



Postgres Analytics Accelerator - PGAA

リリース *1.8.0*

EnterpriseDB Corporation

2026 年 04 月 15 日

目次

第 1 章	Get started	3
1.1	PGAA を使用する	3
1.2	PGD を使用した複製	3
1.3	パフォーマンスと最適化	4
1.4	リファレンス	4
第 2 章	Analytics Accelerator 1.8 release notes	5
2.1	PGAA 1.8	5
2.1.1	機能	5
2.1.2	バグ修正	6
2.1.3	インフラストラクチャとその他の変更	6
2.2	PGAA 1.7	6
2.2.1	機能	6
2.2.2	バグ修正	7
2.2.3	インフラストラクチャとその他の変更	7
2.2.4	非推奨	7
2.3	PGAA 1.6	7
2.3.1	ハイライト	7
2.3.2	機能	7
2.3.3	パフォーマンスとクエリの最適化	7
2.3.4	レプリケーションとデータ型	8
2.3.5	カタログとストレージ管理	8
2.3.6	バグ修正	9
2.3.7	インフラストラクチャとその他の変更	10
2.3.8	非推奨	10
第 3 章	Analytics Accelerator	11
3.1	Analytics Accelerator quickstart guide	11
3.1.1	前提条件	11
3.1.2	Postgres インスタンスへのパッケージのダウンロードとインストール	12
3.1.3	保存場所の構成	12
3.1.4	分析テーブルの作成	13
3.1.5	分析クエリーの実行	13
3.1.6	クエリプランの分析	14
3.1.7	結論	15
3.2	Analytics Accelerator architecture	16
3.2.1	アーキテクチャの概要	16
3.2.2	コアコンポーネント	16
3.2.3	クエリ実行モード	17
3.3	Concepts	17

3.3.1	ストレージとテーブルフォーマット	17
3.3.2	カタログ	18
3.3.3	クエリーの実行	18
3.3.4	キャッシング	18
3.4	Known issues	19
3.5	Analytics Accelerator compatibility	20
3.5.1	Postgres の互換性	20
3.5.2	プラットフォームの互換性	20
3.5.3	スパーク	20
3.5.4	PGD	20
3.5.5	ハイブリッドマネージャー	20
3.5.6	サポートされているストレージ形式とプロバイダー	20
第 4 章	Installing Analytics Accelerator	23
4.1	パッケージのダウンロードとインストール	23
4.2	拡張機能の有効化	23
4.3	インストールの検証	24
第 5 章	Reading tables in object storage	25
5.1	オブジェクトストレージから読み取るように Postgres を構成する	26
5.1.1	保存場所の作成	26
5.1.2	テーブルの Postgres へのマッピング	26
5.1.3	加速の照会と検証	27
5.2	制限事項	27
第 6 章	Integrating with Iceberg catalogs	29
6.1	カタログ接続の追加	29
6.1.1	REST カタログ	30
6.1.2	AWS S3 テーブル	30
6.1.3	カタログ構成の検証	31
6.2	カタログテーブルの登録	31
6.2.1	カタログの添付	31
6.2.2	カタログのインポート	32
6.2.3	カタログ管理テーブルの作成	32
第 7 章	Writing to object storage	35
7.1	スタンドアロンの保存場所を使用する	35
7.1.1	保存場所の作成	36
7.1.2	CTAS を使用したテーブルの作成	36
7.2	Iceberg カタログを使用する	37
7.2.1	カタログ接続の構成	37
7.2.2	カタログ管理 CTAS テーブルの作成	37
7.3	CTAS の進行状況のモニタリング	38
7.4	CTAS の制限	38
第 8 章	Replicating with PGD	39
8.1	テーブルタイプを理解する	39
8.2	階層化ストレージ戦略	40

8.2.1	階層テーブルの実装	40
8.2.2	アナリティクスへの複製	40
8.2.3	アナリティクスへのオフロード	40
8.3	メタデータ管理	40
8.3.1	Getting started	41
8.3.2	Implementing tiered tables	43
8.3.3	Replicating to analytics	48
8.3.4	Offloading to analytics	50
第 9 章	Accelerating with Spark	53
9.1	エグゼキューターエンジンの選択	53
9.2	分散実行オプション	53
9.2.1	Distributed Spark execution	54
9.2.2	GPU-accelerated Spark with NVIDIA RAPIDS	58
第 10 章	Monitoring and maintaining analytical tables	65
10.1	テーブル統計の分析	65
10.2	テーブルパフォーマンスの最適化	66
10.2.1	pgaa.launch_task を使用したテーブルメンテナンスの実行	66
10.2.2	高度なメンテナンス	68
第 11 章	Optimizing query performance	71
11.1	スキャンタイプを理解する	71
11.2	クエリプランの分析	71
11.3	DirectScan フォールバックの管理	72
11.4	コンピューティングブッシュダウンの構成	72
第 12 章	Reference	73
12.1	Configuration parameters	73
12.1.1	エグゼキューターエンジン	73
12.1.2	クエリー動作	74
12.1.3	Seafowl	76
12.1.4	カタログの同期	78
12.1.5	キャッシング	79
12.1.6	Postgres DistributedPGD 統合	80
12.1.7	DataFusion から継承された構成	80
12.1.8	エグゼキューターエンジン	80
12.1.9	クエリー動作	81
12.1.10	Seafowl	82
12.1.11	カタログの同期	83
12.1.12	キャッシング	83
12.1.13	DataFusion から継承された構成	83
12.2	Functions	83
12.2.1	テーブルファンクション	84
12.2.2	カタログファンクション	86
12.2.3	Spark ファンクション	93
12.2.4	バックグラウンドタスクファンクション	95
12.2.5	Postgres DistributedPGD 統合機能	98

12.2.6	ユーティリティファンクション	104
12.3	Table options	105
12.3.1	PGAA テーブルオプション	105
12.3.2	PGD テーブルオプション	106
12.3.3	アクセス方法の管理	107
12.4	Data types and definitions	107
12.4.1	データ型	108
12.4.2	管理用の列挙型とステータス	109
12.5	Benchmark datasets	110

Postgres Analytics AcceleratorPGAA は、Postgres が Delta Lake、Apache Iceberg、Parquet のようなオープンテーブル形式で保存されている大規模なデータを照会できるようにする高性能拡張機能です。PGAA は、大量の分析クエリをベクトル化された実行エンジンにオフロードすることにより、運用データベースとデータレイク間のギャップを埋めます。

第1章 Get started

- *Analytics Accelerator compatibility* サポートされている PostgreSQL バージョン、オペレーティングシステム、およびその他の要件を確認します。
- *Analytics Accelerator architecture* コアアーキテクチャとベクトル化されたエンジンがどのように動作するかを理解します。
- *Concepts* ベクトル化された実行、データレイク統合、および DirectScan の基本原理を理解します。
- *Analytics Accelerator quickstart guide* PGAA をインストールし、保存場所を作成し、サンプルベンチマークデータセットからテーブルを読み取ります。

1.1 PGAA を使用する

- *Installing Analytics Accelerator* 拡張機能をインストールし、Seafoam バックグラウンドワーカーを有効にするためのステップバイステップの手順。
- *Reading tables in object storage* S3、GCS、または Azure Blob Storage に直接接続し、PGFS ストレージの場所を使用して Parquet、Delta、または Iceberg ファイルを照会します。
- *Integrating with Iceberg catalogs* 外部の Iceberg REST カタログと統合して、テーブルメタデータを管理します。
- *Writing to object storage* CREATE TABLE AS SELECT CTAS を使用して、Postgres データをオブジェクトストアに最適化されたレイクハウス形式にエクスポートします。

1.2 PGD を使用した複製

- **階層テーブルの実装** PGD AutoPartition と PGAA を組み合わせて、自動化されたデータライフサイクルを作成します。Postgres テーブルに最近のデータを維持したまま、古いパーティションをオブジェクトストレージに移動します。
- **アナリティクスへの複製** 標準ヒープテーブルを HTAP テーブルに変換します。継続的論理レプリケーションを使用して、データレイク内のトランザクションデータのリアルタイム分析コピーを維持します。
- **アナリティクスへのオフロード** HTAP テーブル全体をコールド層に手動で移動し、ローカルデータを切り捨ててディスク領域をすぐに再利用することにより、サージカルストレージ管理を実行します。

1.3 パフォーマンスと最適化

- **Accelerate with Spark** 大規模なデータセットと複雑な分散結合を Spark Connect を介してリモート Spark クラスターにオフロードします。PGAA は、パフォーマンス要件に応じて 2 つの統合モードを提供します。
 - *Distributed Spark execution* 高コンカレンシーの分析クエリーと分散処理にリモート Spark クラスターを活用します。
 - *GPU-accelerated Spark with NVIDIA RAPIDS* GPU アクセラレーションを活用するために、Apache Spark の NVIDIA RAPIDS アクセラレータと統合します。
- **Monitoring and maintaining analytical tables** ストレージ使用率の監査、テーブルの状態の監視、および PGAA 管理テーブルのテーブルメンテナンスタスクを実行します。
- **Optimizing query performance** DirectScan の実行の管理、コンピューティングプッシュダウンの構成、およびパスフォールバックのトラブルシューティングにより、クエリ速度を最大化します。

1.4 リファレンス

- **Configuration parameters** PGAA 拡張機能の動作は、グランドユニファイド構成 GUC 変数によって制御されます。これらのパラメーターを使用すると、エグゼキューターを切り替え、パフォーマンスの最適化を有効にし、セキュリティ資格情報を管理できます。
- **Functions** PGAA は、新しいテーブルのマッピング、ストレージの状態の監視、メンテナンスバックグラウンドジョブの起動などの管理タスクのための SQL ファンクションスイートを導入しています。
- **Table options** 分析テーブルをマッピングまたは作成するときに、特定のオプションを使用して、オブジェクトストアからデータを読み取る方法、またはオブジェクトストアにデータを書き込む方法を定義できます。
- **Data types and definitions** PGAA は、ネイティブの Postgres データ型をデータレイクの最適化されたカラムナー形式にマップします。
- **Benchmark datasets** 分析データセットの事前構成されたスキーマとデータのロード指示にアクセスして、パフォーマンスのベースラインを設定します。

第2章 Analytics Accelerator 1.8 release notes

Postgres Analytics Accelerator PGAA 1.8 ドキュメントでは、Analytics Accelerator の最新バージョンについて説明しています。サポートされているすべてのバージョンのリリースノートも以下に含まれています。

Version	Release date
<i>Analytics Accelerator 1.8 release notes</i>	9 April 2026
<i>PGAA 1.7</i>	5 March 2026
<i>PGAA 1.6</i>	4 February 2026

2.1 PGAA 1.8

リリース日 2026 年 4 月 9 日

Analytics Accelerator 1.8 には、次の機能強化とバグ修正が含まれています。

2.1.1 機能

- ストレージ場所ベースのレプリケーションのためのパーティションテーブルユニオンビュー以前は REST カタログでのみサポートされていたパーティションテーブルのユニオンビューが、ファイルベースのストレージ場所カタログでもサポートされるようになりました。
- CTAS 操作の進行状況レポート CTAS ジョブは、100 万行または 10 分ごとに NOTICE メッセージを出力するようになり、長期実行クエリステータスを可視化します。
- EPAS のクエリプッシュダウンの改善一般的な EPAS 固有の機能が Seafoal エンジンによって認識されるようになり、Oracle 互換の構文のために以前は CompatScan にフォールバックしたクエリの DirectScan アクセラレーションが有効になります。
- サポートされていないリストタイプのレプリケーションの向上いくつかの追加の Postgres 配列タイプがデータレイクにネイティブにレプリケートされるようになりました。その他のサポートされていない配列型は、単一のテキスト値に文字列化されるのではなく、文字列の配列として複製されます。
- `pgaa.seafoal_version()` ファンクションコミットハッシュ、ビルド時間、最適化レベル、有効な機能を含む詳細な Seafoal エンジンバージョン情報を返す新しいファンクション。

2.1.2 バグ修正

- ゼロ行を返す CTAS クエリが空のテーブルを正しく作成しなかった問題を修正しました。
- 行レベルの削除を使用する Iceberg テーブルのクエリの正しさに影響を与える等価削除述語プッシュダウンのバグを修正しました。
- `pg_dump` が PGAA 管理テーブルのコンテンツを誤ってダンプする問題を修正しました。
- 関連サブクエリを含む特定のクエリで誤った結果を生成する可能性のある過度に積極的なクエリプッシュダウンを修正しました。
- PGAA アクセス方法の DDL を使用してテーブルを作成するときに、PGAA はテーブルフォーマットを正しく推論するようになりました。

2.1.3 インフラストラクチャとその他の変更

- アップストリーム問題 DF-19049 および DF-21292 のバックポートを含む DataFusion 52 へのアップグレード。
- 内部バッチ追加操作を最適化することにより、PGD の初期ロードと CTAS パフォーマンスが向上しました。
- `pgaa._bench_replication()` を拡張して、ストレージの場所ベースのベンチマークに加えて、カタログベースのレプリケーションベンチマークをサポートしました。

2.2 PGAA 1.7

リリース日 2026 年 3 月 5 日 Analytics Accelerator 1.7 には、次の機能強化とバグ修正が含まれています。

2.2.1 機能

- **Iceberg コンパクションメンテナンス** `pgaa.launch_task()` ファンクションを介した Iceberg テーブルのコンパクションメンテナンス操作のサポートを追加しました。このワークフローは、PGAA バックグラウンドワーカーによってネイティブに処理され、コンパクションを実行するための外部 Spark Connect 統合に関する以前の要件が削除されます。これは、ネイティブ PGAA テーブルと PGD でレプリケートされた標準ヒープテーブルの両方に適用されます。
- **PGD レプリケーションパフォーマンス** Seafoal ベースの同期ライタを PGAA エンジンに移行することにより、PGD へのレプリケーション TPS の 2~2.5 倍の増加を達成しました。
- **メモリ管理** `pgaa.max_in_memory_mb` 構成パラメーターを導入して、サイズ関連のライタフラッシュ基準をきめ細かく制御できます。
- **ストレージ認証** 自動資格情報販売をサポートしていないストレージバックエンドの静的資格情報のサポートを有効にしました。

2.2.2 バグ修正

- 残留トランザクションロケーションが永続的 LSN のプロモーションを妨げる問題を解決しました。
- DirectScan エンジン内の日付と期間の減算中に発生するエラーを修正しました。

2.2.3 インフラストラクチャとその他の変更

- `pgaa._bench_replication` SQL ファンクションからアクセスできる、合成 `pgbench` のようなレプリケーションベンチマークを追加しました。
- エンドツーエンドのベンチマークスクリプトを単一の統合ユーティリティ `pgaabench.sh` にマージします。

2.2.4 非推奨

- ファンクション `pgaa.execute_compaction()` は非推奨であり、将来のリリースでは削除される予定です。テーブルのメンテナンスを実行するには、`pgaa.launch_task()` を使用します。

2.3 PGAA 1.6

リリース日 2026 年 2 月 4 日

Analytics Accelerator 1.6 には、次の機能強化とバグ修正が含まれ、バージョン 1.3.1 以降のすべての変更を反映しています。

2.3.1 ハイライト

- **GPU アクセラレーション分析** Spark RAPIDS プラグインのサポートにより、Spark Connect エグゼキューターを使用するときに大量の分析ワークロードに GPU アクセラレーションを活用できます。
- **ネイティブ Azure サポート** PGAA は、Azure Blob Storage および Azure Data Lake Storage Gen2ADLS を公式にサポートするようになりました。これにより、分析ワークロードのプライマリストレージ場所として Azure ベースのデータレイクを活用できます。
- **Parquet のサポートの拡張** Parquet ストレージ形式が、`file://` および `s3://` プロトコルの両方を使用する PGFS ロケーションでサポートされるようになりました。
- **高度なクエリプッシュダウンベクトル化エンジン**に直接ジョインとグループバイ集計をオフロードするための新しいサポートにより、CompatScan の大幅なパフォーマンスの向上。

2.3.2 機能

2.3.3 パフォーマンスとクエリの最適化

- Iceberg ストレージ形式の一般的な最適化を追加しました。
- `analytics_storage_location` が設定されている場合、HTAP テーブルの DirectScan を有効にしました。

- キャッシュオブジェクトストアは、ファイルの読み取りエラーが発生するとすぐにエントリを無効にするようになりましたが、ファイルの書き込みが失敗した場合にメモリ内エントリを無効にしません。
- DataFusion をバージョン 51 に更新して、安定性とクエリパフォーマンスを向上させました。
- NOT LIKE および NOT ILIKE 演算子のネイティブサポートを追加しました。

2.3.4 レプリケーションとデータ型

- レプリケーション中の Postgres UUID タイプのサポートを追加しました。
- PGD 管理を簡単にするために次のラッパー関クションを追加しました。
`pgaa.enable_analytics_replication`、`pgaa.disable_analytics_replication`、`pgaa.convert_to_analytics`、`pgaa.restore_from_analytics`、および `pgaa.convert_to_tiered_table`。
- BIGINT より小さい整数のような型はすべて INTEGER に強制されるようになりました。
- デフォルトの `pgaa.max_replication_lag_s` および `pgaa.flush_task_interval_s` を 5 秒に更新して、データの更新の応答性を高めました。
- BYTEA レプリケーションとスキーマ推論のサポートを追加しました。
- BPCHAR および UUID 配列の特定のサポートを含む、Postgres 配列タイプをネイティブレイクハウスタ
イプにレプリケートするサポートを導入しました。
- サポートされていないタイプを TEXT に変換して、レプリケーションの失敗を防ぐフォールバックメ
カニズムを追加しました。
- オプション `pgd.purge_analytics_target` を `pgaa.enable_analytics_replication` および `pgaa.
convert_to_tiered_table` 関クションに統合して、複製する前に既存の分析をページしました。

2.3.5 カタログとストレージ管理

- Iceberg REST 接続が OAuth2 認証をサポートするようになりました。
- `pgaa.execute_compaction` を追加して、Spark Connect 実行エンジンを使用して分析 Iceberg テー
ブルのパフォーマンスとストレージ効率を向上させます。
- 環境管理のためのヘルパーコマンドを追加しました。テーブルサイズとレプリケーションステータ
スをモニタするための
 - `pgaa.list_analytics_tables()`
 - `pgaa.list_tiered_tables()` すべての階層/パーティションテーブルを表示します。
 - `pgaa.pgaa_version()` 詳細なバージョン情報を取得します。
- 接続と構成を検証するための `pgaa.test_storage_location()` および `pgaa.test_catalog()` フ
ァンクションを追加しました。
- `file://` および `s3://` プロトコルを使用して PGFS 保存場所の Parquet フォーマットのサポートを
追加しました。

- `cascade` パラメーターを `pgaa.detach_catalog()` に追加しました。これは、デタッチする前にカタログによって管理されているすべてのテーブルを削除します。
- テーブルが存在する場合に明示的な `cascade := true` パラメーターを必要とすることにより、`pgaa.delete_catalog()` ファンクションの安全性の向上を含み、偶発的なメタデータの損失を防ぎます。
- `pgaa.add_catalog()` および `pgaa.update_catalog()` ファンクションの自動カタログ接続チェックを追加しました。
- `pgaa.list_catalog_tables()` ファンクションを追加しました。これにより、最初にテーブルをローカルデータベースにインポートまたはアタッチせずに、カタログ内のすべてのテーブルとビューを探索できます。オプションの名前空間フィルタリングを使用して。
- `pgaa.drop_catalog_tables()` ファンクションを追加しました。これは、リモートの Iceberg カタログのテーブルに影響を与えることなく、特定の Iceberg カタログによって管理されているすべてのローカル Postgres テーブルとビュー定義を削除します。
- 名前空間フィルタリングを `pgaa.import_catalog()` に追加し、選択的なスキーマインポートが可能になりました。
- 明示的な値なしで指定された場合、`pgaa.tiered_table` や `pgaa.purge_data_if_exists` などの構成パラメーターのデフォルト値を `true` に変更しました。

2.3.6 バグ修正

- Iceberg REST カタログから読み取る際のオブジェクトストアキャッシュの問題を修正します。
- PGAA autoupgrade ワーカーが PGD DDL レプリケーションをブロックする可能性があるロックアップの問題を修正しました。
- フォロワーノードでのタプルの挿入を防止することにより、マルチノードセットアップでの CTAS 障害を解決しました。
- `CREATE TABLE AS SELECT` 操作での `NULL` 挿入中に発生したセグメンテーションフォールトを修正しました。
- CTAS 実行前に同期収集をリセットすることにより、ページ後のテーブルの再作成の問題を修正しました。
- レプリケーションライタでの誤解を招くバッチサイズの推定を修正しました。
- 追加または削除専用バッチの行スキャッシュロジック。
- 既存のテーブルまたは互換性のないビューがパーティションテーブルで正しくドロップされない問題を解決しました。
- DirectScan クエリ抽出におけるクエリ切り捨ての問題を修正しました。
- `CREATE TABLE AS SELECT CTAS` に障害が発生した後、DirectScan が動作しなくなる問題を解決しました。
- ビューの同期ライタのページを修正しました。
- `pgaa.tiered_table` オプションを変更するときの重複キー制約違反を解決しました。

- SingleTableFileCatalog テーブルを削除する際のオブジェクト削除スコープを修正しました。
- テーブルの切り捨てに関する問題を解決し、テーブル切り捨てがデータを空の Parquet ファイルの代わりに「ファイルなし」に正しく置き換えるようになりました。
- 大規模なテーブルで Out-of-MemoryOOM エラーを発生させた Create Table As SelectCTAS 操作中のメモリリークを解決しました。
- 内部テーブルマッピング操作中に発生するメモリリークを修正しました。
- パーティション列のない修飾子を使用すると、階層化テーブル DML が失敗する問題を修正しました。
- CTAS フックから不要なテーブルの存在警告を削除しました。
- tiered_table オプションを繰り返し設定したときにトリガーされる重複キーエラーを修正しました。

2.3.7 インフラストラクチャとその他の変更

- レプリケーションパフォーマンスの自動ベンチマークを追加しました。
- rustc を 1.92.0 に更新しました。
- DataFusion 50 にアップグレードされました。
- ABI バージョン 300 を導入し、pgaa_tabular_exists やオブジェクトパージなどの新機能を公開します。
- PGAA ABI v300 を介してデータをパージする機能を公開しました。

2.3.8 非推奨

- スタンドアロンのメタストアエージェントを削除しました。
- 6.1 より前の PGD バージョンのサポートを削除しました。
- BigAnimal サイドカー画像を削除しました。
- pgaa.get_all_analytics_table_settings ファンクションを削除しました。
- 1.3 より前のバージョンからのアップグレードパスは削除されました。古いバージョンのユーザーは、1.6.0 に移動する前に、最初に PGAA 1.3.1 にアップグレードする必要があります。

第3章 Analytics Accelerator

Postgres Analytics Accelerator PGAA は、Postgres を高性能分析エンジンに変換します。PGAA は、ベクトル化された実行とオープンテーブルフォーマットを統合することにより、Postgres エコシステムを離れることなく、データレイクで複雑な OLAP クエリを直接実行できます。

次のリソースを使用して、アクセラレータの基礎を理解し、運用ワークロード用に環境を準備します。

- *Analytics Accelerator quickstart guide* PGAA のインストール、保存場所を作成、サンプルベンチマークデータセットからテーブルを読み取るためのステップバイステップガイド。
- *Analytics Accelerator architecture* 分離されたコンピューティングとストレージモデルの詳細。PGAA 拡張機能が Seafoam ベクトル化エンジン、および S3、GCS、Azure Data Lake Storage などのオブジェクトストレージプロバイダーとどのように相互作用するかを説明します。
- *Concepts* ベクトル化された実行、列指向ストレージ、DirectScan と CompatScan 実行パスの違いなど、PGAA の基本原理を検討します。
- *Analytics Accelerator compatibility* サポートされている Postgres バージョン、オペレーティングシステム、およびクラウドプロバイダーの可用性の詳細なマトリックス。
- *Known issues* 現在の制限、サポートされていないケース、および現在対処しているアクティブなバグの透明なリスト。

3.1 Analytics Accelerator quickstart guide

このクイックスタートガイドでは、次のことを行います。

1. スタンドアロンのコミュニティ Postgres インスタンスに PGAA 拡張機能をインストールします。
2. オブジェクトストレージにサンプルベンチマークデータセットを指す保存場所を作成します。
3. 複雑な分析クエリを実行し、Seafoam エンジンが動作していることを確認します。
4. クエリプランの分析を実行します。

3.1.1 前提条件

- Postgres インスタンスを実行している環境。サポートされているプラットフォームの完全なリストについては、*Analytics Accelerator compatibility* を参照してください。
- EDB アクセストークン。

3.1.2 Postgres インスタンスへのパッケージのダウンロードとインストール

EDB リポジトリからパッケージをダウンロードします。

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
curl -sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/standard/
↳setup.deb.sh" | sudo -E bash
sudo apt-get install -y postgresql-<version>-pgaa
```

<version> は Postgres バージョンです。

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
curl -sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/standard/
↳setup.rpm.sh" | sudo -E bash
sudo dnf install -y postgresql-<version>-pgaa
```

