DEIM2021 B33-5

# PostgreSQL におけるビューの増分メンテナンス機能の実装とその評価

長田　悠吾† 　星合　拓馬†† 　石井　達夫† 　三島　健† 　増永　良文†††

† SRA OSS, Inc. 日本支社　〒 171–0022 東京都豊島区南池袋 2–32–8
†† NTT コムウェア株式会社　〒 108–8019 東京都港区港南 1–9–1
††† お茶の水女子大学（名誉教授）　〒 112–8610 東京都文京区大塚 2–1–1
E-mail: †{nagata,ishii,mishima}@sraoss.co.jp, ††hoshiai.takuma@nttcom.co.jp, †††yoshi.masunaga@gmail.com

**あらまし**　マテリアライズドビューはその実体をデータベースに格納することで高速な応答を可能にする一方で，実テーブルに更新があった際にはビューの内容を最新の状態に更新する必要がある．これを一からの再計算によって行うのはオーバヘッドが大きくレスポンスが悪化するため，実テーブルへの変更差分のみをビューに適用する増分メンテナンス（Incremental View Maintenance, IVM）と呼ばれる方法が提案されている．一方で PostgreSQL は広く使われているオープンソースの RDBMS だが，残念なことに，現状の PostgreSQL では未だに IVM はサポートされていない．そこで，我々は PostgreSQL 上に IVM を実装して，開発コミュニティに提案している．TPC-H ベンチマークを使った評価では，ビュー定義で使用される実テーブルの更新性能が低下する影響もあったが，一から再計算するよりも短時間でビューの更新が可能となることが確認できた．本論文では本実装の方法と，性能評価，および今後の課題について報告する．

**キーワード**　マテリアライズドビュー, ビューメンテナンス, リレーショナルデータベース, PostgreSQL，オープンソース開発

## 1 はじめに

リレーショナルデータベースにおけるビュー（view）は，それを定義するクエリを実体としており，ビューに対する問合せが発生したときにこのクエリを実行し，その結果をクライアントに返す．一方，マテリアライズドビュー（materialized view）では問合せの結果がデータベースに格納されており，これにより通常のビューよりも高速な応答を可能にしている．この機能は例えばデータウェアハウスに格納された大量のデータの解析結果を短いレスポンスタイムで取得したい場合などに有用である．一方で，ビュー定義に含まれる実テーブルが更新された場合にはマテリアライズドビューに格納されているデータは古いものとなり実テーブルの内容との一貫性が失われる．そのため，実テーブルが更新された後にはマテリアライズドビューの内容を最新の状態に更新するメンテナンスが必要となる．ただし，これを更新後の実テーブルから完全な再計算によって行うのは時間がかかりコストが高い．そのため，実テーブルに発生した変化分に応じてデータベースに格納されているマテリアライズドビューの一部分だけを更新する増分ビューメンテナンス（incremental view maintenance, IVM）という手法が研究されてきた [2], [3], [1].

一方で，広く使われているオープンソースの RDBMS である PostgreSQL [9] はマテリアライズドビューを作成する機能はサポートしているが，そのメンテナンス法として用意されているのはビュー定義クエリを実行し直すことでビューを更新する方法のみである．PostgreSQL への増分メンテナンスの実装を目的とした報告は過去にも存在する [6] [7] が，現在の PostgreSQL にこの機能はまだ実装されていない．そこで我々は PostgreSQL への増分メンテナンスの機能を実装し，これを PostgreSQL の機能として加えることを開発コミュニティに提案中である．本論文ではその実装法，性能評価，および今後の課題について報告する．

## 2 増分ビューメンテナンス

本節では増分ビューメンテナンスの概要を述べる．

### 2.1 増分メンテナンスと再計算

増分ビューメンテナンスと再計算との違いを図 1 に示す [1]. ここで，マテリアライズドビューを定義する問合せを $Q_v$，ある時点におけるビューの内容を $T_v$，その時点でのデータベースの状態を $D$ とする．このとき $T_v = Q_v(D)$ が成り立つ．添え字の v はビューを意味する．データベース $D$ に更新 $\delta D$ が発生して新しい状態が $D'$ となったとき，更新後のマテリアライズドビューの状態 $Q_v(D')$ を $D'$ と $Q_v$ を用いて求めるのが再計算（recomputation）によるビューメンテンナンスである．これに対し，$\delta D$ を利用して $T_v$ に対する変化分 $\delta V$ を求め，これを $T_v$ に適用することで新しいビューの状態 $T'_v$ を求める方法が増分メンテナンスである．

