



Postgres Analytics Accelerator - PGAA

リリース 1.11.0

EnterpriseDB Corporation

2026 年 09 月 11 日

目次

第 1 章 Introduction	3
1.1 始めましょう	3
1.2 PGAA を使用する	3
1.3 PGD を使用した複製	3
1.4 パフォーマンスと最適化	3
1.5 オペレーション	4
1.6 リファレンス	4
第 2 章 Analytics Accelerator 1.11 release notes	5
2.1 PGAA 1.11.0	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 1.10.0	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
2.3 PGAA 1.9.1	7
2.3.1 バグ修正	7
2.4 PGAA 1.9.0	7
2.4.1 機能	7
2.4.2 バグ修正	8
2.4.3 インフラストラクチャとその他の変更	8
2.4.4 非推奨と削除	8
2.5 PGAA 1.8.0	9
2.5.1 機能	9
2.5.2 バグ修正	9
2.5.3 インフラストラクチャとその他の変更	9
2.6 PGAA 1.7.0	10
2.6.1 機能	10
2.6.2 バグ修正	10
2.6.3 インフラストラクチャとその他の変更	10
2.6.4 非推奨と削除	10
2.7 PGAA 1.6.0	11
2.7.1 ハイライト	11
2.7.2 機能	11
2.7.3 パフォーマンスとクエリの最適化	11
2.7.4 レプリケーションとデータ型	11
2.7.5 カタログとストレージ管理	12
2.7.6 バグ修正	13

2.7.7	インフラストラクチャとその他の変更	13
2.7.8	非推奨	14
第 3 章	Analytics Accelerator	15
3.1	Analytics Accelerator quickstart guide	15
3.1.1	前提条件	15
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	16
3.1.3	保存場所の構成	16
3.1.4	分析テーブルの作成	17
3.1.5	分析クエリーの実行	17
3.1.6	クエリプランの分析	18
3.1.7	結論	19
3.2	Analytics Accelerator architecture	20
3.2.1	アーキテクチャの概要	20
3.2.2	コアコンポーネント	20
3.2.3	クエリ実行モード	21
3.3	Concepts	21
3.3.1	ストレージとテーブルフォーマット	21
3.3.2	カタログ	22
3.3.3	クエリーの実行	22
3.3.4	キャッシング	22
3.4	Analytics Accelerator compatibility	23
3.4.1	Postgres の互換性	23
3.4.2	プラットフォームの互換性	23
3.4.3	スパーク	23
3.4.4	PGD	23
3.4.5	WarehousePG	23
3.4.6	サポートされているストレージ形式とプロバイダー	24
3.5	Known issues	25
3.5.1	メンテナンス	25
3.5.2	スパーク	25
3.5.3	オブジェクトストレージ	25
3.5.4	WarehousePG	26
第 4 章	Installing Analytics Accelerator	27
4.1	パッケージのダウンロードとインストール	27
4.2	拡張機能の有効化	27
4.3	インストールの検証	28
第 5 章	Reading tables in object storage	29
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	30
5.1.1	保存場所の作成	30
5.1.2	テーブルの Postgres へのマッピング	30
5.1.3	加速の照会と検証	31
5.2	制限事項	31
第 6 章	Integrating with Iceberg catalogs	33
6.1	カタログ接続の追加	33

6.1.1	REST カタログ	34
6.1.2	AWS S3 テーブル	35
6.1.3	カタログ構成の検証	36
6.1.4	メタデータキャッシュの構成	36
6.2	カタログテーブルの登録	36
6.2.1	カタログの添付	37
6.2.2	カタログのインポート	37
6.2.3	カタログ管理テーブルの作成	38
第 7 章	Writing to object storage	41
7.1	スタンドアロンの保存場所を使用する	41
7.1.1	保存場所の作成	42
7.1.2	CTAS を使用したテーブルの作成	42
7.2	Iceberg カタログを使用する	43
7.2.1	カタログ接続の構成	43
7.2.2	カタログ管理 CTAS テーブルの作成	43
7.3	CTAS の進行状況のモニタリング	44
7.4	CTAS の制限	44
第 8 章	Replicating with PGD	45
8.1	テーブルタイプを理解する	45
8.2	階層化ストレージ戦略	46
8.2.1	階層テーブルの実装	46
8.2.2	アナリティクスへの複製	46
8.2.3	アナリティクスへのオフロード	46
8.3	メタデータ管理	47
第 9 章	Accelerating with Spark	49
9.1	エグゼキューターエンジンの選択	49
9.2	分散実行オプション	49
9.2.1	Distributed Spark execution	50
9.2.2	GPU-accelerated Spark with NVIDIA RAPIDS	54
第 10 章	Scaling with WarehousePG	61
10.1	機能	61
10.2	WarehousePG で PGAA を使用する場合	62
10.3	アーキテクチャの概要	62
10.3.1	コアコンポーネント	62
10.4	WarehousePG での PGAA の仕組み	63
10.4.1	分析テーブルの作成	63
10.4.2	クエリーの実行	64
第 11 章	Monitoring and maintaining analytical tables	77
11.1	テーブル統計の分析	77
11.2	テーブルパフォーマンスの最適化	78
11.2.1	Iceberg マニフェスト同時実行性の制御	78
11.2.2	pgaa.launch_task を使用したテーブルメンテナンスの実行	78
11.2.3	高度なメンテナンス	80

第 12 章 Optimizing query performance	83
12.1 スキャンタイプを理解する	83
12.2 クエリプランの分析	83
12.3 DirectScan フォールバックの管理	84
12.4 コンピューティングプッシュダウンの構成	84
第 13 章 Backing up and restoring	85
13.1 物理的バックアップの範囲を理解する	85
13.2 バックアップ	86
13.2.1 単一ノードのバックアップ	86
13.2.2 PGD クラスターのバックアップ	86
13.2.3 スタンドアロンの Postgres ノードの復元	87
13.2.4 単一の PGD ノードの分離による復元	87
13.2.5 PGD クラスターの復元	88
13.2.6 既知の制限分析レプリケーションを有効にした後の PITR リストア	88
第 14 章 Reference	89
14.1 Configuration parameters	89
14.1.1 エグゼキューターエンジン	89
14.1.2 クエリー動作	90
14.1.3 海鳥	92
14.1.4 カタログの同期	96
14.1.5 キャッシング	96
14.1.6 Postgres DistributedPGD 統合	97
14.1.7 DataFusion から継承された構成	97
14.2 Functions	97
14.2.1 テーブルファンクション	98
14.2.2 カタログファンクション	101
14.2.3 Spark ファンクション	109
14.2.4 バックグラウンドタスクファンクション	111
14.2.5 Postgres DistributedPGD 統合機能	114
14.2.6 ユーティリティファンクション	120
14.3 Table options	121
14.3.1 PGAA テーブルオプション	121
14.3.2 PGD テーブルオプション	123
14.3.3 アクセス方法の管理	123
14.4 Data types and definitions	124
14.4.1 データ型	124
14.4.2 管理用の列挙型とステータス	126
14.5 Benchmark datasets	126

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 Introduction

1.1 始めましょう

- *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。

1.2 PGAA を使用する

- *Installing Analytics Accelerator* 拡張機能をインストールし、Seafoal 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。

1.3 PGD を使用した複製

- *PGD を使用した複製* PGD と PGAA を組み合わせて、自動パーティションのオフロードやリアルタイムレプリケーションからオブジェクトストレージへのコールドデータの外科的アーカイブに至るまで、完全なデータライフサイクルを管理します。

1.4 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。

1.5 オペレーション

- *Backing up and restoring* Barman を使用して PGAA 分析テーブルをバックアップおよび復元します。スタンドアロンの Postgres および PGD 展開の復元プロシージャ。

1.6 リファレンス

- *Configuration parameters* PGAA 拡張機能の動作は、Grand Unified Configuration 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 1.11 release notes

Postgres Analytics Accelerator PGAA 1.11.0 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>PGAA 1.11.0</i>	28 August 2026
<i>PGAA 1.10.0</i>	7 July 2026
<i>PGAA 1.9.1</i>	14 May 2026
<i>PGAA 1.9.0</i>	7 May 2026
<i>PGAA 1.8.0</i>	9 April 2026
<i>PGAA 1.7.0</i>	5 March 2026
<i>PGAA 1.6.0</i>	4 February 2026

2.1 PGAA 1.11.0

リリース日 2026 年 8 月 28 日

Analytics Accelerator 1.11 には、次の機能、機能強化、およびバグ修正が含まれています。

2.1.1 機能

- **WarehousePG の CTAS:** CREATE TABLE AS SELECT CTAS が WarehousePG で動作するようになりました。単一のライタ、WarehousePG 7.5 より後のコーディネーター、または WarehousePG 7.5 以前の 1 つの指定されたセグメントが結果を書き込みます。[CTAS を使用したオブジェクトストレージへの書き込み](#)を参照してください。
- **WarehousePG の PGAA バックグラウンドワーカー** PGAA メンテナンスワーカーとメタストア同期ワーカーが WarehousePG コーディネーターで実行されるようになり、WarehousePG で [カタログの添付](#) との Iceberg カatalog の自動同期が有効になります。
- **CompatScan プッシュダウンの拡張** CompatScan は、新しい `pgaa.enable_orderby_pushdown`、`pgaa.enable_distinct_pushdown`、および `pgaa.enable_window_pushdown` パラメーターによって制御される、行全体参照、ORDER BY / LIMIT / OFFSET、DISTINCT、ウィンドウファンクション、およびセミ/アンチ結合もプッシュダウンするようになりました。[コンピューティングプッシュダウンの構成](#)を参照してください。
- **Iceberg テーブルメタデータキャッシング** Seafoal は、Iceberg テーブルメタデータを REST カタログクライアントにキャッシュし、条件付き要求を使用してエントリを再検証してクエリプランニングのレイテンシーを削減します。[メタデータキャッシュの構成](#)を参照してください。

- **Iceberg マニフェスト同時実行制御**新しい `pgaa.autostart_seafowl_concurrent_manifest_ops` 構成パラメーターを使用すると、Iceberg の同時実行マニフェストオブジェクトストア操作をバインドし、飛行中の要求およびピークマニフェストデコードメモリを制御できます。[Iceberg マニフェスト同時実行性の制御](#) を参照してください。
- **メタデータベースのプランナー統計** PGAA は、ANALYZE 統計のないテーブルの Iceberg、Delta Lake、および Parquet メタデータからプランナー統計を合成し、新しい `pgaa.enable_metadata_stats` および `pgaa.metadata_stats_ttl_s` パラメーターによって制御される CompatScan カーディナリティ推定を向上させます。

2.1.2 バグ修正

- AWS Athena マニフェストの読み取り互換性を修正しました。
- Iceberg 統計ベースのデータファイルのプルーニングが、非パーティション列と未パーティションのテーブルで機能するようになりました。
- `to_char` および `to_number` は、Postgres フォーマットセマンティクスに従うようになりました。
- PGAA テーブルの `reltuples = 0` を記録する ANALYZE が原因で発生した不正確な WarehousePG クエリプランを修正しました。

