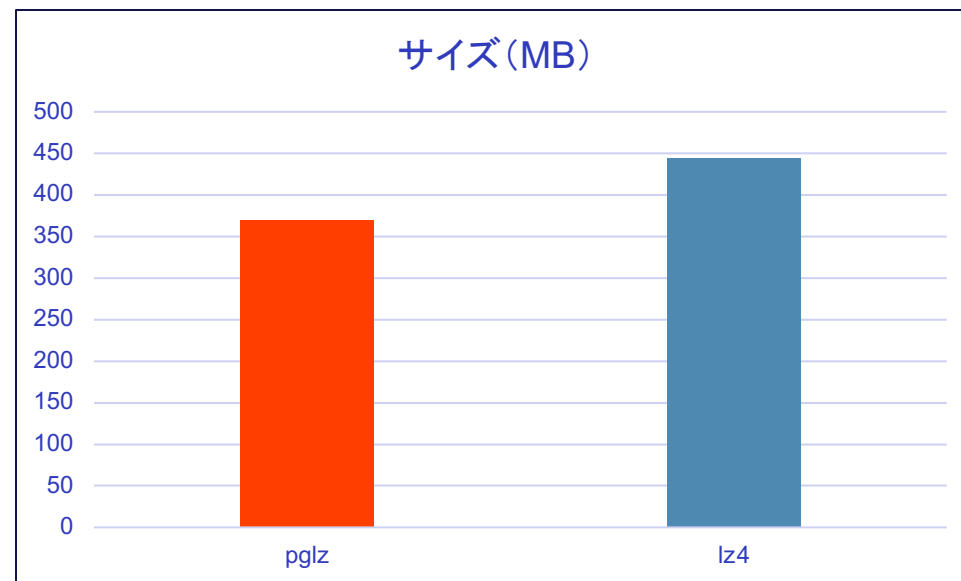
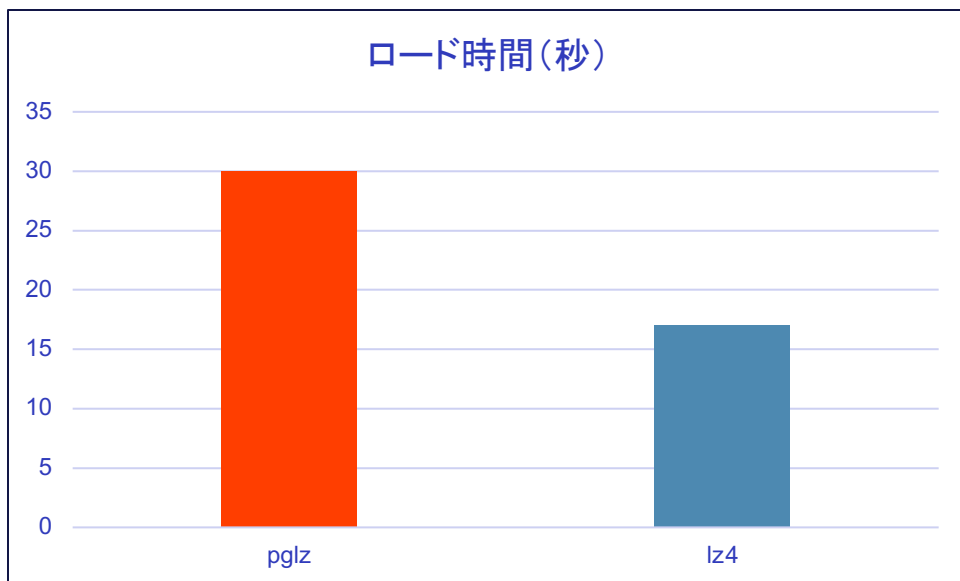


非圧縮の場合は22秒、1190MB

TOASTの圧縮方式にLZ4が指定可能

テキストデータの圧縮効果を計測

- ・ 青空文庫の各文章データを1列にまとめる
- ・ 全16,733件
- ・ 平均列サイズは46kB



運用・管理（DBA）

pg_stat_progress_copyビューの追加

- 新しくCOPYコマンドの進捗状況をレポートするビューが追加
- 現在サポート中の進捗レポートビュー
 - pg_stat_progress_analyze
 - pg_stat_progress_create_index (REINDEXも)
 - pg_stat_progress_vacuum
 - pg_stat_progress_cluster
 - pg_stat_progress_basebackup
 - pg_stat_progress_copy



pg_stat_progress_copyビューの追加

- 1コマンド1行で表示
- pg_dump/pg_restoreも内部ではCOPYコマンドを使っているので、それらの進捗も確認することが可能

```
=# SELECT * FROM pg_stat_progress_copy;
-[ RECORD 1 ]-----+-----
pid          | 29534
datid        | 30977
datname      | testdb
relid        | 30978
command      | COPY FROM
type         | PIPE
bytes_processed | 615087771
bytes_total   | 0
tuples_processed | 13404
tuples_excluded | 0
```