<version> は Postgres バージョンです。

postgresql.conf ファイルを編集して pgaa を shared_preload_libraries に追加し、Postgres バックグラウンドワーカーを介した Seafowl クエリエンジンの自動管理と起動を有効にします。

```
shared_preload_libraries = pgaa
pgaa.autostart_seafowl = on
```

変更を保存した後、Postgres サービスを再起動します。再起動したら、Postgres インスタンスにログインし、PGAA 拡張機能を作成します。

```
CREATE EXTENSION IF NOT EXISTS pgaa CASCADE;
```

CASCADE を使用すると、必要な依存関係 PGFS など作成されます。\\dx を使用して、拡張機能がインストールされていることを確認し、バージョンを確認します。

3.1.3 保存場所の構成

TPC-H ベンチマークデータを含むパブリック S3 バケットを指す保存場所を作成します。

```
SELECT pgfs.create_storage_location(
    quickstart-sample-data,
    s3://beacon-analytics-demo-data-us-east-1-prod,
    {"skip_signature": "true", "region": "us-east-1"}
);
```

保存場所が作成されたことを確認します。

```
SELECT pgfs.list_storage_locations();
```

3.1.4 分析テーブルの作成

次の分析テーブルを作成します。これらのテーブルは、サンプル *Benchmark datasets* の Delta Lake ファイルに直接マッピングします。

```
CREATE TABLE supplier () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
→data, pgaa.path = tpch_sf_1/supplier, pgaa.format = delta);
CREATE TABLE lineitem () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
→data, pgaa.path = tpch_sf_1/lineitem, pgaa.format = delta);
CREATE TABLE orders () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
→data, pgaa.path = tpch_sf_1/orders, pgaa.format = delta);
CREATE TABLE nation () USING PGAA WITH (pgaa.storage_location = quickstart-sample-
→data, pgaa.path = tpch_sf_1/nation, pgaa.format = delta);
```

3.1.5 分析クエリーの実行

このクエリは、マルチサプライヤーの注文の遅延の原因となったサウジアラビアのサプライヤーを特定します。これは、大規模な結合と複数のサブクエリーを含む大量の分析タスクです。

```
SELECT
    s_name,
    COUNT(*) AS numwait
FROM
    supplier,
    lineitem l1,
    orders,
    nation
WHERE
    s_suppkey = l1.l_suppkey
    AND o_orderkey = l1.l_orderkey
    AND o_orderstatus = F
    AND l1.l_receiptdate > l1.l_commitdate
    AND EXISTS (
        SELECT
        - FROM
            lineitem l2
        WHERE
            l2.l_orderkey = l1.l_orderkey
            AND l2.l_suppkey <> l1.l_suppkey
    )
    AND NOT EXISTS (
        SELECT
        - FROM
            lineitem l3
        WHERE
            l3.l_orderkey = l1.l_orderkey
```

(次のページに続く)

(前のページからの続き)

```

        AND l3.l_suppkey <> l1.l_suppkey
        AND l3.l_receiptdate > l3.l_commitdate
    )
    AND s_nationkey = n_nationkey
    AND n_name = SAUDI ARABIA
GROUP BY
    s_name
ORDER BY
    numwait DESC,
    s_name
LIMIT 100;

```

3.1.6 クエリプランの分析

PGAA がこのクエリーをどのように高速化するかを確認するには、上記のステートメントに **EXPLAIN** を先頭に追加し、出力を検査します。

QUERY PLAN

→

→

→

SeafowlDirectScan: Logical Plan

Limit: skip=0, fetch=100

Sort: numwait DESC NULLS FIRST, supplier.s_name ASC NULLS LAST

Projection: supplier.s_name, count(Int64(1)) AS count(*) AS numwait

Aggregate: groupBy=[[supplier.s_name]], aggr=[[count(Int64(1))]]

Filter: supplier.s_suppkey = l1.l_suppkey AND orders.o_orderkey = l1.l_orderkey AND orders.o_orderstatus = Utf8("F") AND l1.l_receiptdate > l1.l_commitdate AND EXISTS (<subquery>) AND NOT EXISTS (<subquery>) AND supplier.s_nationkey = nation.n_nationkey AND nation.n_name = Utf8("SAUDI ARABIA")

Subquery:

Projection: l2.l_orderkey, l2.l_partkey, l2.l_suppkey, l2.l_linenum, l2.l_quantity, l2.l_extendedprice, l2.l_discount, l2.l_tax, l2.l_returnflag, l2.l_linestatus, l2.l_shipdate, l2.l_commitdate, l2.l_receiptdate, l2.l_shipinstruct, l2.l_shipmode, l2.l_comment

Filter: l2.l_orderkey = outer_ref(l1.l_orderkey) AND l2.l_suppkey != outer_ref(l1.l_suppkey)

SubqueryAlias: l2

TableScan: lineitem

Subquery:

Projection: l3.l_orderkey, l3.l_partkey, l3.l_suppkey, l3.l_linenum, l3.l_quantity, l3.l_extendedprice, l3.l_discount, l3.l_tax, l3.l

(次のページに続く)

(前のページからの続き)

```

→returnflag, l3.l_linestatus, l3.l_shipdate, l3.l_commitdate, l3.l_receiptdate, l3.l_
→shipinstruct, l3.l_shipmode, l3.l_comment
      Filter:  l3.l_orderkey = outer_ref(l1.l_orderkey) AND l3.l_suppkey !
→= outer_ref(l1.l_suppkey) AND l3.l_receiptdate > l3.l_commitdate
      SubqueryAlias:  l3
      TableScan:  lineitem
Cross Join:
Cross Join:
Cross Join:
      TableScan:  supplier
      SubqueryAlias:  l1
      TableScan:  lineitem
      TableScan:  orders
      TableScan:  nation
(24 rows)

```

プランの先頭に `SeafowlDirectScan` の存在は、PGAA がクエリ実行の完全な制御を取得し、標準の Postgres エグゼキュータをバイパスしてデータレイクに対して直接クエリを実行していることを示します。

何十億行も Postgres に引き込んで処理するのではなく、結合、フィルタ、集計を含む論理プラン全体が最適化された分析エンジンにプッシュダウンされます。

- サブクエリープッシュダウン `lineitem` テーブルの `EXISTS` および `NOT EXISTS` サブクエリーに注意してください。これらは Postgres によって一つ一つ実行されていません。代わりに、それらは Parquet ファイルに対して並列処理されるグローバル論理プランの一部です。
- ベクトル化されたジョイン `Cross Join` ノードと上位の `Filter` ノードは、エンジンがストレージレイヤーでハッシュジョインまたはネストされたループジョインを実行していることを示します。
- 述語プッシュダウン `n_name = Utf8("SAUDI ARABIA")` や `o_orderstatus = Utf8("F")` のようなフィルタは、初期スキャン中に適用されます。これは、関連するデータのみがレイクから読み取られ、I/O が大幅に削減されることを意味します。
- 最終的な集計とソート `Aggregate` および `Sort` は、パイプラインの最後で、最後の 100 行が Postgres クライアントに返される前に発生します。

3.1.7 結論

PGAA を正常にインストールし、クラウドベースのオブジェクトストレージに到達するように構成し、複雑な SQL ロジックサブクエリや結合を含むがネイティブにオフロードされていることを確認しました。

`Seafowl DirectScan` を活用することで、Postgres インスタンスがペタバイト規模のデータレイクへのゲートウェイとして機能し、トランザクションコンピューティングとは無関係にスケールする分析パフォーマンスを提供できるようになりました。

3.2 Analytics Accelerator architecture

このアーキテクチャページでは、アナリティクスアクセラレーター PGAA がトランザクション Postgres と最新のデータレイク間のギャップを埋める方法に関する高レベルの技術概要を提供します。

3.2.1 アーキテクチャの概要

EDB Analytics Accelerator は、Postgres の機能を拡張して、トランザクション OLTP および分析 OLAP ワークロード用の統合プラットフォームを作成します。システムは、3つの基本原則で動作します。弾力的なスケーリングのためのコンピューティングとストレージの分離、ホットおよびコールドデータ層にわたる透過的なクエリアクセス、およびオープンテーブルフォーマットとのネイティブ互換性。

3.2.2 コアコンポーネント

Postgres アナリティクスアクセラレーター PGAA

PGAA 拡張機能は Postgres クエリプランナーにフックして、分析エンジンにオフロードできるクエリを特定します。分析テーブルのメタデータを管理し、Postgres プロセスとクエリ実行プログラム間の通信を調整します。

PGFS ストレージの抽象化

Postgres ファイルシステム PGFS は、オブジェクトストレージの統合抽象化レイヤーとして機能し、個々のクラウドプロバイダープロトコルの複雑さをマスクしながら、Postgres が AWS S3、GCS、および Azure Blob Storage とシームレスに対話できるようにします。IAM ロールと HMAC キーを介して安全な認証を管理し、自動再試行と接続プーリングでネットワークの復元力を確保し、リモートファイルの場所とスキーマをキャッシュすることによりメタデータのオーバーヘッドを削減することにより、パフォーマンスと信頼性を最適化します。

Executor エンジン

Seafoal は、PGAA のデフォルトの DataFusion ベースのクエリエンジンであり、ステートレスアーキテクチャを利用して、レイクハウスノードは、ローカルデータのストレージまたは移動のオーバーヘッドなしで動的にコンピューティングをスケールアップまたはダウンできます。分析ワークロード専用設計されたベクトル化実行エンジンとして、標準の Postgres で使用される従来の行ごとのアプローチではなく、列型バッチでデータを処理することにより、CPU キャッシュ効率を最大化します。

Postgres インスタンスは、Apache Arrow Flight を使用して Seafoal エンジンと通信します。これは、大規模なデータ転送のために設計された高性能 RPC フレームワークです。Arrow を使用することにより、PGAA は最小限のシリアル化オーバーヘッドでデータベースとクエリエンジン間で列状データを移動できます。

Seafoal は高速 SQL のデフォルトの実行者ですが、PGAA は **Spark Connect** もサポートしています。これにより、組織は非常に複雑な ETL 変換に Apache Spark を活用しながら、使い慣れた Postgres ワイヤプロトコルを介してデータにアクセスできます。

3.2.3 クエリ実行モード

分析テーブルに対してクエリが実行されると、PGAA はクエリ構造とデータの場所に基づいて、最も効率的なスキャンモードを自動的に選択します。

DirectScan クエリーは完全に Seafoal エンジンにオフロードされます。Seafoal は、オブジェクトストレージからリモートファイルを直接読み取り、すべてのロジックフィルタリング、集計、および結合を処理し、最終的な結果セットのみを Postgres に返します。このモードは、ソースでベクトル化された実行を利用することにより、最大のパフォーマンスを提供します。

CompatScan このモードは、クエリがベクトル化エンジンでまだサポートされていない機能を必要とする場合、またはリモートレイクハウスデータをローカルの Postgres テーブルに結合する場合に使用されます。このシナリオでは、Seafoal はリモートデータで初期フィルタリングと投影を実行し、これらの最適化された結果を Postgres にストリーミングして、最終的な結合または複雑な処理をローカルで完了します。

3.3 Concepts

このコンセプトガイドでは、機能レイヤーごとに編成されたアナリティクスアクセラレーター PGAA アーキテクチャの技術的基盤を提供します。

3.3.1 ストレージとテーブルフォーマット

PGFS ストレージ抽象化 Postgres File System (PGFS) は、Postgres とオブジェクトストレージ S3、GCS、Azure 間の統一インターフェイスを提供するクラウドネイティブストレージレイヤーです。認証、ネットワークの復元力、およびリモートオブジェクトのローカルテーブル構造へのマッピングを処理します。

Delta Lake ACID トランザクションをデータレイクに提供するオープンソースのストレージフレームワーク。

トランザクションログ 中心的なメタデータレコード Delta Lake のデルタログまたは Iceberg のスナップショットなどテーブルへのすべての変更を追跡し、データの一貫性を保証します。

タイムトラベル 特定のタイムスタンプまたはログバージョンを参照することにより、以前のバージョンのデータを照会する機能。

スキーマの進化 データセット全体を書き換えずに、テーブルのスキーマを変更する機能たとえば列の追加。

Apache Iceberg 大規模な分析データセット用の高性能テーブル形式。

メタデータファイルレベルの統計 を追跡し、効率的なデータのスキップを可能にする多層マニフェストシステム。

スナップショット 特定の時点でのテーブルのバージョン。クエリは常に一貫したスナップショットに対して実行されます。

パーティションの進化 既存のデータを書き換えずに、テーブルのパーティション化戦略を変更する機能たとえば、日次から月次への変更。

Parquet カラムナストレージ PGAA によって使用される基になるファイル形式。データを行に保存する標準の Postgres とは異なり、Parquet は列にデータを保存します。

圧縮とエンコード カラムナストレージでは、同様のデータ型と一緒に保存され、ストレージのフットプリントと I/O オーバーヘッドが大幅に削減されるため、積極的な圧縮 Snappy または Zstandard のようなエンコードとデルタまたはディクショナリエンコーディングのようなエンコードが可能です。

3.3.2 カタログ

カタログは、テーブルメタデータの信頼できる中心的なソースとして機能します。これらにより、さまざまなコンピューティングエンジン Postgres、Spark、Trino がテーブルを検出、スキーマを解決し、共有データレイク全体でトランザクションの整合性を維持できます。

Iceberg REST PGAA が外部カタログと通信するために使用する標準化された API。これにより、EDB は、Lakekeeper、Tabular、Snowflake Polaris などの REST 仕様をサポートするカタログサービスと統合できます。

AWS S3 テーブル AWS の管理対象カタログおよびストレージサービス。PGAA は S3 テーブルと統合して、サーバーレスメタデータレイヤーを提供し、AWS エコシステム内の Iceberg テーブルの管理を簡素化します。

3.3.3 クエリーの実行

テーブルアクセスメソッド TAM アーキテクチャ PGAA がカスタムストレージエンジンをプラグインできるようにする内部 Postgres API。これにより、Postgres プランナーは、ネイティブローカルテーブルであるかのように、リモートオブジェクトストレージと対話できます。

DirectScan クエリが完全に分析実行エンジン Seafoal または Spark にオフロードされるパフォーマンスの最適化。これは、クエリが分析テーブルのみを参照する場合に発生し、ベクトル化エンジンが操作全体を処理し、最終結果をユーザーに返すことができます。

CompatScan ローカルの Postgres テーブルをリモート分析テーブルに結合するために、または Postgres のみサポートされているファンクションを処理するために使用されるハイブリッド実行モード。

部分プッシュダウン PGAA は、フィルタと投影をオブジェクトストレージレイヤーにプッシュダウンすることによりクエリを最適化し、必要な行のみを Postgres にストリーミングして、結合または複雑なローカル操作を完了します。

コストベースのプランニング Postgres オプティマイザーがレイクハウスメタデータからの統計行数やデータ分散などを使用して、ハッシュ結合またはネスト化の選択など、クエリを実行する最も効率的な方法を決定するプロセスループ。

Seafoal でのベクトル化実行 Seafoal エンジンのコア処理テクノロジー。データを行ごとに処理する代わりに、データをバッチベクターで処理します。これは、最新の CPU 命令 SIMD を活用して、マルチプルのデータポイントで数学的操作を同時に実行し、分析クエリを劇的に高速化します。

3.3.4 キャッシング

ネットワークベースのオブジェクトストレージに固有のレイテンシーを最小限に抑えるために、PGAA は多層キャッシング階層を実装します。これにより、繰り返されるクエリーをローカルストレージと同等の速度で実行できます。

オブジェクトストアキャッシュ このレイヤーは、レイクハウスノードのローカル SSD のデータレイクから取得した実際のデータブロック Parquet ファイルをキャッシュします。頻繁にアクセスされるデータをコンピューティングエンジンの「近く」に保つことにより、PGAA は、同じデータセグメントのリモートオブジェクトストアへの繰り返しの GET 要求のオーバーヘッドを回避します。

メタデータキャッシュ データレイクの構造情報ファイルマニフェストやスキーマ定義などのローカルの高速コピーを維持します。この方法により、エンジンはパーティションプルーニングとファイルレベルのフィ

ルタリングを瞬時に実行でき、リモートストレージプロバイダーを繰り返し照会する高いレイテンシーなしで、取得するデータを正確に特定できます。

メタデータキャッシュは、データレイクテーブルの構造ファイルマニフェスト、スキーマ定義、パーティションの場所を含むを保存します。Iceberg および Delta Lake テーブルは、何千もの個々のファイルで構成される場合があるため、この「マップ」をキャッシュすることにより、クエリプランナーはストレージプロバイダーに連絡せずに無関係なデータをほぼ瞬時にスキップできます。

テーブル統計キャッシュ `pgaa.lakehouse_table_stats()` ファンクションをサポートするために物理テーブルサイズのローカルレコードを維持します。クエリプランナーは、プランの精度を高めるために実行時にストレージプロバイダーから新しいメタデータを取得しますが、このキャッシュにより、ファンクションはライブリモートスキャンを必要とせずにデータレイクオブジェクトのスケールの即時の可視性を提供できます。

3.4 Known issues

これらは、このアナリティクスアクセラレーターリリースで特定された現在既知の問題と制限です。必要に応じて、これらの問題の影響を軽減するために役立つ回避策を含めました。これらの問題は積極的に追跡され、将来のリリースで解決される予定です。

- ファンクション `pgaa.launch_task()` は、Delta Lake z-order、vacuum、purge などの複数のメンテナンスオペレーションをサポートしていますが、現在 Iceberg テーブルの `compaction` オペレーションのみをサポートしています。他の Iceberg メンテナンスルーチンの場合は、`pgaa.spark_sql()` を使用します。
- ファンクション `pgaa.execute_compaction()` は、パラメーター `settings` をサポートしていません。`pgaa.launch_task()` を使用して、圧縮動作をより詳細に制御します。
- Apache Spark との統合は、
 - S3 互換オブジェクトストレージまたは共有 POSIX ファイルシステムの Parquet ファイルの読み取り専用クエリーでサポートされています。
 - Iceberg REST カタログの Iceberg テーブルの読み取り専用クエリー。
- Apache Spark による GPU アクセラレーションは、S3 互換オブジェクトストレージまたは共有 POSIX ファイルシステムの Parquet ファイルの読み取り専用クエリーでのみサポートされています。
- Iceberg Spark ライブラリーバージョン 1.9.2 には、Spark を使用して Iceberg テーブルから読み取る場合の既知の問題があります。特定の条件下では、同時実行中に等価削除がスキップされる場合があります。現在の回避策は、`spark-defaults.conf` ファイルで Spark アプリケーション設定 `spark.sql.iceberg.executor-cache.enabled` を無効にすることです。このキャッシュを無効にすると、すべての削除が正しく処理されるため、データの一貫性が保証されますが、大容量の読み取りワークロードの場合、パフォーマンスに影響を与える可能性があります。
- Amazon S3 テーブルとの統合は、PGD 統合機能階層化テーブル、レプリケーション、オフロードでは現在サポートしていません。PGD 管理のライフサイクルの場合、ストレージ場所または Iceberg REST 準拠のカタログを使用します。

3.5 Analytics Accelerator compatibility

3.5.1 Postgres の互換性

Postgres distribution	Postgres version
Postgres (PG)	16, 17, 18
EDB Postgres Extended Server (PGE)	16, 17, 18
EDB Postgres Advanced Server (EPAS)	16, 17, 18

3.5.2 プラットフォームの互換性

amd64 アーキテクチャでの Debian 11/12 または Ubuntu 22/24、および amd64 と arm64 の両方で RHEL 8/9。

3.5.3 スパーク

Apache Spark バージョン 3.4 以降。

3.5.4 PGD

PGD バージョン 6.1 以降。

3.5.5 ハイブリッドマネージャー

PGAA バージョン 1.8 は、Hybrid Manager Innovation Release 2026.4 に含まれています。

3.5.6 サポートされているストレージ形式とプロバイダー

Data Format	Operation	Object storage	Engine support
Iceberg	Read	Local, S3, GCP, Azure (via storage locations)	Seafoal
Iceberg	Read	Iceberg REST catalog	Seafoal Spark
Delta Lake	Read	Local, S3, GCP, Azure (via storage locations)	Seafoal
Parquet	Read	Local, S3, GCP, Azure (via storage locations)	Seafoal
Parquet	Read	Local, S3 (via storage locations)	Seafoal Spark Spark + RAPIDS

次のページに続く

表 2 – 前のページからの続き

Data Format	Operation	Object storage	Engine support
Iceberg	Write (via <i>CTAS</i>)	Local, S3, GCP, Azure (via storage locations)	Postgres engine via CompatScan
Iceberg	Write (via <i>CTAS</i>)	Iceberg REST catalog	Postgres engine via CompatScan
Delta Lake	Write (via <i>CTAS</i>)	Local, S3, GCP, Azure (via storage locations)	Postgres engine via CompatScan
Iceberg	Write (using PGD)	Local, S3, GCP, Azure (via storage locations)	Seafoal
Delta Lake	Write (using PGD)	Local, S3, GCP, Azure (via storage locations)	Seafoal
Iceberg	Write (using PGD)	Iceberg REST catalog	Postgres engine via CompatScan

第4章 Installing Analytics Accelerator

Postgres Analytics Accelerator PGAA を使用するには、Postgres インスタンスに拡張機能をインストールし、クエリエンジンのバックグラウンドワーカーを有効にする必要があります。サポートされているプラットフォームの完全なリストについては、*Analytics Accelerator compatibility* を参照してください。

4.1 パッケージのダウンロードとインストール

EDB リポジトリからパッケージをダウンロードします。

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=<your-subscription-plan>
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
→SUBSCRIPTION_PLAN/setup.deb.sh" | sudo -E bash
sudo apt-get install -y <postgresql-distribution>-pgaa
```

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
EDB_SUBSCRIPTION_PLAN=<your-subscription-plan>
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
→SUBSCRIPTION_PLAN/setup.rpm.sh" | sudo -E bash
sudo dnf install -y <postgresql-distribution>-pgaa
```

どこで

- \$EDB_SUBSCRIPTION_TOKEN は、EDB サブスクリプションの登録時に受け取ったトークンです。
- \$EDB_SUBSCRIPTION_PLAN は、standard PGE または Community Postgres の場合または enterprise EPAS の場合。
- <postgresql-distribution> は、Postgres ディストリビューション edb-postgrextended、edb-as、または postgres とバージョンを指定します。

4.2 拡張機能の有効化

PGAA を初期化するには、postgresql.conf ファイルを更新する必要があります。pgaa を shared_preload_libraries に追加すると、Postgres はバックグラウンドワーカーとして Seafoal クエリエンジンのライフサイクルを管理できます。

1. postgresql.conf を編集します。

```
shared_preload_libraries = pgaa
pgaa.autostart_seafowl = on
```

1. Postgres サービスを再起動して、新しいライブラリをロードします。
2. Postgres インスタンスにログインします。これが新規インストールであるか更新であるかに応じて、適切なコマンドを実行します。
 - 新規インストールの場合

```
CREATE EXTENSION IF NOT EXISTS pgaa CASCADE;
```

CASCADE を使用すると、Postgres ファイルシステム PGFS などの必要な依存関係が自動的にインストールされます。

- 既存のインストールをアップグレードする場合

```
ALTER EXTENSION pgaa UPDATE;
```

4.3 インストールの検証

\dx メタコマンドまたは version ファンクションを呼び出して、拡張機能がアクティブであることを確認します。

```
SELECT pgaa.pgaa_version();
```

インストールしたら、[保存場所の作成](#) してデータレイクへの PGAA アクセスを付与したり、Postgres *Integrating with Iceberg catalogs* を統合したりできます。

例

寄木細工ファイルをポイントします。

```
SELECT pgaa.test_storage_location (my-storage-location, false);
```

ファンクションは、成功すると NULL を返します。

第5章 Reading tables in object storage

Postgres アナリティクスアクセラレータ PGAA を使用すると、データファイルへの直接接続を確立することにより、Postgres の SQL インターフェイスを使用してオブジェクトストレージのテーブルから高パフォーマンスの分析クエリを実行できます。このモードでは、Postgres はストレージパスと直接対話してテーブル構造を解決し、クラウドバケットからマニフェストファイルを読み取って、基になるデータをスキャンおよび処理する方法を決定することによりメタデータコーディネーターとして機能します。

大規模なデータを照会するために、PGAA は構造化されたオープンテーブル形式と生ファイル形式の両方をサポートしています。これらの標準は、データを編成する方法と Postgres がデータを取得する方法を定義します。

- **Apache Iceberg** 階層メタデータツリーを使用してテーブルの状態を追跡する高性能テーブルフォーマット。これにより、以前のスナップショットへの「タイムトラベル」と、クエリのパーティション列を手動で管理する必要なく、Postgres がファイルをプルーニングできる「隠しパーティショニング」が可能になります。
- **Delta Lake** データレイクに信頼性をもたらすオープンソースストレージレイヤー。トランザクションログ `_delta_log` フォルダーを使用して、テーブルのバージョン履歴を維持し、他のプロセスが同じ場所書き込んでいる場合でも、Postgres が一貫した Parquet ファイルのセットを読み取るようにします。
- **Parquet Iceberg** および **Delta Lake** データレイクで使用される基になるカラムナストレージ形式。管理レイヤーなしでスタンドアロンフォーマットとして使用される場合、PGAA は物理ファイルまたはディレクトリに直接マッピングします。