2.1.3 インフラストラクチャとその他の変更

- DataFusion をバージョン 55 にアップグレードしました。
- オブジェクト全体がキャッシュされた Iceberg マニフェストとマニフェストリスト `.avro` ファイルをオブジェクトストアキャッシュにキャッシュし、Iceberg スキャン計画中のこれらのファイルのキャッシュされない繰り返し読み取りを回避します。

2.2 PGAA 1.10.0

リリース日 2026 年 7 月 7 日

Analytics Accelerator 1.10 には、次の機能、機能強化、およびバグ修正が含まれています。

2.2.1 機能

- **``Seafowl`` の EPAS ファンクションのサポート** 追加の EDB Postgres Advanced Server 互換ファンクションが Seafowl エンジンで認識されるようになり、Oracle 互換の構文を使用するより多くのクエリーに DirectScan アクセラレーションを拡張します。新しい追加は、日付算術 `last_day`、`next_day`、`months_between`、`sysdate`、`systimestamp`、文字列検索 `INSTR`、`INSTRB`、`regexp_substr`、および数値および条件ロジック `to_number`、`width_bucket`、`decode` をカバーします。
- **MinIO AIStor Iceberg REST カタログのサポート** PGAA は、Iceberg REST カタログバックエンドとして MinIO AIStor をサポートするようになりました。
- **``Struct`` および ``Map`` タイプのサポート** PGAA は、Iceberg、Delta Lake、および Parquet テーブルから Struct および Map タイプを読み取り、Postgres の JSON として表示できるようになりました。

2.2.2 バグ修正

- CTAS は、保存場所が見つからない場合、またはパスが許可されない場合、誤解を招くダウンストリームエラーで失敗するのではなく、明確なエラーを発生させるようになりました。
- CREATE TABLE 定義で列のサブセットのみが指定された場合、PGAA テーブルの SELECT * クエリーを修正しました。
- gpstore は、ターゲットデータベースにメタデータが既に存在する PGAA テーブルをスキップし、失敗する代わりに不足しているテーブルのみを復元します。
- 最初の列値が NULL の場合、レプリケーション中のマルチプライマリキー列の解析を修正しました。
- レプリケーション中に完全な忠実度でラウンドトリップするように TIMESTAMPTZ 列を修正しました。
- 階層化テーブルで MERGE を使用する場合のプランナーのクラッシュを修正しました。
- UUID および Decimal 列の Parquet ファイルからの Iceberg 統計のデコードを修正しました。

2.2.3 インフラストラクチャとその他の変更

- オブジェクトストアのキャッシュを拡張して、オフセットおよびサフィックスタイプの GET 要求をカバーしました。
- DataFusion をバージョン 54 にアップグレードしました。

2.3 PGAA 1.9.1

リリース日 2026 年 5 月 14 日

2.3.1 バグ修正

- 定数の結果 JOIN (SELECT MAX(1) AS mx) m ON ... などに還元されるサブクエリーに対して結合するクエリーによってトリガーされる CompatScan 結合プッシュダウンのセグメンテーションフォールトを修正しました。

2.4 PGAA 1.9.0

リリース日 2026 年 5 月 7 日

Analytics Accelerator 1.9 には、次の機能、機能強化、およびバグ修正が含まれています。

2.4.1 機能

- **WarehousePG の PGAA** PGAA が WarehousePG で利用できるようになり、MPP スケールでオブジェクトストレージに対する分散分析クエリーが有効になります。WarehousePG の PGAA テーブルは、Parquet、Delta Lake、および Iceberg フォーマット、Iceberg REST カタログ統合、およびオプションの RAPIDS GPU アクセラレーションを備えた Spark Connect をサポートしています。[Scaling with WarehousePG](#) を参照してください。

- `pgaa.engine_version()` ファンクション `pgaa.seafowl_version()` を置き換え、`pgaa.executor_engine` または Spark Connect によって現在選択されているエグゼキュータエンジンからのバージョン情報を返します。[`pgaa.engine\_version\(\)`](#) を参照してください。
- **DataFusion メモリプール構成** 新しい `pgaa.autostart_seafowl_memory_pool_type` 構成パラメーターを使用すると、Seafowl エンジンが使用する DataFusion メモリプールタイプを選択できます。
- **DataFusion Explain オプション** `explain` 名前空間が `pgaa.datafusion.explain.*` 構成パラメーターを介して公開されるようになり、DataFusion クエリプラン出力を制御できます。

2.4.2 バグ修正

- PGAA テーブルの `VACUUM` および `VACUUM FULL` は、エラーを発生させなくなりました。PGAA データはオブジェクトストレージに存在し、バキューム処理を必要としないため、NOTICE を発行してスキップします。
- `bdr.replicate_ddl` を介して呼び出された場合に `pgaa.delete_catalog` が動作しない問題を修正しました。
- ネストされたフィールドを持つ Iceberg テーブルのマニフェスト解析を修正しました。
- PGAA テーブルへの `INSERT` は、説明的なエラーを返すようになりました。
- `pgaa.spark_sql` で非 UTF-8 マップキーが UTF-8 に正しく強制されなかった問題を修正しました。
- フラッシュ前に発生した PGAA 書き込みエグゼキュータのトランザクション ID ローテーションの問題を修正しました。
- `pgaa.table_mapping` 書き込みが非 PGAA テーブルで誤ってトリガーされる問題を修正しました。
- 再生中の耐久性のある LSN の進歩を修正して、レプリケーションラグを削減します。

2.4.3 インフラストラクチャとその他の変更

- PGAA での行取得パフォーマンスが最大 4 倍向上しました。
- DataFusion をバージョン 53 にアップグレードしました。
- 統合されたログアクセスのために、すべてのライブラリログを Postgres ログに転送しました。
- Flight プロトコルの PGAA と Seafowl 間で gRPC メッセージサイズを調整しました。
- ABIv200 サポートを削除しました。
- バージョン文字列から `-rc.1` サフィックスを削除しました。

2.4.4 非推奨と削除

- `pgaa.seafowl_version()` は削除され、[`pgaa.engine\_version\(\)`](#) に置き換えられました。

2.5 PGAA 1.8.0

リリース日 2026 年 4 月 9 日

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

2.5.1 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向上一般的な EPAS 固有のファンクションが Seafoal エンジンで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。

2.5.2 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。

2.5.3 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。

2.6 PGAA 1.7.0

リリース日 2026 年 3 月 5 日

Analytics Accelerator 1.7 には、次の機能、機能強化、およびバグ修正が含まれています。

2.6.1 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。

2.6.2 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。

2.6.3 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。

2.6.4 非推奨と削除

- 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。
- PGAA へのライタの移行の一環として、`pgaa.flush_task_interval_s` および `pgaa.write_lock_timeout_s` 構成パラメーターを削除しました。代わりに新しい `pgaa.max_in_memory_mb` 構成パラメーターを使用して、ライタのフラッシュ動作を制御します。

2.7 PGAA 1.6.0

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。

2.7.1 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。

2.7.2 機能

2.7.3 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。

2.7.4 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` ファンクションに統合して、複製する前に既存の分析をパージしました。

2.7.5 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。

2.7.6 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、パ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カッシュロジック。
- 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パ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。

2.7.7 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ズ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ズする機能を公開しました。

2.7.8 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 **Seafowl** 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について説明します。
- *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、**Seafowl** 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>
export EDB_SUBSCRIPTION_PLAN=analytics
curl -sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↳SUBSCRIPTION_PLAN/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>
export EDB_SUBSCRIPTION_PLAN=analytics
curl -sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↳SUBSCRIPTION_PLAN/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 Seafoal 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_seafoal = 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 などでも作成されます。\\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

```

↪ -----
↪
↪
`Seafowl`DirectScan: 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_
↪ l2.l_inlinenumber, l2.l2.l_quantity, l2.l2.l_extendedprice, l2.l2.l_discount, l2.l2.l_tax, l2.l2.l_
↪ l2.l_returnflag, l2.l2.l_linestatus, l2.l2.l_shipdate, l2.l2.l_commitdate, l2.l2.l_receiptdate, l2.l2.l_
↪ l2.l_shipinstruct, l2.l2.l_shipmode, l2.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_
↪ l3.l_inlinenumber, l3.l3.l_quantity, l3.l3.l_extendedprice, l3.l3.l_discount, l3.l3.l_tax, l3.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`Seafowl DirectScan`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ています。

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 を活用することにより、Postgres インスタンスはベ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

``Seafowl`` 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 Seafowl 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。

Seafowl は高速 SQL のデフォルトのエグゼキュータですが、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、MinIO AIStor 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`` 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など、データレイクの構造情報のローカルの高速コピーを維持します。Iceberg および Delta Lake テーブルは数千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、すべてのクエリに対してストレージプロバイダーに連絡することなく、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ほぼ瞬時に実行できます。キャッシュされたエントリは、完全に再取得されるのではなく、条件付き ETag 要求で再検証されるため、多くのテーブルを含むカタログ、またはクエリーの同時実行性が低い場合、最もメリットが得られます。[メタデータキャッシュの構成](#) を参照してください。

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 Analytics Accelerator compatibility

3.4.1 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

3.4.2 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。

3.4.3 スパーク

- Apache Spark バージョン 3.5.4 以降。
- Iceberg Spark ライブラリーバージョン 1.9.2 以降

3.4.4 PGD

PGD バージョン 6.1 以降。

3.4.5 WarehousePG

- WarehousePG 7.4 以降。
- whpg-backup バージョン 1.33 以降。

3.4.6 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	WarehousePG
Iceberg	Read		Local, S3, GCP, Azure (via storage locations)	<i>Seafowl</i>	Yes
Iceberg	Read		Iceberg REST catalog	<i>Seafowl</i> Spark	Yes
Delta Lake	Read		Local, S3, GCP, Azure (via storage locations)	<i>Seafowl</i>	Yes
Parquet	Read		Local, S3, GCP, Azure (via storage locations)	<i>Seafowl</i>	Yes
Parquet	Read		Local, S3 (via storage locations)	<i>Seafowl</i> Spark Spark + RAPIDS	<i>Seafowl</i> , Spark, Spark + RAPIDS
Iceberg	Write (via CTAS)	(via	Local, S3, GCP, Azure (via storage locations)	Postgres	Yes
Iceberg	Write (via CTAS)	(via	Iceberg REST catalog	Postgres	Yes
Delta Lake	Write (via CTAS)	(via	Local, S3, GCP, Azure (via storage locations)	Postgres	Yes
Iceberg	Write (using PGD)	(using	Local, S3, GCP, Azure (via storage locations)	<i>Seafowl</i>	No
Delta Lake	Write (using PGD)	(using	Local, S3, GCP, Azure (via storage locations)	<i>Seafowl</i>	No
Iceberg	Write (using PGD)	(using	Iceberg REST catalog	Postgres engine via CompatScan	No

3.5 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。

3.5.1 メンテナンス

- 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。

3.5.2 スパーク

- Apache Spark との統合は以下でサポートされています。
 - 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。
- 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。

3.5.3 オブジェクトストレージ

- 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されていません。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。
- 分析レプリケーション用に構成されたテーブルの名前の変更、またはそれを含むスキーマの名前の変更はサポートされておらず、オブジェクトストレージコピーとの同期を中断します。これらのテーブルまたはそのスキーマの名前を変更しないでください。

3.5.4 WarehousePG

- WarehousePG の PGAA は、標準の Postgres プランナーのみをサポートし、Orca オプティマイザーをサポートしていません。
- 既存のオブジェクトストレージデータにマッピングされたテーブルは読み取り専用です。CTAS を使用して、新しいデータをデータレイクに書き込みます。
- `pgaa.launch_task()` には、`pgcrypto` 拡張機能が必要です。エラー `function gen_random_uuid() does not exist` が表示された場合は、メンテナンス機能を使用する前にコーディネーターで `CREATE EXTENSION pgcrypto;` を実行します。
- すでに PGAA テーブルを含む既存のデータベースに復元する場合、メタデータが既に存在するテーブルはスキップされ、不足しているテーブルのみが復元されます。--redirect-db を使用して、完全なバックアップを新しい空のデータベースに復元します。

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。

!!!注 WarehousePG クラスターに PGAA をインストールする場合、インストールと構成のプロセスが標準の Postgres 展開と異なるため、代わりに *Installing Analytics Accelerator* に従ってください。

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=postgres_distributed
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
→SUBSCRIPTION_PLAN/setup.deb.sh" | sudo -E bash
sudo apt-get install -y <postgres-distribution>-pgaa
```

