D-017

データ仮想化エンジン PGSpider の二種類の実現方式と性能の比較
Performance comparison between two implementation types of data virtualization engines PGSpider

片山 大河† 熊谷 宏樹†
Taiga Katayama Hiroki Kumagai

1. はじめに

CPS（Cyber Physical System）の分野では点在するデータを一箇所に集めずにデータソースからオンデマンドに取得するデータ仮想化技術が注目されている。それらのデータはそれぞれ独立したサービス上で運用されているため、データを管理する異なるシステム（例えばデータベース管理システムやファイルシステム）を跨った検索が必要となることがある。

それを実現する PGSpider は複数のデータソースを並列に横断アクセスする機能を持っていて、実現方式の違いから二種類がある。ここでは通常版と Extension 版と呼ぶこととする。両者には機能差があり、共通の主機能であるマルチテナント機能についてはアーキテクチャの違いから性能に影響があることが想像できた。そこで本稿では、それらの実現方式の違いと機能差を説明し、マルチテナント機能関連の性能差について調査した結果を報告する。

2. 二種類の PGSpider

二種類の PGSpider はいずれもオープンソースとして公開していて、PostgreSQL[1]をベースとしている。まず共通の概念である FDW（Foreign Data Wrapper）とマルチテナント機能について説明してから、二種類の PGSpider の実現方式を説明する。

2.1 FDW による異種性の吸収

PostgreSQL には FDW という拡張機能の仕組みがあり、PostgreSQL から他のデータソースにアクセスできるようになっている。この機能を利用すると、データソースを PostgreSQL 上に表現した外部サーバというオブジェクトを定義でき、さらにデータソース上のテーブルを PostgreSQL 上に表現した外部テーブルというオブジェクトを定義できる。ユーザがこの外部テーブルに対するクエリを実行すると、各データソース向けの FDW モジュール（データソース FDW）によってデータソース向けの問い合わせに変換され、データソース上の実テーブルがアクセスされる。このように異種性は FDW によって吸収される。

データソース FDW は、PostgreSQL 本体から呼ばれるコールバック関数（FDW API）の仕様[2]に基づいて各データソース向けに実装された拡張モジュールである。様々なデータソース向けの FDW が公開されている[4]。

2.2 マルチテナント機能とは

この機能は、複数の外部テーブルをひとつのクエリで高速にアクセスするために発案された機能である。複数の外部テーブル間で同じスキーマを持っているとき、それらを仮想的なひとつのテーブルとして定義可能にする。この仮想テーブルのことをマルチテナントテーブル（以降 MT テーブルと表記）と呼ぶ。ユーザが MT テーブルにクエリすると、PGSpider は個々の外部テーブルへのアクセスにクエリを展開し、それらを並列にアクセスしてデータを取得して、結果をマージする（図 1）。このようにして PGSpider はマルチテナント機能により横断検索を実現する。二種類の PGSpider はマルチテナント機能を実現する仕組みが異なる。

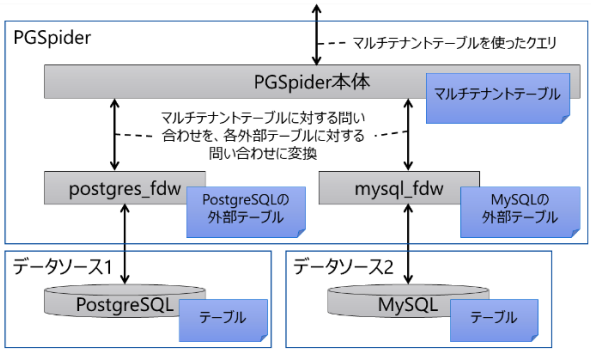

図 1 マルチテナントテーブルと FDW の関係

2.3 通常版

通常版[5]は PostgreSQL のソースコードを改造して実現している。アーキテクチャを図 2 に示す。pgspider_core_fdw というモジュールが MT テーブルを管理していて、2.2 節で説明したクエリ展開や結果マージ処理はこのモジュールが担う。MT テーブルから外部テーブルへの展開は、テーブル名のルールに基づいて決められる。具体的には、外部テーブル名は" MT テーブル名__外部サーバ名__数値"というルールとなっている。個々のデータソースへのアクセスはデータソース FDW が行う。

pgspider_core_fdw 自体も PostgreSQL の FDW 機能を利用して実現している。PostgreSQL 本体から見ると MT テーブルも PostgreSQL 外に存在するテーブルのひとつで、MT テーブルへに対するクエリは FDW モジュール（pgspider_core_fdw）を介して結果を取得する。

