ストリームデータ処理における状態一貫性の保証

小山田昌史† 川島 英之†† 北川 博之††

† 筑波大学システム情報工学科 〒305-8573 茨城県つくば市天王台 1–1–1
†† 筑波大学システム情報系情報工学域 〒305-8573 茨城県つくば市天王台 1–1–1
E-mail: †masa@kde.cs.tsukuba.ac.jp, ††{kawasima,kitagawa}@cs.tsukuba.ac.jp

あらまし 情報技術の発展により，ストリームデータを時々刻々と発信し続けるストリーム情報源は増加の一途にある．これに伴い，イベント検知のような単純な処理だけでなく，様々な分析処理が豊富なストリームデータに対しておこなわれるようになってきた．分析処理の中では，ストリームデータに加えデータベースにおけるリレーションや分類器におけるモデルなど，様々な外部データが用いられる．こうした外部データは更新されうるが，ストリームデータの分析中にデータの更新処理がおこなわれた場合，そのタイミングにより分析処理の結果は異なるものとなってしまう．本稿ではそうした現象を防ぐため，ストリームデータに対して状態一貫性という概念を導入し，これを保証するストリームデータ処理方式としてロック方式とスナップショット方式の二つを提案する．そして，プロトタイプシステムによる評価実験により，継続的クエリ処理のスループットと更新処理のスループットはトレードオフの関係にあり，継続的クエリ処理のスループットが優先される状況においてはロック方式を，更新処理のスループットが優先される状況においてはスナップショット方式を用いた方がよいということを明らかにする．
キーワード ストリームデータ処理，複合イベント処理，一貫性，トランザクション処理

Masafumi OYAMADA†, Hideyuki KAWASHIMA††, and Hiroyuki KITAGAWA††

† Graduate School of Systems and Information Engineering, University of Tsukuba
1-1-1 Tennoudai, Tsukuba City, Ibaraki 305-8573, Japan
†† Faculty of Engineering, Information and Systems, University of Tsukuba
1-1-1 Tennoudai, Tsukuba City, Ibaraki 305-8573, Japan
E-mail: †masa@kde.cs.tsukuba.ac.jp, ††{kawasima,kitagawa}@cs.tsukuba.ac.jp

1. はじめに

センサデータ，ネットワークパケット，アクセスログ，監視カメラからの映像など，継続的に発生するストリームデータは増加の一途にある．今日に至るまで，膨大かつ高い到着レートとなりがちなストリームデータを限られた計算資源で処理するために，作業量の削減をねらったデータの切り捨てやシステムの QoS (Quality of Service) 維持などについて，様々な研究がおこなわれてきた．これに対し，近年では計算資源の発展やストリームデータを発信する情報源の増加もめざましく，多様なストリームデータを蓄積し様々な分析に利用しようという動きが活発となってきている [10,11].

ストリームデータの分析処理では，データベースシステム中のリレーションや，機械学習フレームワーク [12] におけるモデルなど，ストリームデータ以外の様々な外部データが参照される．例えば，商品の購入情報がストリームデータとして流れてくる際，その購入情報と外部のデータベースに格納された商品情報とを結合した上で分析をおこなうという処理がある．この処理において，一つの購買情報に対して複数の商品情報が対応するとき，結合処理の最中に商品情報へ更新が生じると，どのような結果が生じうるだろうか．あるタイミングでは一つの購買情報に対して古い商品情報のみが結合され，またあるタイミングでは一つの購買情報に対して新旧両方の商品情報が結合される．本稿では後者の状況を，ストリームデータ中の単一タプルに対する状態の一貫性が失われていると表現する．

例 1.1 に，単一タプルに対して状態の一貫性が失なわれる例を示す．

例 1.1 (単一タプルに対する一貫性の喪失) 図 *1* は，ストリームデータ中のタプル s_1 と外部リレーション R の結合処理中に，R へ更新が生じた場合の処理結果例をあらわしている．ここで，r は R 中のタプルを，R' は更新処理後のリレーションを，r'_i は R' 中のタプルをそれぞれ意味する．また，この例では，s_1 との結合条件を満たす R 中のタプルは r_1, r_2 の *2* つであるとする．ここで，入れ子ループ結合方式などにおいてリレーション R 中のタプルを順番に s_1 と比較していく際，r_2 までの比較が終了したところで R へ更新が生じ，その結果として更新後のリレーション中のタプル r'_3, r'_4 が結合条件を満たすようになり，一方で r'_1, r'_2 については結合条件を満たさなくなったという状況を考える．このとき，結合処理の結果としては