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=postgres_distributed
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
→SUBSCRIPTION_PLAN/setup.rpm.sh" | sudo -E bash
sudo dnf install -y <postgres-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。
- `<postgres-distribution>` は、ディストリビューションとバージョンのパッケージ名プレフィックスです。たとえば、`postgresql-16` コミュニティ Postgres、`edb-as16` EDB Postgres Advanced Server、または `edb-postgresextended-16` EDB Postgres Extended など。

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 `Seafowl` 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、コンマ区切りを使用して PGAA および PGFS を共有ライブラリのリストに追加します。

```
shared_preload_libraries = <other_libraries>,pgaa,pgfs  
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。新規インストールまたは更新のいずれで作業しているかに応じて、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。

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、`'Seafoal DirectScan'` 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、または MinIO AIStor などの Apache Iceberg REST API に従うサービス。
- **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、MinIO AIStor 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"
  }
);
```

REST カタログは通常、オブジェクトストアの資格情報を保持するか、IMDSv2 や WebIdentity などの方法を使用してオブジェクトストアに対して認証します。クライアントがテーブルを要求すると、カタログはアクセスをチェックし、短期のテーブルスコープの資格情報をクライアントに送ります。カタログはデータアクセスを一元的に管理し、クライアントは長期間のオブジェクトストアの資格情報を保持する必要はなく、カタログで認証するための API トークンのみを保持する必要があるため、この資格情報ベンディングモデルは推奨されるアーキテクチャです。

一部の REST カタログ実装は、資格情報を販売せず、代わりにクライアントが独自のオブジェクトストアの資格情報を直接提供することを想定しています。その場合、オブジェクトストア接続の詳細を使用して `store_config` オブジェクトを `catalog_options` に追加します。

```
SELECT pgaa.add_catalog(
  my_iceberg_catalog,
  iceberg-rest,
  {
    "url": "https://your-catalog-endpoint.com",
    "store_config": {
      "type": "s3",
      "bucket": "your-bucket-name",
      "access_key_id": "your-access-key",
      "secret_access_key": "your-secret-key",
      "region": "us-east-1",
      "endpoint": "https://your-object-store-endpoint.com",
      "allow_http": "true"
    }
  }
);
```

そこで

- `type` オブジェクトストアタイプ、例、S3 互換ストアの `s3`。
- `bucket` カタログの Iceberg テーブルを保持するバケットの名前。
- `access_key_id` および `secret_access_key` バケットへのアクセスに使用される資格情報。
- `region` オブジェクトストアリージョン。
- `endpoint` オブジェクトストアエンドポイント、MinIO などの S3 互換ストアに必要。
- `allow_http` エンドポイントへの非 TLS 接続を許可するには、`"true"` に設定します。

`store_config` は、PGFS と同じオブジェクトストアプロバイダーをサポートします。Azure または Google Cloud Storage の接続オプションについては、PGFS storage provider documentation を参照してください。