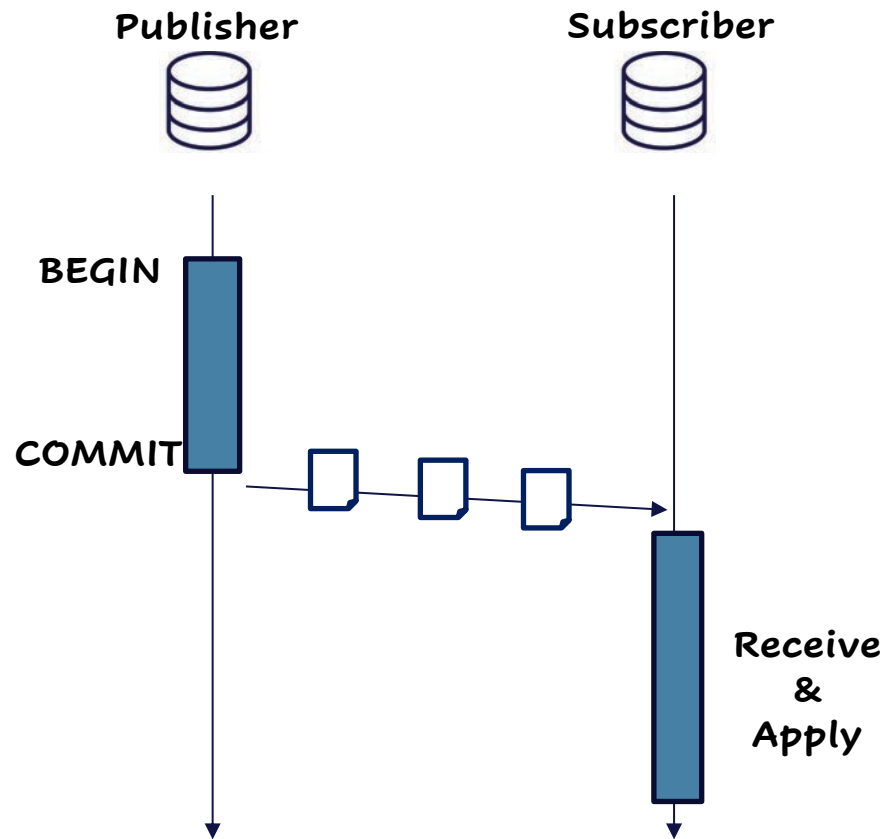
Logical **Streaming** Replicationが追加

- （PG13まで）トランザクションのコミット時にトランザクション全体の変更をサブスクライバ（下流）に送信
- トランザクションの途中でも、サブスクライバに送信可能になった
- CREATE SUBSCRIPTIONコマンドでstreaming = onを指定

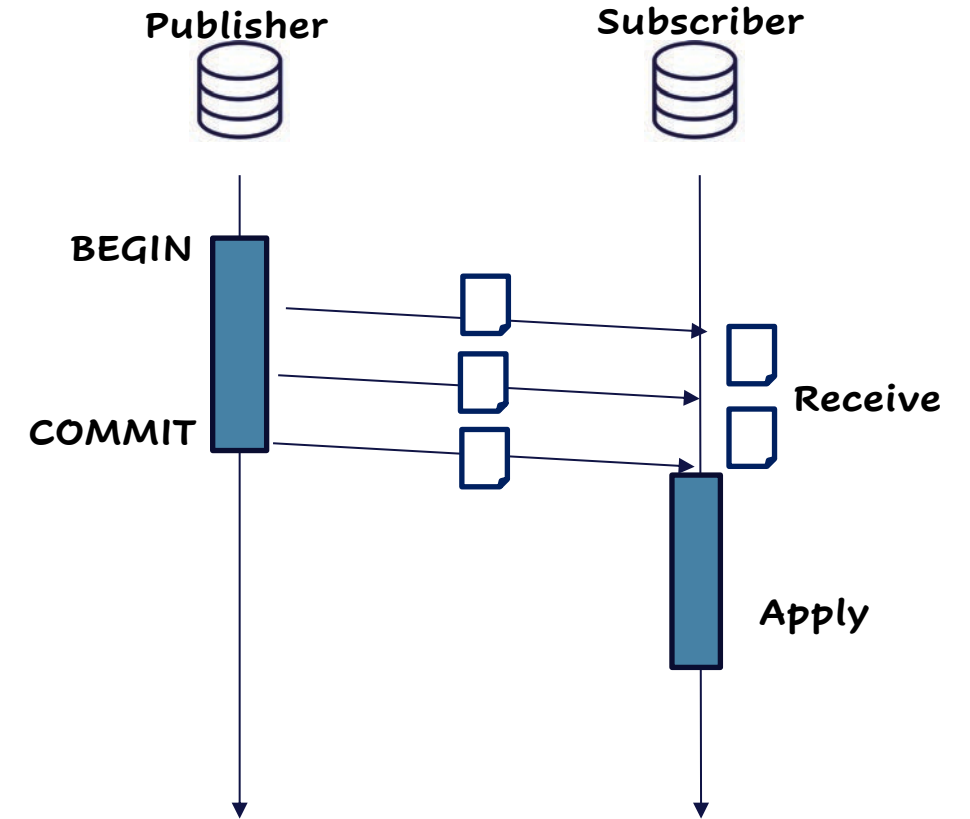


Logical **Streaming** Replicationが追加 (2)