### 2.2 バッグ代数の記法

ビューを定義する問合せ $Q_v$ はバッグ代数（bag algebra）上の演算の組み合わせとして記述することができる [2]．バッグ代数とはタプルの重複が存在しない集合意味論の下で定義される関係代数（relational algebra）をタプルの重複を許すバッグ意味論の下で拡張したものであるが，本論文では関係代数の演算子

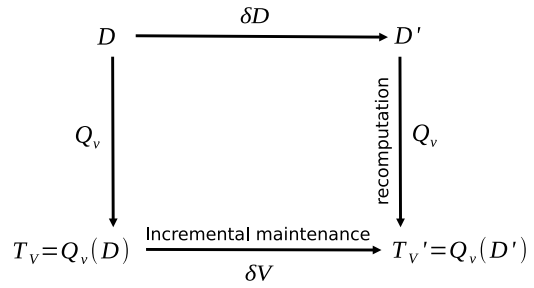


図 1　増分ビューメンテナンスと再計算によるメンテナンス

と同じ記法を用いる．すなわち，選択，射影，直積，内部結合はそれぞれ $\sigma_p$，$\pi_C$，$\times$，$\bowtie_q$ を用いて表す．ここで $p$ は選択の条件を表す述語，$C$ は射影で取り出される属性のリスト，$q$ はテーブルを結合する条件を表す述語を意味する．これらの演算を用いてテーブル $R$ と $S$ の結合 $R \bowtie_p S$ は $\sigma_p(R \times S)$ と表すこともできる．また，ここで射影演算はタプルの重複を除去しないものとする．タプルの重複を除去する演算は $\delta$ で表す．

$\uplus$ は加法和（additive union）を表し，タプルの重複を除去せずに 2 つのテーブルの和集合を求める演算である．たとえば 2 つのテーブル $R = \{a, a, b, b\}$，$S = \{b, c\}$ があるとき $R \uplus S = \{a, a, b, b, b, c\}$ となる．$\dot{-}$ は 2 つのテーブルのモーナス（monus）を表し，タプルの重複を除去せずに 2 つのテーブルの差集合を求める演算である．上記の例では，$R \dot{-} S = \{a, a, b\}$ となる．

### 2.3　SPJ ビューの増分メンテナンス

ここでテーブル $R$，$S$ 上の SPJ ビュー $V_1 = \pi_C \sigma_{p1}(R \bowtie_{p2} S)$ が定義されているときに，あるトランザクションにおいてテーブル $R$ が更新されたとする．このとき $R$ の変化は $R \leftarrow R \dot{-} \nabla R \uplus \Delta R$ と表すことができる．ここで $\nabla R$ は更新の結果として $R$ から削除されたタプル群，$\Delta R$ は $R$ に挿入されたタプル群を表すバッグである．増分ビューメンテナンスはこのときに $V_1$ に発生する差分，すなわち $V_1$ から削除すべきタプル群 $\nabla V_1$ と $V_1$ に挿入すべきタプル群 $\Delta V_1$ を求め，$V_1 \leftarrow V_1 \dot{-} \nabla V_1 \uplus \Delta V_1$ のように適用することで実現できる．これらの差分はビュー定義の $R$ を $\nabla R$，$\Delta R$ に置き換えることで求めることが出来る．この例では $\nabla V_1 = \pi_C \sigma_{p1}(\nabla R \bowtie_{p2} S)$，$\Delta V_1 = \pi_C \sigma_{p1}(\Delta R \bowtie_{p2} S)$ となる．

上記の例は 1 つの実テーブルが更新された場合である．次に複数の実テーブルが同時に更新された場合について述べる．3 つのテーブル $R$,$S$,$T$ 上の直積ビュー $V_2 = R \times S \times T$ を考える．ここで全ての実テーブルにタプルが挿入されたとする．各テーブルの更新前の状態を $R_{pre}$，$S_{pre}$，$T_{pre}$，更新後の状態を $R_{post} = (R_{pre} \uplus \Delta R)$，$S_{post} = (S_{pre} \uplus \Delta S)$，$T_{post} = (T_{pre} \uplus \Delta T)$ とする．このとき，ビューに発生する差分は

$$\Delta V_2 = (\Delta R \times S_{pre} \times T_{pre}) \uplus (R_{post} \times \Delta S \times T_{pre}) \uplus (R_{post} \times S_{post} \times \Delta T)$$