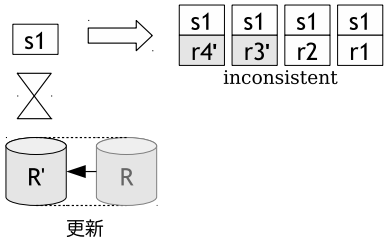

図 1　単一タプルに対する状態の一貫性

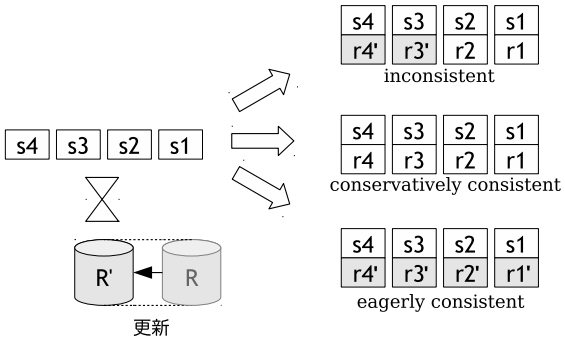

図 2　複数タプルに対する状態の一貫性

$< s_1, r_1 >, < s_1, r_2 >, < s_1, r'_3 >, < s_1, r'_4 >$ が出力されるが，この処理結果には，ストリームデータ中のタプル s_1 が見るリレーションが一貫していないという問題がある．すなわち，もしストリームデータ中のタプル s_1 の見るリレーションが一貫しているならば結合結果は $< s_1, r_1 >, < s_1, r_2 >$ か $, < s_1, r'_3 >, < s_1, r'_4 >$ のどちらかになるはずであるが，得られた結果はそのどちらにも一致しない．

例 1.1 に見られるような一貫性の問題については，過去より議論がおこなわれてきた [5,6,18]．中でも Bornea らは，ストリームデータとリレーショナルデータの効率的な結合方式を提案する際，同時にこの問題について言及し，結合処理の最中の更新処理を禁止するという対策を提案している．

ストリームデータ中の単一タプルに対する状態一貫性について，一貫性が失われる状況と，それを防ぐ先行研究について説明した．本稿では，その一貫性の議論を例 1.2 のようなストリームデータ中の複数タプルに対するものへと発展させる．

例 **1.2 (複数タプルに対する一貫性の喪失)**　図 *2* は，ストリームデータより順に到着するタプル列 s_1, s_2, s_3, s_4 と外部リレーション R の結合中に，R へ更新が生じた場合の処理結果例をあらわしている．ふつう，ストリームデータ中のタプル列と外部リレーションの結合処理は，ストリームデータからタプルが到着するたびにおこなわれる．すなわち，s_i が到着するたびに R の結合条件を満たすリレーション r_i が取得され，結合結果となる $< s_i, r_i >$ が出力される．ここで，図 *2* において *inconsistent* と表記された処理結果例 $< s_1, r_1 >, < s_2, r_2 >, < s_3, r'_3 >, < s_4, r'_4 >$ を考える．この結果においてストリーム中のタプル s_1, s_2, s_3, s_4 はそれぞれ一貫したリレーション R ないし R' を見ているため，前述した単一タプルに対する一貫性を持つといえる．一方で，タプル列 s_1, s_2, s_3, s_4 を一つのまとまりとして考えると，s_1, s_2 が R を参照しているのに対し s_3, s_4 は更新後の R' を参照しており，参照した外部リレーションの状態が一貫していない．

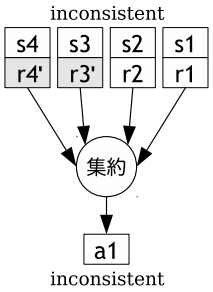

図 3　状態の一貫していない複数タプルの集約

例 1.2 に見られるような複数タプルに対する状態一貫性の喪失は，図 3 のように一貫性の失われたタプル列が集約演算などによりまとめて処理される場合に問題となる．このとき，集約演算に入力されたタプルの内 s_1, s_2 については更新前の状態を参照し，s_3, s_4 については更新後の状態を参照しているが，それらが同列に扱われ一つの出力 a_1 を算出している．本稿では，こうした出力結果 a_1，すなわち一貫性のないストリームデータを処理することにより得られた結果についても，同様に一貫性がないものとして扱う．

以上のような一貫性の問題について，本稿では状態の一貫性をクエリの処理結果に対して保証することを目的とし，次のような流れでこの課題に取り組む．はじめに，2. 節でクエリの処理結果に対して状態の一貫性を定義する．次に，3. 節で出力の一貫性を保証する継続的クエリの実行方式を提案し，4. 節でプロトタイプシステムによる提案手法の性能評価と考察をおこなう．そして，5. 節で関連研究の紹介をおこなったあと，6. 節で本稿のまとめをおこなう．

2.　状態一貫性

ここでは，前節で述べた一貫性の問題を明らかにするため演算の処理結果に対して状態一貫性という概念を導入する．

はじめに，本稿で扱うストリームデータの定義を次のように与える．ストリームデータに対しては，これまでに様々なモデルが提案されているが [2–4, 14]，ここでは Arasu らによる関係代数に基づくモデル [2] を用いる．

定義 **2.1 (ストリームデータ)**　ストリームデータ S を $s{:} < r, t >$ なる要素 s の多重集合として定義する．ここで，r は S のスキーマ S_{schema} に属すタプルを，$t \in T$ は s がシステムに到着した時刻（システム時刻）をそれぞれ意味する．このとき，T はシステムが扱う時刻のドメイン(注1)をあらわす．