並列処理についてはマルチスレッドのモデルを採用している。pgspider_core_fdw は、MT テーブルを構成する外部テーブルごとに子スレッドを作成して処理を担当させる。グローバル変数へのアクセス競合を回避するなど、スレッド化を実現するために PostgreSQL 本体のソースコードに手を加えている。

2.4 Extension 版

Extension 版[6]は、PostgreSQL のソースコードを改造せずに、その名の通り PostgreSQL に備わる Extension 機能の上でマルチテナント機能を実現している。このモジュールを

† 株式会社東芝 TOSHIBA CORPORATION

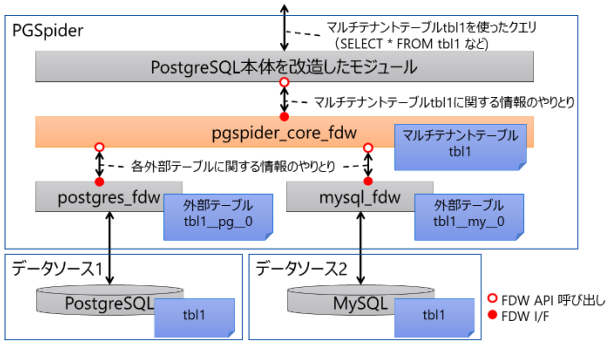

図 2 通常版の PGSpider のアーキテクチャ

pgspider_ext と呼ぶ。通常版に比べて、機能実装の局所化によるメンテナンス性向上や PostgreSQL 本体のバージョンアップへの追従性の向上を目的として本バージョンを考案した。

アーキテクチャを図 3 に示す。テーブル仮想化の実現には、宣言的パーティショニング機能を利用する。パーティションの各子テーブルは pgspider_ext 上のテーブルとし、データソース FDW 上の外部テーブルがその子テーブルと一対一に対応付けられる。子テーブルが外部テーブルを中継する仕組みとした理由は、パーティションを構成するためにはパーティションキーとなるカラムが必要で、外部テーブルだけではそれに該当するカラムが常に存在するとは限らないからである。そこで、pgspider_ext 上のテーブルは外部テーブルのカラムに加えて外部テーブルを識別する特殊カラム（ノード名列）で構成し、このノード列をリストパーティションのキーカラムとするようにしている。

3. 機能的な差異

二種類の PGSpider には機能差がある。主に 2 章で述べた実現方式の違いに起因した代表的な機能差を 3 つ紹介する。開発段階の違いにより、実現は可能だが実装していないだけという機能差もあるがここでは触れない。

3.1 プッシュダウン機能

プッシュダウンとは、クエリの処理の一部をデータソース上で実行させることである。例えば WHERE 条件式のプッシュダウンは条件式の評価をデータソースが行い、PGSpider が条件式に合うデータのみを取得する。プッシュダウン対象は様々あり、その中で通常版の PGSpider ではプッシュダウン可能で、Extension 版では不可能なものがある（通常版の方が優れている）。次の 4 つである。

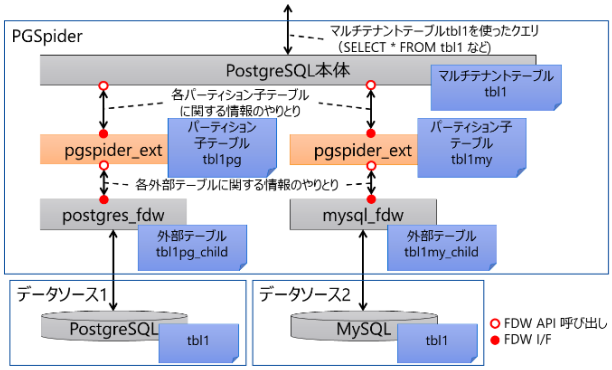

図 3 Extension 版のアーキテクチャ

3.1.1 特殊な集約関数プッシュダウン

MT テーブルは複数の子テーブルで構成されていて、集約関数をそのままデータソースにプッシュダウンすると正しい計算結果が得られないため、特殊な対応が必要になるものがある。例えば平均値を計算する avg の場合、sum と count をプッシュダウンし、それらの結果を受け取った PGSpider が$(\sum_i^n sum_i)/(\sum_i^n count_i)$を計算して avg を求める（$n$ はデータソース数、sum_iと$count_i$は各データソースが返した結果）。