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"
  }
);
```

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。

6.1.4 メタデータキャッシュの構成

REST カタログから取得した Iceberg テーブルメタデータをキャッシュするように **Seafowl** を構成し、クエリプランニング中にカタログサービスへのラウンドトリップを削減します。キャッシュされたエントリは、完全に再取得されるのではなく、条件付き ETag 要求で再検証されるため、多くのテーブルを含むカタログ、またはクエリーの同時実行性が最も高くなります。次のパラメーターを使用してキャッシュを調整します。

- `pgaa.autostart_seafowl_enable_metadata_cache` キャッシュをオンまたはオフにします。デフォルトで有効になっています。
- `pgaa.autostart_seafowl_metadata_cache_capacity_mb` 古いエントリを削除するまでにキャッシュが保持するメタデータの最大量を設定します。多くのテーブルを含むカタログの場合、このパラメーターを上げます。
- `pgaa.autostart_seafowl_metadata_cache_ttl_s` ETag 再検証とは無関係に、**Seafowl** が更新するまでにキャッシュされたエントリを信頼する時間を設定します。ETag 再検証のみに依存するには、`0` に設定します。

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** 静的ワンタ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 カタログの添付

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。

注釈

WarehousePG では、`pgaa.attach_catalog()` はコーディネーターでのみメタストア同期ワーカーを実行します。

`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, tpeds_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 meta-data. 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。

i 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 を使用した複製](#) を参照してください。

!!!注 WarehousePG では、単一のライタ、WarehousePG 7.5 より後のコーディネーター、または WarehousePG 7.5 以前の 1つの指定されたセグメントが、CTAS 結果を書き込みます。[CTAS を使用したオブジェクトストレージへの書き込み](#) 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 する必要があります。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アクセラレーション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 Transactional and Analytics 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。

次のページに続く

表 1 – 前のページからの続き

Table type	Storage location	Behavior
PGAA	Object storage only	デルタまたはアイスバーグ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。

詳細については、PGD ドキュメントの [Implementing tiered tables](#) を参照してください。

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。

警告

分析レプリケーション用に構成されたテーブル、またはそれを含むスキーマの名前を変更しないでください。PGD は名前でレプリケートテーブルを追跡するため、名前を変更するとオブジェクトストレージコピーとの同期が中断され、古いまたは重複したデータが残される場合があります。

詳細については、PGD ドキュメントの [Replicating to analytics](#) を参照してください。

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。

詳細については、PGD ドキュメントの [Offloading to analytics](#) を参照してください。

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。ローカルファイルシステムの格納場所は、マルチノード PGD 展開でのこの方法ではサポートされていません。共有オブジェクトストレージプロバイダー S3、GCS、または Azure を使用します。
- **カタログ管理オフロード** Iceberg REST などのメタデータ

[using a centralized catalog service](#) からデータを分離します。

第9章 Accelerating with Spark

デフォルトでは、Postgres アナリティクスアクセラレーター PGAA は、埋め込み分析エンジン *Seafowl* 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 *Seafowl* 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	<i>Seafowl</i>	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	Seafoal	Spark (CPU)	Connect	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.

Seafoal 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_seafoal_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 **Seafowl** 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** Seafowl 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** Seafowl 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、Seafowl プランは、ネットワークレイテンシーとクラスターコーディネーション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

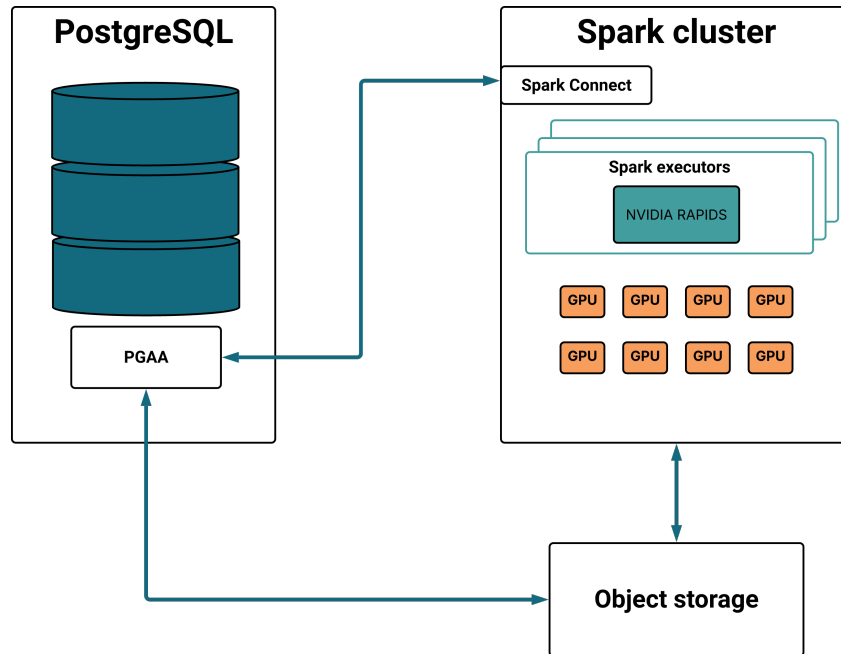


図 1: rapids_arch

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 Scaling with WarehousePG

Postgres Analytics AcceleratorPGAA は、レイクハウス機能で WarehousePG 7 を拡張し、別個のインフラストラクチャやデータの移動をせずにオブジェクトストレージ内のデータに対する分散分析クエリーを可能にします。WarehousePG の PGAA には、WarehousePG 7.4 以降が必要です。

10.1 機能

- **WarehousePG から直接オブジェクトストレージを照会します。** ストレージの場所として S3、Google Cloud StorageGCS、または Azure パスを登録し、それを指す PGAA テーブルを作成し、次の使用で Parquet、Delta Lake、または Iceberg ファイルを照会します。ETL やデータインポートを必要としない標準 SQL。 [ストレージの場所を使用したオブジェクトストレージの照会](#) を参照してください。
- **CTAS を使用してオブジェクトストレージに書き込む** CREATE TABLE AS SELECT CTAS を使用して、クエリ結果をデータレイクの Iceberg または Delta Lake 形式にエクスポートします。 [CTAS を使用したオブジェクトストレージへの書き込み](#) を参照してください。
- **Iceberg カタログに接続する** テーブル検出、スキーマとパーティションのメタデータ、および基になるオブジェクトストレージへの認証のために、外部の Iceberg REST カタログに PGAA を接続します。EDB Postgres 分散 PGD がデータをオブジェクトストレージにレプリケートすると、WarehousePG は共有カタログに接続し、MPP スケールでそのデータを読み取ることができ、両方のプラットフォームで OLTP と分析ワークロードを組み合わせ使用できます。 [Iceberg カタログに接続する](#) を参照してください。
- **Spark での加速** 大規模なワークロードの場合、または既に Spark クラスターを操作している場合、Seafoal ではなく Spark Connect を介してクエリーをルーティングします。クラスターに GPU 対応ノードが含まれる場合、Spark で RAPIDS アクセラレーションを利用でき、コンピューティング集約的なクエリのスループットがさらに向上します。 [Accelerating with Spark](#) を参照してください。
- **バックアップと復元** gpbackup を使用して、PGAA テーブル定義とストレージバインディングをキャプチャします。復元後、テーブルはすぐに正しいオブジェクトストレージパスを参照し、手動の再構成は必要ありません。 [Backing up and restoring PGAA tables on WarehousePG](#) を参照してください。
- **監視と保守** テーブルの状態を監視し、PGAA テーブルで圧縮やバキュームなどのメンテナンス操作を実行します。WarehousePG では、メンテナンスワーカーはコーディネーターでのみ実行されます。 [Monitoring and maintaining analytical tables](#) を参照してください。

現在の制限については、 [Known issues](#) を参照してください。

10.2 WarehousePG で PGAA を使用する場合

次の場合に PGAA を選択します。

- オープンテーブルフォーマット Iceberg、Delta Lake を照会する必要があります。
- PGD などの他のパイプラインと共有する Iceberg REST カタログにアタッチし、カタログ管理の認証を活用して、複製せずに MPP スケールでデータを照会したい。
- より高速な分析クエリーが必要です。PGAA のベクトル化エンジンは、行ごとではなく列方向のバッチでデータを処理します。
- Spark Connect を介してクエリーをルーティングするか、GPU アクセラレーション実行のために RAPIDS を使用したい。

考慮すべき現在の制限事項

- PGAA は、標準の PostgreSQL プランナーのみをサポートし、Orca はサポートしていません。
- 既存のオブジェクトストレージデータにマッピングされたテーブルは読み取り専用です。CTAS を使用して、新しいデータをデータレイクに書き込みます。

10.3 アーキテクチャの概要

WarehousePG の PGAA は、オプションの Iceberg カタログと Spark 統合を使用して、ベクトル化クエリエンジンを通じてクラスターのコーディネーターとセグメントホストをオブジェクトストレージに接続します。

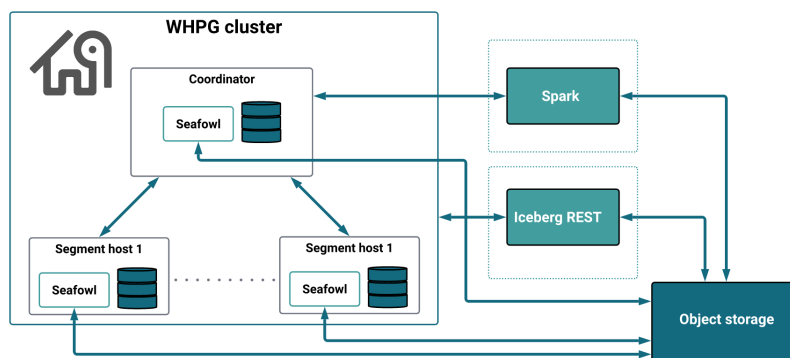


図 1: PGAA on WarehousePG architecture diagram

10.3.1 コアコンポーネント

WarehousePG WHPG

コーディネーターホストと複数のセグメントホストで構成される Postgres ベースの超並列処理 MPP データベース。コーディネーターはクエリのプランニングとディスパッチを処理し、セグメントはクエリを並列実行します。

PGAA 拡張機能

WarehousePG クエリプランナーと統合して、分析エンジンにオフロードできるクエリを特定します。デフォルトのベクトル化実行エンジンとして **Seafowl** をバンドルし、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 拡張機能

オブジェクトストレージ接続のストレージレイヤーを提供する PGAA の依存関係。資格情報管理を処理し、S3、GCS、および Azure へのアクセスを抽象化します。

``Seafowl``

PGAA にバンドルされたベクトル化クエリエンジン。WarehousePG インストールでは、Seafowl は WHPG コーディネーターと各セグメントホストで **systemd** サービスとして実行され、コーディネーターのみと分散クエリーの実行の両方が有効になります。

Iceberg REST カタログオプション

テーブルスキーマ、パーティション、およびスナップショットを追跡する外部メタデータレジストリ。PGAA は、登録されたカタログを介して、基になるオブジェクトストレージへのテーブル検出と認証を有効にします。

Spark クラスターオプション

代替実行エンジンとして使用される外部 Apache Spark クラスター。Spark Connect を介して構成された場合、WHPG コーディネーターは Seafowl の代わりに Spark にクエリーを転送します。

オブジェクトストレージ

分析データのソース。PGAA は、S3、GCS、および Azure をサポートし、WarehousePG へのデータインポートはせずに、クエリ時に Parquet、Delta Lake、および Iceberg ファイルの直接照会をサポートしています。

10.4 WarehousePG での PGAA の仕組み

PGAA は、分析テーブルが定義およびオブジェクトストレージにバインドされるストレージレイヤーと、プランナーとベクトル化クエリエンジンが連携してクラスター全体にクエリを分散および実行するクエリ実行レイヤーの 2 つのレベルで WarehousePG と統合します。

10.4.1 分析テーブルの作成

PGAA は、カスタムテーブルアクセス方法 **pgaa** を提供し、WarehousePG にデータをインポートせずに、標準の SQL を介してオブジェクトストレージにあるテーブルを定義および照会できます。PGAA カタログテーブルは **DISTRIBUTED REPLICATED** ポリシーを使用し、すべてのノードでメタデータを利用できるようにします。

ストレージ場所または Iceberg カタログにマップされたテーブルは読み取り専用です。データは、オブジェクトストレージに既に存在する必要があります、たとえば、PGD または別のパイプラインによって書き込まれます。

CTAS を使用して新しいデータを書き込むこともできます。単一のライター、WarehousePG 7.5 より後の WarehousePG のコーディネーター、または WarehousePG 7.5 以前の 1 つの指定されたセグメントがデータを書き込み、PGAA は結果のテーブルにクラスター全体で一貫性を保つ分散ポリシーを割り当てて、他の

ものと同様に照会および結合できます。他の PGAA テーブル。CTAS を使用したオブジェクトストレージへの書き込みを参照してください。

10.4.2 クエリーの実行

データが特定のセグメントホストに固定されるネイティブの WarehousePG テーブルとは異なり、PGAA データはオブジェクトストレージに存在し、任意のノードからアクセスできるため、PGAA に各クエリーに対して最も効率的な実行戦略を選択する自由度を与えます。

i 注釈

PGAA は、標準の PostgreSQL プランナーのみをサポートし、Orca オプティマイザーはサポートしません。

PGAA は、2つのエンジンのいずれかを介してクエリーを実行します。Seafoal は、PGAA にバンドルされたデフォルトのベクトル化エンジン、または Spark クラスターをポイントするように Spark Connect を構成する場合。

Spark がアクティブなエンジンである場合、コーディネーターは、セグメントホストを関与させずに、Spark Connect を介してクエリーを Spark クラスターに転送します。Spark は、Seafoal スキャンモードとは無関係に、独自の計画と配布を処理します。

Seafoal がアクティブなエンジンの場合、PGAA は、クエリの性質に応じて2つのスキャンモードのいずれかを選択します。

- **DirectScan** PGAA は、セグメントホストを介さずに、コーディネーター上の Seafoal にクエリーを完全にオフロードします。
- **CompatScan** クエリーに、ネイティブ WarehousePG ヒープテーブルとの結合など、Seafoal が処理できない操作が含まれる場合に使用されます。PGAA は、テーブルごとに、2つの候補実行プランを WarehousePG プランナーに提案し、プランナーは安価な方を選択します。
- **Strown locus MPP** スキャンはすべてのセグメントホストに分散され、それぞれがデータの一部を並列して読み取ります。
- **一般軌跡** テーブル全体をコーディネーターとすべてのセグメントで処理できます。[aggregation push-down](#) を有効にすると、コーディネーターは結果を返す前に集計を処理します。これにより、クライアントに返されるデータをさらに削減できます。

両方の戦略を組み合わせると、異なるサイズの PGAA テーブル間の結合に特に役立ちます。たとえば、小さなテーブルを大きなテーブルに対して結合する場合、プランナーは小さなテーブルの一般軌跡を使用し、すべてのセグメントで完全にスキャンしてローカルハッシュマップにロードできますが、大きなテーブルに Strown 軌跡を使用します。各セグメントは、その部分のみを読み取ります。結合は、ホスト間でデータをシャッフルせずに、各セグメントでローカルに実行されます。

Engine	Scan mode	Execution
Seafoal	DirectScan	Coordinator only.

次のページに続く

表 1 – 前のページからの続き

Engine	Scan mode	Execution
<i>Seafowl</i>	CompatScan	Coordinator and every segment scan the full table (General locus), or each segment reads a portion of the data in parallel (Strewn locus), based on cost.
Spark	SparkDirectScan	Spark cluster via coordinator only.

Seafowl スキャンモードの詳細については、[スキャンタイプを理解する](#) を参照してください。

Installing PGAA on WarehousePG

前提条件

PGAA をインストールする前に、次のことを確認します。

- x86_64 RHEL 8 または RHEL 9 で実行されている WarehousePG 7.4 以降のクラスター。
- コーディネーターホストで構成された EDB リポジトリへのアクセス。
- すべてのホストで sudo 特権を持つ gpadmin ユーザー。

特に記載のない限り、すべての手順は gpadmin ユーザーとして実行する必要があります。

拡張機能のインストール

パッケージのダウンロードとインストール

1. コーディネーターで EDB サブスクリプション資格情報を設定します。

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=analytics
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
→SUBSCRIPTION_PLAN/setup.rpm.sh" | sudo -E bash
sudo dnf download edb-whpg7-pgaa
```

edb-whpg7-pgfs は、edb-whpg7-pgaa の宣言された依存関係であり、自動的にダウンロードされます。

1. コーディネーターで、WHPG クラスター内のすべてのホストをリストするファイル all_hosts を作成します。例

```
cdw
scdw
sdw1
sdw2
```

1. すべてのホストにパッケージを転送してインストールします。

```
gpssh -f all_hosts -e "scp gpadmin@$(hostname):~/edb-whpg7-pgaa *.rpm /tmp/ && sudo
→dnf install -y /tmp/edb-whpg7-pgaa* .rpm"
```

1. パッケージがすべてのホストにインストールされていることを確認します。

```
gpssh -f all_hosts -e rpm -q edb-whpg7-pgaa edb-whpg7-pgfs
```

拡張機能の有効化

WarehousePG インストールでは、Seafoal は、拡張機能自分自身によって制御されるバックグラウンドワーカーではなく、systemd サービスとして管理されます。RPM は、systemd ユニットファイルを自動的にインストールします。

1. コーディネーターで、コンマ区切りを使用して共有ライブラリのリストに PGAA および PGFS を追加し、クラスターを再起動します。

```
gpconfig -c shared_preload_libraries -v <other_libraries>,pgaa,pgfs
gpstop -ra
```

1. コーディネーターとすべてのセグメントホストを含むすべてのホストで Seafoal サービスを有効にして開始します。

```
gpssh -f all_hosts -e sudo systemctl enable seafoal && sudo systemctl start seafoal
```

!!!注 systemd が利用できない環境 Docker などにインストールする場合、スーパーバイザプロセスやエントリポイントスクリプトなどの他の手段を介して `Seafoal` ワーカーをデーモン化する必要があります。重要な制約は、ホストごとに 1 つの `Seafoal` ワーカーのみを実行する必要があることです。同じホストで複数のワーカーを実行しているか、いずれかの実行にまったく失敗すると、問題が発生します。.. ::