となる．すなわちテーブルに発生した差分と更新前，更新後の両テーブル状態を用いて実テーブルの数だけの直積演算を行う．タプル削除の場合も同様である．また，同じテーブルがビュー定義の中に複数回現れる自己結合（self join）の場合は，定義上は同一のテーブルを「内容が同じ異なるテーブル」とみなすことにより同じ方法で増分メンテナンスが可能である．

### 2.4　タプル重複の扱い

ビュー，実テーブルおよびこれらの差分はバッグであるためタプルの重複が許される．バッグ内の同じタプルの数はそのタプルの重複度（multiplicity）と呼ばれる．増分メンテナンスにおいてビューに差分を適用する際には加法和 $\uplus$ およびモーナス $\dot{-}$ の演算を用いるが，これらの演算はタプルの重複度同士の加減演算で定義することができる．例えば，バッグ $A$ と $B$ に共通するタプル $t$ があり，その重複度をそれぞれ $n$，$m$ とする．このとき，加法和 $A \uplus B$ の結果に含まれるタプルの重複度は，$A$ と $B$ に共通するタプル $t$ については $n + m$ となり，片方にのみ存在するタプルについては元の重複度がそのまま使われる．また，モーナス $A \dot{-} B$ に関しては，$A$ と $B$ に共通するタプル $t$ の重複度は $max(n - m, 0)$ となる．このとき重複度が 0 となったタプルは結果に含めない．$A$ にのみ存在するタプルについては元の重複度がそのまま使われる．$B$ のみに含まれるタプルについても演算結果に含まれない．

バッグに重複除去演算 $\delta$ を施すと，全てのタプルの重複度は全て 1 となる．一方，重複除去を含むビューの増分メンテナンスを行う際には重複除去を行う前のタプル重複度を保持しておくことが有用である．例えば，テーブル $R = \{a, a, b, c, c\}$ の上にビュー $V_3 = \delta(R)$ が定義されるとき，ビューの内容は $\{a, b, c\}$ となる．このとき $R$ から $\Delta R = \{a, b\}$ が削除されたとする．もしここで $\Delta R$ を直接 $V_3$ から削除すれば $V_3 = \{c\}$ となり正しい状態 $\{a, c\}$ が得られない．しかし，実テーブル $R$ のタプルの重複度が保持されていれば，$\Delta R$ を $V_3$ に適用すると $b$ の重複度が 0 となるため $V_3$ から削除され，$a$，$c$ の重複度は 1 以上であるため $V_3$ に残るということがわかる．

以上のようにタプルの重複度をビュー内に保持することで，タプルの重複および重複除去演算を含むビューの増分メンテナンスを実現する手法は counting アルゴリズムと呼ばれており [4]，本論文における実装ではこの手法を用いている．

### 2.5　集　　約

集約を含むビューの増分メンテナンスはテーブルに発生した差分上の集約値を計算することで実現できる．本実装で対応している集約関数は count, sum, avg, min, max である．

ある集約関数 $f(x)$ を用いたビュー $V_4 = \pi_{G,f(x)} R$ がバッグ $R$ 上に定義されているとする．ここで $x$ は $R$ 上の集約対象の属性で，$G$ はグループ化（group-by）属性である．$R$ に差分 $\nabla R$ および $\Delta R$ が発生したとき，各グループ $G$ に関する差分上の集約値を $\nabla f_G = \pi_{G,f(x)} \nabla R$，$\Delta f_G = \pi_{G,f(x)} \Delta R$ とする．$f$ が count または sum の場合，ビュー内のグループ $G$ の集約値は $f(x) \leftarrow f(x) - \nabla f_G + \Delta f_G$ と計算できる．また $f$ が avg の場合には count および sum の値を別途ビュー内に保持しておくことで $avg(x) \leftarrow sum(x)/count(x)$ と計算できる．f が min または max の場合，タプルが挿入されたときには $min(x) \leftarrow Min(min(x), \nabla min_G)$，$max(x) \leftarrow Max(max(x), \nabla max_G)$

と更新できる．一方でタプルが削除された場合，$min(x) < \Delta min_G$，$max(x) > \Delta max_G$ ならばグループ $G$ のタプルは更新する必要が無いが，それ以外の場合はグループ $G$ の $min(x)$，$max(x)$ を実テーブルから再計算する必要がある．