Logical Replication



Logical **Streaming** Replication



Logical **Streaming** Replicationが追加 (3)

- *logical_decoding_work_mem*で制御可能
- トランザクションを適用するタイミングは変わらない（上流でCOMMITした後）
- パブリッシャ（上流）で一括挿入する場合などに有効
- 同期レプリケーションを利用する場合にも有効

pg_stat_wal, pg_stat_replication_slots

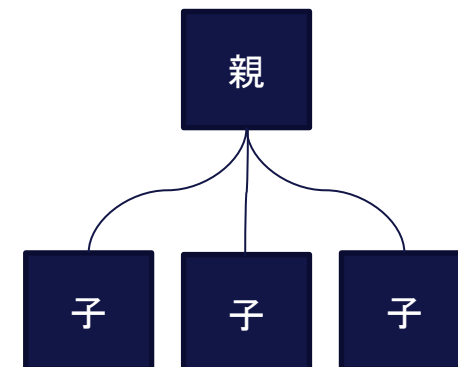
- pg_stat_walビュー
 - データベース全体のWALに関連する統計情報を表示
 - *track_wal_io_timing* = onでWALのI/O時間(wal_write/sync_time)も計測
- pg_stat_replication_slotsビュー
 - レプリケーションスロットを使う機能（論理レプリケーション、論理デコーディング）に関連する統計情報を表示
 - *logical_decoding_work_mem*の調整に便利

```
=# SELECT * FROM pg_stat_wal;
-[ RECORD 1 ]-----+-----
wal_records      | 1784593
wal_fpi          | 8307
wal_bytes        | 499802699
wal_buffers_full | 34573
wal_write        | 118416
wal_sync         | 0
wal_write_time   | 8568.164
wal_sync_time    | 0
stats_reset      | 2021-07-12 23:53:34.753769+09
```

```
=# SELECT * FROM pg_stat_replication_slots
-[ RECORD 1 ]+-----
slot_name      | testslot
spill_txns     | 1
spill_count    | 2
spill_bytes    | 132000000
stream_txns    | 0
stream_count   | 0
stream_bytes   | 0
total_txns     | 43940
total_bytes    | 160693197
stats_reset    |
```

ALTER TABLE ... DETACH [CONCURRENTLY|FINALIZE]

- CONCURRENTLYオプションを付けることで、パーティションテーブルの取り外し（DETACH）がDML（SELECTなど）と競合しない
- これまでは排他ロックを取得していたので同時実行中のDMLと競合していた
- パーティションテーブルの取り付け（ATTACH PARTITION）もDMLと競合しない（デフォルトパーティションを除く）



ALTER TABLE ... DETACH [CONCURRENTLY|FINALIZE]

- 途中でキャンセルされた場合、“DETACH PENDING”という状態になる
- ‘¥d+’メタコマンドやpg_inherits.inhdetachpendingを見ることで確認できる
- FINALIZEでDETACHを完了させる

```
=# ¥d+ p
```

Partitioned table "public.p"

Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target	Description
a	integer				plain			

Partition key: LIST (a)

Partitions: c1 FOR VALUES IN (1),
 c2 FOR VALUES IN (2),
 c3 FOR VALUES IN (3) (DETACH PENDING)

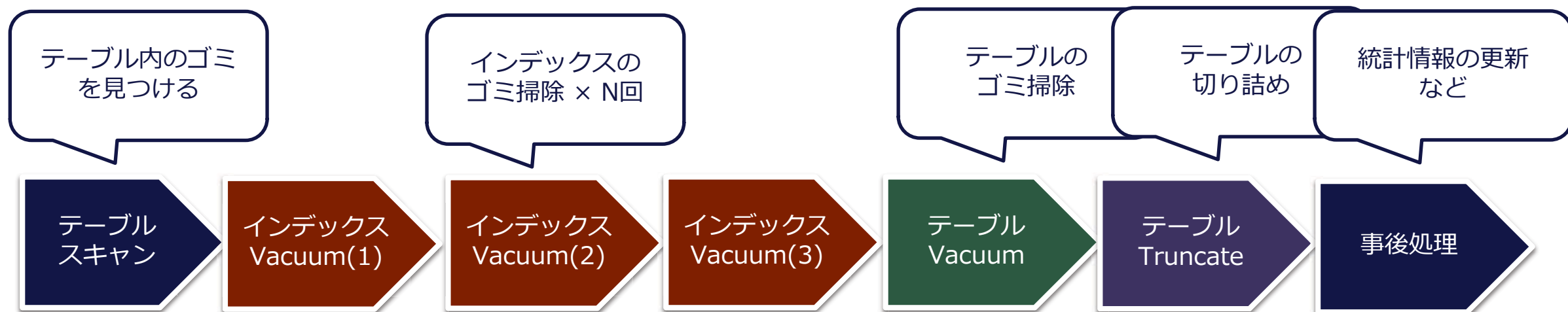
```
=# ALTER TABLE p DETACH PARTITION c3 FINALIZE;  
ALTER TABLE
```