次に，データベースにおけるリレーションや分類器におけるモデルなど，ストリームデータ処理内で参照される更新可能な外部データを形式的に扱うため状態 **(State)**，参照関数 **(Reference Function)**，という概念を導入する．

(注1)：時刻を要素に持つ順序集合．例としては UNIX 時間（秒単位）や西暦（年単位）があげられる．

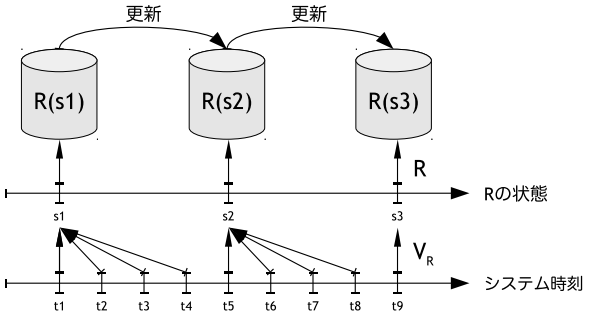

図 4　版関数と参照関数による時刻からの外部データ参照

定義 2.2 (状態)　状態 $\mathfrak{s}$ を半順序集合 $\mathfrak{S}$ の要素として定義する．

定義 2.3 (参照関数)　状態 $\mathfrak{s}$ から各時点での外部データへの写像を参照関数と呼び，$R : \mathfrak{S} \to \mathfrak{D}$ であらわす．このとき，$\mathfrak{D}$ は外部データの定義域を意味する．これは，$\mathfrak{s}$ を R に与えることで，ある時点 $\mathfrak{s}$ での外部データ $R(\mathfrak{s})$ が定まるということを意味しており，$\mathfrak{s}$ は外部データ $R(\mathfrak{s})$ の状態と呼ばれる．また，R について $R^{-1}(R(\mathfrak{s})) = \mathfrak{s}$ が成り立つ，すなわち外部データを R の逆関数へ与えることで，その外部データの状態を得ることができるとする．

さらに，ある時刻における外部データを参照する，すなわち時刻から外部データの状態への対応付けをおこなうために版関数 **(Version Function)** という概念を導入する．

定義 2.4 (版関数)　参照関数 R と時刻 $t \in T$ を与えたときに，t の時点で最新となる R の状態を返す写像として，$V_R : T \to \mathfrak{S}$ を定義する．V_R は参照関数 R の版関数と呼ばれる．

定義 2.2 から定義 2.4 にかけて示した概念のイメージを図 4 に示す．各システム時刻は V_R により状態 $\mathfrak{s}$ へと対応し，各状態は R により外部データへと対応する．

そして，継続的クエリ **(Continuous Query)** を次のように与える．

定義 2.5 (継続的クエリ)　継続的クエリ q を $q : \{S\} \times \{\mathfrak{D}\} \to S$ なる写像として定義する．これは，継続的クエリが任意数のストリームデータと任意数の外部データを用い，処理結果としてストリームデータを出力することをあらわす．

以上の定義を用いて，継続的クエリの処理結果についてその状態一貫性 **(State Consistency)** を次のように与える．

定義 2.6 (状態一貫性)　継続的クエリ q のある処理結果について，その処理内で複数回呼ばれた全ての参照関数に対し入力となった状態が全ての呼び出しで等しいとき，その処理結果は状態一貫性を持つという．反対に，一つでも異なる状態が呼び出しの中で使われていた場合，その処理結果は状態一貫性を持たないという．

状態一貫性は，継続的クエリが処理結果を計算する際に参照した外部データの状態，すなわち版が一貫することを保証する．

3. 状態一貫性を保証したストリームデータ処理

本節では，継続的クエリの処理結果一つ一つが状態一貫性を持つと保証されるストリームデータ処理方式を提案する．なお，本稿では継続的クエリ内にオーバーラッピングウィンドウが含まれる場合を対象としない．オーバーラッピングウィンドウが含まれる場合の状態一貫性については，3.4 節において詳しく述べる．

提案する方式は，ロックを用いた方式と，スナップショットによる方式の二つに分けられる．ここでは，二つの方式の間で共通となる処理について説明したあと，各方式の詳細な説明へと移る．

3.1 継続的クエリ処理

図 5 は，ある継続的クエリにイベントが次々と到着し，処理結果が一つ出力されるまでの内部動作をあらわしている．ここで「集約」と書かれたノードは二つのイベントを受け取り一つのイベントを出力する集約演算を，「`r(x)`」と書かれたノードは一つのイベントを受け取り外部データ `x` を参照して処理をおこなった上でその結果となるイベントを一つ出力する演算を，それぞれあらわす．

