Hewlett Packard
Enterprise

列指向アクセスメソッド徹底比較

Noriyoshi Shinoda

December 6, 2024

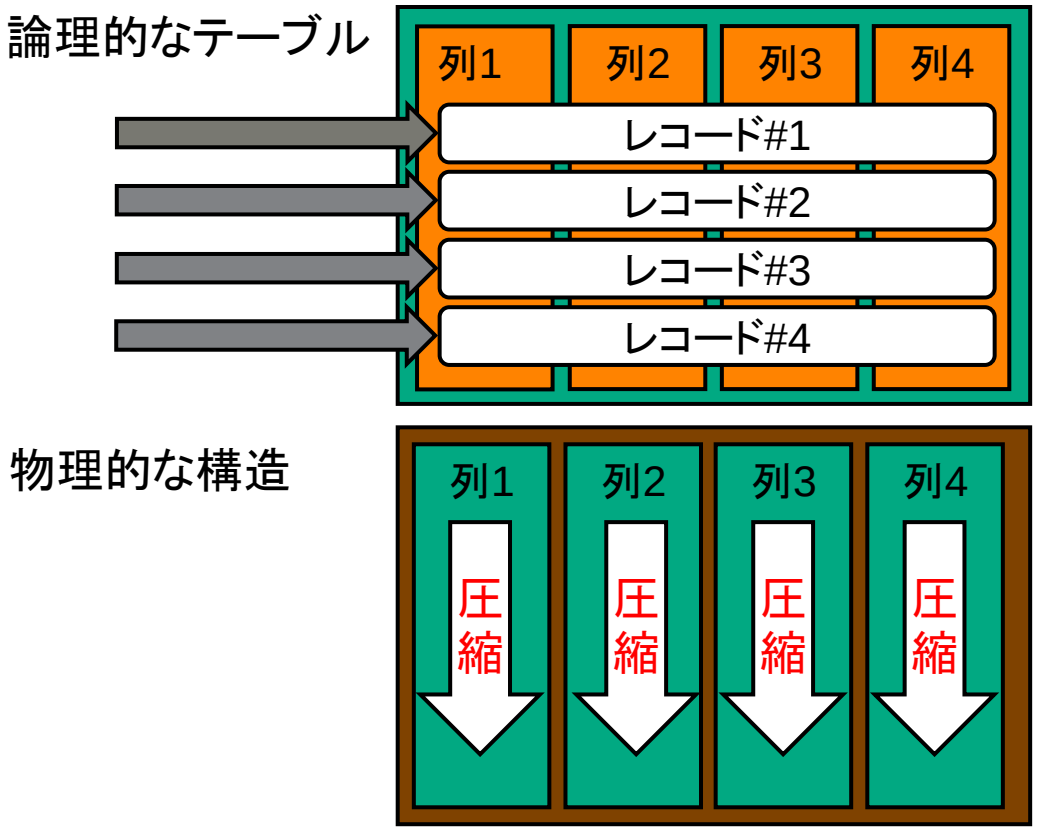
SPEAKER

Featured in
The Redgate 100

- ✓篠田 典良（しのだ のりよし）
- ✓所属
  - ✓日本ヒューレット・パッカード合同会社
- ✓現在の業務など
  - ✓「篠田の虎の巻」作成
  - ✓PostgreSQL 開発（PostgreSQL 10～17, 18 dev）
  - ✓Oracle ACE Pro (2009～)
  - ✓PostgreSQLをはじめ Oracle Database, Microsoft SQL Server, Vertica 等 RDBMS 全般に関するシステムの設計、移行、チューニング、コンサルティング
- ✓関連する URL
  - ✓Redgate 100 in 2022 (Most influential in the database community 2022)
    - ✓https://www.red-gate.com/hub/redgate-100/
  - ✓「PostgreSQL 虎の巻」シリーズ
    - ✓https://github.com/nori-shinoda/documents/blob/main/README.md
  - ✓Oracle ACE Profile
    - ✓https://ace.oracle.com/apex/ace/profile/nshino483

はじめに
列指向テーブルとは？

- ✓列単位でデータの管理を行うテーブル
  - ✓アプリケーションからは透過的
  - ✓データ圧縮による I/O 削減を狙う
  - ✓HTAP にも利用される



はじめに
列指向アクセスメソッド徹底比較

- ✓列指向データを扱う PostgreSQL エクステンション
  - ✓Citus
  - ✓Hydra
  - ✓pg_mooncake
- ✓列指向データを扱うデータベース
  - ✓AlloyDB for PostgreSQL

テーブルアクセスメソッド

テーブルアクセスメソッド
SQL 文に対するデータの動作を決める

✓TableAmRoutine 構造体(src/include/access/tableam.h)

```
typedef struct TableAmRoutine
{
    NodeTag         type;
    const TupleTableSlotOps *(*slot_callbacks) (Relation rel);
    TableScanDesc   (*scan_begin) (Relation rel, ...
    void            (*scan_end) (TableScanDesc scan);
    void            (*scan_rescan) (TableScanDesc scan, struct ScanKeyData *key, …
    bool            (*scan_getnextslot) (TableScanDesc scan, …
    void            (*scan_set_tidrange) (TableScanDesc scan, …
    bool            (*scan_getnextslot_tidrange)…
...

}
```

テーブルアクセスメソッド
SQL 文に対するデータの動作を決める

- ✓テーブルアクセスメソッドの登録
  - ✓TableAmRoutine 構造体を返す FUNCTION を作成
  - ✓CREATE ACCESS METHOD 文でハンドラーを登録
  - ✓pg_am カタログに格納
- ✓pg_mooncake エクステンションの初期化スクリプト

```
CREATE FUNCTION columnstore_handler(internal) RETURNS table_am_handler
    AS 'MODULE_PATHNAME' LANGUAGE C STRICT;
CREATE ACCESS METHOD columnstore TYPE TABLE HANDLER columnstore_handle ;
```

CITUS

CITUS
概要

- ✓Citus Data (https://www.citusdata.com/) が開発
  - ✓Azure Cosmos DB for PostgreSQL の中核機能
  - ✓https://github.com/citusdata/citus
- ✓エクステンション名：citus_columnar
  - ✓分散データベースを構成するエクステンション citus から分離
- ✓Table Access Method (Pluggable Table Storage Interface) として実装

```
postgres=> SELECT * FROM pg_am WHERE amtype='t';
  oid   |  amname  |                amhandler                | amtype
--------+----------+-----------------------------------------+--------
      2 | heap     | heap_tableam_handler                    | t
  16417 | columnar | columnar_internal.columnar_handler      | t
(2 rows)
```