Vacuum改善

インデックスVacuumのスキップ

- インデックスVacuumはもっとも時間がかかる処理の一つ
- ゴミがテーブル全体の2%以内の集中している場合、インデックスVacuumをスキップする
- デフォルトで有効
- INDEX_CLEANUP = ONで強制的にインデックスVacuumをすることも可能



Vacuum改善

インデックスVacuumのスキップ

- インデックスVacuumはもっとも時間がかかる処理の一つ
- ゴミがテーブル全体の2%以内の集中している場合、インデックスVacuumをスキップする
- デフォルトで有効
- INDEX_CLEANUP = ONで強制的にインデックスVacuumをすることも可能

ゴミが狭い範囲に集中している場合はスキップ



Vacuum改善 (2)

Failsafeモードの導入

- XID周回の危機が迫っている場合、XIDの凍結処理 (FREEZE) 以外の処理をスキップ
- さらに、autovacuum遅延も無効にする
- XIDの凍結処理を最優先に
- `vacuum_failsafe_age`、`vacuum_multixact_failsafe_age`で制御可能
 - デフォルトは16億



データ破損に関連する新機能

amcheckモジュールとpg_amcheckコマンド

- contrib/amcheckモジュールがheap（テーブル）にも対応
 - テーブルの破損検知が可能
 - これまではBtreeインデックスのみ対応していた
- pg_amcheckコマンドが追加
 - amcheckモジュールがより便利に使えるようになる
 - テーブル名やスキーマ名でフィルタリング
 - --jobsオプションで並列処理

データ破損に関連する新機能

pg_surgeryモジュール

- contrib/pg_surgeryモジュールが追加
- タブルのメタデータを直接編集するツール
 - Surgery : 手術
- 破損したテーブル等に使える
 - 破損していると論理ダンプすら取れない可能性がある
- UNIQUE制約、外部キー制約等を見捨ててタブルを編集するので使う場面は注意が必要（内部構造を理解している専門家が行うべき）
- `heap_force_kill('tbl'::regclass, '(10, 60)::tid)`
- `heap_force_freeze('tbl'::regclass, '(10, 60)::tid)`



開発(SQL関連)

SQLを使った関数・プロシージャ本体がSQL標準に対応

- これまではSQL関数の本体は、AS \$\$... \$\$ で関数本体を記述していた

~v13

```
CREATE FUNCTION bmi(  
  wight_kg int,  
  height_m float)  
RETURNS float AS  
$$  
    SELECT wight_kg / (height_m^2);  
$$  
LANGUAGE SQL;
```

v14~

```
CREATE FUNCTION bmi(  
  wight_kg int,  
  height_m float)  
RETURNS float  
LANGUAGE SQL  
RETURN wight_kg / (height_m^2);
```



SQLを使った関数・プロシージャ本体がSQL標準に対応

- 複数あるSQL関数本体もBEGIN ATOMICを使うことで、AS \$\$... \$\$は必要なくなった

~v13

```
CREATE PROCEDURE insert_data(val int) AS
$$
    BEGIN
        INSERT INTO tbl_a values (val);
        INSERT INTO tbl_b values (val);
    END;
$$
LANGUAGE SQL;
```

v14~

```
CREATE PROCEDURE insert_data(val
int)
BEGIN ATOMIC
    INSERT INTO tbl_a values (val);
    INSERT INTO tbl_b values (val);
END;
LANGUAGE SQL;
```



SQLを使った関数・プロシージャ本体がSQL標準に対応

- SQL標準に対応
 - DBMS移行時に便利
- 関数定義時にパースされる
- 関数とその中で利用するオブジェクトに依存関係が作られる

```
CREATE PROCEDURE insert_data(val int)
BEGIN ATOMIC
    INSERT INTO tbl_a values (val);
    INSERT INTO tbl_b values (val);
END;
LANGUAGE SQL;
```

SEARCH句とCYCLE句

- 再帰SQL (WITH RECURSIVE) にSEARCH句とCYCLE句が追加
- SQL標準
- SEARCH句：深さ優先探索、幅優先探索を指定
 - 木構造の探索に便利
- CYCLE句：循環を検知
 - グラフ構造の探索に便利