セントラルカタログなしでデータレイクを照会するために、システムは、保存場所、データファイルが存在する特定のバケットまたはディレクトリへのポインターを介したリンクを確立し、AWS S3 および S3 互換 API、Google Cloud 全体の環境をサポートストレージ GCS、Azure Blob Storage ADLS Gen2 を含む、およびローカルファイルシステム。

取得プロセスは、次の 3 つの手順に従います。

- PGAA は、最初に最新のメタデータファイルまたはトランザクションログのストレージパスをスキャンするか、ファイルヘッダーを検査して生の Parquet のスキーマとデータ型を判断します。
- システムはメタデータを解析して、S3、GCS、Azure、またはローカルファイルシステムに関連するオブジェクトの仮想マップを作成します。
- **Seafowl** 分析エンジンは、オブジェクトストアからデータを直接取得し、ソースでベクトル化フィルタと集計を適用します。処理された結果は Postgres エグゼキューターにストリーミングされ、関連するデータのみが転送されます。

5.1 オブジェクトストレージから読み取るように Postgres を構成する

このプロセスには、ストレージの場所を介した物理ストレージアクセスを定義することと、リモートメタデータをローカル定義にマッピングする 2 つの手順が含まれます。

5.1.1 保存場所の作成

ストレージの場所は、クラウドバケットのエイリアスとして機能します。PGFS 格納場所を作成して、バケットの場所を Postgres に指示し、必要な資格情報を提供する必要があります。次の構文を使用して、`pgfs.create_storage_location()` ファンクションで保存場所を作成できます。

```
SELECT pgfs.create_storage_location(name=>storage_location_name,
                                   url=>protocol://path,
                                   options => {},
                                   credentials => {});
```

例、パブリック S3 バケットの場合

```
SELECT pgfs.create_storage_location(
  my-lake-data,
  s3://beacon-analytics-demo-data-us-east-1-prod,
  {"skip_signature": "true", "region": "us-east-1"}
);
```

構成パラメーターとサポートされているストレージプロバイダーの包括的なリストについては、[Creating a PGFS storage location](#) を参照してください。

保存場所を作成した後、次のコマンドでその存在を確認します。

```
SELECT pgfs.list_storage_locations();
```

`pgaa.test_storage_location()` ファンクションを使用して、接続を確認します。2 番目のパラメーターは、読み取りアクセス `false` または書き込みアクセス `true` をテストするかどうかを指定します。

```
SELECT pgaa.test_storage_location(my-storage-location, false);
```

ファンクションは、成功すると `NULL` を返します。

5.1.2 テーブルの Postgres へのマッピング

PGAA アクセス方法を使用して、保存場所をポイントするテーブルを作成します。例

```
CREATE TABLE iceberg_data ()
USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.storage_location = my_lake_data,
```

(次のページに続く)

(前のページからの続き)

```
pgaa.path = path/to/data
);
```

そこで

- `pgaa.format` 基になるテーブルまたはファイル形式を指定します。サポートされている値は、`'iceberg'`、`'delta'`、または `'parquet'` です。
- `pgaa.storage_location` 前の手順で定義した保存場所の名前。
- `pgaa.path` 保存場所のルートからテーブルのメタデータとデータファイルが含まれるディレクトリへの相対パス。Iceberg の場合、これは `/metadata` ディレクトリを含むフォルダーをポイントする必要があります。

注釈

利定義を空の `()` のままにすると、PGAA は、最初にテーブルを照会したときにメタデータファイルからスキーマを自動的に検出し、リモートの Iceberg/Delta メタデータまたは Parquet ヘッダーで見つかったフィールドに名前を列をマップします。

5.1.3 加速の照会と検証

テーブルが作成されたら、標準の SQL でクエリーできます。PGAA のベクトル化エンジンが実際に作業を処理していることを確認するには、`EXPLAIN` を使用します。

```
EXPLAIN (COSTS OFF) SELECT count(*), region FROM iceberg_data GROUP BY region;
```

エンジンがアクティブな場合、`SeafowlDirectScan` というノードが表示されます。これは、クエリが標準の Postgres エグゼキュータによって処理されるのではなく、ベクトル化アクセラレータにオフロードされたことを確認します。

5.2 制限事項

直接格納場所を使用する場合、ローカルの Postgres テーブル定義とリモートデータファイル間のアライメントを管理する必要があります。この手動アプローチでは、Postgres システムカタログに保存されている静的スキーマと、データレイクで定義された実際の動的スキーマ間の競合が発生します。

外部エンジンがデータレイクスキーマを進化させると、クエリの失敗、サイレントデータの省略、または不正確なデータマッピングが発生する可能性があります。

一元化されたカタログなしで作業すると、新しいテーブルの検出、マルチプルのエンジンからの同時書き込みの管理、およびクエリパフォーマンスに影響を与える機能も制限されます。

データレイクの進化に応じて、これらの制限を手動で解決する必要があります。

- Postgres の `ALTER TABLE` コマンドを使用して、リモートスキーマと一致するように列を追加、削除、または変更します。

- 最も一般的な修正は、空のかっこ () を使用してテーブルを削除して再作成することです。最初の照会時に、PGAA はリモートメタデータの新しい検出を実行します。
- ドリフトを防ぐには、オブジェクトストレージファイルと正確に一致するように `CREATE TABLE` ステートメントで列名とタイプを明示的に定義します。

スキーマ干渉を排除する最も効果的な方法は、スタンドアロンのストレージ場所から外部の Iceberg カタログに移動することです。カタログを使用することにより、スキーマ管理の責任をユーザーからシステムに移行します。詳細は、[Integrating with Iceberg catalogs](#) を参照してください。

第6章 Integrating with Iceberg catalogs

カタログサービスは、データレイクの集中化された「頭脳」として機能し、論理テーブル定義から物理ストレージを分離します。メタデータ、スキーマ、およびパーティショニングの単一の信頼できるソースを維持することにより、Postgres、Spark、Trino などの複数の独立したエンジンが、破損やスキーマドリフトのリスクなしで同じデータを共有できます。

Postgres Analytics Accelerator PGAA は、サードパーティのカタログと統合して、Postgres 展開内の分析ワークロードにシームレスなデータレイク接続を提供します。

PGAA は、次のカタログ実装を使用して Iceberg テーブルの読み取りおよび書き込み *Iceberg カタログを使用する* をサポートしています。

- **REST カタログ** Lakekeeper、Tabular、Apache Polaris、Project Nessie など、Apache Iceberg REST API に続くサービス。
- **AWS S3 テーブルクラウドスケール**の分析用に設計された高性能の AWS ネイティブカタログ。

PGAA を使用した外部 Iceberg カタログの管理には、2 つの主なフェーズが含まれます。

1. **カタログ接続の追加**。
2. **カタログテーブルの登録**。

注釈

外部カタログを使用する場合、保存場所を作成する必要はありません。テーブルは、保存場所またはカタログにリンクされています。

6.1 カタログ接続の追加

ファンクション `pgaa.add_catalog()` を使用して、外部 Iceberg カタログを Postgres インスタンスに登録します。このファンクションは、カタログの場所とログイン方法を Postgres に伝えるメタデータリンクを作成します。また、カタログ接続と資格情報を検証し、構成を保存します。

```
SELECT pgaa.add_catalog(  
    catalog_name TEXT,  
    catalog_type TEXT,  
    catalog_options JSON  
);
```

次のカタログタイプのいずれかを指定します。

- `iceberg-rest` Iceberg REST カタログ Lakekeeper、Snowflake Polaris、Tabular の場合。

- iceberg-s3tables AWS S3 テーブルの場合。

6.1.1 REST カタログ

ほとんどの REST カタログサービスでは、JSON オブジェクトを介してエンドポイントとトークンの詳細を提供します。

```
SELECT pgaa.add_catalog(  
  my_iceberg_catalog,  
  iceberg-rest,  
  {  
    "url": "https://your-catalog-endpoint.com",  
    "warehouse_name": "your_warehouse_name",  
    "warehouse": "your_warehouse_id",  
    "token": "your_secret_auth_token"  
  }  
);
```

Snowflake のような一部の Iceberg REST カタログプロバイダーは、認証のより高度な設定を必要とします。PGAA は、安全なローテーション認証を処理するための完全な OAuth2 オプションをサポートしています。例

```
SELECT pgaa.add_catalog(  
  my_iceberg_catalog,  
  iceberg-rest,  
  {  
    "oauth2.grant_type": "client_credentials",  
    "oauth2.client_id": "oauth-client-id",  
    "oauth2.client_secret": "oauth-client-secret",  
    "oauth2.token_uri": "https://EXAMPLE-polaristest.snowflakecomputing.com/  
→polaris/api/catalog/v1/oauth/tokens",  
    "oauth2.scope": "PRINCIPAL_ROLE:read_only"  
  }  
);
```

6.1.2 AWS S3 テーブル

AWS S3 テーブルカタログの場合、Amazon リソースネーム ARN とリージョンを指定します。

```
SELECT pgaa.add_catalog(  
  my_s3_catalog,  
  iceberg-s3tables,  
  {  
    "arn": "arn:aws:s3tables:us-east-1:123456:bucket/analytics",  
    "region": "us-east-1"  
  }
```

(次のページに続く)

(前のページからの続き)

```
}  
);
```

6.1.3 カタログ構成の検証

カタログを追加した後、次の機能を使用して接続を確認し、利用可能なデータを探索します。

- 登録されたカタログをリストします

```
SELECT * FROM pgaa.list_catalogs();
```

- 構成の接続と書き込み権限をテストします。接続が成功した場合、ファンクションは NULL を結果返します。例

```
SELECT pgaa.test_catalog(my_iceberg_catalog, test_writes := FALSE);
```

- インポートする前に、外部カタログ内で使用可能なテーブルを参照します。

```
SELECT pgaa.list_catalog_tables(my_iceberg_catalog);
```

カタログの登録とメンテナンスをサポートするファンクションの完全なリストについては、[カタログファンクション](#) リファレンスセクションを参照してください。

6.2 カタログテーブルの登録

カタログ接続を確立したら、Postgres が外部データを認識および照会できるように、そのメタデータを登録する必要があります。PGAA は、登録のための 3 つの異なるパスを提供します。

- [カタログの添付](#) 動的な自動同期。
- [カタログのインポート](#) 静的ワнтаイムスナップショットの場合。
- [カタログ管理テーブルの作成](#) 高レベルの粒度で単一のテーブルを登録します。

6.2.1 カatalogの添付

カタログを添付すると、Postgres と外部の Iceberg メタデータサービス間の動的リンクが確立されます。静的インポートとは異なり、この方法では、Postgres 環境が進化するデータレイクとの同期を維持します。

カタログをアタッチすると、PGAA はカタログを継続的にポーリングして新しいテーブルまたはスキーマの進化を検出するバックグラウンドワーカーを起動します。PGAA は、カタログ側で定義された名前空間階層を優先します。接続されたカタログに、データベース上の既存のテーブルと名前を共有するテーブルが含まれている場合、バックグラウンドワーカーはそれをスキップして上書きを回避します。

同期プロセスは、2 つの主要な構成パラメーターによって管理されます。

- `pgaa.metastore_sync_poll_rate_s` ポーリング頻度を制御します。
- `pgaa.metastore_sync_stale_duration_s` メタデータの鮮度のしきい値を定義します。

`pgaa.attach_catalog()` ファンクションを使用してカタログを添付します。

```
SELECT pgaa.attach_catalog(my_iceberg_catalog);
```

Postgres 環境でテーブルが作成されたことを確認します。カタログテーブルが Postgres インスタンスに登録されると、標準のテーブルと同じようにアクセスできます。

`pgaa.detach_catalog()` ファンクションを使用して、カタログを切断できます。

```
SELECT pgaa.detach_catalog(my_iceberg_catalog);
```

このファンクションは、バックグラウンド同期ワーカーを停止し、Iceberg メタデータサービスへのダイナミックリンクを削除します。カタログは追加されたままで `pgaa.list_catalogs()` に表示され、後で資格情報を再入力せずに再接続できますが、更新のポーリングや新しいテーブルの自動的なマップは行われません。オプション `CASCADE` を使用する場合、このカタログによって管理されているすべてのテーブルもドロップされます。

6.2.2 カタログのインポート

継続的にポーリングする代わりに、1 回だけの手動同期を実行できます。Postgres はテーブルの現在の状態を取得しますが、インポートを再度実行するまで将来の変更は検出しません。インポートされたカタログに、データベース上の既存のテーブルと名前を共有するテーブルが含まれている場合、PGAA はそのテーブルの作成をスキップして、上書きを回避します。カタログをインポートして、次のバックグラウンドポーリングサイクルを待たずに更新を強制することもできます。

`pgaa.import_catalog()` ファンクションを使用してカタログをインポートします。

```
SELECT pgaa.import_catalog(my_iceberg_catalog);
```

オプションで名前空間フィルタを指定するコマンドを使用して、この特定の Iceberg 名前空間内のテーブルのみがインポートされるようにできます。例

```
SELECT pgaa.import_catalog(my_iceberg_catalog, tpcds_sf_1000);
```

新しいテーブルが Postgres 環境に表示されることを確認します。追加の構成なしで、データレイクに対して標準の SQL クエリを直接実行できます。

6.2.3 カタログ管理テーブルの作成

カタログ管理テーブルでは、既存のカタログ接続を活用しながら、カスタムローカルテーブル名を定義し、特定のスキーマ配置を選択し、列を明示的にマップできます。

手動登録は、カタログ全体を添付するよりも柔軟性があります。Postgres 名前空間を完全に制御できるため、Iceberg カタログ内の場所に関係なく、任意のローカルスキーマにテーブルを配置できます。

手動テーブルの同期は、データレイクで発生する変更の種類によって異なります。

Change type	Synchronization	Behavior details
Data changes	Automatic	Every query resolves the latest Iceberg snapshot. New, updated, or deleted rows are reflected immediately.
Partition evolution	Automatic	Iceberg stores partitioning logic in metadata. If the partitioning strategy changes, PGAA handles the file-pruning adjustments behind the scenes.
Schema evolution	Manual	Structural changes (new or deleted columns) are ignored or will cause errors. You must update the Postgres table using <i>ALTER TABLE</i> to match the new remote schema.

CREATE TABLE ... USING PGAA WITH 構文を使用して、カタログ管理テーブルを作成します。例

```
CREATE TABLE managed_sales_data (
  id BIGINT,
  sale_date TIMESTAMP,
  amount NUMERIC
)
USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
  pgaa.catalog_table = managed_sales_data
);
```

そこで

- `pgaa.format` テーブルフォーマット 'iceberg' を設定します。
- `pgaa.managed_by` 特定の外部カタログ `pgaa.add_catalog` によって以前に定義された外部カタログを指定します。
- `pgaa.catalog_namespace` Iceberg カタログ内でテーブルが存在する名前空間を定義します。
- `pgaa.catalog_table` データレイクに表示されるテーブルの名前を指定します。

注釈

列定義を空の `()` のままにすると、PGAA は Iceberg カタログからスキーマを自動的に検出およびマッピングします。

第7章 Writing to object storage

Postgres アナリティクスアクセラレータ PGAA は、`CREATE TABLE AS SELECT` CTAS 構文を使用したデータレイクへの直接書き込みをサポートしています。これにより、ローカルの Postgres テーブルの変換、既存のオブジェクトストレージファイルの再構築、または両方のデータを結合するにかかわらず、任意のソースからのデータを Iceberg や Delta Lake のような高性能のカラムナー形式に処理および永続化できます。

PGAA は Postgres クエリエグゼキューターと統合されているため、CTAS オペレーションのソースデータは、インスタンスにアクセスできる任意の場所に存在できます。これにより、いくつかの高度なワークフローが有効になります。

- **エクスポート** 標準のローカル Postgres テーブルからデータレイクにデータを移動して、履歴データをアーカイブします。
- **変換** 1つのオブジェクト格納場所から読み取り、結合または集計を実行し、結果を別のクラウドディレクトリに書き込みます。
- **フェデレーションジョイン** ローカルリレーショナルテーブルをリモートオブジェクトストレージテーブルと結合し、結合された結果をクラウドに書き込みます。

PGAA アクセス方法を使用することにより、クエリの結果セットを指定したストレージ場所に直接ストリーミングできます。AWS S3 および S3 互換 API、Google Cloud Storage GCS、Azure Blob Storage ADLS Gen2、およびローカルファイルシステム、または統合された Iceberg カタログに直接。

スタンドアロンのストレージ場所または統合された Iceberg カタログを使用して、CTAS 操作を実行できます。

注釈

CTAS は、不変のライトワンス操作用に設計されています。オブジェクトストレージでの継続的な `INSERT`、`UPDATE`、または `DELETE` 操作など、より高度な書き込み機能については、[PGD integration](#) を参照してください。

7.1 スタンドアロンの保存場所を使用する

この方法は、1 回限りのエクスポートまたはアドホックデータの移動に最適です。PGAA は、Postgres 行からターゲット形式への変換を処理し、必要なメタデータまたはトランザクションログ Iceberg マニフェストまたは Delta トランザクションログなどを生成し、単一のアトミック操作で結果をコミットします。

CTAS を介してオブジェクトストレージにテーブルをプロビジョニングするには、2 段階のプロセスが必要です。最初に、ストレージの場所を介してリモートストレージ環境への物理接続を確立し、次にクエリ結果をその場所にマッピングする SQL コマンドを実行します。

7.1.1 保存場所の作成

オブジェクトストレージに書き込むには、create a PGFS storage location を行う必要があります。これにより、ターゲットバケットが定義され、必要な資格情報が提供されます。例

```
SELECT pgfs.create_storage_location(
    my_lake_data,
    s3://your-bucket-name/path,
    {"aws_region": "us-east-1", "aws_access_key_id": "...", "aws_secret_access_key":
    → "..."}
);
```

注釈

保存場所に関連付けられた資格情報には、宛先バケットの読み取りおよび書き込み権限が必要です。これは、次のファンクションを実行して確認できます。成功した場合は`NULL`を結果ます。

```
SELECT pgaa.test_storage_location (my_lake_data, true);
```

7.1.2 CTAS を使用したテーブルの作成

USING PGAA WITH および AS 句を使用して、結果セットをオブジェクトストレージの構造化テーブルフォーマットに直接エクスポートするテーブルを作成します。例

```
CREATE TABLE s3_export_table
USING PGAA
WITH (
    pgaa.storage_location = my_lake_data,
    pgaa.path = path/in/s3/iceberg_data,
    pgaa.format = iceberg
) AS
SELECT * FROM local_source_table;
```

そこで

- `pgaa.format` 宛先テーブルフォーマットを指定します。サポートされている値は `iceberg` および `delta` です。
- `pgaa.storage_location` 前の手順で定義した保存場所の名前。
- `pgaa.path` 宛先ディレクトリ。PGAA は、このパス内にメタデータとログディレクトリを自動的に作成します。

7.2 Iceberg カタログを使用する

Iceberg カタログを統合している場合は、CTAS を使用してそれには書き込むことができます。このモードでは、カタログはストレージパスとテーブルのバージョン管理を自動的に管理します。

CTAS を介してオブジェクトストレージのテーブルに書き込み、Iceberg カタログを使用するには、2つのステップで構成されます。最初にカタログ接続を構成し、次にそのカタログを介してクエリ結果をマップする SQL コマンドを実行します。

7.2.1 カタログ接続の構成

ファンクション `pgaa.add_catalog()` を使用して、外部 Iceberg カタログを Postgres インスタンスに登録します。例

```
SELECT pgaa.add_catalog(
  my_iceberg_catalog,
  iceberg-rest,
  {
    "url": "https://your-catalog-endpoint.com",
    "warehouse_name": "your_warehouse_name",
    "warehouse": "your_warehouse_id",
    "token": "your_secret_auth_token"
    "danger_accept_invalid_certs": "false"
  }
);
```

構成オプションの完全なリストについては、[カタログ接続の追加](#) を参照してください。

注釈

カタログへの書き込み権限が必要です。これは、次のファンクションを実行して確認できます。成功した場合は `NULL` を結果ます。

```
SELECT pgaa.test_catalog(my_iceberg_catalog, test_writes := TRUE);
```

7.2.2 カタログ管理 CTAS テーブルの作成

USING PGAA WITH および AS 句を使用して、結果セットを構成済みの Iceberg カタログに直接エクスポートするテーブルを作成します。例

```
CREATE TABLE export_local_transactions
USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
```

(次のページに続く)

(前のページからの続き)

```
pgaa.catalog_table = managed_sales_data
) AS
SELECT * FROM local_transactions;
```

そこで

- `pgaa.format` 宛先テーブルフォーマットを設定します。現在、'iceberg' のみがサポートされています。
- `pgaa.managed_by` カタログ名を指定します。
- `pgaa.catalog_namespace` Iceberg カタログ内でテーブルが配置される名前空間またはデータベースを定義します。
- `pgaa.catalog_table` データレイクに作成するテーブルの名前を設定します。

7.3 CTAS の進行状況のモニタリング

大規模な CTAS 操作の場合、PGAA は NOTICE メッセージを発行して、システムビューをポーリングしたりログを照会したりせずに進行状況を追跡できます。進行状況は 100 万行ごとまたは 10 分ごとのいずれか早い方で報告され、その後に最終的な完了メッセージが続きます。

```
NOTICE: CTAS progress: 1000000 rows processed
NOTICE: CTAS progress: 2000000 rows processed
NOTICE: CTAS completed: 2500000 rows processed
```

7.4 CTAS の制限

CTAS はデータの移動と変換のための強力なツールですが、次の制約のある特定の実行モデルで動作します。

- **エンジンの依存関係** CTAS 操作を実行する場合、ソース内部クエリは DirectScan を利用しません。代わりに、データの取得は、CompatScan を介して標準の Postgres エンジンによって処理されます。したがって、SELECT フェーズ内の複雑な結合または集計は、Seafoal エンジンが提供する完全なベクトル化加速の恩恵を受けられません。
- **自動上書きなし** 偶発的なデータの損失を防ぐため、スタンドアロンの保存場所を使用する場合、指定された `pgaa.path` にデータが既に存在する場合、コマンドはデフォルトで失敗します。既存のデータを置き換える場合は、CTAS ステートメントの WITH 句で `pgaa.purge_data_if_exists` パラメーターを `true` に明示的に設定する必要があります。
- **ライトワンス** CTAS は、アトミックな書き込みワンス操作です。データを変更するには、テーブルを削除し、新しいパスで再作成する必要があります。

第8章 Replicating with PGD

Postgres Distributed PGD と Postgres Analytics Accelerator PGAA を統合して、統一アーキテクチャ内で完全なデータライフサイクルを管理します。PGD はホットデータの高可用性トランザクションバックボーンを提供し、PGAA はオブジェクトストレージ内のコールドデータのベクトル化エンジンとクラウドネイティブハンドラーを提供します。

このコラボレーションでは、2つの異なるストレージ階層を作成します。

- ホットティア PGD ミリ秒未満のレイテンシーと完全な ACID コンプライアンスを必要とする現在の操作データ用のローカル PostgreSQL ストレージ。
- コールド層 PGAA 高速分析クエリ用に最適化された Apache Iceberg[®]または Delta Lake のようなカラムナフォーマットを使用した費用対効果の高いオブジェクトストレージ。

これらを組み合わせることで、統合された Postgres インターフェイスを介した透過的なアクセスを維持しながら、高性能トランザクションストレージから費用対効果の高いデータレイクにデータをシームレスに移行できます。このアーキテクチャは、インパクトの高いさまざまなユースケースをサポートしています。

- **規制へのコンプライアンス**古いデータと監査証跡をオフロードして、長期的な保持を保証しながら、無駄のない高性能システムを維持します。- **時系列と IoT** センサーとイベントストリームの費用対効果の高いコールド履歴分析と、ホットな最近のデータを使用したりアルタイムダッシュボードのバランスをとります。- **金融サービス**トランザクションストレージコストを高騰させることなく、必要な大量の履歴取引レコードを管理します。- **小売および季節的なビジネスサイクル**適応ポリシーを使用して、古いレコードをコールドストレージに階層化しながら、カスタマーサービスのためにピークシーズンの注文データをホットに保ちます。

8.1 テーブルタイプを理解する

このアーキテクチャは、2つの異なるストレージ層ホット層ローカル SSD、ミリ秒未満のレイテンシーとコールド層オブジェクトストレージ、カラムナフォーマットを実装しています。データは、次の3つの異なるテーブルタイプのいずれかに存在します。

Table type	Storage location	Behavior
Heap	Local disk only	ホット層の標準のトランザクションテーブル。
Hybrid Analytics and Transactional Processing (HTAP)	Local + object storage	分析コピーにマッピングされたヒープテーブル。PGD は、すべての変更をほぼリアルタイムで複製します。
PGAA	Object storage only	デルタまたはアイスバーク形式で利用可能な純粋な分析コールドテーブル。