### 2.6 ビューメンテナンスのタイミング

ビューの増分メンテナンスを実行するタイミングには大きく 2 つのアプローチがある．即時メンテナンス（immediate maintenance）では実テーブルを更新した同じトランザクションの中でビューが最新の状態に更新される．例えば，テーブルを更新するクエリが発行された直後や，そのトランザクションがコミットされたタイミングでビューを最新化するといった方法がこれに該当する．即時メンテナンスではテーブルが更新する度にビューが最新化されるため一貫性が保たれるが，その分テーブルの更新性能は低くなる．一方，遅延メンテナンス（deferred maintenance）では実テーブルを更新したトランザクションがコミットされた後にビューの最新化が行われる．例えば，ユーザがコマンドを実行したタイミングや，定期的かつ自動的にビューを最新化するといった方法がこれに該当する．

## 3 PostgreSQL への実装

本節では PostgreSQL への増分ビューメンテナンスの実装について述べる．

現状の PostgreSQL は増分ビューメンテナンスをサポートしていないため，我々は本実装による増分ビューメンテナンスを PostgreSQL の機能として取り入れるよう開発コミュニティに提案中である．なお，開発中のコードは GitHub にて公開されている [14]．

### 3.1 機能の概要

本実装では，即時メンテナンスアプローチによる増分ビューメンテナンスを提供する．実テーブルが更新された直後に，マテリアライズドビューは即時にかつ自動的に更新される．ビューは増分メンテナンスで更新されるため，再計算よりも高速なビューの更新を可能とする．遅延メンテナンスを実現するためにはトランザクションで発生したテーブル差分の履歴を管理する機構を実装する必要があるが，即時メンテナンスではこれが不要であり，遅延メンテナンスよりも少ない実装量で実現可能であることが，即時メンテナンスでの実装を採用した理由である．

### 3.2 ビューの作成

増分メンテナンス可能なマテリアライズドビュー（Incrementally maintainable materialized view: IMMV）の作成は以下の SQL コマンドで行う．

```
CREATE INCREMENTAL MATERIALIZED VIEW <view name>
  AS <query>;
```

CREATE MATERIALIZED VIEW はマテリアライズドビューを作成する既存のコマンドである．<view name>にはビューの名前，<query>にはビューを定義する SELECT クエリを指定する．INCREMENTAL というキーワードが本実装でこのコマンドに加えられた拡張であり，これを付与して作成されたマテリアライズドビューは増分メンテナンスの対象となる．

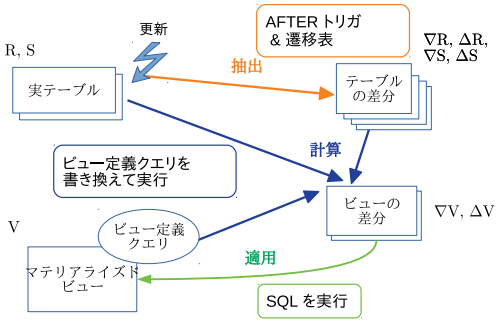

図 2　増分ビューメンテナンスの流れ

ビューの定義に使用可能なクエリとして，選択と射影に相当する通常の SELECT ... FROM ... WHERE ... の他，結合（内部結合，外部結合，自己結合），一部の組み込み集約関数（count, sum, avg, min, max），GROUP BY 句，DISTINCT 句，FROM 句内の単純なサブクエリ，単純な CTE（WITH 句），EXISTS サブクエリに対応している．ここで単純とは，集約，外部結合，DISTINCT 句をクエリに含まないことを意味している．

また，IMMV を作成する際にはユーザが指定した以外のカラムが隠しカラムとして追加される場合がある．例えば，集約や DISTINCT 句が含まれるビューの場合には，count(*) 集約関数で計算されたタプルの重複度が__ivm_count__という名前で作成される．__ivm_で始まるカラム名は IMMV において隠しカラムとして扱われ，SELECT 文のターゲットリストに明示的に指定されない限りクエリ結果には現れない．集約や EXISTS 句が使用されている場合にはこれとは別の隠しカラムも追加される．

### 3.3 増分ビューメンテナンスの流れ

本実装におけるビューの増分メンテナンスは以下の 3 つのステップで行われる（図 2）．

1　テーブルに発生した差分の抽出
2　ビューに発生する差分の計算
3　ビュー差分をビューに適用

以下では各ステップの詳細について述べる．

#### 3.3.1 テーブル差分の抽出

テーブルに発生した差分の抽出には，文レベルの AFTER トリガと遷移表（Transition Table）[12] を用いる．IMMV が定義された際に，ビュー定義に含まれる全ての実テーブルに対して AFTER トリガが作成される．AFTER トリガは実テーブルに対し INSERT, DELETE, UPDATE コマンドが実行された直後に実行され，ビューの更新はトリガ関数の中で行われる．AFTER トリガはテーブルを更新したトランザクションと同じトランザクションの中で実行されるため，本実装は即時メンテナンスを行うものとなる．