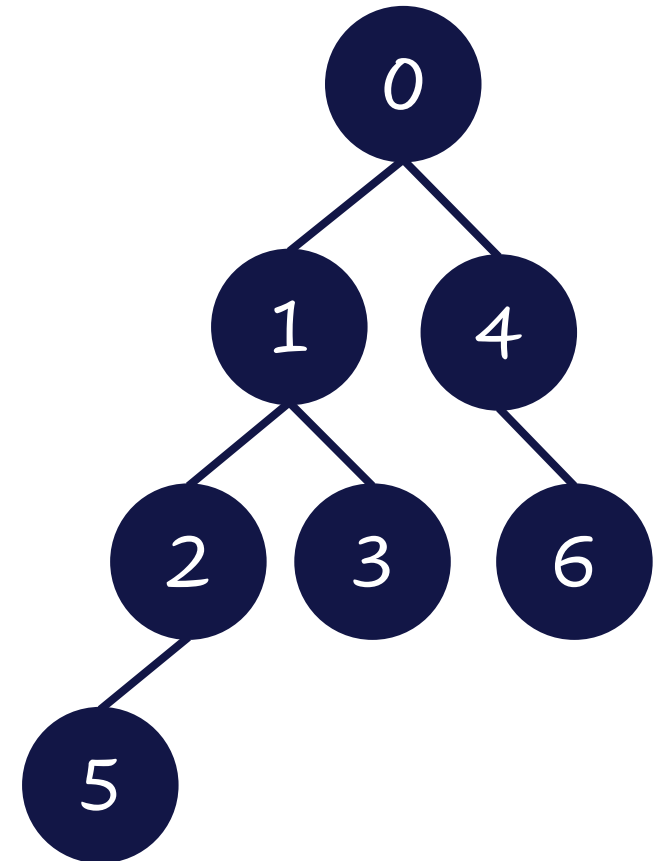
SEARCH句

SEARCH句で深さ優先探索、幅優先探索

```
=# SELECT * FROM tree;
```

id	parent
0	(null)
1	0
2	1
3	1
4	0
5	2
6	4

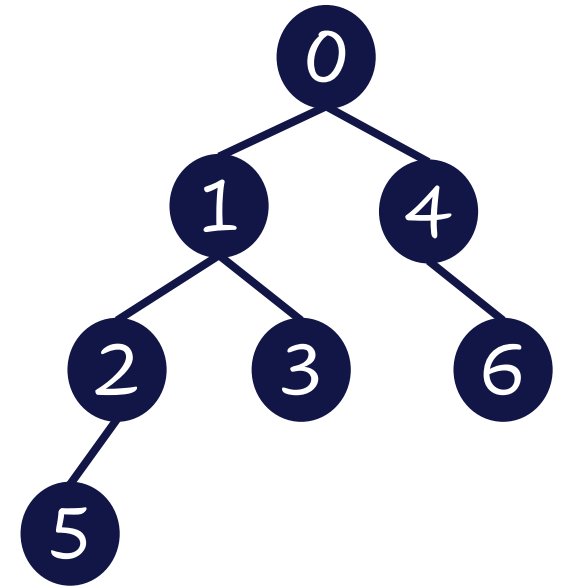
(7 rows)



SEARCH句

深さ優先探索（従来の方法）

```
WITH RECURSIVE search_tree (id, parent, lv, path) as (  
  SELECT t.id, t.parent, 1, array[t.id]  
  FROM tree t  
  WHERE parent is null  
  
  UNION ALL  
  
  SELECT t.id, t.parent, lv + 1, path || t.id  
  FROM tree t, search_tree st  
  WHERE t.parent = st.id  
)  
SELECT * FROM search_tree ORDER BY path;
```

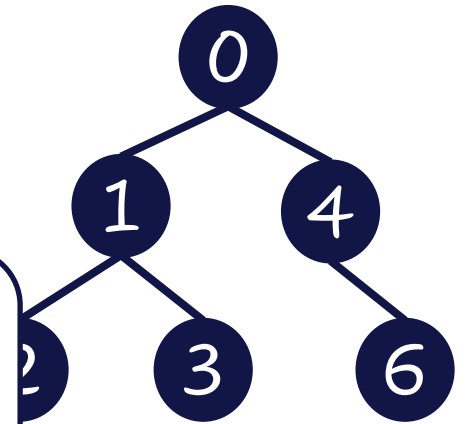


SEARCH句

深さ優先探索（従来の方法）

```
WITH RECURSIVE search_tree (id, parent, lv, path) as  
  SELECT t.id, t.parent, 1, array[t.id]  
  FROM tree t  
  WHERE parent is null
```