8.2 階層化ストレージ戦略

運用ニーズに基づいて、層間でデータを移行するには3つの使用可能な方法があります。

8.2.1 階層テーブルの実装

この方法では、**PGD AutoPartition** を活用して、大容量データのライフサイクルを自動化します。ヒープテーブルは、次のパーティション構造に変換されます。- **ホットパーティション** 最近のデータは、アクティブな処理のヒープまたは **enable_replication** が有効になっている場合は HTAP として残ります。- **コールドパーティション** パーティションが経過期間のしきい値を超えると、PGD はそれらをオブジェクトストレージに大量コピーし、PGAA テーブルに変換し、ローカルディスク領域を再利用して、データを照会可能に維持します。

8.2.2 アナリティクスへの複製

このメソッドは、テーブルをオブジェクトストレージ内の分析コピーと継続的に同期します。ヒープテーブルは HTAP テーブルに変換されます。ユーザーには、標準のトランザクションテーブルのように見えますが、分析コピーはバックグラウンドで維持されます。

トランザクションのスループットに負担を与えることなく、ほぼリアルタイムの同期を保証するために、PGAA は Iceberg Merge-on-ReadMoR を使用します。これは、データブロック全体を書き換えるのではなく、小さな「削除ファイル」で変更を記録し、最小限のオーバーヘッドでデータレイクを最新の状態に保ちます。

8.2.3 アナリティクスへのオフロード

この方法は、HTAP テーブルのローカルディスク領域をすぐに再利用し、オブジェクトストレージ PGAA テーブルにコピーのみを残す 1 回限りの外科的方法を提供します。ローカルヒープコピーは切り捨てられ、テーブルアクセス方法は PGAA に変更されます。これはオブジェクトストレージをポイントするようになりました。

トランザクション書き込み用にデータを戻す必要がある場合は、オブジェクトストレージからローカルの Postgres ヒープにテーブルを復元できます。

メソッド | 目的 | ユースケース | 原点テーブル | 宛先テーブル | ----- | ----- | ----- | ----- |
 ----- | 階層化テーブル | 自動化されたライフサイクル管理 | 履歴の保持が重要かつ自動化されている大量の時系列データ。 | ヒープ | ヒープまたは HTAP | ホット PGAA | コールド レプリケーション | レプリケーション | レプリケーション | レプリケーション | レプリケーション | 継続的な同期 | 一般的な分析のためにほぼリアルタイムでトランザクションテーブルをレプリケート。 | ヒープ | HTAP オフロード | 選択的なアーカイブ | ワンタイムの外科的削除と 特定のテーブルのアーカイブ。 | HTAP | PGAA

8.3 メタデータ管理

PGAA は、PGD と統合するときにオブジェクトストアのメタデータを管理する 2 つの方法をサポートしています。

- **カタログオフロード** *storage locations* を使用してメタデータをデータファイルとともに保存し、オブジェクトストレージに直接接続します。

- **カタログ管理オフロード** Iceberg REST などのメタデータ *Integrating with Iceberg catalogs* からデータを分離します。

8.3.1 Getting started

分散環境内に階層化ストレージを実装するには、最初に EDB Postgres 分散 PGD クラスターとオブジェクトストア間の一貫したリンクを確立する必要があります。

以下のセクションでは、クラスターがストレージプロバイダーをポイントするために必要な前提条件と構成手順を説明します。これらの手順は、PGD データのバージョンがカラムナ形式で永続化される外部の場所を定義します。

ノードグループレベルでこれらの設定を構成することにより、クラスター内のすべてのノードが階層化データの書き込みおよび読み取りの場所を正確に認識する必要があります。

前提条件

- **クラスター PGAA** および PGFS 拡張機能がインストールされた PGD バージョン 6.1 以降。
- **ストレージの場所** Iceberg または Delta Lake 形式を使用するローカル、S3、GCP、または Azure ストレージ。
- **カタログ** 外部カタログサービスを使用する場合、Iceberg REST カタログのみがサポートされます。
- **権限** データベースユーザーには、PGD および PGAA ファンクションに対する CREATE、ALTER、および EXECUTE 特権が必要です。

注釈

保管場所またはカタログサービスに関連付けられている資格情報には、宛先バケットの読み取りおよび書き込み権限が必要です。次のファンクションを実行して、権限を確認できます。成功した場合は `NULL` を結果ます。

- 保存場所の場合

```
SELECT pgaa.test_storage_location (my_storage_location, true);
```

- カタログサービスの場合

```
SELECT pgaa.test_catalog(my_iceberg_catalog, test_writes := FALSE);
```

オブジェクトストレージをポイントする

階層化テーブルを作成する前に、コールドデータの宛先を定義する必要があります。分散 PGD クラスター内で作業しているため、これらの構成呼び出しを `bdr.replicate_ddl_command` でラップして、設定がグループ内のすべてのノードに伝播されるようにする必要があります。

!!!ヒント

ノードグループを表示するには、`SELECT node_group_id FROM bdr.node_group;` を使用します。

1. PGFS extension を使用してストレージの場所を作成し、ノードグループの分析ストレージとして設定します。

```
SELECT bdr.replicate_ddl_command($$
  SELECT pgfs.create_storage_location(
    my-storage-location,
    protocol://my-bucket-name/path/,
    {"region": "region-name"},
    {"access_key_id": "...", "secret_access_key": "..."}
  )
$$);
```

2. analytics_write_catalog をクリアします。analytics_storage_location を割り当てる前に、これは NULL である必要があります。

```
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_write_catalog, NULL);
```

3. ノードグループの分析ストレージの場所を設定します。

```
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_storage_location, my-
→storage-location);
```

1. *Integrating with Iceberg catalogs* への接続を構成します。

```
SELECT bdr.replicate_ddl_command($$
  SELECT pgaa.add_catalog(
    my-catalog-name,
    iceberg-rest,
    {
      "url": "https://my-catalog-rest-endpoint.com",
      "token": "MY_AUTH_TOKEN",
    }
  )
$);
```

2. カタログをデフォルトの分析の書き込み場所として設定します。

```
SELECT bdr.replicate_ddl_command($$
  SELECT bdr.alter_node_group_option(my-pgd-group, analytics_write_catalog, my-
→catalog-name)
$);
```

次のステップ

PGD クラスターのストレージターゲットを定義したため、パフォーマンスと容量の要件に基づいて、テーブルがそのストレージとどのように相互作用するかを決定できます。

- **階層テーブルの実装** ローカルヒープストレージから経過期間のしきい値に基づいてターゲットとされる分析ストレージにパーティションを自動的に移行することにより、ゼロタッチデータライフサイクルを確立します。
- **アナリティクスへの複製** ローカルトランザクションヒープテーブルを維持すると同時に、大量の分析処理のためにカラムナコピーをストレージターゲットに同期します。
- **アナリティクスへのオフロード** アナリティクスにレプリケートされる既存のテーブルからヒープディスク領域を再利用します。

8.3.2 Implementing tiered tables

階層テーブルは、Postgres Distributed PGD AutoPartition と Postgres Analytics Accelerator PGAA を組み合わせて、自動化されたデータライフサイクルを作成することにより、ユニファイドストレージアーキテクチャを提供します。標準ヒープテーブルが階層構造に変換されると、システムは手動介入なしでトランザクションストレージから分析ストレージへのデータの移行を管理し、ストレージコストを最適化するゼロタッチデータライフサイクル管理を提供します。

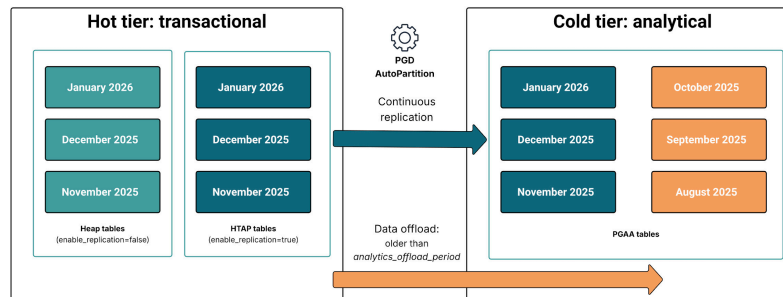


図 1: tiered_tables

このアーキテクチャでは、クエリはホットとコールドの両方のデータ層にシームレスにまたがり、物理的な場所に関係なくデータセット全体への単一アクセスポイントを提供します。最近のデータは、高性能のトランザクション処理のためにローカルヒープパーティションに残りますが、古いコールドパーティションは、定義された経過期間のしきい値を超えると、オブジェクトストレージに透過的にオフロードされます。

`pgaa.convert_to_tiered_table()` を呼び出すことにより、標準ヒープテーブルはハイブリッドパーティション構造に変換されます。

- **ホット層** 最近のデータは、ローカルヒープパーティション、またはレプリケーションが有効になっている場合はハイブリッド分析およびトランザクション処理 HTAP テーブルに残ります。HTAP の場合、PGAA は Iceberg Merge-on-ReadMoR を使用して、軽量の「削除ファイル」として更新と削除をキャプチャします。これにより、大規模なデータブロックの書き換えのオーバーヘッドなしで、クラウド層の同期が維持されます。
- **コールド層** パーティションが `analytics_offload_period` で定義された経過時間しきい値を超えると、データレイクの PGAA テーブルに自動的に変換されます。ローカルストレージは再利用されますが、データは同じテーブルインターフェイスを介して透過的に照会可能なままです。

プロセスの概要

ヒープテーブルを階層テーブルに変換するには、次の手順を実行します。

1. *Getting started* 保存場所またはカタログサービスのいずれかを選択します。
2. `pgaa.convert_to_tiered_table()` ファンクションを使用してヒープテーブルの階層テーブルへの変換。
3. 階層化テーブルのモニタリング 自動オフロード。
4. ストレージニーズの進化に応じた 階層化テーブルの変更と無効化 設定。

ヒープテーブルの階層テーブルへの変換

`pgaa.convert_to_tiered_table()` ファンクションは、自動オフロードプロセスを開始します。`enable_replication` が `true` の場合、継続的レプリケーションのためにアクティブなパーティションを HTAP テーブルに変換します。例

```
CALL pgaa.convert_to_tiered_table(
    relation := my-transactional-table::regclass,
    range_partition_column := date,
    partition_increment := 1 month,
    analytics_offload_period := 1 year,
    initial_lower_bound := 2010-01-01,
    retention_period := 5 years,
    enable_replication := true,
    minimum_advance_partitions := 2,
    maximum_advance_partitions := 5,
    drop_after_retention_period := true,
    purge_analytics_target := true
);
```

そこで

- `relation` 階層化テーブルに変換するヒープテーブルの名前。
- `range_partition_column` レンジパーティション化に使用される列。DATE または TIMESTAMP であり、主キーに含まれる必要があります。
- `partition_increment` 各パーティションのサイズを定義する時間間隔。
- `analytics_offload_period` パーティションがローカルストレージからデータレイクのコールド層に移動される経過期間のしきい値。
- `initial_lower_bound` 最初のパーティションの下限の開始値。
- `retention_period` データがコールド層からパージされる期間。
- `enable_replication` `true` の場合、アクティブなパーティションのオブジェクトストレージへのリアルタイムレプリケーションを HTAP テーブルとして有効にします。`false` の場合、ホットパーティションはヒープテーブルとして残ります。

- `minimum_advance_partitions` 維持する将来のパーティションの最小数。
- `maximum_advance_partitions` 維持する将来のパーティションの最大数。
- `drop_after_retention_period` 保持期間の有効期限が切れた後、パーティションをローカルにドロップするかどうか。
- `purge_analytics_target true` の場合、オブジェクトストアディレクトリまたはカタログエントリに既存のテーブルが削除されます。 `false` に設定されており、既存のテーブルが検出された場合、コマンドはエラーで失敗します PGD 6.3 以降でサポートされています。

i 注釈

レンジパーティションに使用される列は、主キーである必要があります。

シーケンスまたは外部キー制約を含むテーブルは、階層テーブルではサポートされていません。階層テーブルに変換する前に、シーケンスまたはコンストレインを削除する必要があります。

階層化テーブルは、すべてのパーティションの `UNION` を実行する統合ビューを作成し、データライフサイクル全体で単一のクエリインターフェイスを提供します。 `bdr.prefer_analytics_engine` パラメーターを有効にして、階層テーブル全体のクエリーを Seafoal エンジンにルーティングできます。

```
SET bdr.prefer_analytics_engine = TRUE;
```

`enable_replication` が `true` の場合、この設定はパーティションテーブル全体の `DirectScan` を有効にし、Seafoal は単一のパスでオフロードされたコールドデータとレプリケートされたホットデータの両方に対してクエリーを実行できます。

i 注釈

ストレージ場所ベースのレプリケーションを使用する場合、パーティション分割テーブルのユニオンビューは Seafoal エンジンでのみサポートされ、Spark を介してアクセスできません。カタログサービスを使用する場合、Seafoal と Spark の両方を介してユニオンビューにアクセスできます。

階層化テーブルのモニタリング

`pgaa.list_tiered_tables()` ファンクションは、現在自動化されたライフサイクル管理が行われているすべてのテーブルの高レベルの概要を提供します。その出力は、どのヒープテーブルが階層化テーブルに正常に変換されたか、どのパーティション化ロジックが適用されているか、およびローカルストレージとオブジェクトストレージ間の現在のバランス階層化および非階層化データサイズを特定するのに役立ちます。

```
SELECT * FROM pgaa.list_tiered_tables();
```

`bdr.analytics_table` ビューを照会して、Postgres スキーマと基になる分析エンジン間の低レベルのマッピングを表示します。このテーブルは、ローカルノードグループとリモート分析ターゲットの関係を追跡します。

```
SELECT * FROM bdr.analytics_table;
```

次のクエリを使用して、ホットパーティションとコールドパーティションを区別します。

```
SELECT
    relname AS partition_name,
    amname AS access_method
FROM pg_class c
JOIN pg_am am ON c.relam = am.oid
WHERE relname LIKE my-transactional-table%
    AND c.relkind = r
ORDER BY relname;
```

階層化テーブルの変更と無効化

PGD と PGAA コマンドを組み合わせ使用して、ライフサイクルポリシーを変更したり、標準のパーティションテーブルに戻すことができます。

ライフサイクル設定の調整

`bdr.autopartition()` ファンクションを実行する既存の階層テーブルのパーティション化とオフロードロジックを更新できます。

これは、オフロードしきい値を短縮または拡張する必要がある場合に特に役立ちます。たとえば、`analytics_offload_period` を 1 か月から 3 か月に変更するには、次のコマンドを実行します。

```
SELECT bdr.replicate_ddl_command($$
    SELECT bdr.autopartition(
        relation := public.my-transactional-table,
        partition_increment := 1 month,
        partition_initial_lowerbound := 2020-01-01,
        analytics_offload_period := 3 months
    )
$$);
```

詳細は、[AutoPartition in PGD](#) を参照してください。

警告

これらの設定の変更は、新しいパーティションにのみ影響し、既存のデータは再編成されません。パーティションサイズを増やしても、システムは現在の小さなパーティションをマージしません。現在のセットが使い果たされると、より大きなパーティションの作成が開始されます。新しいオフロードタイミングはすべてのパーティションに適用されますが、以前の設定で作成された物理構造は変更されません。!!!

レプリケーションステータスの変更

`enable_replication` の現在の設定を変更する場合は、パーティションのレプリケーションステータスを管理する必要があります。

親テーブルに対して `pgaa.enable_analytics_replication()` または `pgaa.disable_analytics_replication()` ファンクションを使用して、階層全体の一貫性を保証します。

```
CALL pgaa.enable_analytics_replication(my-transactional-table::regclass);
```

親テーブルに対して `ALTER TABLE` を使用すると、将来のパーティションの `enable_replication` の値のみが設定されます。既存の子パーティションは遡って更新されません。既存のすべてのパーティションで `ALTER TABLE` コマンドを手動で実行して、データ全体の一貫性を保証する必要があります。

階層テーブルの無効化

標準のパーティションヒープテーブルに戻すには

1. `enable_replication` がアクティブであった場合、親テーブルのレプリケーションを停止します。

```
CALL pgaa.disable_analytics_replication(my-transactional-table::regclass);
```

1. コールドパーティションを復元します。コールドパーティションごとに、次を実行します。

```
pgaa.restore_from_analytics(partition_name);
```

または `ALTER TABLE partition_name SET ACCESS METHOD heap;` を使用します。

1. 親テーブルから階層テーブルフラグを削除します。

```
ALTER TABLE my-transactional-table SET (pgaa.tiered_table = false);
```

1. `AutoPartition` を停止しますオプション。テーブルが新しいパーティションを完全に作成しないようにするには、次のコマンドを実行します。

```
bdr.drop_autopartition(my-transactional-table);
```

1. テーブルが `SELECT * FROM pgaa.list_tiered_tables()` を使用して階層テーブルとしてリストされていないことを確認します。
2. すべてのパーティションがアクセス方法として `heap` をリストしていることを確認します。

```
SELECT
    c.relname AS partition_name,
    a.amname AS access_method
FROM pg_class c
JOIN pg_am a ON c.relam = a.oid
WHERE c.relname LIKE my-transactional-table%
AND c.relkind = r;
```

8.3.3 Replicating to analytics

Postgres DistributedPGD クラスター内で Postgres アナリティクスアクセラレーター PGAA を使用して、ローカルヒープテーブルをハイブリッド分析およびトランザクション処理 HTAP テーブルに変換することにより、分析データレイクと同期します。

この状態では、すべての INSERT、UPDATE、および DELETE オペレーションがキャプチャされ、ほぼリアルタイムでオブジェクトストレージにストリーミングされます。テーブルはアプリケーションの標準のトランザクションエンドポイントであり続けますが、PGD はバックグラウンドで高忠実度の分析レプリカを維持し、運用パフォーマンスに影響を与えることなく BI ツールとクエリエンジンでデータをすぐに利用できるようにします。

このプロセスの主な利点は次のとおりです。

- **ニアリアルタイム分析** バッチ ETL プロセスを待たずに、データレイクの最新のトランザクションデータにアクセスします。
- **デュアルステートのアクセシビリティ** トランザクション用のローカルの高性能ストレージを維持しながら、大量の分析クエリ用のカラムナバージョンを提供します。
- **統合管理** 標準の SQL および PGAA 機能を介して、データのライフサイクルとレプリケーション設定を直接管理します。
- **マージオンリードの効率** PGAA は、Iceberg Merge-on-ReadMoR を使用してデータを同期します。行が変更されるたびに大きなファイルを書き換える代わりに、単に削除ファイルに変更を記録します。これにより、レプリケーションの高速性が維持され、オブジェクトストレージの負荷が大幅に削減されます。

プロセスの概要

テーブルの継続的レプリケーションを有効にするには、次の手順を実行します。

1. **Getting started** 保存場所またはカタログサービスのいずれかを選択します。
2. **レプリケーションの有効化** データストリームを開始します。
3. **モニタリングレプリケーション** ステータスとヘルス。
4. **レプリケーションの無効化** 同期が必要でなくなった場合。

レプリケーションの有効化

`pgaa.enable_analytics_replication()` ファンクションを使用して、既存のヒープテーブルを構成して、データをオブジェクトストレージにストリーミングします。ファンクションは、ヒープテーブルを分析 HTAP テーブルに変換し、論理レプリケーションワーカーを起動します。

```
CALL pgaa.enable_analytics_replication(my_table::regclass, purge_analytics_target :=  
→ TRUE);
```

レプリケーションが開始される前に、ターゲットの場所にある既存のデータをパージするように `purge_analytics_target := TRUE` を設定します PGD 6.3 以降が必要です。

または、テーブルオブジェクトでレプリケーション状態を直接切り替えます。

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = TRUE);
```

bdr.prefer_analytics_engine パラメーターを有効にして、クエリを PGAA エンジン Seaflowl または Spark にルーティングできます。

```
SET bdr.prefer_analytics_engine = TRUE;
```

無効にすると、クエリーは標準の Postgres エンジンによってローカルヒープに対して実行されます。

モニタリングレプリケーション

レプリケーションが有効になると、テーブルがローカルリレーションリストと分析メタデータの両方に表示されることを確認することにより、HTAP ステータスを確認できます。

ローカルテーブルと分析ターゲット間のマッピングを表示します。

```
SELECT * FROM bdr.analytics_table;
```

現在 PGAA によって管理されているすべてのテーブルをリストします。

```
SELECT * FROM pgaa.list_analytics_tables();
```

レプリケーションの無効化

レプリケーションを無効にすると、論理ワーカーによる変更のキャプチャが停止されます。ローカルテーブルは標準のヒープテーブルに戻り、オブジェクトストレージのリモートコピーはデータの静的スナップショットになります。

pgaa.disable_analytics_replication() ファンクションを使用して同期を停止します。

```
CALL pgaa.disable_analytics_replication(my_table::regclass);
```

または、ALTER TABLE を使用します。

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = FALSE);
```

```
SET bdr.prefer_analytics_engine= FALSE;
```

注釈

以前に無効になったテーブルでレプリケーションを再度有効にすると、システムは完全なリロードを実行します。このプロセスは、オブジェクトストレージ内の既存の分析データを効果的に削除し、現在のローカルヒープデータを全体的に再アップロードし、継続的同期を開始します。!!!

8.3.4 Offloading to analytics

ハイブリッド分析およびトランザクション処理 HTAP テーブルは、Postgres アナリティクスアクセラレーター PGAA を使用して、Postgres DistributedPGD クラスターとオブジェクトストレージの分析コピーを継続的に同期し、両方の冗長ストレージを犠牲にして、トランザクションパフォーマンスに影響を与えることなく、大量の分析を有効にしますローカルヒープとデータレイク。

HTAP テーブルの分析へのオフロードは、ローカルヒープコピーを削除し、ディスク領域を再利用するために使用される手動の外科的操作です。テーブルをオフロードすると、システムは次のことを行います。

- ローカルヒープテーブルを切り捨て、PGD ノードのディスク領域をすぐに解放します。
- データレイクへのリアルタイムレプリケーションを停止します。
- テーブルアクセス方法を PGAA に変更し、Postgres テーブル定義をオブジェクトストレージ内の既存のデータを直接ポイントします。

その結果、Postgres を介して完全に照会可能なままですが、ローカルストレージを消費しないコールドテーブルが生成されます。将来的に再びトランザクション書き込みが必要な場合は、テーブルを復元し、データをレイクからローカルヒープにストリーミングして戻すことができます。

プロセスの概要

テーブルをデータレイクに外科的にオフロードするには、次の手順を実行します。

1. *Getting started* 保存場所またはカタログサービスのいずれかを選択します。
2. **テーブルのオフロード**。
3. トランザクションアクセスが再度必要な場合は、**アナリティクスからの復元**ヒープに戻します。

テーブルのオフロード

`pgaa.convert_to_analytics()` ファンクションを使用して、同期された HTAP テーブルをコールドの分析テーブルに変換し、ローカルディスクからデータを削除します。

```
CALL pgaa.convert_to_analytics(my-table::regclass);
```

ファンクションはローカルデータを削除し、テーブルのアクセス方法を PGAA に再定義して、外部の Iceberg または Delta ファイルのみを指すようにします。

または、`ALTER TABLE` コマンドでアクセス方法を切り替えます。`pgaa.auto_truncate` オプションを使用して、ローカルディスク領域がすぐに解放されるようにします。

保存場所の場合

```
ALTER TABLE public.my_table
SET ACCESS METHOD pgaa,
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.storage_location = my-storage-location,
```

(次のページに続く)

(前のページからの続き)

```
pgaa.path = public.my_table
);
```

カタログ管理テーブルの場合

```
ALTER TABLE public.my_catalog_managed_table
SET ACCESS METHOD pgaa,
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
  pgaa.catalog_table = my_remote_table_name
);
```

アナリティクスからの復元

トランザクション書き込みを再開するか、データをローカルの高性能トランザクションストレージに戻す必要がある場合は、`pgaa.restore_from_analytics()` ファンクションを使用します。

```
CALL pgaa.restore_from_analytics(public.transactional_table);
```

ファンクションは、PGAA テーブルの構成を確認し、オブジェクトストレージ内の外部 Iceberg/Delta パスを特定し、データをテーブルのローカルストレージ構造に書き換えます。データがローカルに存在すると、ファンクションはテーブルのアクセス方法をローカルトランザクションエンドポイントに再度設定します。

`ALTER TABLE` コマンドを使用して、テーブルをデータレイクのコールド状態からローカルトランザクションヒープテーブルに復元することもできます。この手動方法は、ローカルストレージへの移行をきめ細かく制御します。

```
ALTER TABLE public.my_table
SET ACCESS METHOD heap;
```

警告

テーブルをヒープアクセス方法に復元しても、オブジェクトストレージからファイルは削除されません。オブジェクトストレージプロバイダーを介してデータを手動で削除するか、PyIceberg のようなツールを使用してカタログからテーブルを削除します。

第9章 Accelerating with Spark

デフォルトでは、Postgres Analytics Accelerator PGAA は、組み込み分析エンジンである Seafoal を使用してクエリを高速化します。ただし、単一の Postgres インスタンスのリソースを超える大規模なデータ処理の場合、Spark Connect を介してリモートの Apache Spark クラスターに実行をオフロードできます。

Spark Connect は、Spark ドライバーからアプリケーションを分離する Apache Spark のシンクライアントサーバープロトコルです。高速ブリッジとして機能し、Postgres はクエリ指示をリモートの分散 Spark クラスターに送信できます。これにより、データベースと同じマシンで Spark を実行する必要がなく、外部クラスターの膨大な計算能力を活用できます。

9.1 エグゼキューターエンジンの選択

`pgaa.executor_engine` 構成パラメーターは、分析クエリの重労働が発生する場所を決定します。デフォルトの Seafoal エンジンを超えて移動する場合、分散実行には 2 つの主なオプションがあります。

9.2 分散実行オプション

Spark を活用するには、次の 2 つの方法のいずれかで環境を構成できます。

- ***Distributed Spark execution*** : 標準の Apache Spark クラスターを使用して、分散 CPU コアを使用して大規模なデータセットを処理します。これは、大規模なデータ処理と大規模なメンテナンスタスクの標準的なアプローチです。
- ***GPU-accelerated Spark with NVIDIA RAPIDS*** : Apache Spark の NVIDIA RAPIDS アクセラレータを統合します。このオプションは、クエリの実行を GPU にオフロードし、コンピューティング集約的な結合と集計の処理時間とインフラストラクチャコストを大幅に削減します。

Feature	Seafoal	Spark Connect (CPU)	Spark + RAPIDS (GPU)
Architecture	Process alongside Postgres.	Remote Spark cluster.	Remote Spark with NVIDIA GPUs.
Best for	Small/medium datasets.	Petabyte-scale, complex ETL.	Massive joins & aggregations.
Scalability	Single host limits.	Horizontal (Multi-node).	Horizontal + GPU Acceleration.
Complexity	Zero-config.	Requires Spark endpoint.	Requires GPU-enabled nodes.