!!!注 Seafoal は、デフォルトでポート 9090 で Prometheus メトリックエンドポイントを公開します。Prometheus が既に同じホストで実行され、ポート 9090 にバインドされている場合、Seafoal は起動に失敗します。競合を解決するには、影響を受けるホストで `etc/edb/pgaa/seafoal.toml` の `port` の値を編集し、Seafoal サービスをリスタートします。.. :: 1. コーディネーターに接続し、PGAA を使用する各データベースに拡張機能を作成します。

```
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。

拡張機能のアップグレード

WarehousePG 上の既存の PGAA インストールを新しいバージョンにアップグレードするには、[拡張機能のインストール](#) の代わりにこのセクションの手順を完了します。

1. コーディネーターで、新しいパッケージバージョンをダウンロードします。

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=analytics
curl -sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
→SUBSCRIPTION_PLAN/setup.rpm.sh" | sudo -E bash
sudo dnf download edb-whpg7-pgaa
```

1. コーディネーターとすべてのセグメントホストを含むすべてのホストでパッケージを転送およびアップグレードします。

```
gpssh -f all_hosts -e "scp gpadmin@$(hostname):~/edb-whpg7-pgaa *.rpm /tmp/ && sudo
↪dnf install -y /tmp/edb-whpg7-pgaa* .rpm"
```

1. すべてのホストで Seafoal サービスを再起動して、新しいバイナリをロードします。

```
gpssh -f all_hosts -e sudo systemctl restart seafoal
```

1. WarehousePG クラスターを再起動して、新しい pgaa ライブラリをロードします。

```
gpstop -ra
```

1. コーディネーターに接続し、PGAA がインストールされている各データベースの拡張機能を更新します。

```
ALTER EXTENSION pgaa UPDATE;
```

!!!注すべてのホスト、コーディネーター、セグメントでパッケージをアップグレードし、ALTER EXTENSION を実行する前に両方の再起動を完了します。SQL 移行スクリプトは、新しいライブラリにのみ存在するファンクションを想定しているため、まだアップグレードされていない pgaa ライブラリに対して実行すると、could not find function "..._wrapper" in file ... などのエラーで失敗します。

インストールの検証

1. \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();
```

1. Seafoal サービスがすべてのホストで実行されていることを確認します。

```
gpssh -f all_hosts -e sudo systemctl status seafoal --no-pager
```

サービスの状態とログの確認

Seafoal は、systemd を介して構造化 JSON ログを書き込みます。

- 単一ホストでステータスと最近のログ出力を確認するには

```
sudo systemctl status seafoal
```

- ライブログをストリーミングするには

```
sudo journalctl -u seafoal -f
```

- すべてのホストのログを一度に確認するには

```
gpssh -f all_hosts -e sudo journalctl -u seaflowl --no-pager -n 50
```

WarehousePG コーディネーターとセグメントログについては、[Viewing the database server log files](#) を参照してください。

Reading from and writing to object storage on WarehousePG

ストレージの場所または Iceberg カタログを使用してオブジェクトストレージからデータを読み取り、オプションで Spark Connect で高速化し、CTAS を使用して新しいデータをオブジェクトストレージに書き込みます。

ストレージの場所を使用したオブジェクトストレージの照会

ストレージ場所はオブジェクトストレージに直接接続し、Parquet、Delta Lake、および Iceberg ファイルをサポートします。コーディネーターで保存場所と PGAA テーブルを作成し、標準の SQL を使用してデータを照会します。WarehousePG は、最も効率の良い実行モードを自動的に選択し、クエリーの性質に応じて、コーディネーターでクエリーを実行またはセグメントホスト全体に分散します。

!!!注ストレージ場所または Iceberg カタログで作成されたテーブルは読み取り専用で、データは既にオブジェクトストレージに存在する必要があります。新しいデータをオブジェクトストレージに書き込むには、[CTAS を使用したオブジェクトストレージへの書き込み](#) を参照してください。

保存場所の構成の詳細については、[Reading tables in object storage](#) を参照してください。

例

TPC-H SF1 データを含むパブリック S3 バケットに接続し、3つのテーブルを登録し、分析クエリを実行して、DirectScan 純粋な PGAA と CompatScan ネイティブヒープテーブルと結合した PGAA の両方をデモンストレーションします。

1. コーディネーターで保存場所を作成します。

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

1. データを指す PGAA テーブルを作成します。

```
CREATE TABLE customer () USING PGAA
WITH (pgaa.storage_location = sample-data, pgaa.path = tpch_sf_1/customer);

CREATE TABLE orders () USING PGAA
WITH (pgaa.storage_location = sample-data, pgaa.path = tpch_sf_1/orders);

CREATE TABLE lineitem () USING PGAA
WITH (pgaa.storage_location = sample-data, pgaa.path = tpch_sf_1/lineitem);
```

1. 分析クエリを実行します。

```

SELECT
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) AS revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = BUILDING
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < DATE 1995-03-15
    AND l_shipdate > DATE 1995-03-15
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue DESC,
    o_orderdate
LIMIT 10;

```

1. EXPLAIN を使用して、クエリプランを観察します。PGAA は最初に DirectScan を試行し、コーディネーター上の Seafowl に実行を完全にオフロードします。プランには、述語プッシュダウン、集計、ソートを含む完全な Seafowl 論理プランが表示されます。WarehousePG プランナーが関係していないため、Optimizer 行が表示されないことに注意してください。

QUERY PLAN

```

-----
↪ -----
`Seafowl`DirectScan: Logical Plan
  Limit:  skip=0, fetch=10
    Sort:  revenue DESC NULLS FIRST, orders.o_orderdate ASC NULLS LAST
      Projection:  lineitem.l_orderkey, sum(lineitem.l_extendedprice * Int64(1) - ↪
↪ lineitem.l_discount) AS revenue, orders.o_orderdate, orders.o_shippriority
        Aggregate:  groupBy=[[lineitem.l_orderkey, orders.o_orderdate, orders.o_
↪ shippriority]], aggr=[[sum(lineitem.l_extendedprice * (Int64(1) - lineitem.l_
↪ discount))]]
          Filter:   customer.c_mktsegment = Utf8("BUILDING") AND customer.c_custkey = ↪
↪ orders.o_custkey AND lineitem.l_orderkey = orders.o_orderkey AND orders.o_orderdate
↪ < CAST(Utf8("1995-03-15") AS Date32) AND lineitem.l_shipdate > CAST(Utf8("1995-03-15
↪ ") AS Date32)

```

(次のページに続く)

(前のページからの続き)

```

Cross Join:
Cross Join:
  TableScan: customer
  TableScan: orders
  TableScan: lineitem

```

(11 rows)

PGAA は、クエリーにネイティブヒープテーブルとの結合など、Seafoal が処理できない操作が含まれる場合、CompatScan にフォールバックします。例、customer をネイティブ segments テーブルと結合する場合

```

CREATE TABLE segments (seg_id INT, seg_name TEXT) DISTRIBUTED BY (seg_id);
INSERT INTO segments VALUES (1, BUILDING), (2, MACHINERY);

EXPLAIN SELECT c_name, seg_name
FROM customer
JOIN segments ON c_mktsegment = seg_name;

```

PGAA は最初に DirectScan を試行し、次の通知を使用して CompatScan にフォールバックします。

```

NOTICE: PGAA DirectScan failed: QueryPlanningError("Error during planning: table_
→default.public.segments not found"). Running the query in compatibility mode.
QUERY PLAN
-----
→ -----
Gather Motion 4:1 (slice1; segments: 4) (cost=1414.00..246348.40 rows=748960_
→width=64)
  -> Hash Join (cost=1414.00..236986.40 rows=187240 width=64)
      Hash Cond: (customer.c_mktsegment = segments.seg_name)
      -> Custom Scan (`Seafoal`CompatScan) on customer (cost=25.00..1131.25_
→rows=37500 width=64)
          `Seafoal`Plan: Logical Plan
          Projection: r1.c_name, r1.c_mktsegment
          SubqueryAlias: r1
          TableScan: public.customer
          `Seafoal`Query: select r1.c_name, r1.c_mktsegment from public.customer_
→as r1
      -> Hash (cost=769.00..769.00 rows=49600 width=32)
          -> Broadcast Motion 4:4 (slice2; segments: 4) (cost=0.00..769.00_
→rows=49600 width=32)
              -> Seq Scan on segments (cost=0.00..149.00 rows=12400 width=32)
Optimizer: Postgres-based planner
(13 rows)

```

Gather Motion 4:1 は、Stewrn ローカスプランを示します。`Seafoal`CompatScan は、4 つのセグメントホストの各ホストで並列実行され、それぞれが customer データのスライスを読み取り、結果がコーディ

ネーターに収集されます。ヒープテーブルとの結合は、WarehousePG プランナーによって処理されます。下部の Optimizer: Postgres-based planner 行は、Seaflowl がすべてを独立して処理する DirectScan とは異なり、Postgres プランナーがプラン全体を担当したことを確認します。

PGAA が General 遺伝子座と Strewn 遺伝子座を選択する方法の詳細については、[クエリ実行モード](#) を参照してください。

Iceberg カタログに接続する

WarehousePG コーディネーターに外部 Iceberg REST カタログを接続して、クラスター全体でカタログ管理テーブルを照会可能にします。

WarehousePG で PGAA を使用して、EDB Postgres DistributedPGD がオブジェクトストレージにレプリケートしたデータを読み取ることができます。PGD は、Iceberg 形式のデータを書き込み、Iceberg REST カタログに登録します。WarehousePG は、データを移動せずに MPP スケールで接続および照会できます。この設定の PGD 側については、[PGD を使用した複製](#) を参照してください。

!!!注 `pgaa.attach_catalog()` は、メンテナンswーカーと同じ方法で、WarehousePG コーディネーターでのみメタストア同期ワーカーを実行します。PGAA カタログテーブルはレプリケートテーブルポリシーを使用するため、すべてのノードで同期されたメタデータを利用できます。

資格情報構成やサポートされているカタログタイプを含む完全なセットアップ手順については、[Integrating with Iceberg catalogs](#) を参照してください。

例

1. コーディネーターにカタログに登録します。

```
SELECT pgaa.add_catalog(
  my_catalog,
  iceberg-rest,
  {"url": "https://iceberg-catalog.example.com"}
);
```

1. テーブルのメタデータをインポートします。

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

1. オプションで、インポートを特定の名前空間に制限します。

```
SELECT pgaa.import_catalog(my_catalog, my_schema);
```

インポートされると、テーブルは WarehousePG セッションから照会できます。[例](#) で説明されているのと同じクエリプランパターンが適用されます。EXPLAIN を使用して、PGAA が DirectScan または CompatScan を使用しているか、およびどの遺伝子座が選択されているかを観察します。

Spark での加速

Spark Connect を介してクエリーを外部 Apache Spark クラスターにルーティングします。このモードでは、コーディネーターはクエリーを Spark に直接転送し、セグメントホストは実行に関与せず、Spark は WarehousePG プランナーとは無関係に独自の計画と配布を処理します。

完全なセットアップ手順、GPU アクセラレーションオプション、およびパフォーマンスガイダンスについては、[Accelerating with Spark](#) を参照してください。

例

1. `pgaa.spark_connect_url` を設定し、コーディネーターの実行エンジンを切り替えます。

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

1. 例と同じパブリック TPC-H サンプルデータを使用して、ストレージ場所と PGAA テーブルを作成します。Spark Connect は Parquet のみをサポートしているため、`pgaa.format` を明示的に設定する必要があります。

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

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

CREATE TABLE orders () USING PGAA
WITH (pgaa.storage_location = sample-data, pgaa.path = tpch_sf_1/orders, pgaa.format_
  →= parquet);

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

1. 同じ TPC-H Q3 クエリーを実行します。コーディネーターは実行を Spark に転送します。