最初に一度だけ実行する部分



UNION ALL

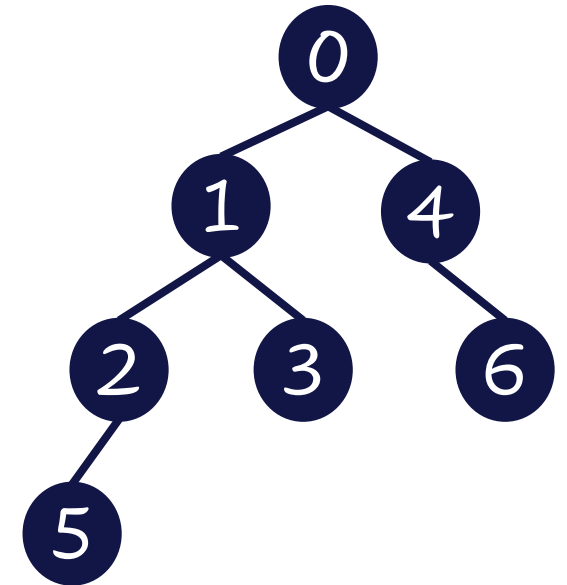
```
SELECT t.id, t.parent, lv + 1, path || t.id  
FROM tree t, search_tree st  
WHERE t.parent = st.id
```

```
)  
SELECT * FROM search_tree ORDER BY path;
```

SEARCH句

深さ優先探索（従来の方法）

```
WITH RECURSIVE search_tree (id, parent, lv, path) as (  
  SELECT t.id, t.parent, 1, array[t.id]  
  FROM tree t  
  WHERE parent is null  
  
  UNION ALL  
  
  SELECT t.id, t.parent, lv + 1, path || t.id  
  FROM tree t, search_tree st  
  WHERE t.parent = st.id  
)  
SELECT * FROM search_tree ORDER BY path;
```

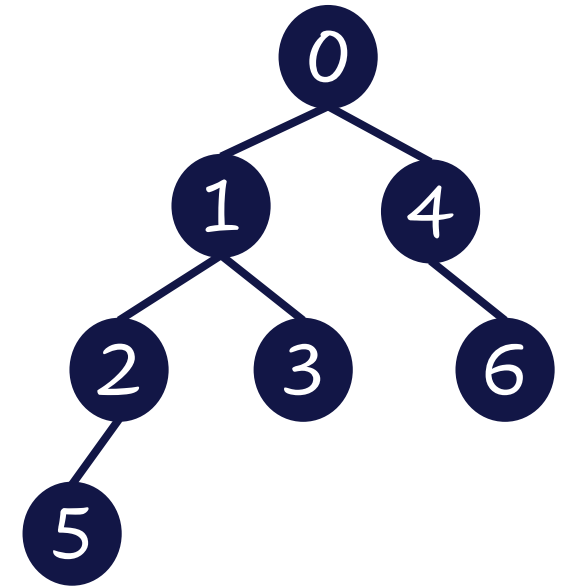


再帰的に
呼ばれる部分

SEARCH句

深さ優先探索（従来の方法）

```
WITH RECURSIVE search_tree (id, parent, lv, path) as (  
  SELECT t.id, t.parent, 1, array[t.id]  
  FROM tree t  
  WHERE parent is null  
  
  UNION ALL  
  
  SELECT t.id, t.parent, lv + 1, path || t.id  
  FROM tree t, search_tree st  
  WHERE t.parent = st.id  
)  
SELECT * FROM search_tree ORDER BY path;
```



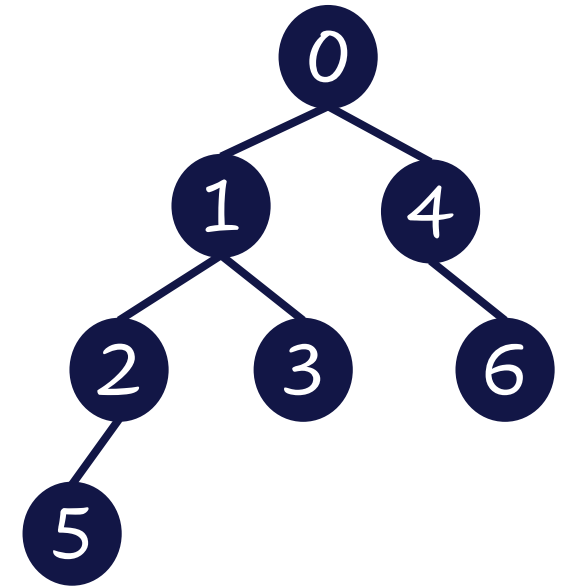
id	parent	lv	path
0		1	{0}
1	0	2	{0,1}
2	1	3	{0,1,2}
5	2	4	{0,1,2,5}
3	1	3	{0,1,3}
4	0	2	{0,4}
6	4	3	{0,4,6}

(7 rows)

SEARCH句

幅優先探索（従来の方法）

```
WITH RECURSIVE search_tree (id, parent, lv, path) as (  
  SELECT t.id, t.parent, 1, array[t.id]  
  FROM tree t  
  WHERE parent is null  
  
  UNION ALL  
  
  SELECT t.id, t.parent, lv + 1, path || t.id  
  FROM tree t, search_tree st  
  WHERE t.parent = st.id  
)  
SELECT * FROM search_tree ORDER BY lv;
```



id	parent	lv	path
0		0	{0}
1	0	1	{0,1}
4	0	1	{0,4}
2	1	2	{0,1,2}
3	1	2	{0,1,3}
6	4	2	{0,4,6}
5	2	3	{0,1,2,5}