CITUS
テーブルの作成

✓アクセスメソッド columnar を利用
- ✓CREATE TABLE 文に USING columnar 句を指定して作成
- ✓圧縮方法や圧縮率は独自 GUC で指定する(後述)

✓作成例

```
postgres=> CREATE TABLE citus1(c1 NUMERIC, c2 VARCHAR(10)) USING columnar;
CREATE TABLE
postgres=> ¥d+
                                    List of relations
 Schema |     Name      | Type  |  Owner   | Persistence | Access method |  Size   | …
--------+---------------+-------+----------+-------------+---------------+---------+-…
 public | citus1        | table | demo     | permanent   | columnar      | 16 kB   |
(1 row)
```

CITUS
特徴

✓データ更新

✓UPDATE 文／DELETE 文／MERGE 文は実行不可

```
postgres=> DELETE FROM citus1 WHERE c1=0;
ERROR:  UPDATE and CTID scans not supported for ColumnarScan
postgres=> UPDATE citus1 SET c2='update' WHERE c1=0;
ERROR:  UPDATE and CTID scans not supported for ColumnarScan
postgres=> MERGE INTO citus1 AS c USING data1 AS d ON c.c1 = d.c1
                WHEN MATCHED THEN DO NOTHING;
ERROR:  UPDATE and CTID scans not supported for ColumnarScan
```

✓その他

✓インデックス利用可能

✓主キー、一意キー、NOT NULL 等制約も利用可能

CITUS
圧縮設定の変更

✓圧縮オプションの指定

オプション	説明	デフォルト	設定範囲
columnar.compression	圧縮方法	zstd	zstd, lz4, pglz, none
columnar.compression_level	圧縮レベル	3	1～19
columnar.stripe_row_limit	ストライプ行数	150,000	1,000～10,000,000
columnar.chunk_group_row_limit	チャンクグループ行数	10,000	1,000～100,000

✓テーブルの圧縮オプション確認

```
postgres=> SELECT * FROM columnar.options;

 regclass | chunk_group_row_limit | stripe_row_limit | compression_level | compression
----------+-----------------------+------------------+-------------------+-------------
 citus1   |                 10000 |           150000 |                 3 | zstd
(1 row)
```

CITUS
圧縮の効果

✓圧縮効果（2 列／1,000 万タプル）
- ✓citus1 : 列指向テーブル／一括コミット
- ✓citus2 : 列指向テーブル／10 タプル単位コミット
- ✓data1 : Heap テーブル

✓一括 INSERT による圧縮効果が高い

```
postgres=> ¥d+
                                        List of relations
 Schema |     Name      | Type  |  Owner   | Persistence | Access method |  Size   |…
--------+---------------+-------+----------+-------------+---------------+---------+…
 public | citus1        | table | demo     | permanent   | columnar      | 28 MB   |
 public | citus2        | table | demo     | permanent   | columnar      | 7813 MB |
 public | data1         | table | demo     | permanent   | heap          | 422 MB  |
(3 rows)
```

CITUS
圧縮の効果

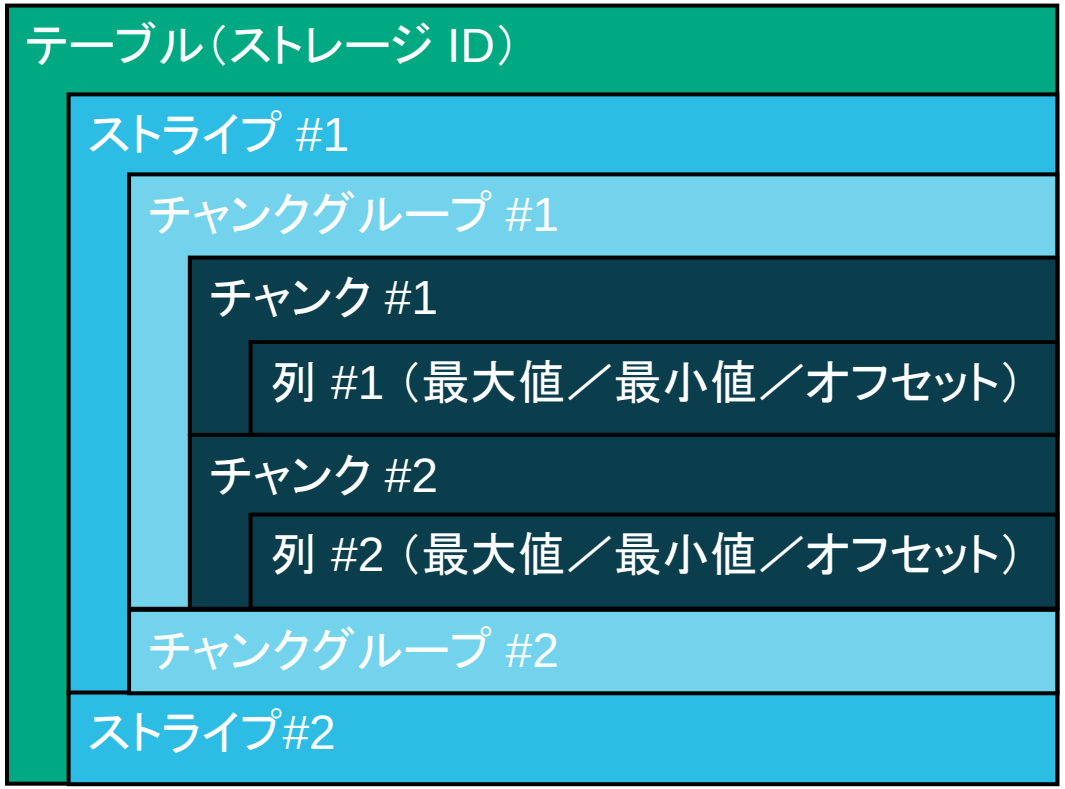
✓圧縮効率の確認
- ✓パラメーターはデフォルト
- ✓5 列／1,000 万タプルで検証（Heap テーブルは 575 MB）
- ✓単一トランザクション