```
SELECT
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) AS revenue,
  o_orderdate,
  o_shippriority
FROM
  customer,
  orders,
```

(次のページに続く)

(前のページからの続き)

```

lineitem
WHERE
  c_mktsegment = BUILDING
  AND c_custkey = o_custkey
  AND l_orderkey = o_orderkey
  AND o_orderdate < DATE 1995-03-15
  AND l_shipdate > DATE 1995-03-15
GROUP BY
  l_orderkey,
  o_orderdate,
  o_shippriority
ORDER BY
  revenue DESC,
  o_orderdate
LIMIT 10;

```

1. EXPLAIN を使用して、クエリプランを観察します。Spark がアクティブなエンジンの場合、PGAA は SparkDirectScan ノードを生成します。このプランは、Spark クラスター内で完全に処理される述語プッシュダウン、結合、集計を含む Spark 独自の実行を示します。Optimizer 行は表示されず、WarehousePG プランナーが関与していないことを確認します。

QUERY PLAN

```

-----
->-----
SparkDirectScan: Logical Plan
  AdaptiveSparkPlan: isFinalPlan=false
    +- TakeOrderedAndProject: (limit=10, orderBy=[revenue DESC, o_orderdate ASC])
      +- HashAggregate: (keys=[l_orderkey, o_orderdate, o_shippriority],
->functions=[sum(...)])
        +- HashAggregate: (keys=[l_orderkey, o_orderdate, o_shippriority],
->functions=[partial_sum(...)])
          +- Project: [o_orderdate, o_shippriority, l_orderkey, l_extendedprice, l_
->discount]
            +- SortMergeJoin: [o_orderkey], [l_orderkey], Inner
              :- Sort [o_orderkey ASC NULLS FIRST]
              : +- Exchange hashpartitioning(o_orderkey, 200)
              : +- BroadcastHashJoin [c_custkey], [o_custkey], Inner,
->BuildLeft
                :- BroadcastExchange
                : +- Filter (c_mktsegment = BUILDING)
                : +- FileScan parquet [c_custkey, c_mktsegment]
                : Location: s3a://beacon-analytics-demo-data-us-
->east-1-prod/tpch_sf_1/customer
                : +- Filter (o_orderdate < 1995-03-15)
                : +- FileScan parquet [o_orderkey, o_custkey, o_orderdate,

```

(次のページに続く)

(前のページからの続き)

```

o_shippriority]
      :                               Location: s3a://beacon-analytics-demo-data-us-
→east-1-prod/tpch_sf_1/orders
      +- Sort [l_orderkey ASC NULLS FIRST]
      +- Exchange hashpartitioning(l_orderkey, 200)
      +- Filter (l_shipdate > 1995-03-15)
      +- FileScan parquet [l_orderkey, l_extendedprice, l_
→discount, l_shipdate]
      Location: s3a://beacon-analytics-demo-data-us-east-1-
→prod/tpch_sf_1/lineitem
(22 rows)

```

1. 完全な Spark 実行プランを検査するには、ブラウザーで `http://<spark-host>:4040` の Spark UI を開きます。[SQL] タブに移動して、ステージ境界、シャッフル操作、ステージごとのメトリックを含むクエリプランの視覚的な内訳を表示します。

CTAS を使用したオブジェクトストレージへの書き込み

CREATE TABLE AS SELECT CTAS を使用して、スタンドアロンストレージ場所または Iceberg カタログをターゲットに、他の PGAA インストールと同じ構文でクエリ結果をオブジェクトストレージに書き込みます。単一のライタ、WarehousePG 7.5 より後のコーディネーター、または WarehousePG 7.5 以前の 1 つの指定されたセグメントが、すべてのセグメントに並列して書き込むのではなく、データを書き込みます。PGAA は、結果のテーブルに、クラスター全体で一貫性を保つ分散ポリシーを提供するため、他の PGAA テーブルと同様に照会および結合できます。

CTAS 構文、保存場所、および Iceberg カタログの詳細については、[Writing to object storage](#) を参照してください。

例

分散 WarehousePG テーブルを作成し、CTAS を使用してオブジェクトストレージにエクスポートし、結果を元のテーブルに対して結合できることを確認します。

1. コーディネーターに保存場所を作成します。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。

```

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": "..."}]
→.
);

```

次のファンクションを実行して、資格情報に書き込みアクセスがあることを確認します。成功した場合は NULL を結果ます。

```

SELECT pgaa.test_storage_location(my_lake_data, true);

```

1. 分散ソーステーブルを作成し、CTAS でエクスポートします。

```
CREATE TABLE orders_local (order_id int, amount numeric) DISTRIBUTED BY (order_id);
INSERT INTO orders_local SELECT i, i * 10 FROM generate_series(1, 100) AS i;

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

1. 行数を確認し、元の分散テーブルに対して新しいテーブルを結合します。

```
SELECT count(*) FROM orders_export;

SELECT count(*)
FROM orders_export e
JOIN orders_local l ON e.order_id = l.order_id;
```

Backing up and restoring PGAA tables on WarehousePG

PGAA テーブルは、WarehousePG クラスターではなく、オブジェクトストレージにデータを保存します。whpg-backup パッケージが提供する [gpbakup utility](#) は、PGAA と通常のヒープテーブルの両方を含む混合データベースをサポートします。PGAA テーブルの場合、gpbakup はメタデータテーブルマッピング、格納場所、パスバインディングのみをキャプチャし、オブジェクトストレージの基になるデータはキャプチャしません。このメタデータは gpstore によって自動的に復元されるため、手動で再構成せずに、復元後にすぐに PGAA テーブルを照会できます。

オブジェクトストレージの基になるデータの保護は、オブジェクトストレージプロバイダーの責任です。

前提条件

whpg-backup バージョン 1.33 以降が WarehousePG クラスターにインストールされている。

バックアップ

コーディネーターで gpbakup を実行します。

```
gpbakup --dbname mydb --backup-dir /path/to/backup-dir
```

オプションの完全なリストについては、[gpbakup](#) リファレンスページを参照してください。

出力からのバックアップタイムスタンプをメモします。復元手順に必要になります。例

```
[INFO]:-Backup Timestamp = 20260417100433
```

復元

`--create-db` を使用して `gprestore` にターゲットデータベースを作成し、`--redirect-db` を使用してバックアップされたデータベース以外のデータベースにリストアします。`gprestore` は、`pgaa` および `pgfs` 拡張機能を自動的に作成します。

既に PGAA テーブルがある既存のデータベースに復元する場合、メタデータが既に存在する場合テーブルはスキップされるため、不足しているテーブルのみが復元されます。

1. `gprestore` を実行します。

```
gprestore --backup-dir /path/to/backup-dir --timestamp <backup-timestamp> \  
--redirect-db mydb_restored --create-db
```

オプションの完全なリストについては、[gprestore](#) リファレンスページを参照してください。

1. `gprestore` 出力の `Tables restored` をチェックして、リストアが成功したことを確認します。
2. ストレージ場所が復元され、PGAA テーブルがオブジェクトストレージに対してすぐに照会可能であることを確認します。

```
SELECT * FROM pgfs.list_storage_locations();  
SELECT count(*) FROM my_table;
```

第11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。

11.1 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」で何が起き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。

11.2 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。

11.2.1 Iceberg マニフェスト同時実行性の制御

小さいファイル問題の影響を受けるテーブルには、多くの Iceberg マニフェストエントリも蓄積されます。**Seafowl** は、これらのマニフェストを読み取ってデコードして、スキャンとメンテナンス操作を計画します。一度に多数のマニフェストに対してこれを行うと、メモリの使用が増加し、オブジェクトストア要求のバーストが生成される可能性があります。この同時実行性を `pgaa.autostart_seafowl_concurrent_manifest_ops` で制限します。これは、同時マニフェストオブジェクトストア操作の数を制限します。オブジェクトストアとホストに容量がある場合、これを下げてピークメモリを削減し、非常に大きなマニフェストを持つテーブルのバーストを要求するか、値を上げて多くの小さなマニフェストを持つテーブルのプランニングを高速化します。

11.2.2 `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。

i 注釈

WarehousePG では、バックグラウンドメンテナンスワーカーはコーディネーターでのみ実行され、セグメントホストでは実行されません。 `pgaa.launch_task()` を呼び出す前に、コーディネーターに `'pgcrypto'` 拡張機能 `'CREATE EXTENSION IF NOT EXISTS pgcrypto;'` をインストールします。