通常版では、MT テーブルに関する集約関数情報が pgspider_core_fdw に渡ってくるため、上記のような特殊対応が実現可能である。

一方、Extension 版では、PostgreSQL 本体が MT テーブル（＝パーティション親テーブル）に対するクエリを各子テーブルに展開する処理を行い、各パーティション子テーブルに関するクエリ情報を作成した後、その情報が pgspider_ext に渡ってくる。このとき、PostgreSQL 本体はこのような特殊な集約関数をプッシュダウンできないと判断する。つまり、元々の集約関数の情報が pgspider_ext まで渡ってこないため、pgspider_ext 上だけではこのような集約関数をプッシュダウンすることができない。なお、特殊な操作を必要としない集約関数（例えば count 関数）は、データソース向けの集約関数情報が pgspider_ext に渡されるためプッシュダウン可能である。

3.1.2 SQL 関数プッシュダウン

SELECT の射影で SQL 関数（集約ではない関数）を使った場合、SQL 関数の情報は FDW レイヤーに渡らない。SQL 関数の引数にあたるカラムの情報が FDW レイヤーに渡され、PostgreSQL 本体で SQL 関数を計算する。

従来版は PostgreSQL 本体のコードを改修し、SQL 関数の情報を pgspider_core_fdw に渡るようにしているため、プッシュダウン可能になっている。一方で Extension 版では PostgreSQL 本体を改造しないポリシーのため、本機能は実現できない。

3.1.3 LIMIT・OFFSET プッシュダウン

通常版では子テーブルがひとつだけの場合に限り LIMIT・OFFSET をプッシュダウンする。集約関数のプッシュダウンの状況と同じく、LIMIT・OFFSET の情報が pgspider_ext のレイヤーまで渡ってこないため、Extension 版では本機能は未サポートとなる。

3.1.4 ジョインプッシュダウン

ジョインのプッシュダウンも LIMIT・OFFSET のプッシュダウンと同様の状況である。通常版ではジョイン処理がひとつのデータソース内で完結する場合（つまりふたつのジョイン対象のテーブルの子テーブルがひとつの子テーブルしか持たず、同一のデータソース上にある場合）にのみジョインをプッシュダウンする。Extension 版ではジョイン情報が pgspider_ext のレイヤーまで渡らないため本機能は未サポートとなる。

3.2 ノード名列の値の決め方

この機能差は仕様の違いである。2.4 節で述べたように、Extension 版ではノード名列がパーティションのキーカラムになっていて、この値はユーザが任意に設定することが可能となっている。一方で通常版は pgspider_core_fdw が自動的に設定する仕様となっていて、データソースを識別する

値として外部サーバ名を基に文字列を生成するようになっている。

3.3 設定パラメータの影響

Extension 版がパーティションを前提としたアーキテクチャであることに起因して、MT テーブルへのアクセスの観点で設定パラメータの影響が通常版と異なる項目がある。影響がある設定項目は以下の 3 つである。

1. enable_partitionwise_aggregate
   パーティションの子テーブル単位で部分的に集約関数の計算を実施するかどうかを設定する項目である。従来版は本設定項目が影響しないため、可能な限り集約関数をプッシュダウンしようとする。Extension 版では本設定が有効で、集約関数をプッシュダウンしないように設定することも可能である。
2. constraint_exclusion
   テーブルに制約が設定されている場合、その情報をもとにクエリを最適化され、不要なテーブル走査をスキップする機能がある。
   この設定項目はデフォルト値が partition なため、ふたつの PGSpider でデフォルト設定時の挙動が異なる。通常版では設定を変更しない限り MT テーブルへのアクセスは本最適化が行われない。一方で、Extension 版では設定を変更しなくとも本最適化が行われる。
3. parallel_leader_participation
   クエリ処理を複数の子プロセス（ワーカープロセスと呼ばれる）によって並列処理する仕組みがある。ワーカープロセスを取りまとめているリーダープロセスが、ワーカープロセスとしても動作させるかの設定が行える（[3] parallel_leader_participation）。
   通常版では 2.3 節で述べたとおり、並列処理は PostgreSQL 本体の仕組みとは独立した仕組みで実現しているため、本設定は影響しない。Extension 版の場合、この設定を有効にしたところ正しい結果が得られなかったため、無効化する必要があることがわかった。明確な情報を持っていないが、pgspider_ext を使わない場合（データソース FDW とパーティション機能を組み合わせて使用した場合）でも正しい結果が得られなかったため、PostgreSQL の不具合の可能性があると考えている。