✓圧縮レベルの指定は zstd 以外では変化が見られなかった

圧縮メソッド	圧縮レベル 1 (MB)	圧縮レベル 19 (MB)	備考
zstd	31	16	
lz4	46	46	
pglz	50	50	
none	504	504	

CITUS
ストレージ構造

✓Heap テーブルと異なる構造を持つ



- ✓テーブル
  - ✓ストレージ ID を持つ
- ✓ストライプ
  - ✓圧縮単位
  - ✓トランザクションまたは columnar.stripe_row_limit の単位
- ✓チャンクグループ
  - ✓チャンクの集合
  - ✓複数列をまとめて管理
  - ✓columnar.chunk_group_row_limit の単位で作成
- ✓チャンク
  - ✓列値、最大値／最小値／オフセット

CITUS
カタログ

✓ストレージ構造を示すカタログが用意されている

✓旧バージョンでは「ストレージID」とテーブルを関連づけるカタログが無かった（VACUUM VERBOSEで確認）

✓Citus 12 では columnar.storage ビューが提供（Hydra には無い）

カタログ名	内容	備考
columnar.chunk	チャンク一覧、列最大値、最小値など	全ユーザ参照可
columnar.chunk_group	チャンクグループ一覧、削除行数など	全ユーザ参照可
columnar.options	テーブル圧縮オプション一覧	全ユーザ参照可
columnar.storage	ストレージIDとテーブルの関係	全ユーザ参照可
columnar.stripe	ストライプ一覧	全ユーザ参照可

CITUS
アクセス・メソッドの変更

✓既存の Heap テーブルからの変更も可能

```
postgres=> ALTER TABLE data1 SET ACCESS METHOD columnar;
ALTER TABLE
postgres=> ¥d+ data1
                                          Table "public.data1"
 Column |         Type          | Collation | Nullable | Default | Storage  | …
--------+-----------------------+-----------+----------+---------+----------+-…
 c1     | integer               |           | not null |         | plain    | …
 c2     | character varying(10) |           |          |         | extended | …
Indexes:
    "data1_pkey" PRIMARY KEY, btree (c1)
Access method: columnar
```

✓ただし columnar.options カタログに情報が反映されない（バグ？）

CITUS
実行計画

✓Custom Scan によるアクセス

```
postgres=> EXPLAIN ANALYZE SELECT SUM(c1) FROM column1 WHERE c1 = 10000 ;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------…
 Aggregate  (cost=25.88..25.89 rows=1 width=8) (actual time=8.042..8.043 rows=1 loops=1)
   ->  Custom Scan (ColumnarScan) on citus1  (cost=0.00..25.88 rows=1 width=4) (actual…
         Filter: (c1 = 10000)
         Rows Removed by Filter: 9999
         Columnar Projected Columns: c1
         Columnar Chunk Group Filters: (c1 = 10000)
         Columnar Chunk Groups Removed by Filter: 999
 Planning Time: 2.667 ms
 Execution Time: 8.081 ms
(9 rows)
```

CITUS
実行計画

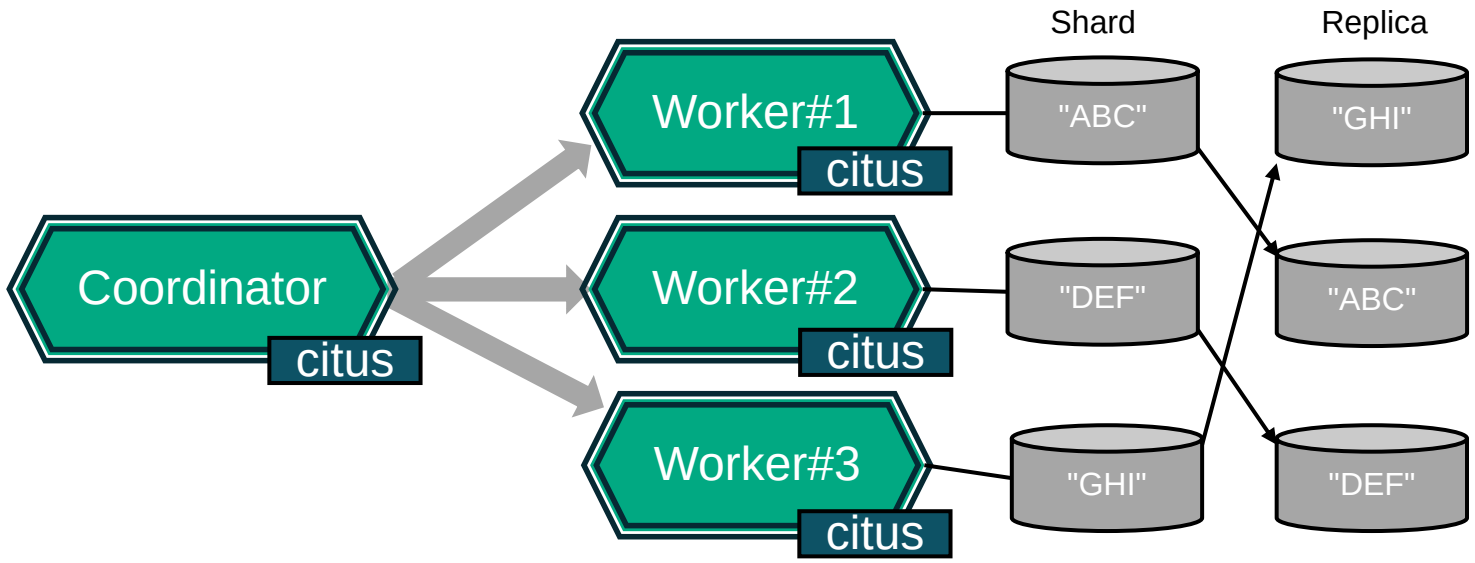
✓パラレルクエリーはサポートされない

✓強制的にパラレル実行しようとすると実行計画が Single Copy となる

```
postgres=> SET debug_parallel_query = on;
SET
postgres=> EXPLAIN SELECT * FROM citus1;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------
 Gather  (cost=1000.00..1004467.71 rows=10000000 width=10)
   Workers Planned: 1
   Single Copy: true
   ->  Custom Scan (ColumnarScan) on citus1  (cost=0.00..3467.71 rows=10000000 width=10)
         Columnar Projected Columns: c1, c2
(5 rows)
```