遷移表はテーブルに発生した変化を捕捉する AFTER トリガ

の機能である．遷移表は AFTER トリガ関数で通常のテーブルと同じように参照することができる.「テーブルから削除されたタプル」と「テーブルに挿入されたタプル」を含む 2 つの遷移表が存在し，これらはそれぞれ 2.3 で述べた $\nabla R$，$\Delta R$ に相当する．DELETE 文が実行された場合には前者のみが，INSERT 文が実行された場合には後者のみが，UPDATE 文が実行された場合にはこれらの両方の遷移表が生成される．

#### 3.3.2 ビュー差分の計算

ビューに発生する差分は，上で得られたテーブル差分とビュー定義クエリを用いて計算する．具体的には 2.3 で述べたように，更新の発生したテーブルを遷移表と置き換えるようにクエリを書き換え，これを実行することで，ビューの差分となるタプルを得ることができる．またこのときにはタプルの重複を考慮し，ビュー差分に含まれるタプルの重複度を計算するために count 集約関数を使用する．

例えば，

```
SELECT R.i, S.j FROM R, S WHERE R.x = S.x
```

というクエリで定義されたビューがあり，テーブル R が更新され，その遷移表が dR であった場合には，ビュー差分を求めるクエリは以下の通りとなる．

```
SELECT dR.i, S.j, count(*) __ivm_count__
 FROM dR, S WHERE dR.x = S.x
 GROUP BY dR.i, S.j
```

ここで __ivm_count__は重複度である．このクエリを 2 つの遷移表それぞれを用いて実行することで,「ビューから削除すべきタプル」と「ビューに挿入すべきタプル」を含む 2 つのビュー差分が得られる．これらはそれぞれ 2.3 で述べた $\nabla V$，$\Delta V$ に相当する．

なお，このクエリの書き換えは SQL で書かれたクエリ文字列を書き換えるのではなく，クエリをパースした結果である内部表現（Query 構造体）を書き換えることで行う．

複数のテーブルが同時に更新された場合には，2.3 で述べた通り，更新前のテーブル状態を知る必要がある．これはテーブルに発生した差分を逆に適用することで求めることができる．例えばテーブル R から削除されたタプルの遷移表が dR_old，挿入されたタプルの遷移表が dR_new の場合，テーブル R の更新前の状態 R から dR_new 内のタプルを取り除き，dR_old 内のタプルを加えることで求めることができる．しかし，R と dR_new に共通するタプルを探索するのはコストがかかるため，本実装では PostgreSQL のシステムカラムである xid，cid を用いる．これらはテーブル内の各タプルに存在するシステムカラムであり，それぞれそのタプルを挿入したトランザクションおよびコマンドの ID が格納されているので，これらの値を用いることでテーブル更新より前に存在してたタプルを抽出することができる．したがって，テーブルが更新されたトランザクションとコマンドの ID を pre_xid, pre_cid とすると，テーブル R の更新前の状態は以下のクエリで求めることができる．[1]

```
SELECT * FROM R
  WHERE (age(R.xmin) - age(pre_xid) > 0) OR
        (R.xmin = pre_xid AND R.cmin < pre_cid)
 UNION ALL
SELECT * FROM dR_old
```

#### 3.3.3 ビュー差分の適用

上で得られたビュー差分は SQL クエリを実行することによりマテリアライズドビューに適用する．集約または DISTINCT 句を含まない SPJ ビューの場合は，削除すべきタプル（$\nabla V$ に含まれるタプル）を DELETE 文で削除し，挿入すべきタプル（$\Delta V$ に含まれるタプル）を INSERT 文で挿入する．また，その際には重複度が考慮される．DELETE の際には指定された重複度の数だけのタプルをビューから削除する．これはビュー内の同じ内容のタプルを row_number ウィンドウ関数 [10] で番号付けし，その番号が重複度より小さいタプルのみを削除することで実現している．INSERT の際には重複度の数だけタプルを増やしてビューに挿入する．これは generate_series 関数 [11] を用いて実現している．

集約を含むビューの場合は 2.5 で述べた方法に基づき，ビュー内の行の中の集約値を UPDATE 文を用いて更新する．avg の集約値を更新するためには count および sum の値が必要となるが，これらの値は IMMV を定義する際に隠しカラムとして作成されビューに格納されている．