4. 性能的な差異

プッシュダウンの可否が性能に影響を与える要因の一つとなる。また、プッシュダウンの可否が同一であっても実現方式の違いにより性能に差が出る可能性がある。そこで、マルチテナント機能の性能差を確認するために実験を行った。MT テーブルを使った問い合わせ方法は両 PGSpider で同じため、同一の SQL 文を実行している（ただし MT テーブルの作成手順は異なる）。実行環境はひとつの仮想マシン上に構築し、CPU が Intel Xenon Gold 5118 (2.30 GHz)、12GB RAM、128 HDD、OS は CentOS 7 を使用した。

4.1 実験の構成とデータ

まず実験のソフトウェア構成について説明する。データソースは PostgreSQL13.5 とし、データソース数による性能への影響を確認したかったため、データソース数が 1、2、4 および 8 の 4 種類（ジョインクエリの測定の場合は 1 および 2 の 2 種類）のモデルを用意した（Model1～4）。それらを仮想化する PGSpider があり、postgres_fdw によって接続している。従来版のソースコードは[7]（v2.0.0 相当）を、Extension 版は[8]（v1.0.0 相当）を使用した。

データ量はいずれのモデルも全体で 100 万件を格納し、1 ノードあたりのデータ量がデータソース数に反比例するようにした。例えば、8 データソースの場合、1 データソースあたり 12,5000 件となる（図 4）。

テーブルはふたつのカラム（integer 型のカラム c1 と長さ 20 の text 型のカラム c2）を持つものを用意した。それらで MT テーブル t1 を構成する。各レコードは c1 に一意の値を持つ。その値は、WHERE 条件で範囲指定したときに、各データソースでヒットする件数が均一になるようにしている。

ジョインクエリの測定の場合はもうひとつ別のテーブル t2 を用意した。スキーマは t1 と同一で、全体で 1000 件のデータを持つ。インナージョインの結果が 1000 件となるように t2 のデータを作成した。

4.2 測定クエリ

以下に挙げる様々なパターンの SQL を実行させた。

1. SELECT * FROM t1;
   全件を無条件に取得するクエリで、100 万件の大量データを取得するケースである。
2. SELECT * FROM t1 WHERE c1 < *n*;
   WHERE 条件で取得件数を絞り込み、少量のデータを取得するケースである。*n* の値を変えて取得データ量が 8 件、8000 件あるいは 8000 件となる 3 種類の SQL を測定した。
3. SELECT *agg*(c1) FROM t1;
   agg は集約関数（count あるいは stddev）で、両 PGSpider ともプッシュダウンできるケースと一方しかプッシュダウンできないケースである。通常版はどちらの関数もプッシュダウン可能だが、Extension 版では stddev をプッシュダウンできない。
4. SELECT t1.c1, t1.c2, t2.c1, t2.c2 FROM t1 JOIN t2 ON t1.c1 = t2.c1;
   通常版ではデータソースがひとつのモデルの場合のみジョインをプッシュダウンでき、その他の場合はプッシュダウンできないケースである。

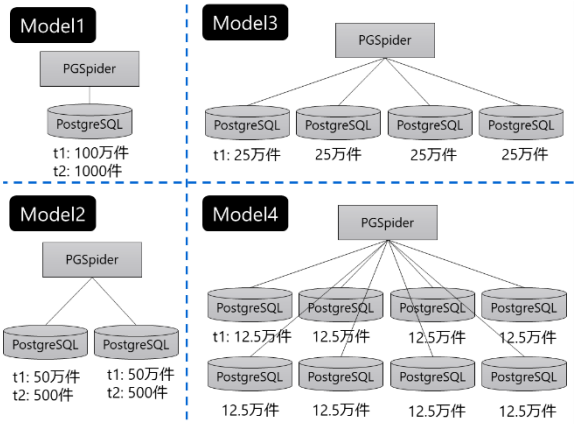

図 4 データソース構成

4.3 測定結果

4.3.1 大量データ取得

このケースではデータソース数が少ないほど通常版が高速で、データソース数が 8 の場合で同等という結果が得られた（表 1）。通常版が高速な要因として考えられることは、PGSpider 上での処理の並列度が挙げられる。

表 1 大量データ取得ケースの測定結果（ミリ秒）

モデル	通常版	Extension 版
Model1	2,945	4,920
Model2	1,798	2,776
Model3	1,555	2,853
Model4	1,611	1,554