CITUS
実行計画

✓並列処理したければ citus エクステンションの分散テーブル（Distributed Table）を利用



CITUS
検索性能

✓検索性能の確認
- ✓5 列 / 3 億タプルで検証（Heap: 17 GB / Columnar: 937 MB)
- ✓パラメーターはデフォルト
- ✓Linux のキャッシュクリア、PostgreSQL インスタンス再起動後に実行
- ✓Heap テーブルの検索はパラレルクエリー（Parallel Seq Scan／並列度 2）

✓検証結果

SQL	Heap (s)	Columnar (s)	備考
SELECT c2	44,249.4	22,519.7	EXPLAIN ANALYZE 結果
SELECT COUNT(c1)	46,143.3	14,779.4	一意列
SELECT COUNT(c2)	49,391.6	13,592.6	単一値列
SELECT SUM(c1)	42,268.1	13,110.4	

CITUS
検索性能

✓結合性能の確認
- ✓heap テーブル同士の Inner Join + COUNT
- ✓columnar テーブル同士の Inner Join + COUNT

✓検証結果

結合条件	Time (ms)	備考
Heap Only	472,994.7	Parallel Hash Join (2 Worker)
Columnar Only	285,759.9	Hash Join

CITUS
更新性能

✓検証結果

✓単一トランザクション

SQL	Heap (ms)	Columnar (ms)	備考
INSERT	624,153.8	663,986.4	generate_series 関数で 3 億件一括
TRUNCATE	1,022.8	89.0	

✓総評

✓検索の高速性を確認できた

✓5 列 / 3 億タプル（Heap: 17 GB）は Citus から見ると下限

✓5 列 / 1 億タプル（Heap: 5 GB）では Heap テーブルの方が高速

✓タプル数、列数が多くなると性能差が出る

HYDRA

HYDRA
概要

- ✓Hydra (https://docs.hydra.so/overview) が開発
  - ✓一部 citus の機能を流用
  - ✓https://github.com/hydradatabase/hydra
- ✓エクステンション名：columnar
- ✓Table Access Method (Pluggable Table Storage Interface) として実装

```
postgres=> SELECT * FROM pg_am WHERE amtype='t';
  oid  |  amname  |          amhandler          | amtype
-------+----------+-----------------------------+--------
     2 | heap     | heap_tableam_handler        | t
 16425 | columnar | columnar.columnar_handler | t
(2 rows)
```

HYDRA
テーブルの作成

Citusと同じ

✓アクセスメソッド columnar を利用
- ✓CREATE TABLE 文に USING columnar 句を指定して作成
- ✓圧縮方法や圧縮率は後述する GUC で指定する

✓作成例

```
postgres=> CREATE TABLE hydra1(c1 NUMERIC, c2 VARCHAR(10)) USING columnar;
CREATE TABLE
Postgres=> ¥d+
                                    List of relations
 Schema |     Name      | Type  |  Owner   | Persistence | Access method |  Size   | …
--------+---------------+-------+----------+-------------+---------------+---------+-…
 public | hydra1        | table | demo     | permanent   | columnar      | 16 kB   |
(1 row)
```

HYDRA
テーブルの作成

Citusとほぼ同じ

✓ストレージオプションの指定

オプション	説明	デフォルト	設定範囲
columnar.compression	圧縮方法	zstd	zstd, lz4, pglz, none
columnar.compression_level	圧縮レベル	3	1～19
columnar.stripe_row_limit	ストライプ行数	150,000	1,000～10,000,000
columnar.chunk_group_row_limit	チャンクグループ行数	10,000	1,000～100,000,000

✓テーブルの圧縮オプション確認

```
postgres=> SELECT * FROM columnar.options;

 regclass | chunk_group_row_limit | stripe_row_limit | compression_level | compression
----------+-----------------------+------------------+-------------------+-------------
 hydra1   |                 10000 |           150000 |                 3 | zstd
(1 row)
```

HYDRA
テーブルの作成

✓その他オプションの指定

オプション	説明	デフォルト	設定範囲
columnar.column_cache_size	列キャッシュのサイズ（MB）	200MB	20MB ～ 20000MB
columnar.enable_column_cache	列キャッシュの有効化	off	off ～ on
columnar.min_parallel_processes	最小パラレルプロセス数	8	1 ～ 32
columnar.enable_parallel_execution	パラレルクエリー有効化	on	off ～ on
columnar.enable_vectorization	ベクトル化の有効化	on	off ～ on
columnar.enable_dml	DELETE/UPDATE 無効	on	off ～ on
columnar.enable_columnar_index_scan	カスタム索引スキャン	off	off ～ on
columnar.planner_debug_level	デバッグレベル指定	debug3	debug5 ～ log

HYDRA
特徴

✓更新 DML を制限できる

✓enable_dml を false に設定すると、UPDATE 文／DELETE 文は実行されない（エラーは発生しない）

```
postgres=> SET columnar.enable_dml = false;
SET
postgres=> DELETE FROM hydra1 WHERE c1=0;
DELETE 0
postgres=> UPDATE hydra1 SET c2='update' WHERE c1=0;
UPDATE 0
postgres=> SELECT * FROM hydra1 WHERE c1=0;
 c1 |  c2
----+------
  0 | zero
(1 row)
```

HYDRA
物理構成

Citusとほぼ同じ

- ✓物理ファイル
  - ✓PostgreSQL の標準に従う
  - ✓pg_class カタログの relfilenode 列に指定されたファイルを作成
  - ✓1 GB 単位で分割
  - ✓Citus も同じ
- ✓その他
  - ✓インデックス利用可能
  - ✓制約利用可能

HYDRA
カタログ

Citusとほぼ同じ

✓カタログ
- ✓基本的には Citus と共通
- ✓row_mask テーブルはテーブル書き込み状況を確認するため Hydra 独自
- ✓一部のカタログは SUPERUSER 権限が必要になるよう変更
- ✓Citus と異なり、ストレージ ID とテーブル名を関連付けるカタログが無い（VACUUM VERBOSE で表示）