DISTINCT 句を含むビューの場合は 2.4 で述べたように，ビュー内の行の重複度を UPDATE 文を用いて更新する．重複度が 0 となった行はビューから削除される．

外部結合を含むビューの場合は，内部結合のみ含むビューの場合の処理に加えて，ダングリングタプル（dangling tuple）と呼ばれるタプルの処理が追加で必要となる．本実装では Larson and J. Zhou (2007) のアルゴリズム [5] をタプル重複を許すように拡張した方法 [8] でこれに対処している．

EXISTS 句を含むビューの場合は，EXISTS 相関サブクエリの結果に現れるタプルの重複度をビューの各行に隠しカラムとして格納することで対応している [8]．サブクエリの中で使用されているテーブルが更新された場合には，UPDATE 文によりこの重複度を更新し，これが 0 となった行はビューから削除される．

ビューに対して DELETE あるいは UPDATE を実行する際には，タプルの検索が発生する．そのため，ビューのサイズが大きい場合にビュー差分の適用に長時間かかる可能性がある．これを軽減するため，可能な場合には IMMV を定義する際に一意インデックス（unique index）が自動で作成される．具体的には，ビュー定義で使用されている全てのテーブルの主キーのカラムが全てビューのターゲットリストに含まれている場合には，これらのカラムにインデックスを作成する．集約を含むビューの場合には GROUP BY で参照されているカラムに一意インデックスを作成する．また，DISTINCT 句を含むビューの場合には，

1：age() は指定された ID のトランザクションがどれだけ古いかを返す関数である．また，実際のクエリでは xid, cid の型のキャストが必要となるが，ここでは省略している．

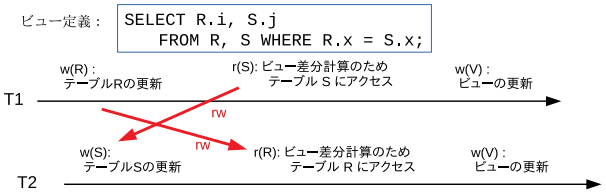


図 3　同時実行トランザクションによる不整合の発生

表 1　性能評価環境

| | |
|---|---|
| PC | Panasonic Let ’s note CF-SV7 |
| CPU | Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz (8 core) |
| DRAM | 15.5GB |
| ストレージ | SSD |
| OS | Ubuntu 18.04.4 LTS (64bit) |

全カラムに一意インデックスが作成される．それ以外の場合にはインデックスの自動作成はされないため，効率的なビューメンテナンスのためにはユーザが適切なインデックスを手動で作成することが求められる．

### 3.4　同時実行トランザクションによるビュー更新

テーブル更新と同じトランザクションにおいてビューの更新を行う即時メンテナンス方式では，複数のトランザクションが同時にビューを更新する際に不整合が発生する可能性がある．図 3 にその例を示す．テーブル R, S を結合するビューが定義されているとき，テーブル R を更新するトランザクション T3 とテーブル S を更新するトランザクション T2 が同時に実行されたとする．このとき，トランザクション T1 の中ではテーブル R の更新（ w(R)）の後，ビュー差分計算のためテーブル S が参照（r(S)）される．同様に，T2 ではテーブル S の更新（w(S)）とテーブル R の参照（r(R)）が行われる．この結果，T1 から T2 への rw-conflict と、T2 から T1 への rw-conflict の両方が発生し，rw-conflict の閉路が形成されるため，ビュー差分の計算に不整合が発生しうる．この不整合を防止するために，トランザクション T1, T2 でビュー差分の計算を同時に進行されないようにする．READ COMMITTED トランザクション分離レベルの場合には，排他ロックを取得して他のトランザクションが終了するのを待ってから，ビュー差分の計算および適用を行う．ただし，ビュー定義の中で参照されるテーブルが 1 つだけの場合には，ビュー差分の計算時にテーブルを参照することがなく上述の rw-conflict が発生しないため，この排他ロックは取得しない．この排他ロック回避の効果は 4.2 節で検証する．また，SERIALIZABLE、REPEATABLE READ トランザクション分離レベルの場合は不整合を回避できないため，他のトランザクションが同じビューを更新中であった場合にはトランザクションをアボートする．

## 4　性 能 評 価

本節では本実装による増分ビューメンテナンスの効果を確認するために行った性能評価について述べる．本実験で用いた環境は表 1 の通りである．

表 2　TPC-H: Q01 と Q09 の実行時間

| | クエリの実行 | ビューの参照 | REFRESH | 増分メンテナンス |
|---|---|---|---|---|
| Q01 | 10.311 s | 1.585 ms | 36.811 s | 17.104 ms |
| Q09 | 3.124 s | 2.331 ms | 6.153 s | 29.301 ms |