並列処理はふたつの視点があり、ひとつは複数のデータソースへのアクセスに対する並列性、もうひとつは結果の取り込み処理と取り込んだ結果に対して行う処理の同時実行性である。まず前者に関しては、通常版ではデータソースの数だけ子スレッドを作成するようになっている。一方で Extension 版は PostgreSQL のパラレルクエリの仕組みに基づいていて、今回の場合、Model1～4（データソース数が 1、2、4、8）に対するデータソースにアクセスするプロセス数は 1、2、3、4 であった。

また後者の並列性については、通常版では子スレッドがデータソースから受信した各レコードは、キューを介してメインスレッドが受け取るためストリーム的に動作できるようになっている。これは単一データソースの場合でも効果を発揮する。一方で、Extension 版では結果の取り込み処理の完了を待って、取り込んだ結果に対して行う処理を開始するため逐次処理になるようである（[3] parallel_leader_participation）。以上のことから、通常版の方が並列性が高いため高速になったと考えている。

4.3.2 少量データ取得

このケースでは単一データソースの場合は同等で、データソース数が増えると性能差が現れ、通常版の方が高速という結果が得られた（表 2）。クエリの実行時間が 100 ミリ程度と短いため、Extension 版は子プロセスの起動コストが大きく性能低下に影響したと考えている。

表 2 少量データ取得ケースの測定結果（ミリ秒）

取得件数	モデル	通常版	Extension 版
8	Model1	99	89
8	Model2	43	80
8	Model3	29	92
8	Model4	34	94
8,000	Model1	131	135
8,000	Model2	59	109
8,000	Model3	36	117
8,000	Model4	34	108
80,000	Model1	131	135
80,000	Model2	59	109
80,000	Model3	36	117
80,000	Model4	34	108

4.3.3 集約関数

結果を表 3 に示す。count 関数は両 PGSpider がプッシュダウン可能な集約関数で、傾向としては少量のデータ取得のケースと同じくデータソース数が増えると通常版が高速になった。一方、通常版ではプッシュダウンできるが Extension 版ではプッシュダウンできない stddev 関数を実行したときは、想像できるように、プッシュダウン可能な通常版が 15 倍以上高速だった。

表 3 集約関数クエリの測定結果（ミリ秒）

集約関数	モデル	通常版	Extension 版
count	Model1	114	104
count	Model2	54	85
count	Model3	37	98
count	Model4	32	98
stddev	Model1	278	4,251
stddev	Model2	70	2,194
stddev	Model3	45	2,022
stddev	Model4	37	1,171

4.3.4 ジョイン

ジョインクエリの結果を表 4 に示す。単一データソースの場合は通常版ではプッシュダウン可能なため、Extension 版と比べて約 32 倍と圧倒的に高速だった。複数データソースの場合でも通常版の方が約 1.3 倍高速だった。通常版が高速な要因は、4.3.1 節で説明したストリーム的な処理が影響していると考えている。

表 4 ジョインクエリの測定結果（ミリ秒）

モデル	通常版	Extension 版
Model1	143	4,565
Model2	3,518	4,547

5. おわりに

二種類の PGSpider について実現方式の違い、機能差および性能差について検証し報告した。通常版はマルチスレッドモデルや多くのプッシュダウン機能に対応しているため性能面では Extension 版より有利だが、Extension 版は PostgreSQL 本体のソースコードを変更することなく実現しているためメンテナンス性が高い。

性能は全体的に Extension 版より通常版の方が 1.5～3.5 倍程度高速で、プッシュダウンの機能差が影響するシナリオでは特に性能差が大きく、15～45 倍高速だった。

性能やメンテンス性など重視されるポイントが適用システムによって異なるため、本検証で PGSpider 選定のための有益な情報が得られたと考えている。

参考文献

[1] PostgreSQL, https://www.postgresql.org/
[2] PostgreSQL 文章 第 56 章 外部データラッパの作成, https://www.postgresql.jp/document/13/html/fdwhandler.html
[3] PostgreSQL 文書 19.7. 問い合わせ計画, https://www.postgresql.jp/document/13/html/runtime-config-query.html
[4] Foreign data wrappers, https://wiki.postgresql.org/wiki/Foreign_data_wrappers
[5] Taiga Katayama, PGSPIDER - HIGH-PERFORMANCE SQL CLUSTER ENGINE., FOSDEM PGDAY 2020
[6] 片山大河, 廣瀬繁雄, 金松基孝, PostgreSQL によるデータ仮想化エンジンを実現する拡張機能 PGSpider, FIT2021
[7] 通常版 PGSpider, https://github.com/pgspider/pgspider
[8] Extension 版 PGSpider, https://github.com/pgspider/pgspider_ext