オプション	内容	備考
columnar.chunk	チャンク一覧、列最大値、最小値など	SUPERUSERのみ
columnar.chunk_group	チャンクグループ一覧、削除行数など	全ユーザ参照可
columnar.options	テーブル圧縮オプション一覧	全ユーザ参照可
columnar.row_mask	書き込みチャンクを認識	SUPERUSERのみ
columnar.stripe	Stripe一覧	全ユーザ参照可

HYDRA
実行計画

✓Custom Scan によるアクセス（集約）

```
postgres=> EXPLAIN ANALYZE SELECT SUM(c1) FROM hydra1;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------…
Finalize Aggregate  (cost=4819.85..4819.86 rows=1 width=8) (actual time=81.054..81.929   …
->  Gather  (cost=4819.12..4819.83 rows=7 width=8) (actual time=76.702..81.922 rows=8    …
        Workers Planned: 7
        Workers Launched: 7
        ->  Parallel Custom Scan (VectorAggNode)  (cost=3819.12..3819.13 rows=1 width=8 …
              ->  Parallel Custom Scan (ColumnarScan) on hydra1  (cost=0.00..247.69 row …
                    Columnar Projected Columns: c1
 Planning Time: 0.305 ms
 Execution Time: 81.995 ms
(9 rows)
```

HYDRA
実行計画

✓Custom Scan によるアクセス(条件)

```
postgres=> EXPLAIN ANALYZE SELECT * FROM hydra1 WHERE c1 BETWEEN 100000 AND 20000000;
                                   QUERY PLAN
-----------------------------------------------------------------------------------------…
 Gather  (cost=1000.00..992036.70 rows=9905460 width=10) (actual time=0.791..434.999 row …
   Workers Planned: 7
   Workers Launched: 7
   ->  Parallel Custom Scan (ColumnarScan) on hydra1  (cost=0.00..490.70 rows=1428571 wi …
         Columnar Projected Columns: c1, c2
         Columnar Chunk Group Filters: ((c1 >= 100000) AND (c1 <= 20000000))
         Columnar Chunk Groups Removed by Filter: 9
         Columnar Vectorized Filter: ((c1 >= 100000) AND (c1 <= 20000000))
 Planning Time: 1.451 ms
 Execution Time: 607.679 ms
(10 rows)
```

HYDRA
実行計画

✓列キャッシュを有効化した場合の実行計画

```
postgres=> EXPLAIN ANALYZE SELECT COUNT(*) FROM hydra1;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------...
Finalize Aggregate  (cost=4819.85..4819.86 rows=1 width=8) (actual time=81.054..81.929   ...
         ->  Parallel Custom Scan (ColumnarScan) on hydra1 (cost=0.00..247.69 rows=1428  ...
                     Columnar Projected Columns: c1
                     Cache Hits: 0
                     Cache Misses: 286
                     Cache Evictions: 0
                     Cache Writes: 286
                     Cache Maximum Size: 11440000
                     Cache Ending Size: 11440000
                     Total Cache Entries: 286
...
```

HYDRA
実行計画

✓独自のインデックス・スキャン（デフォルト無効）

```
postgres=> EXPLAIN SELECT * FROM hydra1 WHERE c1=10000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using hydra1_pkey on hydra1  (cost=0.43..59.96 rows=1 width=9)
   Index Cond: (c1 = 10000)
(2 rows)

postgres=> SET columnar.enable_columnar_index_scan = on;
SET
postgres=> EXPLAIN SELECT * FROM hydra1 WHERE c1=10000;
                              QUERY PLAN
-----------------------------------------------------------------------
 Custom Scan (ColumnarIndexScan)  (cost=0.43..8.45 rows=1 width=9)
   ColumnarIndexScan using : hydra1_pkey
   Index Cond: (c1 = 10000)
(3 rows)
```

HYDRA
実行計画

- ✓実行計画の特徴
  - ✓何が何でもパラレルクエリー
  - ✓columnar.min_parallel_processes (default 8) で並列度を指定
    - ✓ ワーカー数の計算 columnar.min_parallel_processes - parallel_leader_participation
  - ✓max_parallel_workers (default 8) は効く
  - ✓max_parallel_workers_per_gather は効かない

HYDRA
検索性能

✓検索性能の確認
- ✓5 列 / 1 億タプルで検証
- ✓パラメーターはデフォルト
- ✓Linux のキャッシュクリア、PostgreSQL インスタンス再起動後に実行
- ✓Heap テーブルの検索はパラレルクエリー（Parallel Seq Scan／並列度 2）

✓検証結果

SQL	Heap (ms)	Columnar (ms)	備考
SELECT c2	13,797.2	7,123.8	EXPLAIN ANALYZE 結果
SELECT COUNT(c1)	4,171.8	881.3	一意列
SELECT COUNT(c2)	4,199.0	827.9	単一値列
SELECT SUM(c1)	4,149.0	886.7	合計

HYDRA
検索性能

✓結合性能の確認
- ✓heap テーブル同士の Inner Join + COUNT
- ✓columnstore テーブル同士の Inner Join + COUNT

✓検証結果

結合条件	Time (ms)	備考
Heap Only	104,745.0	Parallel Hash Join (2 Worker)
Columnar Only	34,103.8	Parallel Hash Join (7 Worker)

HYDRA
更新性能

✓更新性能の確認
- ✓単一トランザクション
- ✓DELETE 文、UPDATE 文はかなり遅くなる

✓検証結果

SQL	Heap (ms)	Columnar (ms)	備考
INSERT	236,782.0	112,421.6	generate_series 関数で 1 億件一括
UPDATE	3,165.6	713,977.1	1,000,000 件更新（主キー Index Scan)
DELETE	496.9	2,777.6	1,000,000 件削除（主キー Index Scan)
TRUNCATE	1,022.8	84.8	

✓総評
- ✓検索の高速性を確認できた
- ✓更新は遅くなるが、検索／結合性能は向上する
- ✓パラレルクエリー並列度が上がることが寄与していると思われる

PG_MOONCAKE

PG_MOONCAKE
概要

- ✓mooncake (https://mooncake.dev/) が開発
  - ✓https://github.com/Mooncake-Labs/pg_mooncake
- ✓DuckDB の機能を利用
  - ✓https://github.com/duckdb/duckdb
- ✓エクステンション名：pg_mooncake
- ✓Table Access Method (Pluggable Table Storage Interface) として実装

```
postgres=> SELECT * FROM pg_am WHERE amtype='t';
  oid  |    amname     |       amhandler        | amtype
-------+---------------+------------------------+--------
     2 | heap          | heap_tableam_handler   | t
 16390 | columnstore   | columnstore_handler    | t
(2 rows)
```