### 4.1　TPC-H による評価

評価は TPC-H のクエリを用いて行った．TPC-H は業界標準の意思決定支援系のベンチマークであり，このようなデータ解析クエリの実行には時間がかかるためマテリアライズドビューを利用した応答時間短縮が効果的である．まず 22 個のうちいくつのクエリが本実装における増分メンテナンス可能なマテリアライズドビューの定義で使用できるか確認した所，使用可能なクエリは 9 個であった．その他のクエリは，集約を含むサブクエリを含んでいるなど現状の実装では対応できなかったため，ビューを定義する段階でエラーとなった．

使用可能なクエリのうち，とくにクエリ実行の所要時間が長かった Q01 と Q09 を評価の対象とした．Q01 は大きな 1 つのテーブルに対する集約，Q09 は 6 つのテーブルを結合した上の集約を行うクエリである．また，TPC-H のスケールファクタは 1 を用いた．

表 2 に結果を示す．まずクエリを直接実行したときの応答時間は Q01 で 10.311 秒，Q09 で 3.124 秒であった．これに対し，それぞれ増分メンテナンス可能なマテリアライズドビューを作成し，ビューにアクセスしたときの応答時間は Q01 で 1.585 ミリ秒，Q09 で 2.331 ミリ秒であった．これはマテリアライズドビューを作成することで短い応答時間で結果を得られたことを意味している．

次にこれらのマテリアライズドビューに対し以下のコマンドを実行した時の応答時間を見る．

```
REFRESH MATERIALIZED VIEW <view name>;
```

REFRESH MATERIALIZED VIEW は PostgreSQL においてクエリを再実行することでビューを最新化するコマンドである．このコマンドの応答時間は Q01 では 36.811 秒，Q09 では 6.153 秒であり，再計算によるビューのメンテナンスには長時間かかることが確認された．

次に，Q01 と Q09 の定義で使用されているテーブルである lineitem を 1 行更新したときの応答時間を調べたところ，Q01 で 17.104 ミリ秒，Q09 で 29.301 ミリ秒であった．テーブルを更新した直後にビューの内容は自動的に増分メンテナンスで更新されるため，これはテーブルの更新とビューの更新の両方を含んだ処理時間となっている．それにもかかわらず，REFESH MATERIALIZED VIEW コマンドを実行した場合に比べると，Q01 で 2000 倍，Q09 で 200 倍以上高速であるという結果になった．

### 4.2　テーブル更新性能への影響

本実装ではテーブルが更新される度にビューが更新される．そのため，ビュー更新のオーバヘッドがテーブル更新性能に影響する．特に，排他ロックが取得される場合には同時接続数が

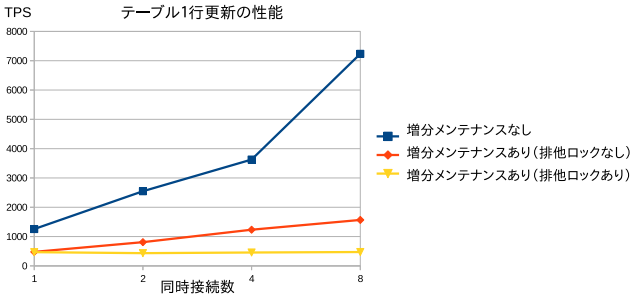


図 4　テーブル更新性能

増えても実行性能が向上しないことが予測される．その一方で，3.4 節で述べたように，ビュー定義で参照されるテーブルが 1 つだけの場合には排他ロックが回避できるため，この場合には性能の低下が抑制できることが期待できる．これを確認するため，IMMV が作成されている状況下でのテーブル更新性能を評価した．

TPC-H の Q01 クエリで参照されているテーブル lineitem の 1 行をランダムに選んで更新するトランザクションを繰り返し実行したときの tps（Transaction per second）を計測した．tps は IMMV が存在しない場合と，Q01 のクエリを用いて定義された IMMV が 1 つ存在する場合とを比較した．また，このビューで参照されるテーブルは lineitem のみなので本来は排他ロックは回避されるが，排他ロックの影響を見るためにこの機能を意図的に無効にした場合の性能も計測した．tps の計測には PostgreSQL の標準ベンチマークツールである pgbench [13] を用いた．同時接続数は 1, 2, 4, 8 を用いた．