次のページに続く

表 1 – 前のページからの続き

Feature	Seafowl	Spark Connect (CPU)	Spark + RAPIDS (GPU)
Cost efficiency	Best for datasets < 1 TB.	Best for datasets between 1 TB and 3 TB.	Best for datasets 3 TB or higher.

Seafowl はシングルノードのパフォーマンス用に高度に最適化されていますが、次の場合は Spark ベースのエグゼキューターへの切り替えを検討します。

- **メモリ制約** クエリーが `pgaa.autostart_seafowl_max_memory_mb` 制限を超えています。
- **メンテナンスが多い** Iceberg テーブルで大規模なコンパクションまたは Z-オーダーリングなどのリソースを大量に使用する操作を実行しています。
- **極端なスケール** 外部クラスターの分散コンピューティングを必要とするデータセット全体の結合と集計を処理する必要があります。

9.2.1 Distributed Spark execution

Postgres Analytics AcceleratorPGAA を Apache Spark と統合して、Postgres をマルチテラバイトのデータセットの CPU ボトルネックを克服する高性能プラットフォームに変換します。

PGAA を介してクエリ実行を Apache Spark にオフロードすると、アーキテクチャの利点がすぐに得られます。

- **意図的なハンドオフ** 実行エンジンを明示的に選択することにより、完全な制御を維持します。PGAA は次に、SQL クエリーをパッケージ化して、Spark Connect エンドポイントにルーティングします。
- **分散処理** ワークロードを Spark クラスターに移動することにより、分散コンピューティング能力を活用して、単一のデータベースインスタンスをボトルネックにしてしまう大規模なデータセットを処理します。
- **ゼロインデックス** 手動インデックス作成や定期的なメンテナンスを必要とせずにピークの分析パフォーマンスを達成し、操作オーバーヘッドを大幅に削減します。

!!!注

現在、Apache Spark との統合は以下をサポートしています。

- S3 互換オブジェクトストレージまたは共有 POSIX ファイルシステムの Parquet ファイルの読み取り専用クエリー。
- Iceberg REST カタログの Iceberg テーブルの読み取り専用クエリー OAuth2 認証はサポートされていません。

1. *Spark* 環境の構成
2. *Postgres* を *Spark* に接続する
3. ソースデータの構成
4. 分析クエリの実行

5. クエリプランの検査

Spark 環境の構成

3. 4以降を実行している Spark クラスターインストールを利用またはインストールできます。

次の依存関係で構成した実行中の Spark Connect サーバーが必要です。

```
org.apache.spark:spark-connect_2.12:3.5.6,\
io.delta:delta-spark_2.12:3.3.1,\
org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.9.2,\
org.apache.iceberg:iceberg-aws-bundle:1.9.2,\
org.apache.hadoop:hadoop-aws:3.3.4
```

!!!重要

Iceberg Spark ライブラリーバージョン 1.9.2 には、Spark を使用して Iceberg テーブルから読み取る場合に既知の問題があります。特定の条件下では、同時実行中に等価削除がスキップされる場合があります。

現在の回避策は、`spark-defaults.conf` ファイルで Spark アプリケーション設定 `spark.sql.iceberg.executor-cache.enabled` を無効にすることです。

このキャッシュを無効にすると、すべての削除が正しく処理されるため、データの一貫性が保証されますが、大容量の読み取りワークロードの場合、パフォーマンスに影響を与える可能性があります。

1. 実行エンジンを設定し、Postgres セッションで Spark Connect URL を定義します。

```
SET pgaa.executor_engine = spark_connect;
SET pgaa.spark_connect_url = sc://spark-connect:15002;
```

`spark-connect` は、Spark Connect サービスアドレスを指します。

2. PGAA インターフェイスを介して単純なバージョンチェックを実行することにより、接続を検証します。

```
SELECT pgaa.spark_sql(SELECT version());
```

成功した場合、コマンドは Spark クラスターのバージョン文字列を結果ます。

ソースデータの構成

PGAA は、データの保存場所として S3 互換ストレージおよびローカルファイルシステムからの Parquet ファイルの読み取り、または Iceberg REST カタログの Iceberg テーブルの読み取りをサポートしています。

オプション 1 Parquet ファイルからの読み取り

1. Postgres データベースに、分析データが含まれるバケットを指す PGFS storage location を作成します。
例、パブリックバケットの場合

```
SELECT pgfs.create_storage_location(
my-sample-data,
s3://beacon-analytics-demo-data-us-east-1-prod,
{"skip_signature": "true", "region": "us-east-1"}
);
```

2. PGAA アクセス方法を使用してテーブルを作成します。前の手順で作成した保存場所、単一の Parquet ファイルまたは複数の Parquet ファイルを含むディレクトリへのパス、およびフォーマット現在 Parquet を指定します。例

```
CREATE TABLE customer () USING PGAA WITH (pgaa.storage_location = my-sample-data,
→pgaa.path = tpch_sf_1/customer, pgaa.format = parquet);
CREATE TABLE orders () USING PGAA WITH (pgaa.storage_location = my-sample-data, pgaa.
→path = tpch_sf_1/orders, pgaa.format = parquet);
CREATE TABLE lineitem () USING PGAA WITH (pgaa.storage_location = my-sample-data,
→pgaa.path = tpch_sf_1/lineitem, pgaa.format = parquet);
CREATE TABLE nation () USING PGAA WITH (pgaa.storage_location = my-sample-data, pgaa.
→path = tpch_sf_1/nation, pgaa.format = parquet);
```

複数の Parquet ファイルを含むディレクトリを指定すると、PGAA はすべてのファイルを処理のために単一のテーブルに自動的に結合します。

オプション 2 Iceberg テーブルからの読み取り

1. カタログ接続を構成します。例

```
SELECT pgaa.add_catalog(
my_iceberg_catalog,
iceberg-rest,
{
"url": "https://iceberg.example.com/api/v1",
"warehouse": "a1b2c3d4-e5f6-g7h8",
"token": "secret_token_abc123"
}
);
```

2. 必要なカタログ管理テーブルを作成します。または、カタログを添付またはインポートします。

```
CREATE TABLE customer () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
→iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = customer);
CREATE TABLE orders () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
→iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = orders);
CREATE TABLE lineitem () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
→iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = lineitem);
CREATE TABLE nation () USING PGAA WITH (pgaa.format = iceberg,pgaa.managed_by = my_
→iceberg_catalog,pgaa.catalog_namespace = public,pgaa.catalog_table = nation);
```

Iceberg カタログを使用してテーブルへのアクセスを構成する方法の詳細については、[Integrating with Iceberg catalogs](#) を参照してください。

分析クエリの実行

この例のクエリーは、特定の四半期内に最高の収益を達成した顧客を特定することにより、トップ 20 収益レポートを生成します。これを Spark にオフロードすることにより、PGAA は複雑な条件付き集計を並列化し、述語プッシュダウンを利用してオブジェクトストレージからのデータ転送を最小限に抑えます。

```
SELECT
    c_custkey,
    c_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
FROM
    customer,
    orders,
    lineitem,
    nation
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate >= 1993-10-01
    AND o_orderdate < 1994-01-01
    AND l_returnflag = R
    AND c_nationkey = n_nationkey
GROUP BY
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
ORDER BY
    revenue DESC,
    c_custkey
LIMIT 20;
```

クエリプランの検査

Spark が有効になっていない場合、PGAA はローカルベクトル化実行のためにデフォルトの Seafoal エンジンを使用します。EXPLAIN をステートメントの前に追加して、ローカルのシングルノード実行から Spark Connect を介したリモートの分散処理モデルへの移行を確認します。

これらの 2 つのクエリプランを比較すると、ローカルのシングルノード実行から Spark Connect を介したリモートの分散処理モデルへの移行が強調表示されます。

- **結合戦略** Seafoal は、データベースに行をシークエンシャルに比較することを強制する「ネストループ」アプローチ Cross Join + Filter を使用します。Spark はこれを BroadcastHashJoin にアップグレードします。ここでは、小さなテーブルがすべてのワーカーノードに送信されて、大きなファクトテーブルを並列処理します。
- **集計ロジック** シングルノードシークエンシャル aggregate の代わりに、Spark は 2 ステージ HashAggregate を使用します。これは、個々のワーカーノードで partial_sum を計算し、Exchange シャッフルを実行して結果を結合し、大規模データのホリゾンタルスケーラビリティを提供します。
- **データアクセス** Seafoal は、使用可能なすべてのデータに対して標準の TableScan を実行します。Spark は、Iceberg REST カタログを介してメタデータを認識する BatchScan を使用します。これは、Iceberg マニフェストファイルを読み取り、S3 から取得される前に無関係なデータブロックを完全にスキップします。

Spark プランは、大規模なデータセットのクラスター全体で水平にスケーリングするように設計された洗練された分散ブループリントです。逆に、Seafoal プランは、ネットワークレイテンシーとクラスター調整のオーバーヘッドが実際の処理時間を上回る場合に、小規模なデータセット用に最適化された高速のローカライズされたブループリントです。

9.2.2 GPU-accelerated Spark with NVIDIA RAPIDS

NVIDIA RAPIDS を Spark に接続した Postgres Analytics Accelerator PGAA 環境と統合して、データレイクハウスのハードウェアレベルのアクセラレーションを実現します。Spark はクラスター全体の分散ワークロードをオーケストレーションしますが、RAPIDS は集中的な計算タスクを GPU にオフロードし、大規模なデータセットの複雑なクエリの大幅なスループットの向上を提供します。

この GPU 高速化アプローチにより、PGAA は、次のコア機能を介して標準の分散処理を超越できます。

- **大並列実行** 何千もの NVIDIA GPU コアが列状データを同時に処理し、標準の Postgres テーブルより最大 100 倍の速度で複雑なクエリを完了します。
- **ハードウェアアクセラレーションによるスケーリング** Spark は 100 GB でディストリビューションを管理しますが、RAPIDS は 3 TB 以上でプライマリパフォーマンスドライバーとなり、データの増加に応じてハードウェア効率のギャップが拡大します。
- **ゼロオーバーヘッドパフォーマンス** メモリ内の列型データを直接操作し、手動のインデックス作成、バキューム処理、または従来のデータベースチューニングなしでピークパフォーマンスを提供します。
- **インテリジェントフォールバック** ユーザーの介入なしで、サポートされていない操作を Spark CPU または Postgres カーネルに自動的にシフトして戻すことにより、クエリの完了を保証します。

!!!注

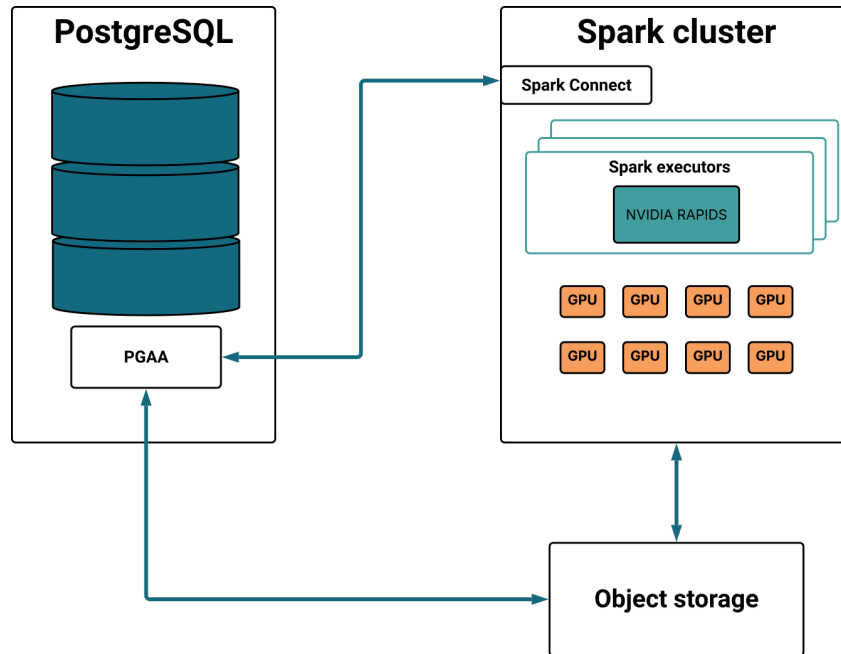


図 1: rapids_arch

現在、Apache Spark による GPU アクセラレーションは、S3 互換オブジェクトストレージまたは共有 POSIX ファイルシステムの Parquet ファイルの読み取り専用クエリに対して最適化されています。Iceberg 読み取り/書き込みのサポートが間もなく登場します。

コアアーキテクチャは、PGAA 拡張機能を備えた Postgres サーバー、NVIDIA RAPIDS で構成された Apache Spark クラスタ、および Parquet 形式のデータを含む互換性のあるオブジェクトストレージで構成されています。

この環境を実装するには、既存のインフラストラクチャを活用するか、手動インストールを実行するか、[ready-to-run Docker Compose stack](#) を使用して必要なすべてのコンポーネントをすばやく展開できます。このスタックには、NVIDIA Brev プラットフォーム用に最適化された事前構成された例が含まれており、クラウドにセルフサービスの GPU インスタンスを簡単に展開できます。互換性のある GPU が検出された場合、NVIDIA RAPIDS は自動的にハードウェアアクセラレーションを提供します。それ以外の場合、システムは透過的に CPU ベースの処理にフォールバックします。

NVIDIA RAPIDS を使用した PGAA の構成は、次の手順で構成されます。

1. **前提条件** を確認します。
2. **NVIDIA RAPIDS の有効化** が Spark クラスタ内で適切にインストールされ、アクティブであることを確認します。
3. **Postgres** を **Spark** に接続する。
4. **ソースデータの構成**。
5. **分析クエリの実行** し、ワークロードが GPU アクセラレーションを利用していることを確認します。

前提条件

- **Postgres PGAA** および **PGFS** 拡張機能がインストールされたバージョン 16 以降。
- ストレージ場所現在、S3 互換ストレージとローカルファイルシステムのみがデータ保存場所としてサポートされています。
- **Apache Spark** バージョン 3.4 以降。
- **Spark Connect** 次の依存関係で構成された実行中の **Spark Connect** サーバー。

```
org.apache.spark:spark-connect_2.12:3.5.6,\
io.delta:delta-spark_2.12:3.3.1,\
org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.9.2,\
org.apache.iceberg:iceberg-aws-bundle:1.9.2,\
org.apache.hadoop:hadoop-aws:3.3.4
```

- **NVIDIA RAPIDS** バージョン 2025.10 以降が Spark クラスターにインストールされています。詳細は、[NVIDIA RAPIDS の有効化](#) を参照してください。
- ハードウェア NVIDIA GPU を搭載したマシンへのアクセス。

NVIDIA RAPIDS の有効化

GPU アクセラレーションを利用するには、Apache Spark クラスター内に NVIDIA RAPIDS を正しくインストールしてアクティブ化する必要があります。次の統合ガイドラインに従って、ハードウェアアクセラレーションをサポートするように Spark 環境を構成します。

1. **RAPIDS JAR** のダウンロード Spark 環境が必要な

RAPIDS JAR files をダウンロードすることを確認します。ジョブを投入するときにプラグインバージョン **com.rapids-4-spark_2.12:25.10.0** を指定します。

2. **GPU Discovery:** Spark が使用可能な GPU を識別できるようにする

Install the GPU discovery script。

3. **Spark 構成を適用します。** 少なくとも次のパラメーターを有効にします。最適化設定の完全なリストについては、

[RAPIDS configuration](#) および [Tuning guide](#) を参照してください。

```
spark.rapids.sql.enabled=true
spark.rapids.filecache.enabled=true
spark.executor.resource.gpu.amount=1
spark.plugins=com.nvidia.spark.SQLPlugin
spark.shuffle.manager=com.nvidia.spark.rapids.spark356.RapidsShuffleManager
```

4. **コンテナプロビジョニング Docker** を介して Spark を展開する場合、コンテナマニフェストを変更してホスト GPU への Spark アクセスを許可します。

上記の手順-検出スクリプトのボリュームマウント、環境レベルの Spark 構成の設定、および NVIDIA コンテナランタイムの構成を含む-を実装する完全な例については、サンプル

`docker-compose.yaml` を参照してください。

Postgres を Spark に接続する

1. Postgres のターミナルから、実行エンジンとして Spark を使用するように PGAA を設定し、Spark Connect エンドポイントを定義します。

```
SET pgaa.executor_engine = spark_connect;
SET pgaa.spark_connect_url = sc://spark-connect:15002;
```

`spark-connect` は、Spark Connect サービスアドレスを指します。

2. PGAA インターフェイスを介して簡単なバージョンチェックを実行することにより、Postgres が Spark クラスターと通信できることを確認します。

```
SELECT pgaa.spark_sql(SELECT version());
```

成功した場合、コマンドは Spark クラスターのバージョン文字列を結果ます。

ソースデータの構成

1. Postgres データベースに、分析データが含まれるバケットを指す PGFS storage location を作成します。
例、パブリックバケットの場合

```
SELECT pgfs.create_storage_location(
my-sample-data,
s3://beacon-analytics-demo-data-us-east-1-prod,
{"skip_signature": "true", "region": "us-east-1"}
);
```

2. PGAA アクセス方法を使用してテーブルを作成します。前の手順で作成した保存場所、単一の Parquet ファイルまたは複数の Parquet ファイルを含むディレクトリへのパス、およびフォーマット現在 Parquet を指定します。例

```
CREATE TABLE store () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/store, pgaa.
→format = parquet);
CREATE TABLE customer () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/customer,
→pgaa.format = parquet);
CREATE TABLE date_dim () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/date_dim,
→pgaa.format = parquet);
CREATE TABLE store_returns () USING PGAA
  WITH (pgaa.storage_location = my-sample-data, pgaa.path = tpcds_sf_10/store_
→returns, pgaa.format = parquet);
```

複数の Parquet ファイルを含むディレクトリを指定すると、PGAA はすべてのファイルを処理のために単一のテーブルに自動的に結合します。

分析クエリの実行

定義したテーブルに対して大規模な分析クエリを実行して、GPU アクセラレーションを正常に活用していることを確認します。

クエリは Spark クラスターにオフロードされ、NVIDIA RAPIDS がアクティブな場合、結合および集計操作は数千の GPU コアにわたって並列実行されます。

次の例では、TPC-DS からインスピレーションを得た分析クエリを実行し、店舗の合計収益を地域平均と比較することにより、高収益の顧客を特定します。これには、複雑な共通テーブル式 CTE、関連サブクエリー、およびマルチステージ集計が含まれます。正確には、ベクトル化された実行を通じて PGAA が加速する CPU 集中型のワークロードのタイプです。

```
WITH customer_total_return AS (
    SELECT
        sr_customer_sk AS ctr_customer_sk,
        sr_store_sk AS ctr_store_sk,
        SUM(sr_return_amt) AS ctr_total_return
    FROM store_returns, date_dim
    WHERE sr_returned_date_sk = d_date_sk
        AND d_year = 2000
    GROUP BY sr_customer_sk, sr_store_sk
)
SELECT c_customer_id
FROM customer_total_return ctr1, store, customer
WHERE ctr1.ctr_total_return > (
    SELECT AVG(ctr_total_return) * 1.2
    FROM customer_total_return ctr2
    WHERE ctr1.ctr_store_sk = ctr2.ctr_store_sk
)
AND s_store_sk = ctr1.ctr_store_sk
AND s_state = TN
AND ctr1.ctr_customer_sk = c_customer_sk
ORDER BY c_customer_id
LIMIT 100;
```

NVIDIA RAPIDS がクエリをアクティブに高速化していることを確認するには、Spark Connect UI を介して実行プランをモニタします。

1. `http://<spark_driver_host>:4040` で Spark UI にアクセスします。
2. [SQL] タブに移動し、最近実行したクエリーを選択します。
3. 計画を確認します。RAPIDS がアクティブな場合、クエリノードには **Gpu** の接頭辞が付けられます。例、GpuFilter、GpuBatchScan、GpuUnion など。

クエリプランを超えて、Spark UI 内で GPU 固有のパフォーマンスメトリックを直接分析し、潜在的なボトルネックを特定できます。これらのメトリックの詳細な内訳とそれらを最適化する方法については、

[NVIDIA tuning guide](#) を参照してください。

第10章 Monitoring and maintaining analytical tables

Postgres Analytics Accelerator PGAA は、Postgres プロンプトから直接ストレージ使用率、テーブルフォーマット、およびデータレイクテーブルのメタデータの状態を監査するのに役立つビルトイン機能を提供します。

10.1 テーブル統計の分析

数百の標準の Postgres テーブルがある環境では、PGAA によって管理されているテーブルを特定することが困難になる場合があります。pgaa.list_analytics_tables() ファンクションは、すべてのレイクハウス統合テーブルのグローバル概要を提供します。

```
SELECT * FROM pgaa.list_analytics_tables();
```

ファンクション出力は、次の洞察を提供します。

- **ID とフォーマットスキーマ**、テーブル名、およびストレージフォーマット Iceberg、Delta、または Parquet などを表示します。
- **ストレージの場所** データファイルが存在する場所への正確なパス。
- **カタログのメタデータ** 外部カタログを使用する場合、このファンクション出力には外部カタログサービス名、名前空間、およびリモートテーブル名が表示されます。
- **レプリケーションステータス** データ移動の現在の状態を表示します PGD レプリケーションシナリオの場合のみ。
- **ストレージメトリック**
 - object_storage_snapshot_size_bytes 最新のライブスナップショットのサイズ。これは、SELECT * 中にスキャンされたデータの実際の量を表します。
 - object_storage_total_size_bytes 現在のバージョンで使用されていないメタデータ、マニフェスト、および履歴データファイルを含むバケット内の合計フットプリント。
 - object_storage_total_size_bytes メタデータを含むバケット内の合計フットプリント、マニフェスト、および履歴データファイルは使用されません。

pgaa.lakehouse_table_stats() ファンクションは、テーブルの物理的特性に関する特定の情報を提供します。これは、キャパシティプランニングとクエリパフォーマンスの微調整に重要です。

```
SELECT * FROM pgaa.lakehouse_table_stats(my_schema.my_table::regclass);
```

このファンクションは、基になるオブジェクトストレージを照会して、特定のマッピングの「内部」で何が起こっているのかを確認します。それはあなたに役立ちます

- **ライブデータとデッドデータを区別する**古いトランザクションログまたは圧縮されていない小さなファイルによってストレージが「無駄」になっている量を正確に特定します。
- **成長の監視**新しいデータが外部ソースから取り込まれ、スナップショットサイズが時間の経過とともにどのように進化するかを追跡します。
- **キャッシュ認識**このファンクションの結果は、`pgaa.lakehouse_table_stats_cache_ttl_s` 構成パラメーターによって制御されます。絶対的な最新の統計が必要であるのに表示されない場合は、キャッシュ TTL が 0 に設定されていることを確認します。

`latest_snapshot_size` と `total_size` を比較することは、パフォーマンスとコストのバランスを保つために重要です。スナップショットのサイズは、「ライブ」データ、クエリー中にスキャンされる実際のボリュームを表しますが、合計サイズには、履歴バージョン、トランザクションログ、非圧縮行などのパケットフットプリント全体が含まれます。

Iceberg や Delta Lake のような形式では、合計サイズがスナップショットのサイズを超えて、タイムトラベルとロールバックをサポートします。これらの値の大きなギャップは、履歴オーバーヘッドの蓄積を示し、**VACUUM** またはコンパクションを実行する信号として機能します。これらのメトリックを監視することは、無駄のない高パフォーマンスのデータレイクを維持するのに役立ちます。