PG_MOONCAKE
テーブルの作成

- ✓アクセスメソッド columnstore を利用
  - ✓CREATE TABLE 文に USING columnstore 句を指定して作成
  - ✓圧縮方法や圧縮率は指定できない
- ✓作成例

```
postgres=> CREATE TABLE moon1(c1 NUMERIC, c2 VARCHAR(10)) USING columnstore;
CREATE TABLE
Postgres=> ¥d+
                                        List of relations
 Schema |     Name      | Type  |  Owner   | Persistence | Access method |  Size   | …
--------+---------------+-------+----------+-------------+---------------+---------+-…
 public | moon1         | table | demo     | permanent   | columnstore   | 0 bytes |
(1 row)
```

PG_MOONCAKE
特徴

✓テーブルの保存形式
- ✓データの保存は Parquet 形式
- ✓DuckDB が持つ Delta Lake / Iceberg アクセス機能を利用
- ✓ローカル・ファイルシステムとリモート S3 バケットにデータ保存可能

✓SQL 文の実行
- ✓テーブルデータ更新可能（ただし MERGE 文、SELECT FOR UPDATE 文実行不可）
- ✓ANALYZE 文／VACUUM 文実行不可

✓その他
- ✓インデックス利用不可
- ✓主キー、一意キー制約利用不可
- ✓NOT NULL 制約はテーブル定義可能だが機能しない

PG_MOONCAKE
圧縮の効果

✓圧縮効果（5 列／1 億タプル）
- ✓data1 : Heap テーブル
- ✓moon1 : 列指向テーブル／一括コミット ⇒ 0 bytes になっているが、実際は 382 MB
- ✓moon2 : 列指向テーブル／10 レコード単位コミット ⇒ 30 GB 以上

✓一括 INSERT による圧縮効果が高い

```
postgres=> ¥d+
                                        List of relations
 Schema |     Name      | Type  |  Owner   | Persistence | Access method |  Size    |…
--------+---------------+-------+----------+-------------+---------------+----------+…
 public | data1         | table | demo     | permanent   | heap          | 5746 MB  |
 public | moon1         | table | demo     | permanent   | columnstore   | 0 bytes  |
 public | moon2         | table | demo     | permanent   | columnstore   | 0 bytes  |
(3 rows)
```

PG_MOONCAKE
COPY 文による転送

✓COPY FROM / TO 文による Parquet ファイルの入出力

✓ローカル・ストレージまたは S3 バケット上の Parquet ファイルを利用可能

✓実行例

```
postgres=# CREATE TABLE copy1(c1 TEXT, c2 TEXT) USING columnstore;
CREATE TABLE
postgres=# COPY copy1 FROM 's3://bucket1/copy1.parquet' ;
COPY 1000
postgres=# COPY copy1 TO '//tmp/copy1.parquet' ;
COPY 1000
```

PG_MOONCAKE
カタログ

✓列指向テーブルを参照できるカタログ

✓パブリック・クラウド上の S3 バケットを参照するシークレットを保存するカタログ

オプション	内容	備考
mooncake.cloud_secrets	secret テーブルを参照するビュー	
mooncake.columnstore_tables	secrets テーブルを参照するビュー	
mooncake.data_files	Parquet ファイル一覧	SUPERUSER のみ参照可
mooncake.secrets	Secret 情報の保存 （mooncake.create_secret 関数）	SUPERUSERのみ参照可
mooncake.tables	テーブルと保存ディレクトリー覧	SUPERUSERのみ参照可

PG_MOONCAKE
オプションの指定

✓オプション（mooncake 関連）

パラメーター名	デフォルト	説明	備考
mooncake.allow_local_tables	on	ローカル・テーブルの有効化	
mooncake.enable_local_cache	on	ローカル・キャッシュの有効化	
mooncake.default_bucket	''	デフォルト S3 バケット名	

PG_MOONCAKE
オプションの指定

✓オプション（DuckDB 関連）

パラメーター名	デフォルト	説明
duckdb.force_execution	false	実行ターゲットを強制選択
duckdb.enable_external_access	true	外部アクセスを許可
duckdb.allow_unsigned_extensions	true	外部エクステンションの実行許可
duckdb.max_memory	4GB	最大メモリー使用量
duckdb.memory_limit	4GB	最大メモリー使用量
duckdb.disabled_filesystems	LocalFileSystem	特定のファイルシステムのアクセス禁止
duckdb.motherduck_postgres_database	postgres	MotherDuck に保存するデータベース
duckdb.threads	-1	スレッド数
duckdb.max_threads_per_postgres_scan	1	？？？
duckdb.motherduck_enabled	auto	MotherDuck 有効化
duckdb.postgres_role	''	？？？
duckdb.motherduck_token	''	MotherDuck アクセストークン

PG_MOONCAKE
データ保存（データファイル）

✓データファイルの保存

✓「${PGDATA}/mooncake_local_tables/mooncake_{DBNAME}_{TABLENAME}_{OID} 」ディレクトリ

✓ファイル名は「ランダム UUID」から構成　例「ffdef017-66e1-4148-a33d-af0a84a1416d.parquet」

✓データファイル名一覧例

```
$ cd ${PGDATA}/mooncake_local_tables/mooncake_postgres_moon1_24737
$ ls -1t
_delta_log
e8ffeda5-8d37-438f-8335-2c117c1a63bf.parquet
4de4f8df-17dc-456e-ab92-2c50ca8714ef.parquet
3aaeea14-c9df-4e6d-9ec0-5033eebb5695.parquet
92950bef-cf1e-4468-ba92-aaddba5bee2b.parquet
09cc9b58-3d31-4f58-839f-9e7a54a096f0.parquet
95b6e046-3c3a-4710-a8d3-1c8d74e6ef4d.parquet
…
```