はじめ，出力部は集約演算に処理結果を一つ要求する (a)．処理結果の要求は図 5 中，破線の矢印であらわされている．集約演算は処理結果の要求を受け取ると，自身がその処理結果を一つ出力するために二つの入力イベントを必要とすることから，自分の上流にある `r(x)` に処理結果を一つ要求する．`r(x)` も同様に自分の上流にある入力部へと処理結果を一つ要求するが，入力部は要求をブロッキング・キューで管理しており，このときにはまだイベントが一つも到着していないため，そこで要求はブロックされる (b)．このように，処理結果の要求が出力部よりおこなわれ，それが入力部までさかのぼりながら伝搬していくところがポイントとなる．

やがて，イベント `A` が入力部へと到着しキューへと格納される．すると，さきほどブロックされていた `r(x)` からの要求が受け付けられ，イベント `A` はデキューされて下流の `r(x)` へと渡される (c)．このとき，デキュー前に **Begin** 処理と呼ばれる処理がおこなわれるが，これは前述の二方式によって処理内容が異なるため，詳細は後述する．`r(x)` はイベント `A` を受け取ると，外部データ `x` を参照しながら処理をおこない，処理結果である `A'` を集約演算に渡す (d)．

ここで，集約演算は結果を出力するために二つの入力イベントを必要とするため，再度処理結果の要求を `r(x)` へとおこなう．こうして再び要求が出力部まで伝わりイベント `B` がブロッキング・キューより取り出されるが，このとき Begin 処理はおこなわれないところが前回とは異なる．そして，イベント `B'` が集約演算に渡り (e)，二つのイベントがそろった集約演算は集約処理をおこなって，その結果の `A'+B'` を出力部へと渡す．

出力部は結果を `r(x)` より受け取ると，それを出力した上で，**End** 処理と呼ばれる処理をおこなう (f)．この処理についても Begin 処理と同じく前述の二方式によって処理内容が異なるため，詳細は後述する．Begin 処理を終えると，出力部は再び

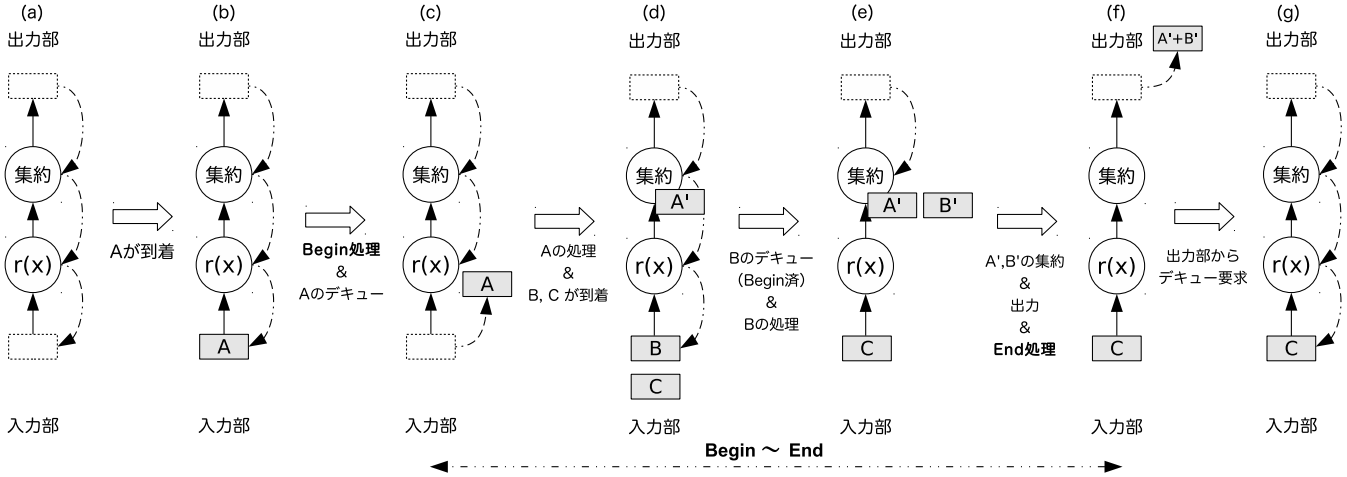

図 5　処理結果を一つ出力するまでの流れ

Algorithm 1 ロック方式における Begin 処理

```
/* 参照しうる全ての外部データについて */
for d in D do
    ReadLock(d) /* 読み込みロックを要求 */
end for
```

Algorithm 2 ロック方式における End 処理

```
/* 参照しうる全ての外部データについて */
for d in D do
    Unlock(d) /* ロックを開放 */
end for
```

Algorithm 3 スナップショット方式における Begin 処理

```
/* 参照しうる全ての外部データについて */
for d in D do
    ReadLock(d) /* 読み込みロックを要求 */
    SnapShotData[d] ← d /* Begin 時点での，d を保存 */
    Unlock(d) /* ロックを開放 */
end for
```

r(x) に処理結果を要求する (g)．この繰り返しで，ストリームデータの処理がおこなわれていく．

以上のような処理方式を取った場合，状態の一貫性を保証するためには Begin 処理から End 処理の間で参照される外部データの状態が全て同じであることを保証すればよい，ということが状態一貫性の定義からいえる．