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

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。

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。

11.2.3 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($$
    CALL preexisting.system.rewrite_data_files(
        table => "preexisting"."ns-1"."table-1",
        strategy => sort,
        sort_order => value DESC,
        options => map(rewrite-all, true)
    )
$$);
```

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12章 Optimizing query performance

Postgres アナリティクスアクセラレータ PGAA は、分析ワークロードを **Seafowl** 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。

12.1 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: **Seafowl** 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 クエリーに、**Seafowl** がネイティブに処理できない 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、**Seafowl** 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。

- **Seafowl** はファイルを読み取り、許可されたプッシュダウン join や group-by など进行处理します。
- Postgres は **Seafowl** から最適化されたストリームを受信して、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。

12.2 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** `Seafowl DirectScan` または **SparkDirectScan** ノードを探します。これは、**Seafowl** 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 に返すことを示します。
- **CompatScan** `Seafowl CompatScan` または **SparkCompatScan** ノードが表示される場合、クエリーは「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。

12.3 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。

12.4 コンピューティングプッシュダウンの構成

次の設定を使用して、リソースを大量に使用する操作をオフロードして、ハイブリッドモードでも高いパフォーマンスを維持します。これらの設定はパフォーマンスのセーフティネットとして機能し、デフォルトで有効になっています。

- `pgaa.enable_join_pushdown` システムがセミ結合およびアンチ結合を含む結合をリモート実行者 **Seafowl** または **Spark** に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。
- `pgaa.enable_orderby_pushdown` ORDER BY および定数 LIMIT / OFFSET のオフロードを有界トップ N 操作として承認するため、ソートと切り捨ては、完全な結果を Postgres にストリーミングした後ではなく、ソースで発生します。
- `pgaa.enable_distinct_pushdown` プレーン SELECT DISTINCT のリモート実行プログラムへのオフロードを承認します。
- `pgaa.enable_window_pushdown` ウィンドウ機能のリモート実行プログラムへのオフロードを承認します。

`to_json(r.*)` などの行全体参照は、リモート実行プログラムに自動的にプッシュダウンされ、専用のトリガーはありません。

!!!注 `pgaa.enable_distinct_pushdown` は WarehousePG には適用されません。そこでは、MPP プランナーが PGAA のプッシュダウンステージが実行される前に、スキャン上のモーションと HashAggregate で DISTINCT を重複排除します。

第13章 Backing up and restoring

Barman を使用して、分析テーブルをバックアップします。Barman は、PGAA 拡張機能がインストールされている Postgres ノードの物理バックアップを実行します。何がバックアップされるかは、展開内のテーブルの種類によって異なります。

13.1 物理的バックアップの範囲を理解する

PGAA テーブルは、展開に応じてさまざまなカテゴリに分類されます。

- **分析テーブル** オブジェクトストレージ内のデータを完全にポイントする PGAA アクセス方法を使用するテーブル。スタンドアロンの Postgres と Postgres DistributedPGD 展開の両方で利用できます。
- **階層テーブル** PGD のみローカル Postgres ヒープ EDb_tran_2 が有効になっている場合、オブジェクトストレージにもレプリケートされるホットパーティションと、オブジェクトストレージにオフロードされたコールドパーティションを含むパーティションベースのテーブル。
- **HTAP テーブル** PGD のみ Postgres ヒープにフルコピーとオブジェクトストレージにフルコピーを持つテーブル。
- **内部メタデータテーブル** データディレクトリに通常のヒープテーブルとして保存される PGAA 独自のシステムテーブル `pgaa.table_mapping` や `pgaa.catalog` など。すべての展開で利用できます。

標準の Postgres 物理バックアップは、以下をキャプチャします。

- 基になるデータ自分自身ではなく、オブジェクトストレージを参照するメタデータとしての分析テーブル定義。
- 完全にバックアップされた、ローカルデータディレクトリに保存されている階層化テーブルホットパーティションと HTAP テーブルからのヒープデータ。
- PGAA の内部メタデータテーブル `pgaa.table_mapping`、`pgaa.catalog` など。

バックアップではキャプチャされません。

- オブジェクトストレージ内の Iceberg、Delta、または Parquet ファイル。
- Iceberg REST カタログエントリ Lakekeeper が保持するものなど。

たとえば、PGD 階層化テーブルの場合、Barman はホットデータを完全にバックアップしますが、コールドデータへのポインターのみをバックアップし、コールドデータ自分自身はバックアップしません。

ディザスタリカバリーを完全にカバーするには、Postgres バックアップとともにオブジェクトストアとカタログサービスのスナップショットを作成します。

13.2 バックアップ

`backup_method = postgres` を使用して Barman を構成します。これは、`pg_basebackup` を実行し、カタログ管理、保持ポリシー、WAL アーカイブ、および `max_worker_processes` 調整を自動的に処理する `barman recover` ワークフローを追加します。

`barman` および `streaming_barman` Postgres ロールとレプリケーションスロットを含む完全なサーバーサイドの設定については、[Barman documentation](#) を参照してください。

13.2.1 単一ノードのバックアップ

PGAA を使用するスタンドアロンの Postgres ノードの場合、バックアップはメタデータとオブジェクトストレージへのポインタのみをキャプチャします。すべての PGAA テーブルデータはオブジェクトストレージにあり、Postgres からレプリケートされることはないため、検討するホットデータはありません。

1. PGAA 対応 Postgres ノードのサーバーエントリを `/etc/barman.d/<node>.conf` に追加します。

```
[pgaa1]
description = "PGAA-enabled Postgres node"
conninfo = host=pgaa1 user=barman dbname=mydb
streaming_conninfo = host=pgaa1 user=streaming_barman
backup_method = postgres
streaming_archiver = on
slot_name = barman
create_slot = auto
```

1. バックアップをとり、カタログに記録されることを確認します。

```
barman backup pgaa1
barman list-backups pgaa1
```

13.2.2 PGD クラスターのバックアップ

PGD 環境では、クラスター内のノードごとに個別のサーバーエントリを追加します。

1. `/etc/barman.d/<node>.conf` の各 PGD ノードのサーバーエントリを追加します。

```
[pgd1]
description = "PGD node pgd1 with PGAA"
conninfo = host=pgd1 user=barman dbname=pgdtest
streaming_conninfo = host=pgd1 user=streaming_barman
backup_method = postgres
streaming_archiver = on
slot_name = barman
create_slot = auto
```

1. バックアップをとり、カタログに記録されることを確認します。

```
barman backup pgd1
barman list-backups pgd1
```

1. クラスター内のノードごとに繰り返します。

!!!注

PGAA のカタログテーブルは、双方向レプリケーション BDR の論理レプリケーションストリームの一部ではありません。各ノードは独自のコピーを保持します。実行時には、BDR のグローバル DDL ロックと、すべてのノードで起動する PGAA の `object_access_hook` を介して一貫性が維持されます。ただし、バックアップは、各ノードの物理コピーを個別にキャプチャします。単一ノードのバックアップは、カタログのそのノードのビューのみをキャプチャします。PGD クラスターの完全な復元には、すべてのノードからのバックアップが必要です。

`barman recover` を使用して、PGAA テーブルを復元します。復元手順は、展開スタンドアロンの Postgres ノード、分離して復元された単一の PGD ノード、または完全な PGD クラスターによって異なります。

13.2.3 スタンドアロンの Postgres ノードの復元

`barman recover` を使用してノードを復元します。

```
barman recover pgaa1 <backup-id> /var/lib/edb-pge/18/data
```

PGAA テーブルは、リストア後すぐに照会可能です。すべての PGAA テーブルデータはオブジェクトストレージにあるため、リストアはメタデータとポインターのみをキャプチャします。

13.2.4 単一の PGD ノードの分離による復元

クラスターに再接続せずに、単一の PGD ノードを復元します。この手順は、カタログの状態の検査、開発環境の復旧、または運用バックアップから分離環境を作成する場合に役立ちます。

```
barman recover pgd1 <backup-id> /var/lib/edb-pge/18/data
```

リストアでは、PGAA カatalogテーブルとパーティションごとのアクセス方法が保持されます。復元は、オブジェクトストレージに接続するために必要なすべてのメタデータをキャプチャするため、復元後も分析クエリへのアクセスは可能です。

復元されたノードは、元のクラスターピアを参照したままですが、それらに接続できないため、起動時に BDR コンセンサスエラーをログに記録します。これらのエラーは、クエリ提供には影響しません。Postgres は通常どおりクエリに回答し続け、PGAA はローカルデータディレクトリからカタログを読み取ります。

`pgaa.list_analytics_tables()` は BDR に依存し、分離して PGD ノードを復元する場合に使用できないため、このシナリオで分析テーブルをリストするには、カタログテーブルを直接照会します。

```
SELECT * FROM pgaa.table_mapping;
SELECT * FROM pgaa.tiered_table;
SELECT * FROM pgaa.catalog;
```

13.2.5 PGD クラスターの復元

PGD クラスターの [physical restore procedure](#) に従います。この手順では、クラスター内のすべてのノードから調整されたバックアップがあることを前提としています。詳細は、[PGD クラスターのバックアップ](#) を参照してください。すべてのノードを一緒に復元すると、PGAA カタログの状態は復元されたノード全体で一貫しており、復元後の書き込みは通常にレプリケートされます。次の PGAA 固有のポイントに注意してください。

- 手動リストアの場合、Postgres を起動する前に `max_worker_processes` を設定します。 `barman recover` は、この調整を自動的に処理します。詳細については、PGD ドキュメントの

Setting `max_worker_processes` on a manual restore <https://www.enterprisedb.com/docs/pgd/latest/lifecycle/backup-restore/#setting-max_worker_processes-on-a-manual-restore>`_ を参照してください。

- 各ノードで次のクエリを実行し、結果が同じであることを確認することにより、復元後にノード全体の PGAA カタログ状態を確認します。