PG_MOONCAKE
データ保存（トランザクション情報）

✓トランザクション情報の保存
- ✓「${テーブル保存ディレクトリ}/_delta_log」ディレクトリ
- ✓トランザクションのコミット単位でファイルが増える

✓ファイル一覧例

```
$ ls -1 _delta_log/
00000000000000000000.json
00000000000000000001.json
00000000000000000002.json
```

✓1 レコード INSERT した時の JSON ファイル例

```
{"add":{"path":"1a141fc6-89a2-4d1a-a789-8c3262b66c7b.parquet","partitionValues":{},
   "size":288,"modificationTime":0,"dataChange":true,"stats":null,"tags":null,"deletionVector":null,
   "baseRowId":null,"defaultRowCommitVersion":null,"clusteringProvider":null}}
{"commitInfo":{"timestamp":1732018398277,"operation":"WRITE",
   "operationParameters":{"mode":"Append"},"clientVersion":"delta-rs.0.21.0"}}
```

PG_MOONCAKE
データ保存の単位

✓更新 DML の単位でデータファイルが生成される（≠トランザクション）

Tx#	操作	更新件数	データファイル	サイズ
1	CREATE TABLE	-	-	-
2	INSERT INTO	1 億	ffdef017-66e1-4148-a33d-af0a84a1416d.parquet	400,533,812
3	INSERT INTO	1	ffdef017-66e1-4148-a33d-af0a84a1416d.parquet 1a141fc6-89a2-4d1a-a789-8c3262b66c7b.parquet	400,533,812 608
4	DELETE	100 万	ffdef017-66e1-4148-a33d-af0a84a1416d.parquet 1a141fc6-89a2-4d1a-a789-8c3262b66c7b.parquet 14b488ae-8156-4852-bad1-0ad0326c90c1.parquet	400,533,812 608 396,528,530
5	UPDATE	100 万	ffdef017-66e1-4148-a33d-af0a84a1416d.parquet 1a141fc6-89a2-4d1a-a789-8c3262b66c7b.parquet 14b488ae-8156-4852-bad1-0ad0326c90c1.parquet 1e50732f-70bc-4f43-a0b5-f3f1eeef655a.parquet	400,533,812 608 396,528,530 396,528,463

PG_MOONCAKE
ファイル形式

✓圧縮方法の指定

✓SNAPPY 圧縮がハードコード

✓columnstore_table.cpp の一部

```
class DataFileWriter {
public:
    DataFileWriter(ClientContext &context, FileSystem &fs, string file_name, vector<LogicalType> types,
                   vector<string> names, ChildFieldIDs field_ids)
        : collection(context, types, ColumnDataAllocatorType::HYBRID),
          writer(context, fs, std::move(file_name), std::move(types), std::move(names),
                 duckdb_parquet::format::CompressionCodec::SNAPPY /*codec*/, std::move(field_ids), {} /*kv_metadata*/,
                 {} /*encryption_config*/, 1.0 /*dictionary_compression_ratio_threshold*/, {} /*compression_level*/,
                 true /*debug_use_openssl*/) {
        collection.InitializeAppend(append_state);
    }
```

PG_MOONCAKE
実行計画

✓実行計画の表示はわかりにくい

```
postgres=> EXPLAIN ANALYZE SELECT COUNT(*) FROM moon1;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Custom Scan (DuckDBScan)  (cost=0.00..0.00 rows=0 width=0) (actual time=11.908..11.926 rows=1 …
   DuckDB Execution Plan:

┌─────────────────────────┐
│ Query Profiling Information │
└─────────────────────────┘

EXPLAIN ANALYZE SELECT count(*) AS count FROM pgmooncake.public.moon1

┌─────────────────────┐
│ Total Time: 0.0067s │
└─────────────────────┘

…
```

PG_MOONCAKE
検索性能

✓検索性能の確認
- ✓5 列 / 1 億タプルで検証
- ✓パラメーターはデフォルト
- ✓Linux のキャッシュクリア、PostgreSQL インスタンス再起動後に実行
- ✓Heap テーブルの検索はパラレルクエリー（Parallel Seq Scan／並列度 2）

✓検証結果

SQL	Heap (ms)	Columnar (ms)	備考
SELECT c2	13,797.2	190.3	EXPLAIN ANALYZE 結果
SELECT COUNT(c1)	4,171.8	866.5	一意列
SELECT COUNT(c2)	4,199.0	236.2	単一値列
SELECT SUM(c1)	4,149.0	388.7	合計

PG_MOONCAKE
検索性能

✓結合性能の確認
- ✓heap テーブル同士の Inner Join + COUNT
- ✓columnstore テーブル同士の Inner Join + COUNT
- ✓Mooncake の検索は非常に高速

✓検証結果

結合条件	Time (ms)	備考
Heap Only	104,745.0	Parallel Hash Join (2 Worker)
Columnar Only	7,926.4	Hash Join

PG_MOONCAKE
更新性能

✓更新性能の確認
- ✓単一トランザクション
- ✓INSERT 文、DELETE 文、UPDATE 文は遅くなる

✓検証結果

SQL	Heap (ms)	Columnar (ms)	備考
INSERT	236,782.0	547,976.6	INSERT SELECT 文による（1 億件）
UPDATE	3,165.6	21,862.8	1,000,000 件更新、Mooncake は全件検索
DELETE	496.9	21,025.4	1,000,000 件更新、Mooncake は全件検索
TRUNCATE	1,022.8	7.2	

✓総評
- ✓更新が遅くなる、制約が無い等の制限は多いが、検索性能は非常に高い

ACCESS METHOD 比較

ACCESS METHOD 比較
エクステンションの機能比較

✓実行可能な DML

SQL 文	Citus	Hydra	pg_mooncake	備考
VACUUM	○	○	×	
ANALYZE	○	○	×	
SELECT	○	○	○	
SELECT FOR UPDATE	×	○	×	
INSERT	○	○	○	
UPDATE	×	○	○	
DELETE	×	○	○	
COPY	○	○	○	Parquet ファイル対応可
TRUNCATE	○	○	○	
MERGE	×	○	×	

ACCESS METHOD 比較
エクステンションの機能比較

✓機能比較

機能	Citus	Hydra	pg_mooncake	備考
圧縮効果	高	高	中	
インデックス	○	○	×	
制約	○	○	×	
トランザクション	○	○	○	
物理フォーマット	独自	独自	Parquet	
S3 バケット保存	×	×	○	
テーブルサイズ確認	○	○	×	
圧縮メソッド指定	○	○	×	
パラレルクエリー	×	○	×	
マネージド環境	Azure	Hydra Cloud	Neon	