ここでのポイントは以下の通りである．

- 処理結果の要求が出力部よりおこなわれ，それが入力部までさかのぼりながら伝搬していく．
- 入力部のブロッキング・キューからイベントが取り出される際には Begin 処理と呼ばれる処理がおこなわれる．ただし，既に Begin 処理が呼ばれており，End 処理がまだ呼ばれていない場合は，その限りでない．
- 出力部が処理結果を出力する際には，End 処理と呼ばれる処理がおこなわれる．
- 状態の一貫性を保証するためには Begin 処理から End 処理の間で参照される外部データの状態が全て同じであることを保証すればよい．

3.2　ロック方式

状態の一貫性を保証するためには Begin 処理から End 処理の間で参照される外部データの状態が全て同じであることを保証すればよいと述べた．ここでは，そうした保証をおこなう一手法としてロック方式を提案する．ロック方式では，アルゴリズム 1 とアルゴリズム 2 に示すように，Begin 処理が始まってから End 処理が終わるまで外部データへロックをかけることによって，外部データが更新されて異なる状態となってしまうことを防ぐ．

ここで，D は継続的クエリが参照する全ての外部データを意味する．また，ReadLock(d) は外部データ d に対して読み込みロックを要求し，コンフリクトするロックが既にかかっていた場合はロックが開放されるまで処理をブロックする関数をあらわし，Unlock(d) は外部データ d に対して自らがかけたロックを開放する関数をあらわす．

ロック方式は正しく動作するが，ストリームデータの性質上，出力部が処理結果を一つ出力するまでにかかる時間，すなわち外部データに読み込みロックをかける時間が非常に長くなることもあり，その間の更新処理を阻害してしまうという欠点を持つ．

3.3　スナップショット方式

更新処理を阻害するロック方式に対し，本稿では更新処理を阻害しない状態一貫性の保証方式として，外部データのスナップショットによる方式を提案する．スナップショット方式は，ロック方式のように外部データに対する更新を防ぐのではなく，Begin 処理の時点での外部データをスナップショットとして保存しておき，Begin 処理から End 処理の間で参照される外部データの状態が一貫するよう，スナップショットを参照しながら処理をおこなう．アルゴリズム 3 とアルゴリズム 4 とに，スナップショット方式における Begin 処理と End 処理の内容を示す．

Algorithm 4 スナップショット方式における End 処理

```
/* 参照しうる全ての外部データについて */
for d in D do
  /* 保存されていた外部データを削除 */
  DeleteSnapShotData(SnapShotData[d])
end for
```

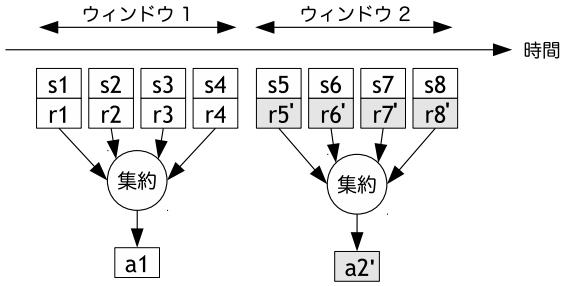

図 6　ノンオーバーラッピングウィンドウの例

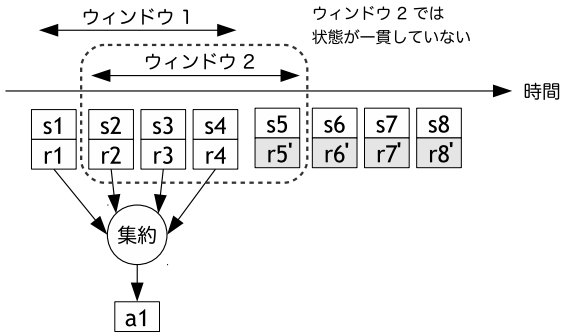

図 7　オーバーラッピングウィンドウの例

スナップショット方式では，Begin 処理が呼ばれた時点での各外部データ d をスナップショットとして SnapShotData[d] へと保存する．そして，演算処理の中で外部データを参照する際には，SnapShotData[d] に保存されているスナップショットを参照する．これにより，Begin 処理から End 処理内で参照される外部データの状態が一貫することが保証される．また，処理結果を出力した時点で保存されているスナップショットは必要なくなるため，End 処理内でそれらを削除することにより，使用するリソース量を抑えることができる．

このように，スナップショット方式には読み込みロックを長時間保持しないため外部データの更新処理を阻害しにくいという利点がある．一方で，外部データのスナップショットを保存する処理が必要となるため，頻繁に更新が発生する状況での性能悪化が予想される．