```
SELECT count(*), md5(string_agg(t::text, , ORDER BY t::text))
FROM pgaa.table_mapping t;
```

- `pgaa.autostart_seafowl` がオンの場合、アクションは必要なくリカバリが完了すると、Seafowl が自動的に起動します。

13.2.6 既知の制限分析レプリケーションを有効にした後の PITR リストア

Barman の `--target-time` オプションを使用して [point-in-time recovery \(PITR\)](#) を実行する場合、`pgaa.enable_analytics_replication()` オペレーション中にターゲット時間が低下すると、展開が部分的に復旧した状態になる場合があります。この制限は、すべての PGD リストアプロシージャに適用されます。

`pgaa.enable_analytics_replication()` で分析レプリケーションを有効にすると、ヒープテーブルが HTAP テーブルに変換され、Iceberg データファイルと REST カタログエントリのオブジェクトストレージへの書き込みが開始されます。リカバリ中に、Postgres は `pgaa.table_mapping` 挿入をロールバックしますが、既にオブジェクトストレージに書き込まれている Iceberg データファイルと REST カタログエントリをロールバックすることはできません。ヒープテーブルは元の行を保持しますが、Iceberg データとカタログエントリはオーファンとして残ります。

孤立した Iceberg データとカタログエントリをクリーンアップするには

1. `pgaa.table_mapping` のリカバリ前スナップショットまたはオブジェクトストアインベントリから、孤立した Iceberg パスを特定します。
2. そのパスにある Iceberg データファイルを削除します。
3. Iceberg REST カタログから孤立したテーブルを削除します。

`pgaa.enable_analytics_replication()` フローの詳細については、[Replicating to analytics](#) を参照してください。

第14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、Agent 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。

14.1 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。

14.1.1 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。

14.1.2 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 **Seafowl** 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 Seafoal 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 Seafoal 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.enable_orderby_pushdown

CompatScan では、ORDER BY および定数 LIMIT / OFFSET を制限されたトップ N 操作として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。

デフォルトは on です。

pgaa.enable_distinct_pushdown

CompatScan では、SELECT DISTINCT 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。

デフォルトは on です。

!!!注 pgaa.enable_distinct_pushdown は WarehousePG には適用されません。そこでは、MPP プランナーが PGAA のプッシュダウンステージが実行される前に、スキャン上のモーションと HashAggregate で DISTINCT を重複排除します。

pgaa.enable_window_pushdown

CompatScan では、ウィンドウ機能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.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_sort_cost_factor

Postgres クエリプランナーが使用する乗数は、Postgres 独自のタプルソート推定と比較して、リモートバックエンド Spark または Seafowl によって作成されたプッシュダウンソートのコストを重み付けます。

デフォルト 0.5 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 Seafowl 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_seafowl_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、Seafowl 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.enable_metadata_stats

実際の **ANALYZE** 統計が欠如しているテーブルの場合、行数推定、NULL 分数、平均列幅、個別値カウント、最小/最大ヒストグラムを含むプランナー統計を Iceberg、Delta Lake、または Parquet メタデータから合成します。クエリプランナーは、CompatScan カーディナリティ推定とクエリプランにこれらの合成統計を使用します。

デフォルトは on です。

14.1.3 海鳥

!!!注 **pgaa.autostart_seafowl** パラメーターは、WarehousePG の PGAA では使用できません。WarehousePGでは、Seafowl は各ホストで **systemd** サービスとして管理されます。詳細については、[WarehousePG installation](#) セクションを参照してください。

pgaa.autostart_seafowl

バックグラウンドワーカーによって管理される Seafowl 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_seafowl_max_memory_mb

組み込み Seafowl 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_seafowl_memory_pool_type

組み込み Seafowl エンジンが使用する DataFusion メモリプールタイプを設定します。受け入れられる値は **greedy** および **fair_spill** です。 **greedy** は、最高のクエリーパフォーマンスを得るために、できるだけ早くメモリを割り当てます。 **fair_spill** は、コンカレントコンシューマ全体にメモリを公平に分散し、メモリが使い果たされると中間データをディスクに流出します。これにより、ある程度のパフォーマンスコストで高いコンカレンシーの下でのメモリ不足障害が削減されます。

デフォルトは **greedy** です。

pgaa.autostart_seafowl_track_memory_consumers

Seafowl メモリプールを、メモリ不足エラーメッセージの上位メモリコンシューマを含む追跡レイヤーでラップし、どのクエリまたはオペレーションがメモリを使い果たしたかの診断を簡単にします。

デフォルトは off です。

pgaa.autostart_seafowl_enable_metrics

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き、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ことにより、Seafowl プロセスの 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 Seafoal 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_enable_metadata_cache

Seafoal が REST カタログクライアントに Iceberg テーブルのメタデータをキャッシュするかどうかを決定します。キャッシュされたエントリは条件付き ETag 要求で再検証され、クエリプランニングのレイテンシーが削減されます。

デフォルトは on です。

pgaa.autostart_seafowl_metadata_cache_capacity_mb

Seafoal メタデータキャッシュに保持される Iceberg テーブルメタデータの最大合計サイズをメガバイト単位で設定します。

デフォルト 256 です。

pgaa.autostart_seafowl_metadata_cache_ttl_s

キャッシュされた Iceberg テーブルのメタデータエントリが更新されるまでの時間を秒単位で設定します。値 0 は有効期限を無効にし、ETag 再検証にのみ依存します。

デフォルト 0 です。

pgaa.autostart_seafowl_concurrent_manifest_ops

Seafoal プロセスでの同時アイスバークマニフェストオブジェクトストア操作の最大数を設定し、スキャン計画とメンテナンス中のフライト中の要求とピークマニフェストデコードメモリの両方を制限します。

デフォルト 64 です。

pgaa.autostart_seafowl_log_level

バックグラウンドの Seafoal 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

バックグラウンドの Seafowl プロセスがリスンする 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

クエリーの実行中にバックグラウンドの Seafowl 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ー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

Seafowl 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 Seafowl 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 Seafowl 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 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 `http://localhost:47470` です。

14.1.4 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。

14.1.5 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.metadata_stats_ttl_s

PGAA が `pgaa.enable_metadata_stats` によって合成されたプランナー統計を、Iceberg、Delta Lake、または Parquet のメタデータから再取得する前にキャッシュする時間を秒単位で制御します。これを 0 に設定すると、常に統計が再取得されます。

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 LiveTTL 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。

14.1.6 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。

14.1.7 DataFusion から継承された構成

PGAA は Seafowl 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、オブティマイザールール、クエリの Explain 出力などの低レベルの動作を制御します。公開される名前空間は execution、explain、および optimizer です。

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。

14.2 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。

14.2.1 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`、 <i>iceberg</i> 、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</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 `initial_offload`。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。

注釈

PGD 環境では、この機能は BDR に依存しており、単一のノードを分離して復元する場合は使用できません。代替方法については、[単一の PGD ノードの分離による復元](#) を参照してください。

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>REG-CLASS</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.lakehouse_table_stats_uncached()

pgaa.lakehouse_table_stats() と同じストレージ統計を返しますが、キャッシュされた結果を返すのではなく、常にオブジェクトストレージを直接照会します。現在の状態が必要で、*pgaa.lakehouse_table_stats_cache_ttl_s* の有効期限が切れるまで待てない場合に、このファンクションを使用します。

概要

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

パラメーター

Parameter	Type	Description
<i>relation</i>	<i>REG-CLASS</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.engine_version()

現在構成されている実行エンジンからバージョン情報を結果ます。pgaa.executor_engine Seafoal または Spark Connect によって選択されたバックエンドエンジンを照会し、そのバージョンの詳細を返します。Seafoal の場合、ビルドとバージョンのメタデータを結果ます。Spark Connect の場合、Spark バージョンと、利用可能な場合は Spark ドライバーからの Java、Scala、および OS 情報を結果ます。

PGAA 1.9 で削除された pgaa.seafoal_version() を置き換えます。

概要

```
SELECT pgaa.engine_version();
```

パラメーター

なし。

戻り値

アクティブな実行エンジンのバージョンの詳細を含む文字列。

14.2.2 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>	VAR-CHAR	PGAA 内のカタログの一意の名前。
<i>catalog_type</i>	<i>pgaa.catalog</i>	カタログタイプ。サポートされている値は、iceberg-rest Iceberg REST カタログ、および `iceberg-s3tables` AWS S3 テーブルです。
<i>catalog_options</i>	JSON	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。

ほとんどの REST カタログは、短期のテーブルスコープのオブジェクトストアの資格情報をクライアントに販売するため、追加の構成は必要ありません。REST カタログが資格情報を販売しない場合は、オブジェクトストア接続の詳細を使用して `store_config` オブジェクトを追加します。

```
{
  "url": "https://your-catalog-endpoint.com",
  "store_config": {
    "type": "s3",
    "bucket": "your-bucket-name",
    "access_key_id": "your-access-key",
    "secret_access_key": "your-secret-key",
    "region": "us-east-1",
    "endpoint": "https://your-object-store-endpoint.com",
    "allow_http": "true"
  }
}
```

そこで

- `store_config.type` オブジェクトストアタイプ、例、S3 互換ストアの `s3`。
- `store_config.bucket` カタログの Iceberg テーブルを保持するバケットの名前。
- `store_config.access_key_id` および `store_config.secret_access_key` バケットへのアクセスに使用される資格情報。
- `store_config.region` オプション オブジェクトストアリージョン。
- `store_config.endpoint` オプションのオブジェクトストアエンドポイント、MinIO などの S3 互換ストアに必要。

- `store_config.allow_http` エンドポイントへの非 TLS 接続を許可するには、`"true"` に設定します。

`store_config` は、PGFS と同じオブジェクトストアプロバイダーをサポートします。Azure または Google Cloud Storage の接続オプションについては、PGFS storage provider documentation を参照してください。

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の識別子である 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>	VAR-CHAR	PGAA 内のカタログの一意の名前。
<i>new_options</i>	JSON	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",
    "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、この関数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>VAR-CHAR</i>	削除するカタログの名前。
<i>cascade</i>	<i>BOOLEAN</i>	`true`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`false`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。このファンクションは、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>	VAR-CHAR	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>	VAR-CHAR	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 `NULL` 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 `NULL` 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。

注釈

WarehousePG では、`pgaa.attach_catalog()` はコーディネーターでのみメタストア同期ワーカーを実行します。

概要

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

パラメーター

Parameter	Type	Description
<i>catalog_name</i>	VAR- CHAR	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>	VAR- CHAR	切り離す登録済みカタログの名前。
<i>cascade</i>	BOOLEAN	オプション。`true` 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`false`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>	TEXT	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`detached`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>text_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>	VAR-CHAR	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>	BOOLEAN	`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>	TEXT	テーブルが存在したローカル Postgres スキーマ。
<i>table_name</i>	TEXT	削除されたローカルテーブルの名前。

14.2.3 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ー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>	TEXT	実行する Spark SQL ステートメント。

次のページに続く

表 20 – 前のページからの続き

Parameter	Type	Description
<i>catalog</i>	<i>TEXT</i> or <i>TEXT[]</i>	クエリーに使用する単一のカタログ名、またはカタログ名の配列。

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。

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>REG-CLASS</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);
```

14.2.4 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。

i 注釈

WarehousePG では、バックグラウンドメンテナンスワーカーはコーディネーターでのみ実行され、セグメントホストでは実行されません。さらに、このファンクションは内部で `gen_random_uuid()` を使用するため、コーディネーターに `pgcrypto` 拡張機能が必要です。このファンクションを呼び出す前に、コーディネーターで `CREATE EXTENSION IF NOT EXISTS pgcrypto;` を実行します。

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
<i>table_name</i>	REG- CLASS	タスクを実行する分析テーブルの名前または OID。
<i>task_type</i>	TEXT	メンテナンスオペレーション <i>compaction</i> 、 <i>zorder</i> 、 <i>vacuum</i> 、または <i>purge</i> 。
<i>task_options</i>	JSONB	タスクタイプに固有の構成以下を参照してください。
<i>scheduled_at</i>	TIMES- TAMP	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 <i>NULL</i> です。

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 Σ 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,
    {
        "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
);
```

14.2.5 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>REG-CLASS</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ジング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、[Implementing tiered tables](#) 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>	REG-CLASS	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>	TEXT	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>	TEXT	`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>	INTERVAL	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。

次のページに続く

表 24 – 前のページからの続き

Parameter	Type	Description
<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>INTER- VAL</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。
<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_partitions</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_partitions</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_period</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。詳細については、[Replicating to analytics](#) 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>REG-CLASS</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、[Replicating to analytics](#) 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>REG-CLASS</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 26 – 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。

注釈

Barman を使用して、HTAP テーブルを含む PGD クラスターをバックアップする場合、この操作中にターゲット時間を含むポイントインタイムリカバリ PITR は、展開を部分的に復旧した状態のままにする場合があります。詳細は、[既知の制限分析レプリケーションを有効にした後の PITR リストア](#) 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_namespace</i>	<i>TEXT</i>	HTAP カタログ内の名前空間/スキーマ。

[次のページに続く](#)

表 27 – 前のページからの続き

Parameter	Type	Description
<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>	ストレージで使用されるデータ形式。

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。

次のページに続く

表 29 – 前のページからの続き

Column	Type	Description
<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>REG-CLASS</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。詳細は、*Table options* を参照してください。

14.2.6 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。
<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。

14.3 Table options

14.3.1 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_namespac</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。

次のページに続く

表 33 – 前のページからの続き

Option	Type	Description
<code>pgaa.auto_truncate</code>	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。
<code>pgaa.purge_data_if_exists</code>	boolean	CREATE TABLE AS の実行時に既存の分析データをパージします。
<code>pgaa.tiered_table</code>	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 TIMESTAMPTZ,
  amount NUMERIC
)
USING PGAA
WITH (
  pgaa.format = iceberg,
  pgaa.managed_by = my_iceberg_catalog,
  pgaa.catalog_namespace = public,
  pgaa.catalog_table = managed_sales_data
);
```

i 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を列をマップします。

- 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
);
```

14.3.2 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。
<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);
```

14.3.3 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、[Offloading to analytics](#) 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;
```

14.4 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。

14.4.1 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>	

[次のページに続く](#)

表 35 – 前のページからの続き

Natively Supported	Coerced
<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>JSONB OID</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>	

14.4.2 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>re-fresh_retry</i> , and <i>re-fresh_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。
<i>pgaa.task_status</i>	<i>pending</i> , <i>running</i> , <i>success</i> , and <i>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。

14.5 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
<i>tpch_sf_1000</i>	TPC-H, Scale Factor 1000
<i>clickbench</i>	ClickBench, 100 million rows
<i>brc_1b</i>	Billion row challenge

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。