結果を図 4 に示す．IMMV が定義されている場合には，テーブル更新性能が低下することが確認された．また，排他ロックを取得する場合には同時接続数を増やしても tps が上昇しないが，3.4 節で述べた排他ロック回避を実施した場合には同時接続数に応じて tps が上昇することが確認された．同時接続数 4 までは，IMMV が定義されていない場合に比べて 1/3 程度の性能低下が見られた．これは，テーブルの更新に比べて，ビューからのタプル削除，およびビューへのタプル挿入という追加の処理を行っているためだと考えられる．同時接続数 8 で性能低下の割合が 1/3 よいり大きいのは，同時実行トランザクション数が増えることでビューからタプルを削除する際に取得される行ロック競合による待ちの発生率が上昇したためだと思われる．

## 5　おわりに

本論文では PostgreSQL の開発コミュニティに提案中の増分ビューメンテナンス機能の実装方法とその性能評価について報告した．性能評価の結果，ビュー定義で使用される実テーブルの更新性能が低下する影響もあったが，一から再計算するよりも短時間でビューの更新が可能となることが確認できた．このことから，本実装による増分ビューメンテナンスは，テーブルの更新が大量に発生するような状況では不向きである一方で，テーブルの更新が少ないが，いざ更新された際には最新のクエリの結果をすぐに欲しいといった状況で有用である．

本実装ではマテリアライズドビューはテーブルが更新された直後に自動的に更新される．すなわち，即時メンテナンスアプローチによる増分ビューメンテナンスを提供しているが，遅延メンテナンスの機能は未実装である．本論文でも確認された通り，即時メンテナンスではテーブル更新の度にビューの更新を行うため，遅延メンテナンスに比べるとテーブル更新性能に与える影響が大きい．そのため，テーブルの更新が多い状況における有用性を向上させるためには，遅延メンテナンスのサポートが今後の課題としてあげられる．

また TPC-H ベンチマークのクエリを使った検証では，その半数以上のクエリが現状の実装では増分メンテナンス可能なマテリアライズドビューの定義として使用できなかった。より現実的で複雑なクエリをサポートするために，ビュー定義に使用できるクエリの制限を緩和していくことも今後の課題の 1 つにあげられる．

## 文　献

[1] R. Chirkova and J. Yang, “Materialized Views,” Foundations and Trends in Databases 4(4), pp.295-405, 2012. SIGMOD Record, 27(3), September 1998.
[2] T. Griffin and L. Libkin, “Incremental Maintenance of Views with Duplicates,” Proc. ACM SIGMOD ’95, pp.328–339, 1995.
[3] A. Gupta and I. S. Mumick, “Maintenance of materialized views: Problems, techniques, and applications,” IEEE Data Engineering Bulletin, vol. 18, no. 2, pp. 3-18, 1995.
[4] A. Gupta, I. S. Mumick and V.S. Subrahmanian, “Maintaining Views Incrementally,” In SIGMOD Conference, 1993.
[5] P-Å Larson and J. Zhou, “Efficient Maintenance of Materialized Outer-Join Views,” In ICDE, pp.56-65, 2007.
[6] Quoc Vinh, N. T. “Synchronous incremental update of materialized views for PostgreSQL,” Programming and Computer Software 42(5), pp.307-315, 2016.
[7] 長田悠吾, 石井達夫, 増永良文, “OID を用いた Incremental View Maintenance 方式の PostgreSQL への試験的実装,” 第 11 回データ工学と情報マネジメントに関するフォーラム (DEIM Forum), 2019.
[8] 長田悠吾, 星合拓馬, 石井達夫, 増永良文, “タプル重複のもとで外部結合および準結合を含むビューの増分メンテナンスとその PostgreSQL への実装,” 第 12 回データ工学と情報マネジメントに関するフォーラム (DEIM Forum), 2020.
[9] PostgreSQL: The world’s most advanced open source database https://www.postgresql.org/
[10] PostgreSQL: Documentation: devel: 9.22. Window Functions https://www.postgresql.org/docs/devel/functions-window.html
[11] PostgreSQL: Documentation: devel: 9.25. Set Returning Functions https://www.postgresql.org/docs/devel/functions-srf.html
[12] PostgreSQL: Documentation: devel: 39.1. Overview of Trigger Behavior https://www.postgresql.org/docs/devel/trigger-definition.html
[13] PostgreSQL Documentation: devel: pgbench, https://www.postgresql.org/docs/devel/pgbench.html
[14] IVM (Incremental View Maintenance) development for PostgreSQL https://github.com/sraoss/pgsql-ivm/