10.2 テーブルパフォーマンスの最適化

分析パフォーマンスは、頻繁なデータの更新により何千もの小さなファイルが作成され、メタデータのオーバーヘッドが増加する「小さいファイル問題」によって大きな影響を受けます。

10.2.1 pgaa.launch_task を使用したテーブルメンテナンスの実行

Delta テーブルと Iceberg テーブルのメンテナンスは、`pgaa.launch_task()` ファンクションを使用してネイティブバックグラウンドワーカーによって非同期に処理されます。これにより、アクティブな SQL セッションをブロックせずに、負荷の高い操作をトリガーできます。

```
SELECT pgaa.launch_task(
  table_name => my_delta_table::regclass,
  task_type => compaction, -- or z-order, vacuum, purge
  options => {"target_size": "128MB"}
);
```

注釈

Delta Lake は使用可能なすべてのタスクタイプをサポートしていますが、Iceberg テーブルは現在 `compaction` オペレーションのみをサポートしています。さらに、Iceberg コンパクションは、`dry_run` を除き、Delta Lake で使用可能な詳細な構成キーをサポートしていません。

サポートされているタスクタイプは次のとおりです。

- **コンパクション** 小さなファイルをより大きな最適化されたブロック `target_size` で定義にマージして、分析スキャンを高速化します。
- **Z-オーダーデルタのみ** 複数の列にわたってデータを再編成して、関連する値が物理的に近くに保存されます。これにより、エンジンがクエリフィルタと一致しないファイル全体を無視するデータスキップが有効になります。
- **バキューム Delta のみ** アクティブなスナップショットに関連付けられなくなり、指定された保持期間より古いデータファイルを削除します。これにより、オブジェクトストレージ領域が再利用されます。
- **パージデルタのみ** 特定のストレージパスからデータを明示的に削除する管理ツール、失敗したエクスポートの手動クリーンアップに役立ちます。

例

- Delta Lake テーブルに対するコンパクションを使用して、小さなファイルをより大きな 256MB ブロックにマージして、スキャンパフォーマンスを高速化します。

```
SELECT pgaa.launch_task(
  table_name => lakehouse.sales_data::regclass,
  task_type  => compaction,
  options    => {"target_size": "256MB"}
);
```

- z オーダーを使用してデータを物理的に再編成して、`customer_id` および `region` の同様の値を持つ行が一緒に保存され、エンジンが無関係なファイルをスキップできるようにします。

```
SELECT pgaa.launch_task(
  table_name => lakehouse.sales_data::regclass,
  task_type  => z-order,
  options    => {"columns": ["customer_id", "region"]}
);
```

- バキュームを使用して、アクティブテーブル状態の一部でなくなり、7 日より古いファイルを削除して、記憶領域を再利用します。

```
SELECT pgaa.launch_task(
  table_name => lakehouse.sales_data::regclass,
  task_type  => vacuum,
  options    => {"retention_period": "7 days"}
);
```

サポートされているオプションと例の完全なリストについては、`pgaa.launch_task()` のファンクションリファレンスセクションを参照してください。

10.2.2 高度なメンテナンス

ネイティブバックグラウンドワーカーでまだサポートされていない複雑なメンテナンス操作の場合、次の機能を使用します。

`pgaa.spark_sql()`

構成済みの Spark Connect エンドポイントを介して、Postgres で使用できない Iceberg 圧縮ルーチンまたは Spark ファンクションを実行します。使用可能なプロシージャのリストについては、[Spark procedures](#) を参照してください。

```
SELECT pgaa.spark_sql(query, catalog_name);
```

注釈

このファンクションを使用するには、`pgaa.spark_connect_url` 構成パラメーターを設定して、使用可能な Spark Connect サービスをポイントする必要があります。

`pgaa.execute_compaction()` 非推奨

`pgaa.execute_compaction()` ファンクションは、断片化したデータファイルをより大きくて効率的な Parquet ファイルに統合し、分析 Iceberg または Delta テーブルのパフォーマンスとストレージ効率を向上させます。

```
SELECT pgaa.execute_compaction(my_iceberg_table::regclass);
```

注釈

ファンクション `pgaa.execute_compaction()` は非推奨で、`pgaa.launch_task()` を優先します。

例

- 標準のテーブル圧縮を使用して、断片化したデータファイルをより大きなファイルにマージし、Iceberg マニフェストを更新します。

```
SELECT pgaa.execute_compaction(lakehouse.inventory_iceberg::regclass);
```

- `pgaa.spark_sql()` を使用して、`rewrite_data_files` Spark タスクを介してメタデータのオーバーヘッドを削減します。

```
SELECT pgaa.spark_sql($${
  CALL preexisting.system.rewrite_data_files(
    table => "preexisting"."ns-1"."table-1",
    strategy => sort,
    sort_order => value DESC,
    options => map(rewrite-all, true)
  )
})
```

(次のページに続く)

(前のページからの続き)

```
)  
$$);
```

サポートされているオプションと例の完全なリストについては、[Functions](#) を参照してください。

第11章 Optimizing query performance

Postgres Analytics AcceleratorPGAA は、分析ワークロードを Seafoal と、大規模向けに設計されたベクトル化実行エンジンである Apache Spark にオフロードすることにより、高いパフォーマンスを実現します。PGAA が実行パスを選択する方法を理解することにより、クエリを「高速パス」に維持して、最大の高速化を実現できます。

11.1 スキャンタイプを理解する

分析テーブルを照会すると、PGAA は 2 つの異なる実行パスを介してデータの取得と処理を管理します。

DirectScan Seafoal は、オブジェクトストレージからデータを直接読み取り、ソースですべてのフィルタと集計を適用し、最終的な結果セットのみを Postgres に返します。これにより、ベクトル化された加速を完全に利用し、ネットワークトラフィックを最小限に抑えることにより、パフォーマンスを最大化します。

CompatScan クエリに Seafoal がネイティブで処理できない SQL 機能またはカスタムファンクションが含まれている場合、システムはハイブリッドモードにフォールバックします。このシナリオでは、Seafoal は引き続きデータソースとして機能しますオブジェクトストレージからデータをストリーミングし、基本的なフィルタリングを実行しますが、Postgres は最終的な複雑な処理のオーケストレーターとして機能します。

CompatScan では、ワークロードは次のように分割されます。

- Seafoal はファイルを読み取り、許可されたプッシュダウンを処理します結合またはグループバイなど。
- Postgres は Seafoal から最適化されたストリームを受け取り、特殊な機能、最終的な並べ替え、またはベクトル化エンジンがまだサポートしていない複雑なロジックを実行します。

DirectScan は最大速度の目標ですが、CompatScan は信頼できるフォールバックとして機能し、一部のパフォーマンスを犠牲にして Postgres 機能の完全な互換性を保証します。

11.2 クエリプランの分析

PGAA がクエリーを実行している方法を確認するには、EXPLAIN コマンドを使用します。出力により、どのエンジンが重い作業を実行しているかがわかります。

- **DirectScan** SeafoalDirectScan または SparkDirectScan ノードを探します。これは、Seafoal が操作全体フィルタリング、結合、および集計を処理し、最終結果のみを Postgres に返すことを示しています。
- **CompatScan** SeafoalCompatScan または SparkCompatScan ノードが表示される場合、クエリは「ハイブリッド」モードで実行されています。Postgres は最終的な実行をオーケストレーションして、完全な機能の互換性を保証しますが、プッシュダウン設定が有効になっている場合、リモートエンジンはクエリの一部を高速化できます。

11.3 DirectScan フォールバックの管理

クエリがより遅い CompatScan パスにフォールバックする場合は、次の手順に従ってトラブルシューティングを行ってください。

1. 現在のセッションで DirectScan が許可されていることを確認します。

```
SHOW pgaa.enable_direct_scan;  
-- If off, enable it:  
SET pgaa.enable_direct_scan=on;
```

2. 強制エラー報告 障害動作を変更しますフォールバックの原因を明らかにできます。

```
SET pgaa.direct_scan_fail_behavior = error;
```

Postgres は、互換性のない SQL 構文またはデータ型を識別する明示的なエラーをスローします。

3. クエリが複雑な場合は、句を 1 つずつ削除して、ハイブリッドモードを強制する特定の「互換性トリガー」を分離します。

11.4 コンピューティングプッシュダウンの構成

ハイブリッドモードでも、リソースを大量に使用する操作をオフロードすることにより、高いパフォーマンスを維持できます。次の設定はデフォルトで有効になっており、パフォーマンスのセーフティネットとして機能します。

pgaa.enable_join_pushdown システムがリモート実行者 Seafoal または Spark への参加をオフロードしようとすることを承認します。データが存在する場所でデータを結合することにより、大規模な生データセットを Postgres メモリにプルすることを回避し、ネットワークトラフィックを大幅に削減します。

pgaa.enable_groupby_pushdown 集約 SUM、COUNT、AVG および GROUP BY 句のオフロードを承認します。ソースで 10 億行を 10 のグループに集約すると、データ転送が数桁削減され、帯域幅と Postgres CPU の両方が節約されます。

第12章 Reference

リファレンスセクションでは、Postgres Analytics AcceleratorPGAA の詳細な技術仕様を提供します。これらのリソースを使用して、構成を微調整し、サポートされているデータ型を理解し、分析テーブルを管理します。

保存場所を管理するための PGFS 機能については、[AI Factory PGFS documentation](#) を参照してください。

- *Configuration parameters* PGAA 拡張機能の動作は、Grand Unified ConfigurationGUC 変数によって制御されます。これらのパラメーターを使用すると、エグゼキューターを切り替え、パフォーマンスの最適化を有効にし、セキュリティ資格情報を管理できます。
- *Functions* PGAA は、新しいテーブルのマッピング、ストレージの状態の監視、メンテナンスバックグラウンドジョブの起動などの管理タスクのための SQL ファンクションスイートを導入しています。
- *Table options* 分析テーブルをマッピングまたは作成する場合、特定のオプションを使用して、オブジェクトストアからデータを読み取るまたは書き込む方法を定義できます。
- *Data types and definitions* PGAA は、ネイティブの Postgres データ型をデータレイクの最適化されたカラムナー形式にマップします。
- *Benchmark datasets* 分析データセットの事前構成されたスキーマとデータのロード指示にアクセスして、パフォーマンスのベースラインを設定します。

12.1 Configuration parameters

このリファレンスガイドでは、Postgres Analytics AcceleratorPGAA 拡張機能が提供する構成パラメーターについて説明します。これにより、実行エンジンの調整、リモート接続の管理、およびデータレイク全体のクエリパフォーマンスの最適化ができます。

12.1.1 エグゼキューターエンジン

`pgaa.executor_engine`

分析操作の実行を担当するコンピューティングバックエンドを決定します。

サポートされている値は `seafowl`、および `spark_connect` です。

デフォルトは `seafowl` です。

pgaa.spark_connect_url

Spark Connect サービスの接続エンドポイントを定義します。リモート Spark Connect サービスの URL を指定します。pgaa.executor_engine が spark_connect に設定されている場合、この設定は必須です。

値は、特定の Spark Connect URI 形式 `sc://<host>:<port>` に従う必要があります。

デフォルトは NULL です。

pgaa.spark_connect_extra_config

pgaa.executor_engine= 'spark_connect' を使用する場合、詳細な調整パラメーターとランタイム設定を JSON オブジェクトとして Spark セッションに直接提供します。オプションのリストについては、[Spark Connect configuration](#) を参照してください。

例

```
SET pgaa.spark_connect_extra_config = {  
  "spark.sql.execution.arrow.maxRecordsPerBatch": "65536"  
};
```

デフォルトは NULL です。

12.1.2 クエリー動作

pgaa.enable_direct_scan

分析テーブルのクエリ実行戦略を制御します。PGAA がクエリ全体を最適化された Seafoal エンジンにオフロードしようとするか、標準の Postgres 互換性レイヤーを使用して処理するかどうかを決定します。

デフォルトは on で、これは、PGAA が CompatScan にフェイルオーバーする前に、最初に DirectScan を試行することを意味します。

pgaa.direct_scan_fail_behavior

ダイレクトスキャンが試行されたが、サポートされていない SQL 機能または構文のために完了できなかった場合のシステムの反応を定義します。

サポートされている値は次のとおりです。

- **ignore** システムはサイレントに CompatScan にフォールバックします。
- **warn** デフォルトクエリは互換性フォールバックを使用して実行されますが、Postgres は DirectScan が失敗したことを示す警告通知を発行します。
- **error** DirectScan を使用できない場合、クエリの実行は完全に中止されます。

DirectScan が要求されたが失敗した場合の動作、ignore、warn、error のいずれか。デフォルトは warn です。

pgaa.enable_join_pushdown

CompatScan では、分析テーブル間の結合操作の最適化戦略を制御します。結合を Postgres 内でローカルに処理するか、リモート実行エンジン Seafowl または Spark にオフロードするかどうかを決定します。

デフォルトは on です。

pgaa.enable_groupby_pushdown

CompatScan では、集計操作 SUM、COUNT、AVG、GROUP BY などをリモート実行エンジン Seafowl または Spark にオフロードしようとします。

デフォルトは on です。

pgaa.ctas_intermediate_flush_size

PGAA 管理テーブルへの大規模な INSERT または CREATE TABLE AS (CTAS) オペレーション中のバッチ動作を制御します。

デフォルト 8192 です。

pgaa.flush_task_interval_s

PGAA バックグラウンドプロセスが永続ストレージへの分析データの保留中のフラッシュをチェックして実行する頻度を秒単位で指定します。

デフォルト 5 です。

pgaa.max_in_memory_mb

ストレージへの強制的なフラッシュがトリガーされる前に、ライタがデータのバッファーに利用できるメモリの最大量をメガバイト単位で設定します。

デフォルトは 3072MB です。

pgaa.scan_aggregation_cost_factor

ローカルの Postgres 実行と比較して、リモートバックエンド Spark または Seafowl で集計を実行するコストを重み付けるために Postgres クエリプランナーが使用する乗数。

デフォルトは 0.005 です。

pgaa.scan_per_tuple_cost

ローカルの Postgres 実行と比較して、リモートバックエンド Spark または Seafowl で単一行タプルをスキャンするコストを重み付けるために Postgres クエリプランナーが使用する乗数。

デフォルト 0.02 です。

pgaa.scan_startup_cost

ローカルの Postgres 実行と比較して、リモートバックエンド Spark または Seafoal でスキャンを開始する固定「スタートアップ」コストを重み付けるために Postgres クエリプランナーが使用する乗数。

デフォルト 25 です。

pgaa.use_seafoal_cost_estimates

Postgres クエリプランナーが、Postgres によって内部収集された統計に依存するのではなく、Seafoal エンジンから直接リアルタイムのコスト推定を要求するかどうかを決定します。

デフォルトは on です。

pgaa.write_lock_timeout_s

タイムアウトして操作を中止する前に、プロセスが書き込みロックの取得を待機する最大時間を秒単位で指定します。

デフォルト 3 です。

12.1.3 Seafoal

pgaa.autostart_seafoal

バックグラウンドワーカーによって管理される Seafoal プロセスの自動開始を制御します。

デフォルトは on です。

pgaa.autostart_seafoal_max_memory_mb

組み込み Seafoal エンジンが消費できる RAM の量の上限をメガバイト単位で設定し、分析エンジンがメインの Postgres プロセスのリソースを枯渇させないようにします。

デフォルトは 2147483647 です。

pgaa.autostart_seafoal_enable_metrics

標準の可観測性スタックを使用して分析エンジンの状態、クエリスループット、およびリソース使用率をモニタするリアルタイムパフォーマンスメトリックを提供する専用の HTTP エンドポイントを開くことにより、Seafoal プロセスで Prometheus メトリックを有効にします。

デフォルト値は on です。

pgaa.autostart_seafowl_metrics_host

メトリックエンドポイントのバインドホスト。

pgaa.autostart_seafowl_enable_metrics= on の場合、Prometheus メトリックサービスが到達可能なホスト名または IP アドレスを指定します。

デフォルトは 127.0.0.1 で、これはセキュリティのためのローカルループバックをポイントし、メトリックがデフォルトで公開されないことを保証します。

pgaa.autostart_seafowl_metrics_port

pgaa.autostart_seafowl_enable_metrics= on のときに Prometheus メトリックサービスに使用される特定のネットワークポートを定義します。

サポートされている値の範囲は 1024～65535 です。デフォルトは 9090 です。

pgaa.autostart_seafowl_enable_object_store_cache

S3、Azure、または GCP の外部ストレージの場所の Seafowl オブジェクトストアキャッシュを有効にするかどうかを決定します。

デフォルトは on です。

pgaa.autostart_seafowl_log_level

バックグラウンド Seafowl プロセスのロギングの詳細度を指定します。

サポートされている値には、error、warn、info、debug、および trace が含まれます。

デフォルトは info です。

pgaa.autostart_seafowl_path

Seafowl 実行可能ファイルへの絶対ファイルパスを指定します。これにより、バックグラウンド分析エンジンを起動するときにどのバイナリを起動するかを PGAA スーパーバイザに正確に指示します。このパラメーターの変更は、別のマシンまたはコンテナで Seafowl サービスを実行しているカスタム構成の場合にのみ必要です。

デフォルトは空の文字列です。

pgaa.autostart_seafowl_port

バックグラウンド Seafowl プロセスがリッスンする TCP ポートを指定します。PGAA 拡張機能はこのポートを使用して、ローカル実行エンジンと通信します。

デフォルトは 47470 です。

pgaa.autostart_seafowl_tmp_dir

クエリの実行中にバックグラウンドの Seafoal プロセスが一時ファイルを保存するディレクトリを指定します。

デフォルトは空の文字列です。

pgaa.enable_maintenance_worker

PGAA バックグラウンドスーパバイザが専用のメンテナンswーカープロセスを起動および管理するかどうかを制御します。

デフォルトは off です。

pgaa.maintenance_worker_sleep_interval

現在実行する保留中の作業がない場合に、新しいタスクをチェックする前にメンテナンswーカープロセスが待機する時間を秒単位で定義します。

デフォルトは 30 秒です。

pgaa.seafowl_client_retry_base_ms

Seafoal プロセスへの失敗した要求を再試行する前に Postgres クライアントが待機する初期遅延をミリ秒単位で指定します。後続の障害が発生するたびに、待機時間は 2 倍になります。

デフォルト 100 です。

pgaa.seafowl_url

PGAA 拡張機能が Seafoal サービスと通信するために使用するエンドポイント URL を指定します。デフォルトでは、PGAA はデータベースと同じホストで Seafoal プロセスを自動的に管理します。このパラメーターの変更は、別のマシンまたはコンテナで Seafoal サービスを実行しているカスタム構成の場合にのみ必要です。

デフォルトは `http://localhost:47470` です。

12.1.4 カタログの同期

pgaa.metastore_sync_poll_rate_s

現在接続されているカタログの状態をポーリングする間隔を秒単位で設定します。これを 0 に設定すると、自動バックグラウンド同期が完全に無効になります。

デフォルトは 60 秒です。

pgaa.metastore_sync_stale_duration_s

接続されたカタログのローカルメタデータが古いと考えられるしきい値を定義します。ローカルキャッシュのメタデータがこの期間よりも古い場合、システムは更新を優先してデータレイクとの一貫性を保証します。

デフォルトは 60 秒です。

pgaa.enable_metastore_sync_worker

PGAA が専用のバックグラウンドワーカーを生成して、Iceberg REST カタログなどのリモートプロバイダーからのメタデータを継続的に同期するかどうかを決定します。

デフォルトは off です。

12.1.5 キャッシング

pgaa.lakehouse_table_stats_cache_ttl_s

`pgaa.lakehouse_table_stats()` ファンクションをサポートするために物理テーブルのサイズがキャッシュされる期間秒単位を制御します。これにより、メタデータのオブジェクトストアへの頻繁な要求が防止されます。これを 0 に設定すると、統計キャッシュが無効になり、ファンクションはストレージプロバイダーから直接データを取得します。

デフォルトは 300 秒です。

pgaa.autostart_seafowl_object_store_cache_capacity_mb

Seafowl プロセスが保存場所からデータをキャッシュするために使用できるローカルディスク領域の最大量をメガバイト単位で指定します。

デフォルト 1024 です。

pgaa.autostart_seafowl_object_store_cache_min_fetch_size_kb

`pgaa.autostart_seafowl_enable_object_store_cache` が有効になっているときにオブジェクトストアに送信されるアウトバウンド要求の最小サイズを定義します。

デフォルト 1024 です。

pgaa.autostart_seafowl_object_store_cache_ttl_s

Seafowl オブジェクトストアキャッシュに保存されているオブジェクトの Time To Live(TTL) を指定します。このパラメーターは、キャッシュされたデータチャンクがエビクションの対象となるまでに有効な時間を決定します。

デフォルト 3600 です。

12.1.6 Postgres DistributedPGD 統合

`pgaa.max_replication_lag_s`

フラッシュが強制されるまでの最大許容遅延秒。

デフォルト 5 です。

`pgaa.flush_task_interval_s`

システムがフラッシュ要件をチェックする頻度秒。

デフォルト 5 です。

12.1.7 DataFusion から継承された構成

PGAA は Seafoal クエリエンジンの基になる実行エンジンとして Apache DataFusion を使用するため、PGAA はクエリ実行エンジンを調整できる複数の `pgaa.datafusion.*` 構成パラメーターを提供します。

これらの設定は、メモリ管理、実行並列処理、オプティマイザールールなどの低レベルの動作を制御します。

次を実行して、PGAA が組み込む DataFusion 構成パラメーターの完全なリストを確認できます。

```
SELECT name, setting, short_desc, context FROM pg_settings WHERE name LIKE %pgaa.data
→%;
```

キー、デフォルト値、およびパフォーマンスへの影響の包括的なリストについては、[DataFusion documentation](#) を参照してください。DataFusion ドキュメントを参照する場合、`pgaa.` を接頭辞として付けることにより、ドキュメントキーを PGAA 構成パラメーターに変換できます。

12.1.8 エグゼキューターエンジン

`pgaa.executor_engine`

分析操作の実行を担当するコンピューティングバックエンドを決定します。

サポートされている値は `seafowl`、および `spark_connect` です。

デフォルトは `seafowl` です。

`pgaa.spark_connect_url`

Spark Connect サービスの接続エンドポイントを定義します。リモート Spark Connect サービスの URL を指定します。 `pgaa.executor_engine` が `spark_connect` に設定されている場合、この設定は必須です。

値は、特定の Spark Connect URI 形式 `sc://<host>:<port>` に従う必要があります。

デフォルト値は `NULL` です。

pgaa.spark_connect_extra_config

pgaa.executor_engine= 'spark_connect' を使用する場合、詳細な調整パラメーターとランタイム設定を JSON オブジェクトとして Spark セッションに直接提供します。オプションのリストについては、[Spark Connect configuration](#) を参照してください。

例

```
SET pgaa.spark_connect_extra_config = {
  "spark.sql.execution.arrow.maxRecordsPerBatch": "65536"
};
```

デフォルト値は NULL です。

12.1.9 クエリー動作

pgaa.enable_direct_scan

分析テーブルのクエリ実行戦略を制御します。PGAA がクエリ全体を最適化された Seafoal エンジンにオフロードしようとするか、標準の PostgreSQL 互換性レイヤーを使用して処理するかどうかを決定します。

デフォルト値は on で、これは、PGAA が CompatScan にフェイルオーバーする前に、最初に DirectScan を試行することを意味します。

pgaa.direct_scan_fail_behavior

ダイレクトスキャンが試行されたが、サポートされていない SQL 機能または構文のために完了できなかった場合のシステムの反応を定義します。

サポートされている値は次のとおりです。

- **ignore** デフォルトシステムはサイレントに CompatScan にフォールバックします。
- **warn** クエリーは互換性フォールバックを使用して実行されますが、PostgreSQL は、DirectScan が失敗したことを示す警告通知を発行します。
- **error** DirectScan を使用できない場合、クエリの実行は完全に中止されます。

DirectScan が要求されたが失敗した場合の動作、ignore、warn、error のいずれか。デフォルトは warn です。

pgaa.enable_join_pushdown

CompatScan では、分析テーブル間の結合操作の最適化戦略を制御します。ジョインが PostgreSQL 内でローカルに処理されるか、リモート実行エンジン Seafoal または Spark にオフロードされるかどうかを決定します。

デフォルト値は on です。

pgaa.enable_groupby_pushdown

CompatScan では、集計操作 SUM、COUNT、AVG、GROUP BYなどをリモート実行エンジン Seafoal または Spark にオフロードしようとします。

デフォルト値は on です。

12.1.10 Seafoal

pgaa.autostart_seafoal

バックグラウンドワーカーによって管理される Seafoal プロセスの自動開始を制御します。

デフォルト値は on です。

pgaa.autostart_seafoal_max_memory_mb

組み込み Seafoal エンジンが消費できる RAM の量の上限をメガバイト単位で設定し、分析エンジンがメインの PostgreSQL プロセスのリソースを枯渇させないようにします。