(7 rows)



SEARCH句

深さ優先探索 (SEARCH句)

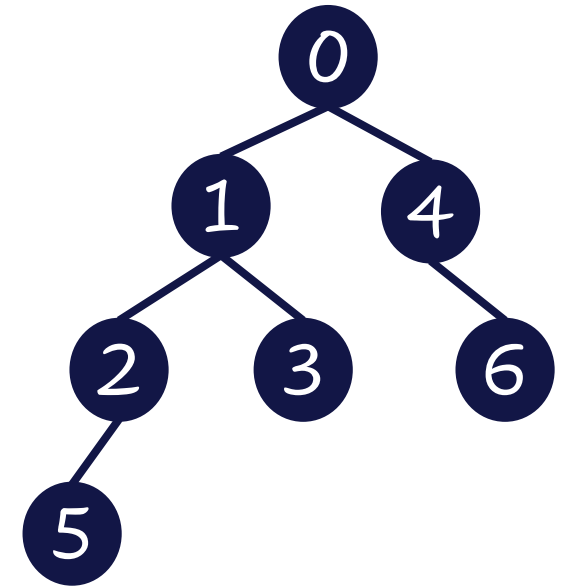
```
WITH RECURSIVE search_tree (id, parent, lv, path) as (  
  SELECT t.id, t.parent, 1, array[t.id]  
  FROM tree t  
  WHERE parent is null
```

UNION ALL

```
  SELECT t.id, t.parent, lv + 1, path || t.id  
  FROM tree t, search_tree st  
  WHERE t.parent = st.id
```

) **SEARCH DEPTH FIRST BY id SET ordercol**

```
SELECT * FROM search_tree ORDER BY ordercol;
```

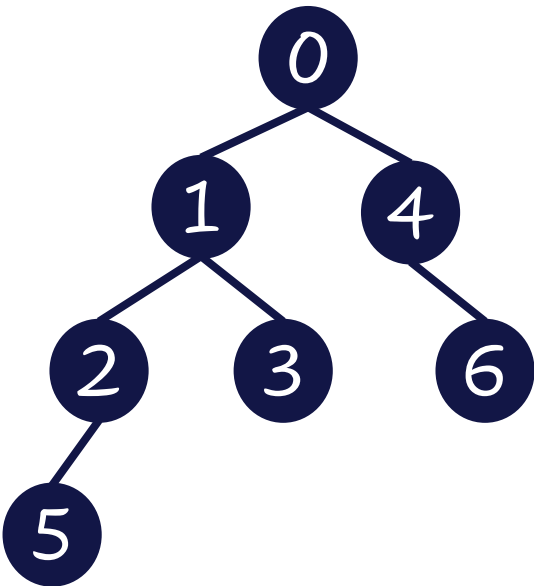


id	parent	lv	path	ordercol
0		1	{0}	{(0)}
1	0	2	{0,1}	{(0),(1)}
2	1	3	{0,1,2}	{(0),(1),(2)}
5	2	4	{0,1,2,5}	{(0),(1),(2),(5)}
3	1	3	{0,1,3}	{(0),(1),(3)}
4	0	2	{0,4}	{(0),(4)}
6	4	3	{0,4,6}	{(0),(4),(6)}

SEARCH句

幅優先探索 (SEARCH句)

```
WITH RECURSIVE search_tree (id, parent, lv, path) as (  
  SELECT t.id, t.parent, 1, array[t.id]  
  FROM tree t  
  WHERE parent is null  
  
  UNION ALL  
  
  SELECT t.id, t.parent, lv + 1, path || t.id  
  FROM tree t, search_tree st  
  WHERE t.parent = st.id  
) SEARCH BREADTH FIRST BY id SET ordercol  
SELECT * FROM search_tree ORDER BY ordercol;
```



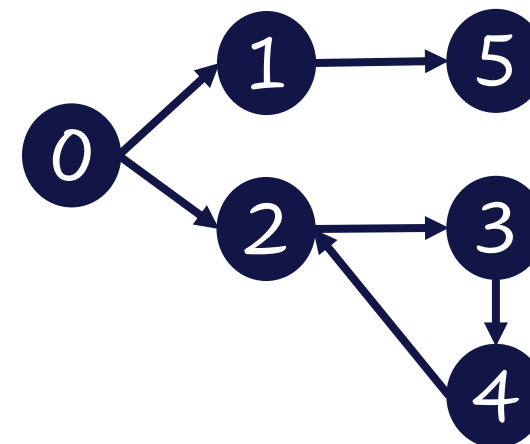
id	parent	lv	path	ordercol
0		1	{0}	(0,0)
1	0	2	{0,1}	(1,1)
4	0	2	{0,4}	(1,4)
2	1	3	{0,1,2}	(2,2)
3	1	3	{0,1,3}	(2,3)
6	4	3	{0,4,6}	(2,6)
5	2	4	{0,1,2,5}	(3,5)