3.4　オーバーラッピングウィンドウへの対応

本節で提案した状態一貫性の保証方式は，継続的クエリ内に登場するウィンドウ演算が図 6 にあらわされるようなノンオーバーラッピングウィンドウとなる，すなわちスライド幅とウィンドウ幅が同じとなることを前提としており，図 7 にあわらされるようなオーバーラッピングウィンドウが含まれる場合を対象としていない．一方で，実際のストリームデータ処理においてはしばしばオーバーラッピングウィンドウが用いられるため，オーバーラッピングウィンドウが含まれる場合にも状態一貫性の保証をおこなうことは，極めて重要であるといえる．

例 3.1 に，オーバーラッピングウィンドウにより状態一貫性が喪失される例を示す．

例 3.1 (オーバーラッピングウィンドウによる状態一貫性の喪失) 図 7 は，ウィンドウ幅 *4*，スライド幅 *1* の集約演算に対するオーバーラッピングウィンドウが，ストリームデータからウィンドウを二つ切り取った様子をあらわしている．ここでは s_4 の到着後に外部データが更新され，s_1から s_4 の参照した外部データと s_5から s_8 の参照した外部データの状態が異なるとしている．このとき，ウィンドウ *1* により切り取られるストリームデータは全て更新前の外部データを参照しているため状態一貫性を持つが，ウィンドウ *2* により切り取られるストリームデータは s_2, s_3, s_4 が更新前の外部データを参照しているのに対し，s_5 が更新後の外部データを参照しているため，状態一貫性を持たない．

現在，オーバーラッピングウィンドウへの対応方式については考察をおこなっている最中であるため，本稿では具体的な対策を述べず，手法の提案については今後の課題とする．

4. 実　　験

本節ではプロトタイプシステムを用いた実験をおこない，前節で提案した二つの状態一貫性保証方式の特徴について考察する．

4.1　実 験 環 境

実験に用いた計算機は，プロセッサが Intel Core2 Duo P8700 2.53Ghz，メモリが 4GB，オペレーションシステムが Linux 2.5.32-37 となっている．実験に用いたプロトタイプシステムは，3.1 節に示した方式で動作するストリームデータ処理システムと，主記憶上で動作するデータベースマネージャからなる．ストリームデータ処理システムとデータベースマネージャは同一プロセス上，異なるスレッドで動作する．実装言語には C++を用い，コンパイルは GNU g++ 4.4.3 でおこなった．このとき，コンパイル時の最適化オプションは-O2 に設定している．

4.2　実 験 概 要

4.2.1　継続的クエリ

実験で用いる継続的クエリとして，状態の一貫性が喪失されうる単純な例を考え，図 8 のような関係演算の処理木をストリームデータ処理システムに登録した．以下に，その処理内容を記す．

（1） はじめに，ストリームデータは外部リレーションと等結合される．このとき，結合のアルゴリズムとしては入れ子ループ結合を用いており，一つの結合処理が開始してから終了するまで，外部リレーションに対して読み込みロックを取得する．全ての実験において一律で，外部リレーション中のタプル数は 1,000，結合演算の結合選択率は 1.0 とした．

（2） 次に，結合結果は選択演算により選択される．選択演算の選択率は全ての実験において一律で 0.5 とした．

（3） そして，選択された結合結果は集約演算へと入力され

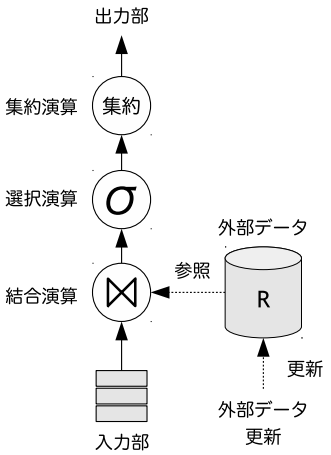

図 8　実験に用いる処理木

る．集約演算は w_{size} タプル毎に属性値のヒストグラムを作成してその結果を出力する．すなわち，ウィンドウ幅 w_{size}，スライド幅 w_{size} のウィンドウ演算をおこなう．

4.2.2　外部リレーションの更新処理

また，本稿で対象としたストリームデータ処理中に外部データが更新されうる状況を再現するために，外部リレーションを定期的に更新するスレッドを走らせる．このスレッドは，ストリーム処理システムが上記の処理木を継続的に処理する間，一定の間隔 u_{interval} で更新処理をおこなう．更新処理は，外部リレーション中からランダムに一つのタプルを選択し，そのタプル中の特定の属性値をランダムに書き換えるものとなっている．このとき，一つの更新処理に要する時間は平均して 10 マイクロ秒であった．更新の際にはリレーション全体について書き込みロックが取得され，この間ストリームデータ処理システムがリレーションを参照しようとした場合には，更新処理が終わるまでストリームデータ処理システムの処理はブロックされる．

4.2.3　スループットの計測

以上の設定を用い，本節では各提案手法毎，各パラメータ毎に次のような流れで計測をおこなった．ストリームデータ処理システムに対しストリームデータを一定の到着レート s_{rate} で送信し，ストリームデータ処理システムが一定数のストリームデータを処理し終えるまでの時間を計測した上で，継続的クエリ処理のスループットを算出する．また，この間に外部リレーション更新スレッドがおこなった外部リレーションの更新回数も同時に計測し，外部リレーション更新処理のスループットを算出する．