デフォルト値は 2147483647 です。

pgaa.autostart_seafoal_enable_metrics

標準の可観測性スタックを使用して分析エンジンの状態、クエリスループット、およびリソース使用率をモニタするリアルタイムパフォーマンスメトリックを提供する専用の HTTP エンドポイントを開くことにより、Seafoal プロセスで Prometheus メトリックを有効にします。

デフォルト値は on です。

pgaa.autostart_seafoal_metrics_host

メトリックエンドポイントのバインドホスト。

pgaa.autostart_seafoal_enable_metrics= on の場合、Prometheus メトリックサービスが到達可能なホスト名または IP アドレスを指定します。

デフォルト値は 127.0.0.1 で、これはセキュリティのためのローカルループバックをポイントし、メトリックがデフォルトで公開されないことを保証します。

pgaa.autostart_seafoal_metrics_port

pgaa.autostart_seafoal_enable_metrics= on のときに Prometheus メトリックサービスに使用される特定のネットワークポートを定義します。

サポートされている値の範囲は 1024 ~ 65535 です。デフォルト値は 9090 です。

12.1.11 カタログの同期

`pgaa.metastore_sync_poll_rate_s`

カタログの更新のために接続されている現在の状態をポーリングする間隔を秒単位で決定します。

接続されたカタログの状態をポーリングする間隔を秒単位で設定します。これを 0 に設定すると、自動バックグラウンド同期が完全に無効になります。

デフォルト値は 60 秒です。

`pgaa.metastore_sync_stale_duration_s`

接続されたカタログのローカルメタデータが古いと考えられるしきい値を定義します。ローカルキャッシュのメタデータがこの期間よりも古い場合、システムは更新を優先してデータレイクとの一貫性を保証します。

デフォルト値は 60 秒です。

12.1.12 キャッシング

`pgaa.lakehouse_table_stats_cache_ttl_s`

テーブル統計ルックアップの結果がメモリに保持される時間を秒単位で制御します。クエリが計画されている場合、PostgreSQL はこれらの統計を必要にして、最も効率の高い実行パスを選択します。それらをキャッシュすることにより、システムは、すべてのクエリーに対してオブジェクトストアへの高遅延ネットワーク要求を行うことを回避します。これを 0 に設定すると、キャッシュ統計が完全に無効になります。

デフォルト値は 300 秒です。

12.1.13 DataFusion から継承された構成

PGAA は Seafoam クエリエンジンの基になる実行エンジンとして Apache DataFusion を使用するため、PGAA はクエリ実行エンジンを調整できる複数の `pgaa.datafusion.*` 構成パラメーターを提供します。

これらの設定は、メモリ管理、実行並列処理、オプティマイザールールなどの低レベルの動作を制御します。

次を実行して、PGAA が組み込む DataFusion 構成パラメーターの完全なリストを確認できます。

```
SELECT name, setting, short_desc, context FROM pg_settings WHERE name LIKE %pgaa.datafusion%;
```

キー、デフォルト値、およびパフォーマンスへの影響の包括的なリストについては、[DataFusion documentation](#) を参照してください。DataFusion ドキュメントを参照する場合、`pgaa.` を接頭辞として付けることにより、ドキュメントキーを PGAA 構成パラメーターに変換できます。

12.2 Functions

リファレンスセクションは、Analytics Accelerator で使用可能な機能のリストです。

PGAA の保存場所を作成する方法の詳細については、PGFS functions for Pipelines を参照してください。

12.2.1 テーブルファンクション

pgaa.list_analytics_tables()

PGAA によって管理されているすべての分析テーブルのリストを返します。構成、ストレージ統計、現在のレプリケーションステータスを含みます。

概要

```
SELECT * FROM pgaa.list_analytics_tables();
```

パラメーター

なし。

戻り値

Column	Type	Description
<i>nspoid</i>	<i>OID</i>	テーブルを含む名前空間スキーマの OID。
<i>reloid</i>	<i>OID</i>	テーブルレプリケーションの OID。
<i>schema_name</i>	<i>TEXT</i>	Postgres スキーマの名前。
<i>table_name</i>	<i>TEXT</i>	分析テーブルの名前。
<i>format</i>	<i>pgaa.table_format</i>	保存形式 `delta`、`iceberg`、または `parquet`。
<i>object_storage_snapshot_size</i>	<i>BIGINT</i>	オブジェクトストレージ内の最新のテーブルスナップショットのバイト単位のサイズ。
<i>object_storage_total_size_in_bytes</i>	<i>BIGINT</i>	オブジェクトストレージ内のこのテーブルのすべてのバージョン/ファイルのバイト単位の合計サイズ。
<i>replication_status</i>	<i>pgaa.replication_status</i>	データ移動の現在の状態 <i>enabled</i> 、 <i>disabled</i> 、または <i>initial_offload</i> 。PGD 統合が必要です。
<i>storage_location_name</i>	<i>TEXT</i>	テーブルに関連付けられている PGFS 格納場所の名前。
<i>storage_location_path</i>	<i>TEXT</i>	データが存在するストレージ場所内のパス。
<i>catalog_name</i>	<i>TEXT</i>	外部カタログの名前該当する場合。
<i>catalog_namespace</i>	<i>TEXT</i>	外部カタログ内の名前空間。
<i>catalog_table</i>	<i>TEXT</i>	外部カタログで定義されたテーブル名。

テーブルには、格納場所またはカタログを持つことができます。両方を持つことはできません。

カタログの詳細は、オブジェクトストレージへのカタログ管理オフロードまたはレプリケーションを実行する場合はレプリケーションターゲット、または [Integrating with Iceberg catalogs](#) の場合はソースカタログの場所を参照します。

pgaa.lakehouse_table_stats()

特定の分析テーブルのストレージ統計を返します。最新のアクティブなスナップショットのサイズ、オブジェクトストレージに保存されているすべてのデータバージョン履歴データとログの合計累積サイズを含みます。

概要

```
SELECT * FROM pgaa.lakehouse_table_stats(table_name::regclass);
```

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	調査する分析テーブルの名前または OID。

戻り値

Column	Type	Description
<i>latest_snapshot_size</i>	<i>BIGINT</i>	テーブルの最新のアクティブなスナップショットのバイト単位のサイズ。
<i>total_size</i>	<i>BIGINT</i>	メタデータ、トランザクションログ、履歴スナップショットを含む、オブジェクトストレージのテーブルに関連付けられているすべてのファイルのバイト単位の合計サイズ。

pgaa.test_storage_location()

定義された保存場所の接続と構成をテストし、読み取りまたは書き込み権限を確認します。

概要

```
SELECT pgaa.test_storage_location(location_name, test_writes:= true);
```

パラメーター

Parameter	Type	Description
<i>name</i>	<i>TEXT</i>	テストする保存場所の名前。
<i>test_writes</i>	<i>BOOLEAN</i>	`true` の場合、ファンクションは書き込み操作を試行して権限を確認します。`false` の場合、読み取りのみがテストされます。

戻り値

テストが成功した場合は NULL を返します。テストが失敗した場合、説明的なエラーメッセージを返します。

pgaa.pgaa_version()

特定のビルドバージョン、ビルド日、Git コミットハッシュを含む EDB Postgres Analytics Accelerator のバージョン情報を返します。

概要

```
SELECT pgaa.pgaa_version();
```

パラメーター

なし。

戻り値

現在のインストールの一意の Git 識別子、セマンティックバージョン、ビルドタイムスタンプ、および一意の Git 識別子を含む文字列。

pgaa.seafowl_version()

PGAA に埋め込まれている Seafowl 分析エンジンの詳細なバージョン情報を返します。コミットハッシュ、ビルド時間、最適化レベル、有効な機能などを含まれます。

概要

```
SELECT pgaa.seafowl_version();
```

パラメーター

なし。

戻り値

Seafowl コミットハッシュ、ビルドタイムスタンプ、最適化レベル、および現在のビルドで有効になっている機能セットを含む文字列。

12.2.2 カタログファンクション

pgaa.add_catalog()

新しい Iceberg カタログを PGAA に登録します。この機能は、カタログがシステムに登録される前に、自動接続チェックを実行して、資格情報とアクセス可能性を検証します。

概要

```
SELECT pgaa.add_catalog(catalog_name, catalog_type, catalog_options::json);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	PGAA 内のカタログの一意の名前。
<i>catalog_type</i>	<i>pgaa.catalog_type</i>	カタログタイプ。サポートされている値は、 <i>iceberg-rest</i> Iceberg REST カタログ、および <i>iceberg-s3tables</i> AWS S3 テーブルです。
<i>catalog_options</i>	<i>JSON</i>	接続パラメーターと認証パラメーターを含む JSON オブジェクト。

使用するストレージに応じて、`catalog_options` の JSON ファイルは別のオプションを指定する必要があります。

- REST カタログの場合

```
{
  "url": "https://your-catalog-endpoint.com",
  "warehouse_name": "your_warehouse_name",
  "warehouse": "your_warehouse_id",
  "token": "your_secret_auth_token"
  "danger_accept_invalid_certs": "false"
}
```

そこで

- `url` REST カタログサービスのベース HTTP(S) エンドポイント。
- `warehouse_name` ウェアハウス ID に代わる人間が判読可能な代替。構成を簡単にするために一部の REST プロバイダーでサポートされています。
- `warehouse` カタログサービス内の特定のウェアハウスの一意の識別子。
- `danger_accept_invalid_certs true` に設定されている場合、Postgres は SSL 証明書の検証をスキップします。これは、内部テストまたは自己署名証明書を使用する場合にのみ使用します。機密のパブリック接続には決して使用しないでください。

Snowflake Open Catalog のような一部の Iceberg REST カタログプロバイダーは、別のパラメーターを想定する場合があります。次の OAuth オプションを指定します。

```
{
  "oauth2.grant_type": "client_credentials",
  "oauth2.client_id": "oauth-client-id",
  "oauth2.client_secret": "oauth-client-secret",
  "oauth2.token_uri": "https://EXAMPLE-polaristest.snowflakecomputing.com/polaris/api/
  ↪catalog/v1/oauth/tokens",
  "oauth2.scope": "PRINCIPAL_ROLE:read_only"
}
```

そこで

- `oauth2.grant_type` OAuth2 フローを指定します。サービス間通信の最も一般的な値は `client_credentials` です。
- `oauth2.client_id` ID プロバイダーに登録されたアプリケーションの一意の識別子。
- `oauth2.client_secret` クライアント ID の認証に使用される秘密キー。
- `oauth2.token_uri` PGAA が資格情報を送信してアクセストークンを要求する完全な HTTP(S) エンドポイント。

- `oauth2.scope` オプションクライアントがアクセスを要求している権限またはリソースのスペース区切りリスト。
- AWS S3 テーブルの場合

```
{
  "arn": "arn:aws:s3tables:us-east-1:1234567890:bucket/my-bucket",
  "region": "us-east-1"
}
```

そこで

- `arn` S3 テーブルバケットの一意的識別子である Amazon リソースネーム ARN を指定します。- `region` S3 テーブルバケットが存在する AWS データセンターの物理的な場所を指定します。

戻り値

成功するとカタログの名前を返します。

`pgaa.update_catalog()`

既存の Iceberg カタログの構成オプション JSON オブジェクトを更新します。`pgaa.add_catalog()` と同様に、この関数は検証チェックを実行して、変更を適用する前に新しい接続パラメーターが機能していることを確認します。

概要

```
SELECT * FROM pgaa.update_catalog(catalog_name, new_options::json);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<code>VARCHAR</code>	PGAA 内のカタログの一意的名前。
<i>new_options</i>	<code>JSON</code>	更新された接続パラメーターと認証パラメーターを含む JSON オブジェクト。

各カタログタイプに必要な JSON フィールドの詳細については、[pgaa.add_catalog\(\)](#) を参照してください。

戻り値

更新が成功すると、カタログの名前を返します。

例

OAuth2 トークンをローテーションします。

```
SELECT * FROM pgaa.lakehouse.update_catalog(
  marketing_lakehouse,
  {
    "url": "https://iceberg.example.com",
```

(次のページに続く)

(前のページからの続き)

```

    "oauth2.client_id": "pgaa_service_user",
    "oauth2.client_secret": "new_secure_secret_2026",
    "oauth2.token_uri": "https://auth.example.com/token"
  }::json
);

```

pgaa.delete_catalog()

登録されたカタログをデータベースから削除します。偶発的なデータの損失を防ぐため、カタログで管理されているテーブルがまだ存在する場合、このファンクションではカスケードパラメーターを明示的に `true` に設定する必要があります。

概要

```
SELECT * FROM pgaa.delete_catalog(catalog_name, cascade := true);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	削除するカタログの名前。
<i>cascade</i>	<i>BOOLEAN</i>	<code>true</code> の場合、PGAA は、カタログエントリを削除する前に、このカタログで管理されているすべてのローカルテーブル定義を自動的に削除します。デフォルトは <code>false</code> です。

戻り値

正常に完了すると、削除されたカタログの名前を返します。

pgaa.list_catalogs()

システム内に登録されているすべてのカタログのリストを返します。接続構成、メタデータ同期タイムスタンプ、現在の動作ステータスを含みます。

概要

```
SELECT * FROM pgaa.list_catalogs();
```

パラメーター

なし。

戻り値

Column	Type	Description
<i>name</i>	<i>TEXT</i>	カタログの名前。
<i>type</i>	<i>pgaa.catalog_type</i>	カタログタイプ。サポートされている値は、 <i>iceberg-rest</i> Iceberg REST カタログ、および <i>iceberg-s3tables</i> AWS S3 テーブルです。
<i>options</i>	<i>JSON</i>	このカタログに使用される接続パラメーター URL、ARN など。
<i>status</i>	<i>pgaa.catalog_status</i>	カタログの現在のヘルス <i>detached</i> 、 <i>attached</i> 、 <i>refresh_retry</i> 、または <i>refresh_failed</i> 。
<i>created_at</i>	<i>TIMESTAMPTZ</i>	カタログが最初に登録されたときのタイムスタンプ。
<i>last_refreshed_at</i>	<i>TIMESTAMPTZ</i>	PGAA が最後にこのカタログからメタデータを正常に同期した時間。

pgaa.import_catalog()

ワンタイムスキャンを実行し、登録済みの Iceberg カタログから Postgres にテーブル定義のインポートを実行します。この関数は、PGAA がリモートテーブルを照会するために必要なローカルメタデータを作成します。これは手動で 1 回限りのインポートであり、自動で継続的な同期は有効になりません。

概要

```
SELECT pgaa.import_catalog(catalog_name, namespace_filter);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	インポートする以前に登録したカタログの名前。
<i>namespace_filter</i>	<i>VARCHAR</i>	指定した場合、この特定の Iceberg 名前空間内のテーブルのみがインポートされます。省略または <i>NULL</i> の場合、すべての名前空間のすべてのテーブルがインポートされます。デフォルトは <i>NULL</i> です。

戻り値

なし。

pgaa.attach_catalog()

以前に登録した Iceberg カタログの継続的なメタデータの同期を有効にします *pgaa.add_catalog()* を使用接続されると、PGAA はリモートカタログの変更を自動的に監視し、それに応じてローカル Postgres メタデータを更新します。ポーリングレート構成については、[カタログの同期](#) を参照してください。

概要

```
SELECT pgaa.attach_catalog(catalog_name);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	同期を開始する登録済みカタログの名前。

戻り値

なし。

pgaa.detach_catalog()

登録された Iceberg カタログの継続的なメタデータの同期を停止し、`detached` 状態に移動します。 `cascade` が `true` の場合、デタッチする前にカタログによって管理されているすべてのテーブルを削除します。 `cascade` が `false` の場合、テーブルはデータベースに残りますが、同期は停止します。

概要

```
SELECT * FROM pgaa.detach_catalog(catalog_name, cascade := true);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	切り離す登録済みカタログの名前。
<i>cascade</i>	<i>BOOLEAN</i>	オプション。 <code>true</code> の場合、PGAA は、デタッチする前に、このカタログに関連付けられているすべてのローカルテーブル定義を自動的に削除します。デフォルトは <code>false</code> です。

戻り値

ファンクションは、デタッチされるカタログの `pgaa.catalog` システムテーブルから行を結果ます。

Column	Type	Description
<i>name</i>	<i>TEXT</i>	分離されたカタログの名前。
<i>type</i>	<i>pgaa.catalog_type</i>	カタログタイプ。
<i>status</i>	<i>pgaa.catalog_status</i>	新しいステータスは <code>detached</code> になります。

pgaa.test_catalog()

登録された Iceberg カタログの接続と構成をテストします。このファンクションは、Postgres インスタンスがリモートカタログエンドポイントと通信できることを確認し、オプションで提供された資格情報に書き込み権限があることを検証します。

概要

```
SELECT pgaa.test_catalog(catalog_name, test_writes:=true);
```

パラメーター

Parameter	Type	Description
<i>name</i>	<i>TEXT</i>	テストする登録済みカタログの名前。
<i>test_writes</i>	<i>BOOLEAN</i>	`true`の場合、ファンクションはカタログメタデータサービスへの書き込み操作を試行して、権限を確認します。`false`の場合、読み取り権限のみがテストされます。

戻り値

テストが成功した場合は `NULL` を返します。テストが失敗した場合、説明的なエラーメッセージを返します。

pgaa.list_catalog_tables()

登録された Iceberg カタログで使用可能なすべてのテーブルとビューのリストを返します。この機能を使用すると、最初にテーブルをローカルデータベースにインポートまたは接続することなく、リモートカタログのコンテンツを探索できます。

概要

```
SELECT * FROM pgaa.list_catalog_tables(catalog_name, namespace_filter);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<i>TEXT</i>	探索する登録済みカタログの名前。
<i>namespace_filter</i>	<i>TEXT</i>	提供する場合、フィルターする Iceberg 名前空間スキーマを指定します。省略または `NULL` の場合、カタログ内のすべてのテーブルが返されます。デフォルトは `NULL` です。

戻り値

Column	Type	Description
<i>schema_name</i>	<i>TEXT</i>	リモートの Iceberg 名前空間/スキーマ。
<i>table_name</i>	<i>TEXT</i>	テーブルまたはビューの名前。

pgaa.drop_catalog_tables()

特定の Iceberg カタログによって管理されているすべてのローカル Postgres テーブルとビュー定義を削除します。cascade が true の場合、依存オブジェクトを含みます。リモートの Iceberg カタログのテーブルには影響しません。

概要

```
SELECT * FROM pgaa.drop_catalog_tables(catalog_name, cascade := true);
```

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	<i>VARCHAR</i>	管理テーブルが削除される登録済みカタログの名前。
<i>cascade</i>	<i>BOOLEAN</i>	`true` の場合、ファンクションはこれらのテーブルに依存するオブジェクトを自動的に削除します。デフォルトは `false` です。

戻り値

Column	Type	Description
<i>schema_name</i>	<i>TEXT</i>	テーブルが存在したローカル Postgres スキーマ。
<i>table_name</i>	<i>TEXT</i>	削除されたローカルテーブルの名前。

12.2.3 Spark ファンクション

pgaa.spark_sql()

構成済みの Spark Connect エンドポイントを介して Postgres クラスターで Spark SQL クエリーを直接実行します。これにより、Postgres で使用できない Iceberg 圧縮ルーチンまたは Spark ファンクションを実行できます。

使用可能なプロシージャのリストについては、[Spark procedures](#) を参照してください。

このファンクションを実行するには、構成パラメーター `pgaa.spark_connect_url` を設定して、使用可能な Spark Connect サービスをポイントする必要があります。

概要

単一のカタログの場合

```
SELECT pgaa.spark_sql(query, catalog_name);
```

複数のカタログの場合

```
SELECT pgaa.spark_sql(query, ARRAY[catalog1, catalog2]);
```

パラメーター

Parameter	Type	Description
<i>query</i>	<i>TEXT</i>	実行する Spark SQL ステートメント。
<i>catalog</i>	<i>TEXT</i> or <i>TEXT[]</i>	クエリーに使用する単一のカatalog名、またはカatalog名の配列。

戻り値

JSON オブジェクトとしてフォーマットされた Spark クエリの結果セット。

例

rewrite_data_files Spark タスクを介してメタデータのオーバーヘッドを削減します。

```
SELECT pgaa.spark_sql($$
    CALL preexisting.system.rewrite_data_files(
        table => "preexisting"."ns-1"."table-1",
        strategy => sort,
        sort_order => value DESC,
        options => map(rewrite-all, true)
    )
$$);
```

pgaa.execute_compaction()

圧縮を実行して、分析テーブルのパフォーマンスとストレージ効率を向上させます。ネイティブバックグラウンドタスクとして実行され、Iceberg テーブルと Delta テーブルの両方をサポートしています。

i 注釈

ファンクション `pgaa.execute\_compaction()` は非推奨で、*pgaa.launch_task()* を優先します。

概要

```
SELECT pgaa.execute_compaction(table_name::regclass, settings::json);
```

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	圧縮する分析テーブルの名前または OID。
<i>settings</i>	<i>JSON</i>	現在サポートされていません。圧縮動作を定義するオプションの JSON オブジェクト。

i 注釈

パラメーター `settings` は現在サポートされていません。 `pgaa.spark_sql()` を使用して、圧縮動作をより詳細に制御します。

戻り値

なし。

例

```
SELECT pgaa.execute_compaction(lakehouse.inventory_iceberg::regclass);
```

12.2.4 バックグラウンドタスクファンクション**pgaa.launch_task()**

分析デルタまたはアイスパークテーブルのバックグラウンドメンテナンスタスクをスケジュールします。

i 注釈

Delta Lake は使用可能なすべてのタスクタイプをサポートしていますが、Iceberg テーブルは現在 `compaction` オペレーションのみをサポートしています。さらに、Iceberg コンパクションは、`dry_run` を除き、Delta Lake で使用可能な詳細な構成キーをサポートしていません。

概要

```
SELECT pgaa.launch_task(  
    table_name::regclass,  
    task_type,  
    task_options::jsonb,  
    scheduled_at::timestamp  
);
```

パラメーター

Parameter	Type	Description
<code>table_name</code>	<code>REGCLASS</code>	タスクを実行する分析テーブルの名前または OID。
<code>task_type</code>	<code>TEXT</code>	メンテナンスオペレーション <code>compaction</code> 、 <code>zorder</code> 、 <code>vacuum</code> 、または <code>purge</code> 。
<code>task_options</code>	<code>JSONB</code>	タスクタイプに固有の構成以下を参照してください。
<code>scheduled_at</code>	<code>TIMESTAMP</code>	指定された場合、タスクはこの時間まで待機して実行されます。デフォルトは <code>NULL</code> です。

次のページに続く

表 20 – 前のページからの続き

Parameter	Type	Description
-----------	------	-------------

JSONB `task_options` の値は、各 `task_type` によって異なります。`columns` を必要とする `zorder` と、`storage_location` と `path` の両方を必要とする `purge` を除き、すべてのオプションはオプション。

- `compaction` 小さなファイルを大きなファイルにマージして、分析スキャンを高速化します。使用可能な `task_options` は次のとおりです。

```
{
  "target_size": 536870912,
  "preserve_insertion_order": true,
  "max_concurrent_tasks": 10,
  "max_spill_size": 2147483648,
  "min_commit_interval": 60,
  "dry_run": false
}
```

そこで

- `target_size` デルタのみ出力ファイルのサイズをバイト単位で指定します。
- `preserve_insertion_order` デルタのみ行の既存のソート順を維持するかどうか。
- `max_concurrent_tasks` デルタのみ エグゼキューターが実行できる並列タスクの数を制限します。
- `max_spill_size` デルタのみプロセス中にディスクに流出できる最大データサイズをバイト単位で設定します。
- `min_commit_interval` デルタのみ 最小待機時間を秒単位で設定します。デルタログの更新をコミットする。
- `dry_run` `true` の場合、オペレーションを計算およびログに記録しますが、圧縮は実行しません。
- `zorder` デルタのみ複数の列にわたってデータを再編成して、これらの列にフィルタを使用したクエリの「データスキップ」を向上させるクラスタリング手法。使用可能な `task_options` は次のとおりです。

```
{
  "columns": ["customer_id", "transaction_date"],
  "target_size": 1073741824,
  "preserve_insertion_order": false,
  "max_concurrent_tasks": 4,
  "max_spill_size": 2147483648,
  "min_commit_interval": 30
}
```

そこで

- `columns` 必須 Z オーダーに使用される列を表す文字列の配列。
- `target_size` 出力ファイルのサイズをバイト単位で指定します。
- `preserve_insertion_order` 行の既存のソート順を維持するかどうか。
- `max_concurrent_tasks` エグゼキューターが実行できる並列タスクの数を制限します。
- `max_spill_size` プロセス中にディスクに流出できる最大データサイズをバイト単位で設定します。
- `min_commit_interval` デルタログの更新をコミットするまでの最小待機時間を秒単位で設定します。

- vacuum Delta のみデルタトランザクションログによって参照されなくなった古いデータファイルを削除し、オブジェクトストレージの領域を解放します。使用可能な task_options は次のとおりです。

```
{
  "retention_period": "168 hours",
  "dry_run": false,
  "enforce_retention_duration": true
}
```

そこで

- retention_period 参照されていないファイルが削除の対象となる年齢を定義します。- dry_run true の場合、削除されるファイルを計算およびログに記録しますが、削除は実行しません。- enforce_retention_duration true の場合、タスクはシステムのグローバル最小安全制限に対して retention_period を検証します。
- purge デルタのみ特定のストレージパスからデータを明示的に削除します。使用可能な task_options は次のとおりです。