ACCESS METHOD 比較
エクステンションの性能比較

✓性能比較

機能	Citus	Hydra	pg_mooncake	備考
全件検索	低	中	高	
部分検索（索引）	中	中	-	
更新	-	低	中	

ACCESS METHOD 比較
その他の比較対象

✓その他

名前	アクセス方法	URL
pg_analytics	Foreign Data Wrapper	https://github.com/paradedb/pg_analytics
pg_duckdb	Table Access Method	https://github.com/duckdb/pg_duckdb
pg_parquet	COPY hook	https://github.com/CrunchyData/pg_parquet/

ALLOYDB

ALLOYDB
概要

- ✓Google Cloud が提供する PostgreSQL 互換データベース
  - ✓オンプレミス版は AlloyDB Omni
  - ✓列指向エンジンを持つ
- ✓Alloy DB の列指向エンジン
  - ✓デフォルトでは無効
  - ✓デフォルトでは列指向データはメモリー上に展開される
  - ✓列指向データは自動選択され、定期的に構築される
  - ✓オプティマイザは自動的に検索元データを選択する

ALLOYDB
オプションの指定

✓主なオプション（google_columnar_engine スキーマ）

パラメーター名	デフォルト	説明	備考
enabled	off	列指向エンジンの有効化	
relations	''	列指向メモリーへ投入対象テーブル	
enable_auto_columnarization	on	自動	
auto_columnarization_schedule	''	列指向メモリーへ投入間隔	デフォルト 1 時間
enable_columnar_scan	on	列指向データにアクセスするか	
memory_size_in_mb	30%	列指向データ用メモリー領域（MB）	
refresh_threshold_percentage	50	列ストアのデータ更新しきい値%	
refresh_threshold_scan_count	5	列ストアのデータ参照回数閾値	

ALLOYDB
列指向メモリー

✓メモリー設定 GUC 群
- ✓列指向データは共有メモリー領域に格納されると思われる。
- ✓memory_size_in_mb を設定しない場合は「30% of instance memory」
- ✓8 GB メモリーのホストから Docker で起動した場合の設定値

パラメーター名	自動設定値	説明	備考
shared_buffers	6370MB	OS メモリーの 75 % を自動設定	
shared_memory_size	1737MB	実際に確保されている共有メモリー	
shared_buffers_init_alloc_size	1GB	初期確保される共有メモリー	
shared_memory_expand_ratio	0.25	共有メモリーを拡張する割合	
shared_memory_overflow_buffer	100kB	オーバーフローバッファのサイズ	
shared_buffers_active	68352	アクティブな共有メモリー	8kB単位
shared_memory_size_in_huge_pages	869	コンテナにもホストでも HugePages設定はしていないが？	

ALLOYDB
列指向メモリー

✓列指向メモリーの内訳

```
postgres=> SELECT * FROM g_columnar_memory_usage;
     memory_name      | memory_total | memory_available | memory_available_percentage | reserved_memory
----------------------+--------------+------------------+-----------------------------+-----------------
 main_pool            |   1073741640 |       1073741392 |                          99 |               0
 recommendation_pool  |     14568224 |         14567232 |                          99 |         3059327
 work_pool            |     57671496 |         57592888 |                          99 |         5190434
 column_set           |      1048576 |          1048472 |                          99 |               0
(4 rows)
```

ALLOYDB
列指向メモリーへの投入

✓列指向メモリーへの投入は自動的に行うこともできる

✓自動選択されたテーブルを列指向メモリーへ追加

```
postgres=> SELECT * FROM g_columnar_recommended_columns;
 database_name | schema_name | relation_name | column_name | estimated_size_in_bytes
---------------+-------------+---------------+-------------+-------------------------
 postgres      | public      | data1         | c1          |               960169115
(1 row)

postgres=> SELECT * FROM google_columnar_engine_recommend();
 total_size_in_mb |          columns
------------------+---------------------------
             1024 | postgres.public.data1(c1)
(1 row)

postgres=> SELECT relation_name, column_name, status, size_in_bytes FROM g_columnar_columns;
 relation_name | column_name | status | size_in_bytes
---------------+-------------+--------+---------------
 data1         | c1          | Usable |     800139680
```

ALLOYDB
列指向メモリーへの投入

✓メモリーに乗らない場合

```
postgres=> SELECT google_columnar_engine_add('data1', 'c1');
WARNING:  Skipping the already populated column c1
WARNING:  All specified columns are present in the columnar cache.
 google_columnar_engine_add
----------------------------
                          0
(1 row)
```

ALLOYDB
実行計画

✓Custom Scan と、元のテーブルも参照して Append 操作を行う

```
postgres=> EXPLAIN SELECT COUNT(c2) FROM data1;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=13517.53..13517.54 rows=1 width=8)
   ->  Gather  (cost=13517.31..13517.52 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=12517.31..12517.32 rows=1 width=8)
               ->  Parallel Append  (cost=20.00..87505.16 rows=4998924 width=6)
                     ->  Parallel Custom Scan (columnar scan) on data1  (cost=20.00..87501.16 rows=…
                           Columnar cache search mode: native
                     ->  Parallel Seq Scan on data1  (cost=0.00..4.00 rows=1 width=6)
```

ALLOYDB
検索性能

✓集約関数の使用
- ✓5 列 / 1 億タプルで検証
- ✓パラメーターはほぼデフォルト
- ✓列指向メモリーを使用した場合と使用しない場合を比較

SQL	Heap (ms)	Columnar (ms)	備考
SELECT COUNT(c1)	5,726.1	21.2	一意列
SELECT COUNT(c2)	5,855.3	8.2	単一値列
SELECT DISTINCT c2	5,430.7	5.6	

まとめ

まとめ

✓列指向テーブルにも多数の選択肢が提供されている
  ✓機能には大きな差がある
  ✓列指向テーブルは更新できる場合でも速度やストレージ容量に十分な配慮が必要

✓みんなちがって、みんないい

THANK YOU

Mail : noriyoshi.shinoda@hpe.com
X(Twitter) : @nori_shinoda
Qiita : @plusultra
GitHub : @nori-shinoda