以後に本節で示す実験結果は，特に断りのない限り同じ測定を五回おこなったものの平均値となっている．

4.3　更新処理が継続的クエリ処理に与える影響

はじめに，外部リレーションの更新処理が継続的クエリ処理に与える影響を観察するため，更新処理のレートを変化させたときの継続的クエリ処理のスループットの変化を計測した．図 9 に更新処理の到着レート（更新間隔 u_{interval} の逆数）を 1 query/sec から 100,000 query/sec まで変化させたときの，継続的クエリ処理のスループットの変化を示す．

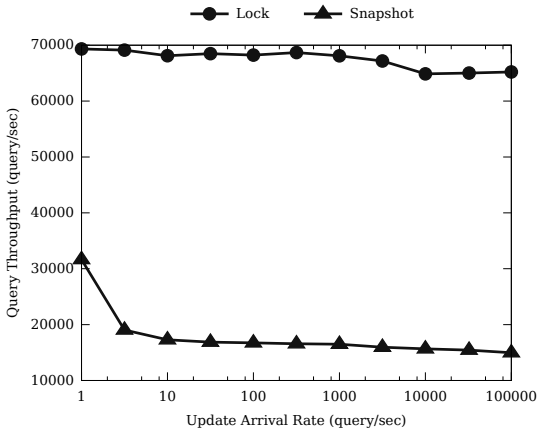

図 9　更新処理の到着レートを変化させた際の継続的クエリ処理のスループット変化

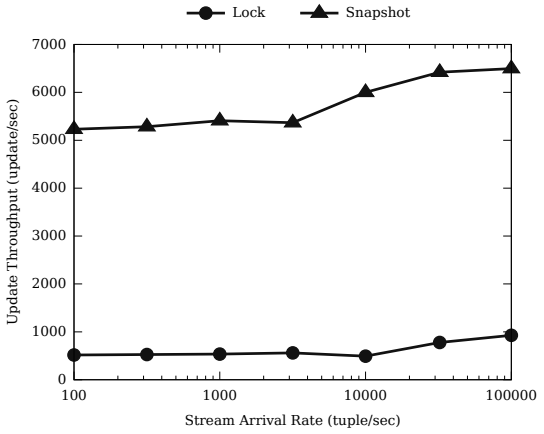

図 10　ストリームデータの到着レートを変化させた際の更新処理のスループット変化

図 9 を見ると，ロック方式，スナップショット方式どちらにおいても更新処理の到着レートが上がるごとに，継続的クエリ処理のスループットが低下していくことが確認できる．これは，更新処理が外部リレーションへ書き込みロックをかける頻度が上がるにつれて，継続的クエリ中の結合演算が読み出しロックを外部リレーションへかけようとしてブロックされる頻度が上がったためと考えられる．

また，ロック方式がスナップショット方式と比べ数倍以上のスループットを示していることも観察できるが，これは Begin 処理の計算量の違いによるものと考えられる．ロック方式の Begin 処理は外部リレーションに対するロックを取得するものであり定数時間で済むが，スナップショット方式の Begin 処理は現在の外部リレーションを保存するものであり，計算量は外部リレーションのサイズに左右されて非常に大きいものとなりうる．こうした計算量の違いが，スループットの差にあらわれたと予想される．

4.4　ストリームデータ到着レートが更新処理に与える影響

次に，ストリームデータの到着レートが更新処理に与える影響を観察するため，ストリームデータの到着レートを変化させたときの，更新処理のスループットの変化を計測した．図 10 にストリームデータの到着レート s_{rate} を 100 tuple/sec から

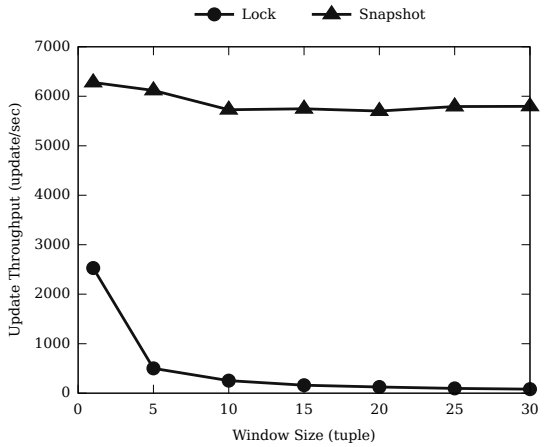

図 11　ウィンドウサイズを変化させた際の更新処理のスループット変化

100,000 tuple/sec まで変化させたときの，更新処理のスループットの変化を示す．

図 10 を見ると，ストリームデータ処理システムが状態一貫性の保証にスナップショット方式を用いた場合，ロック方式と比較して数十倍となるスループットを外部データの更新処理が示していることが確認できる．すなわち，スナップショット方式はロック方式と比較してはるかに外部データの更新処理を阻害しにくいといえる．これは，ロック方式が Begin 処理から End 処理の間で外部データに対して読み込みロックをかけ続けるのに対して，スナップショット方式は外部データを保存する間のみ読み出しロックをかけるためであると考えられる．