```
{
  "storage_location": "s3_main",
  "path": "archive/2023/temp/"
}
```

そこで

- storage_location 必須。保存場所の名前。- path 完全に削除する必要がある保存場所内の相対ディレクトリパスまたはファイル接頭辞。

戻り値

タスクの一意のタスク ID。

pgaa.background_task テーブルと提供されたタスク ID を照会することにより、タスクのステータスを確認できます。

例

- 過去 7 日間より前の古いデータファイルを削除します。

```
SELECT pgaa.launch_task(
  sales.transactions,
  vacuum,
  {"retention_period": "7 days", "dry_run": false}::jsonb
);
```

- 圧縮を実行します

```
SELECT pgaa.launch_task(
  telemetry.logs,
  compaction,
```

(次のページに続く)

(前のページからの続き)

```
{
  "target_size": 536870912,
  "max_concurrent_tasks": 2
}::jsonb
);
```

- 列 `region` および `customer_id` でクラスタリングすることによりデータ行を再編成します。

```
SELECT pgaa.launch_task(
  crm.customers,
  zorder,
  {
    "columns": ["region", "customer_id"],
    "target_size": 1073741824
  }::jsonb
);
```

12.2.5 Postgres DistributedPGD 統合機能

`pgaa.convert_to_analytics()`

PGD レプリケートテーブル HTAP を PGAA アクセス方法に変換します。このファンクションは、レプリケーションを介して同期されたオブジェクトストレージアイスバークまたはデルタ形式にある分析データを直接ポイントします。

概要

```
CALL pgaa.convert_to_analytics(relation_name::regclass);
```

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	アナリティクスに変換する PGD レプリケートテーブル HTAP の名前または OID。

戻り値

なし。

注釈

ヒープ分析テーブルを PGAA テーブルに変換するために、`ALTER TABLE` コマンドと ``SET ACCESS METHOD PGAA`` を実行することもできます。詳細は、[Table options](#) を参照してください。

pgaa.convert_to_tiered_table()

自動パーティション化と分析オフロードを使用して、ヒープテーブルを PGD 管理の階層テーブルに変換します。この機能は、パーティション化されたステー징テーブルの作成を自動化し、既存のデータを移行し、ホット層とコールド層の両方のライフサイクルポリシーを構成します。詳細は、[階層テーブルの実装](#)を参照してください。

概要

```
CALL pgaa.convert_to_tiered_table(
  relation := relation_name::regclass,
  range_partition_column := column_name,
  partition_increment := interval_string,
  analytics_offload_period := interval,
  initial_lower_bound := start_date_string,
  retention_period := interval DEFAULT NULL,
  enable_replication := boolean DEFAULT false,
  minimum_advance_partitions := integer DEFAULT 2,
  maximum_advance_partitions := integer DEFAULT 5,
  drop_after_retention_period := boolean DEFAULT true,
  purge_analytics_target := boolean DEFAULT false
);
```

注釈

レンジパーティションに使用される列は、主キーである必要があります。

シーケンスまたは外部キー制約を含むテーブルは、階層テーブルではサポートされていません。階層テーブルに変換する前に、シーケンスまたはコンストレインを削除する必要があります。

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	階層テーブルに変換するヒープテーブルの名前または OID。
<i>range_partition_column</i>	<i>TEXT</i>	レンジパーティション化に使用される列。 `DATE`または`TIMESTAMP`であり、主キーに含まれる必要があります。
<i>partition_increment</i>	<i>TEXT</i>	`1 month` など、各パーティションのサイズを定義する時間間隔。
<i>analytics_offload_period</i>	<i>INTERVAL</i>	パーティションがローカルストレージからデータレイクのコールド層に移動されるまでのしきい値は経過時間です。
<i>initial_lower_bound</i>	<i>TEXT</i>	最初のパーティションの下限の開始値。
<i>retention_period</i>	<i>INTERVAL</i>	(オプション) データがコールド層からパージされるまでの期間。デフォルトは`NULL`です。

[次のページに続く](#)

表 22 – 前のページからの続き

Parameter	Type	Description
<i>enable_replication</i>	<i>BOOLEAN</i>	(オプション) `true` の場合、アクティブなパーティションのオブジェクトストレージへのリアルタイムレプリケーションを HTAP テーブルとして有効にします。デフォルトは `false` です。
<i>mini-mum_advance_partition</i>	<i>INTEGER</i>	(オプション) 維持する将来のパーティションの最小数。デフォルト 2 です。
<i>maxi-mum_advance_partition</i>	<i>INTEGER</i>	(オプション) 維持する将来のパーティションの最大数。デフォルト 5 です。
<i>drop_after_retention_pe</i>	<i>BOOLEAN</i>	(オプション) 保持期間の有効期限が切れた後、パーティションをローカルにドロップするかどうか。デフォルトは `true` です。
<i>purge_analytics_target</i>	<i>BOOLEAN</i>	(オプション) `true` の場合、オブジェクトストアディレクトリまたはカタログエントリに既存のテーブルが削除されます。`false` に設定されており、既存のテーブルが検出された場合、コマンドはエラーで失敗します。PGD 6.3+ でサポートされています。デフォルトは `false` です。

戻り値

なし。

例

ヒープテーブルを階層構造に変換し、データが月ごとにパーティション化され、リアルタイム分析のためにクラウドにレプリケートされ、3 か月以上古いとコールドストレージに自動的にオフロードされます。

```
CALL pgaa.convert_to_tiered_table(
    relation           := public.sensor_readings::regclass,
    range_partition_column := reading_time,
    partition_increment := 1 month,
    analytics_offload_period := 3 months,
    initial_lower_bound  := 2026-01-01,
    retention_period     := 2 years,
    enable_replication   := true,
    minimum_advance_partitions := 3,
    maximum_advance_partitions := 6,
    drop_after_retention_period := true,
    purge_analytics_target := true
);
```

pgaa.disable_analytics_replication()

HTAP テーブルのオブジェクトストレージへの PGD レプリケーションを無効にします。このファンクションは、オブジェクトストレージへの連続レプリケーションストリームを停止することにより、テーブルを HTAP 状態からヒープ状態に戻します。詳細は、[アナリティクスへの複製](#) を参照してください。

概要

```
CALL pgaa.disable_analytics_replication(relation_name::regclass);
```

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	分析レプリケーションを無効にする HTAP テーブルの名前または OID。

戻り値

なし。

注釈

HTAP テーブルのレプリケーションを無効にするには、`ALTER TABLE` コマンドと ``SET pgd.replicate_to_analytics = FALSE`` を実行することもできます。詳細は、[Table options](#) を参照してください。

pgaa.enable_analytics_replication()

分析用のオブジェクトストレージへのヒープテーブルの PGD レプリケーションを有効にします。このファンクションは、分析エンジンへの連続レプリケーションストリームを開始することにより、標準のヒープテーブルを HTAP テーブルに移行します。詳細は、[アナリティクスへの複製](#) を参照してください。

概要

```
CALL pgaa.enable_analytics_replication(
    relation regclass,
    purge_analytics_target boolean DEFAULT false
);
```

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	分析レプリケーションを有効にするヒープテーブルの名前または OID。

[次のページに続く](#)

表 24 – 前のページからの続き

Parameter	Type	Description
<i>purge_analytics_target</i>	<i>BOOLEAN</i>	(オプション) `true` の場合、レプリケーションを開始する前に分析ターゲットの既存のデータをパージします。PGD 6.3 以降が必要です。デフォルトは `false` です。

戻り値

なし。

注釈

ヒープテーブルのレプリケーションを有効にするには、`ALTER TABLE` コマンドと `SET pgd.replicate_to_analytics = TRUE` を実行することもできます。詳細は、[Table options](#) を参照してください。

pgaa.htap_table_stats_uncached()

PGD 環境の HTAP テーブルの統計を結果ます。

概要

```
SELECT * FROM pgaa.htap_table_stats_uncached(
    htap_catalog text,
    htap_catalog_namespace text,
    htap_catalog_table text,
    htap_storage_location text,
    htap_storage_path text,
    htap_format text
);
```

パラメーター

Parameter	Type	Description
<i>htap_catalog</i>	<i>TEXT</i>	HTAP カタログの名前。
<i>htap_catalog_namespac</i>	<i>TEXT</i>	HTAP カタログ内の名前空間/スキーマ。
<i>htap_catalog_table</i>	<i>TEXT</i>	HTAP カタログ内の特定のテーブル名。
<i>htap_storage_location</i>	<i>TEXT</i>	オブジェクト保存場所の URI または識別子。
<i>htap_storage_path</i>	<i>TEXT</i>	データが存在するストレージ場所内の特定のパス。
<i>htap_format</i>	<i>TEXT</i>	ストレージで使用するデータ形式。

戻り値

Column	Type	Description
<i>latest_snapshot_size</i>	<i>BIGINT</i>	最新のデータスナップショットのバイト単位のサイズ。
<i>total_size</i>	<i>BIGINT</i>	テーブルに関連付けられているすべてのデータの合計累積サイズのバイト単位。

pgaa.list_tiered_tables()

パーティション設定、データ保持ポリシー、分析オフロード構成、ストレージサイズメトリックなど、PGD 管理の階層化テーブルに関する詳細情報を返します。

概要

```
SELECT * FROM pgaa.list_tiered_tables();
```

パラメーター

なし。

戻り値

Column	Type	Description
<i>schema_oid</i>	<i>OID</i>	スキーマの内部オブジェクト識別子。
<i>table_oid</i>	<i>OID</i>	テーブルの内部オブジェクト識別子。
<i>schema_name</i>	<i>TEXT</i>	階層化テーブルを含むスキーマの名前。
<i>table_name</i>	<i>TEXT</i>	階層テーブルの名前。
<i>partition_increment</i>	<i>INTERVAL</i>	自動パーティション化のために定義された間隔。
<i>retention</i>	<i>INTERVAL</i>	データがドロップされるまで保持される期間。
<i>analytics_offload</i>	<i>INTERVAL</i>	データが分析エンジンにオフロードされるまでの期間。
<i>replication_enabled</i>	<i>BOOLEAN</i>	アナリティクスへのレプリケーションがアクティブかどうかを示します。
<i>tiered_data_size</i>	<i>BIGINT</i>	オブジェクトストレージに格納されているデータの合計サイズのバイト単位。
<i>untiered_data_size</i>	<i>BIGINT</i>	ローカルストレージに残っているデータの合計サイズのバイト単位。

pgaa.restore_from_analytics()

オブジェクトストレージにある PGAA 分析テーブルを標準の PostgreSQL ヒープアクセス方法に変換します。

概要

```
CALL pgaa.restore_from_analytics(relation_name::regclass);
```

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REGCLASS</i>	ヒープテーブルに復元する PGAA 分析テーブルの名前または OID。

戻り値

なし。

注釈

PGAA 分析テーブルをヒープに変換するために、`ALTER TABLE` コマンドと ``SET ACCESS METHOD heap`` を実行することもできます。詳細については、[Table options](#) を参照してください。

12.2.6 ユーティリティファンクション

`pgaa._bench_replication()`

pgbench のような合成ベンチマークを実行して、指定された保存場所またはカタログへのデータの取り込みのパフォーマンスを測定します。初期データのロード速度と持続的なトランザクションスループットを計算します。

概要

```
SELECT * FROM pgaa._bench_replication(
    storage_location text DEFAULT NULL,
    catalog text DEFAULT NULL,
    format text DEFAULT iceberg,
    scale integer DEFAULT 100,
    transactions integer DEFAULT 100000,
    dry_run boolean DEFAULT false
);
```

パラメーター

Parameter	Type	Description
<i>storage_location</i>	<i>TEXT</i>	ベンチマークデータが書き込まれる PGFS 宛先保存場所の名前。`catalog` と相互排他的。
<i>catalog</i>	<i>TEXT</i>	ベンチマークターゲットとして使用する Iceberg カタログの名前。`storage_location` と相互排他的。
<i>format</i>	<i>TEXT</i>	ベンチマークに使用するテーブル形式。デフォルトは `iceberg` です。
<i>scale</i>	<i>INTEGER</i>	ベンチマークのスケール係数、初期データセットのサイズを決定します。デフォルト 100 です。

次のページに続く

表 29 – 前のページからの続き

Parameter	Type	Description
<i>transactions</i>	<i>INTEGER</i>	スループットテスト中にシミュレートするトランザクションの合計数。デフォルトは 100000 です。
<i>dry_run</i>	<i>BOOLEAN</i>	`true` の場合、ファンクションは実際のベンチマークを実行せずにパラメーターと接続を検証します。デフォルトは `false` です。

戻り値

Column	Type	Description
<i>initial load speed (MB/s)</i>	<i>DOUBLE PRECISION</i>	初期データセットがストレージに書き込まれた速度。
<i>throughput (TPS)</i>	<i>DOUBLE PRECISION</i>	ベンチマーク中に達成された 1 秒ごとのトランザクションの持続数。

12.3 Table options

12.3.1 PGAA テーブルオプション

CREATE TABLE、ALTER TABLE ステートメントは、USING PGAA WITH ... 句と次のオプションを使用して実行できます。

Option	Type	Description
<i>pgaa.storage_location</i>	text	PGFS ストレージ場所の名前。
<i>pgaa.path</i>	text	保存場所内のパス。
<i>pgaa.format</i>	enum	テーブル形式。有効な値は `delta`、`iceberg`、および `parquet` です。機能によってはサポートされていない場合があります。
<i>pgaa.managed_by</i>	text	カタログ管理テーブルのカタログ名。
<i>pgaa.catalog_namespace</i>	text	カタログ内の名前空間/スキーマ。
<i>pgaa.catalog_table</i>	text	カタログ内のテーブル名。
<i>pgaa.auto_truncate</i>	boolean	アクセス方法が PGAA に切り替わると、ヒープデータが切り捨てられます。PGD 統合が必要です。詳細については、 アナリティクスへのオフロード を参照してください。
<i>pgaa.purge_data_if_exists</i>	boolean	`CREATE TABLE AS` の実行時に既存の分析データをパージします。
<i>pgaa.tiered_table</i>	boolean	階層テーブルとしてマークします。PGD 統合が必要です。詳細については、 階層テーブルの実装 を参照してください。

例

- カタログ管理テーブルの作成

```
CREATE TABLE managed_sales_data (
  id BIGINT,
  sale_date TIMESTAMP,
  amount NUMERIC
)
USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
  pgaa.catalog_table = managed_sales_data
);
```

注釈

列定義を空の `()` のままにすると、PGAA は、最初にテーブルを照会したときにメタデータファイルからスキーマを自動的に検出し、リモートの Iceberg/Delta メタデータまたは Parquet ヘッダーで見つかったフィールドに名前で行をマップします。

- 前の例からテーブルを変更して、PGFS ストレージ場所をポイントします。

```
ALTER TABLE managed_sales_data
SET ACCESS METHOD PGAA,
SET (
  pgaa.format = iceberg,
  pgaa.storage_location = my_lake_data,
  pgaa.path = warehouse_data
);
```

12.3.2 PGD テーブルオプション

ヒープまたは HTAP テーブルに対して CREATE TABLE、ALTER TABLE ステートメントを実行し、次の PGD オプションを指定できます。

Option	Type	Description
<code>pgd.replicate_to_analyti</code>	boolean	オブジェクトストレージへの継続的レプリケーションを有効または無効にします。詳細については、 アナリティクスへの複製 を参照してください。

次のページに続く

表 32 – 前のページからの続き

Option	Type	Description
<code>pgd.purge_analytics_tar</code>	boolean	既存のテーブルをオブジェクトストアディレクトリまたはカタログエントリから削除します。詳細については、 階層テーブルの実装 を参照してください。

例

- 既存のテーブルを変更して、オブジェクトストレージへの継続的レプリケーションを有効にします。

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = TRUE);
```

12.3.3 アクセス方法の管理

テーブルのアクセス方法を変更することにより、トランザクションストレージと分析ストレージ間でテーブルを移行します。これは、ローカルの Postgres ディスクとオブジェクトストレージ間でデータを手動で移動する直接方法を提供します。詳細は、[アナリティクスへのオフロード](#)を参照してください。

HTAP から PGAA への切り替え

HTAP テーブルをオブジェクトストレージにオフロードし、ローカルディスク領域を再利用するには、アクセス方法を PGAA に変更します。ローカルデータは削除され、データはデータレイクにのみ残ります。これは、自動化されたライフサイクルオフロードの手動代替です。

```
ALTER TABLE sales SET ACCESS METHOD PGAA;
SET (
  pgaa.auto_truncate = true,
  pgaa.format = iceberg,
  pgaa.storage_location = my_lake_data,
  pgaa.path = analytical_data
);
```

PGAA からヒープへの切り替え

分析テーブルをローカルトランザクションテーブルに復元するには、アクセス方法を PGAA からヒープに変更します。Postgres はオブジェクトストレージからデータをプルし、ローカルストレージページに再構成します。これは、以前にオフロードされたデータに高頻度の更新を実行する必要がある場合に使用します。

```
ALTER TABLE sales_history SET ACCESS METHOD HEAP;
```

12.4 Data types and definitions

EDB Analytics Accelerator PGAA は、ネイティブの Postgres データタイプをデータレイクの最適化されたカラムナー形式にマッピングします。このページには、サポートされているタイプをリストし、PGAA 管理ビューで見つかるステータスコードを定義します。

12.4.1 データ型

PGAA は、データが Postgres に読み込まれる方法と、データレイクに書き込まれる方法 PGD 統合を使用して独自に管理します。このアーキテクチャは、複雑な形式またはサポートされていない形式との広範な互換性を保証しながら、パフォーマンスを最大化します。

アナリティクスからの読み取り

PGAA テーブルを照会すると、エンジンはレイクハウスからデータを取得し、スキーマで定義されたターゲット Postgres タイプに強制しようとします。ネイティブバイナリデータは、効率を最大限に高めるために Postgres タイプに直接マッピングされますが、他のタイプは内部 Postgres 変換機能を使用してターゲットタイプに強制的に戻されます。

Natively Supported	Coerced
<i>BOOLOID</i>	<i>CSTRINGOID</i>
<i>BYTEAOID</i>	<i>TEXTOID</i>
<i>CHAROID</i>	<i>VARCHAROID</i>
<i>INT2OID</i>	<i>BPCHAROID</i>
<i>INT4OID</i>	<i>JSONOID</i>
<i>INT8OID</i>	<i>JSONBOID</i>
<i>OIDOID</i>	<i>UUIDOID</i>
<i>FLOAT4OID</i>	<i>ANYENUMOID</i>
<i>FLOAT8OID</i>	<i>CSTRINGARRAYOID</i>
<i>NUMERICOID</i>	<i>TEXTARRAYOID</i>
<i>DATEOID</i>	<i>VARCHARARRAYOID</i>
<i>TIMESTAMPOID</i>	<i>BPCHARARRAYOID</i>
<i>TIMESTAMPTZOID</i>	<i>UUIDARRAYOID</i>
<i>INTERVALOID</i>	Any other type
<i>BOOLARRAYOID</i>	
<i>INT2ARRAYOID</i>	
<i>INT4ARRAYOID</i>	
<i>INT8ARRAYOID</i>	
<i>FLOAT4ARRAYOID</i>	
<i>FLOAT8ARRAYOID</i>	
<i>DATEARRAYOID</i>	
<i>TIMEARRAYOID</i>	
<i>TIMESTAMPARRAYOID</i>	
<i>TIMESTAMPTZARRAYOID</i>	
<i>NUMERICARRAYOID</i>	

アナリティクスへの書き込み

データをデータレイクに複製する場合 PGD 統合を使用すると、PGAA は効率を最大限に高めるためにネイティブバイナリビルダーを利用します。基になるストレージ形式でネイティブにサポートされていない列タイプが発生した場合、PGAA はそれを UTF-8 テキスト表現に自動的に変換して、一貫した信頼性の高い書き込みを保証します。

Natively supported	Converted to UTF-8 text
<i>BOOLOID</i>	<i>CSTRINGOID</i>
<i>BYTEAOID</i>	<i>TEXTOID</i>
<i>CHAROID</i>	<i>VARCHAROID</i>
<i>INT2OID</i>	<i>BPCHAROID</i>
<i>INT4OID</i>	<i>JSONOID</i>
<i>INT8OID</i>	<i>JSONBOID</i>
<i>OIDOID</i>	<i>UUIDOID</i>
<i>FLOAT4OID</i>	<i>ANYENUMOID</i>
<i>FLOAT8OID</i>	<i>CSTRINGARRAYOID</i>
<i>TIMESTAMPOID</i>	<i>TEXTARRAYOID</i>
<i>TIMESTAMPTZOID</i>	<i>VARCHARARRAYOID</i>
<i>DATEOID</i>	<i>BPCHARARRAYOID</i>
<i>INTERVALOID</i>	<i>UUIDARRAYOID</i>
<i>NUMERICOID</i>	All other types
<i>BOOLARRAYOID</i>	
<i>INT2ARRAYOID</i>	
<i>INT4ARRAYOID</i>	
<i>INT8ARRAYOID</i>	
<i>FLOAT4ARRAYOID</i>	
<i>FLOAT8ARRAYOID</i>	
<i>DATEARRAYOID</i>	
<i>TIMEARRAYOID</i>	
<i>TIMESTAMPARRAYOID</i>	
<i>TIMESTAMPTZARRAYOID</i>	
<i>NUMERICARRAYOID</i>	

12.4.2 管理用の列挙型とステータス

次のパラメーターと値は、主に PGAA 管理ビューにあります。表に、それらの説明とサポートされている値をリストします。

Type	Supported values	Description
<i>pgaa.table_format</i>	<i>delta</i> , <i>iceberg</i> , and <i>parquet</i> .	オブジェクトストレージ内のテーブルの格納形式。
<i>pgaa.catalog_type</i>	<i>iceberg-rest</i> (Iceberg REST catalog), and <i>iceberg-s3tables</i> (AWS S3 Tables).	カタログサービスの種類。
<i>pgaa.catalog_status</i>	<i>detached</i> , <i>attached</i> , <i>refresh_retry</i> , and <i>refresh_failed</i> .	Postgres とカタログ間の現在の同期の状態。
<i>pgaa.replication_status</i>	<i>disabled</i> , <i>initial_offload</i> , and <i>enabled</i> .	階層テーブルのデータの移動の状態。

次のページに続く

表 35 – 前のページからの続き

Type	Supported values	Description
<i>pgaa.task_status</i>	<i>pending, running, success, and failure.</i>	バックグラウンドワーカータスクのライフサイクル状態。

12.5 Benchmark datasets

サンプルベンチマークデータセットを含む AWS のパブリック S3 バケットを使用できます。これらは、次のエンドポイントを介して利用できます。

- `s3://beacon-analytics-demo-data-ap-east-1-prod`
- `s3://beacon-analytics-demo-data-ap-northeast-1-prod`
- `s3://beacon-analytics-demo-data-ap-northeast-2-prod`
- `s3://beacon-analytics-demo-data-ap-south-1-prod`
- `s3://beacon-analytics-demo-data-ap-south-2-prod`
- `s3://beacon-analytics-demo-data-ap-southeast-1-prod`
- `s3://beacon-analytics-demo-data-ap-southeast-2-prod`
- `s3://beacon-analytics-demo-data-ap-southeast-3-prod`
- `s3://beacon-analytics-demo-data-ca-central-1-prod`
- `s3://beacon-analytics-demo-data-eu-central-1-prod`
- `s3://beacon-analytics-demo-data-eu-south-2-prod`
- `s3://beacon-analytics-demo-data-eu-west-1-prod`
- `s3://beacon-analytics-demo-data-eu-west-2-prod`
- `s3://beacon-analytics-demo-data-us-east-1-prod`
- `s3://beacon-analytics-demo-data-us-east-2-prod`
- `s3://beacon-analytics-demo-data-us-west-2-prod`

スキーマ名で参照することにより、これらのデータセットのテーブルを照会できます。

Schema name	Dataset
<i>tpcds_sf_1</i>	TPC-DS, Scale Factor 1
<i>tpcds_sf_10</i>	TPC-DS, Scale Factor 10
<i>tpcds_sf_100</i>	TPC-DS, Scale Factor 100
<i>tpcds_sf_1000</i>	TPC-DS, Scale Factor 1000
<i>tpch_sf_1</i>	TPC-H, Scale Factor 1
<i>tpch_sf_10</i>	TPC-H, Scale Factor 10
<i>tpch_sf_100</i>	TPC-H, Scale Factor 100

次のページに続く

表 36 – 前のページからの続き

Schema name	Dataset
<i>tpch_sf_1000</i>	TPC-H, Scale Factor 1000
<i>clickbench</i>	ClickBench, 100 million rows
<i>brc_1b</i>	Billion row challenge

i 注釈

ClickBench データセットのデータ列 `EventData` は、日付ではなく整数です。ClickBench の列名には大文字が含まれるため、引用符で囲む必要がありますが、Postgres の引用符のない識別子は小文字と大文字が区別されません。