CYCLE句

従来の方法

```
WITH RECURSIVE search_graph(id, next, is_cycle, path) as (  
  SELECT g.id, g.next, false, array[g.id]  
  FROM graph g  
  WHERE g.id = 0  
  
  UNION ALL  
  
  SELECT g.id, g.next, g.id = ANY(path), path || g.id  
  FROM graph g, search_graph sg  
  WHERE g.id = sg.next AND NOT is_cycle  
)  
SELECT * FROM search_graph;
```



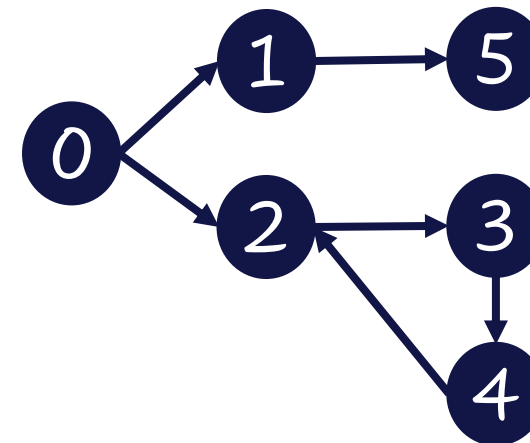
id	next
0	1
1	5
0	2
2	3
3	4
4	2

(6 rows)

CYCLE句

従来の方法

```
WITH RECURSIVE search_graph(id, next, is_cycle, path) as (  
  SELECT g.id, g.next, false, array[g.id]  
  FROM graph g  
  WHERE g.id = 0  
  
  UNION ALL  
  
  SELECT g.id, g.next, g.id = ANY(path), path || g.id  
  FROM graph g, search_graph sg  
  WHERE g.id = sg.next AND NOT is_cycle  
)  
SELECT * FROM search_graph;
```



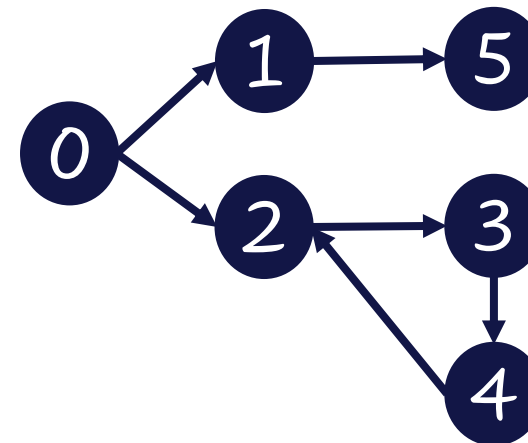
id	next	is_cycle	path
0	1	f	{0}
0	2	f	{0}
1	5	f	{0,1}
2	3	f	{0,2}
3	4	f	{0,2,3}
4	2	f	{0,2,3,4}
2	3	t	{0,2,3,4,2}

(7 rows)

CYCLE句

CYCLE句を使った循環検知

```
WITH RECURSIVE search_graph(id, next) as (  
  SELECT g.id, g.next  
  FROM graph g  
  WHERE g.id = 0
```



```
UNION ALL
```

```
  SELECT g.id, g.next  
  FROM graph g, search_graph sg  
  WHERE g.id = sg.next  
) CYCLE id SET is_cycle USING path  
SELECT * FROM search_graph;
```

id	next	is_cycle	path
0	1	f	{(0)}
0	2	f	{(0)}
1	5	f	{(0), (1)}
2	3	f	{(0), (2)}
3	4	f	{(0), (2), (3)}
4	2	f	{(0), (2), (3), (4)}
2	3	t	{(0), (2), (3), (4), (2)}

(7 rows)



まとめ

- 9/30にリリース済み
- 約200個の新機能や改善
 - 問い合わせのパイプライン化
 - REFRESH MATERIALIZED VIEWがパラレルクエリ対応
 - postgres_fdwがクエリの並列処理をサポート
 - multirageデータ型
- ぜひ使ってみてください！

ここをもっと便利にしてほしい、こういう機能がほしいなどのコメントもお待ちしています！



参考資料

- PostgreSQL 14 Release Note
 - <https://www.postgresql.org/docs/14/release-14.html>
- 篠田の虎の巻「PostgreSQL 14 Beta 1 新機能検証結果」
 - https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/lcc/pdf/PostgreSQL14Beta1_New_Features_ja_20210521-1.pdf
- Waiting for PostgreSQL 14
 - <https://www.depesz.com/tag/pg14/>
- Postgres 14 highlight
 - <https://paquier.xyz/>
- PostgreSQL 14 がやってくる
 - https://qiita.com/nuko_yokohama
- Configurable LZ4 TOAST compression
 - <https://www.enterprisedb.com/blog/configurable-lz4-toast-compression>
- Pg14_sql_standard_function_body
 - <https://www.slideshare.net/kasaharatt/pg14sqlstandardfunctionbody>



Thank you

masahiko.sawada@enterprisedb.com