4.5　継続的クエリの実行時間が更新処理に与える影響

最後に，継続的クエリが処理結果を一つ出力するまでの時間が更新処理に与える影響を観察するため，継続的クエリの実行時間につながる集約演算のウィンドウ幅を変化させたときの，更新処理のスループットの変化を計測した．図 11 に集約演算のウィンドウ幅 w_{size} を 1 tuple から 30 tuple まで変化させたときの，更新処理のスループットの変化を示す．

図 11 を見ると，ロック方式ではウィンドウ幅が大きくなるにつれて更新処理のスループットが大幅に低下していくのに対し，スナップショット方式では更新処理のスループットが低下していない．これは，ロック方式が Begin 処理から End 処理までの間ロックを保持し続けるのに対し，スナップショット方式が Begin 処理の中で即座にロックを解除してしまうためであると考えられる．ウィンドウサイズが大きくなるにつれて Begin 処理から End 処理までの時間は長くなるため，ロック方式ではロックを保持し続ける時間も長くなり，更新処理を妨げやすくなってしまう．一方，スナップショット方式では Begin 処理から End 処理までの時間がいくら長くなろうと，その間ロックを保持しないので更新処理を妨げることはない．

以上より，継続的クエリにおける Begin 処理から End 処理の実行時間が更新処理に影響を与えるのはロック方式を用いた場合のみであるということができる．

4.6　考　　察

継続的クエリ処理のスループットと更新処理のスループットとはトレードオフの関係にあり，継続的クエリ処理のスループットが優先される状況においてはロック方式を，更新処理のスループットが優先される状況においてはスナップショット方式を用いた方がよいと考えられる．

5. 関 連 研 究

本稿のはじめに述べたストリームデータと外部のリレーションの結合については，効率的に結合処理をおこなう方式が数多く提案されているものの [7, 15–17]，外部のリレーションに更新が生じるという状況が考えられてこなかった．これに対し，Bornea らは効率的な結合手法を提案するとともに，外部のリレーションに更新が生じる場合の処理方法に触れている [5]．彼女らの提案する方式は次のとおりである．

（1） まず，結合処理がおこなわれている間は，リレーションに対する更新処理を保留する．

（2） そして，結合処理が終わると，保留されていた更新処理をおこなう．

（3） その後，先ほどと同様に更新処理がおこなわれている間は後続する結合処理を保留し，更新処理が終了したところで結合処理をおこなう．

この繰り返しにより，ストリームデータ中の単一タプルの見るリレーションの状態が一貫していない，という問題を防ぐことができる．彼女らはこれを「ストリーム中の単一タプルとリレーションの結合処理がトランザクションを構成する」と表現している(注2)．彼女らの提案する方式は結合オペレータについての状態一貫性を保証するが，継続的クエリ全体の状態一貫性は保証しない．

```
Select P.price
From   Items [Rows 5] as I, PriceTable as P
Where  I.itemID = P.itemID
```

図 12　一貫性の問題を含む CQL のクエリ例

Arasu らは，ストリームデータとそれに対する問合せ言語 CQL のセマンティクスを定める際，一貫性の問題について言及している [2]．彼らは，図 12 のような継続的クエリを例としてあげた．このクエリ中，Items は購入された商品のストリームを，PriceTable は商品の価格が格納されたリレーションをそれぞれあらわす．このとき，Items から幅 5 タプルの論理ウィンドウ(注3)によってタプル列が切り取られ PriceTable と結合処理がおこなわれるが，もし結合処理をおこなっている最中に PriceTable へ更新が生じた場合，その出力結果はどうすべきかというものが，彼らの提起する問題である．彼らはこれについて明確なセマンティクスを定めていない．

Arasu らの提起した問題は，状態の参照処理と状態の更新処理の分離性（Isolation）をどの単位で維持するかという問題と密接に関係する．これまでに開発されてきたストリームデータ

(注2)：“The effect of these semantics is equivalent to considering that the join of a stream tuple with the relation forms a transaction.” [5].
(注3)：タプルベースウィンドウとも呼ばれる．

処理システムの内いくつかでは，単一のイベント毎 [8]，同じ到着時刻を持つイベント群 [1]，同じウィンドウ内のイベント群 [9,13]，などの単位で分離性を指定することができる．しかしながら，こうしたシステムでは各オペレータに対して分離性を維持する単位が指定できるのみであり，本稿で扱ったような，最終的な出力結果に対する状態一貫性の保証はおこなえない．

Botan らは，分離性の問題について包括的な調査をおこない，ストリームデータに対するトランザクションモデルを提案した [6]．彼女らはストリームデータとリレーションを更新可能な情報源として等しく扱うことで，従来のトランザクション処理における技術を適用している．彼女らの提案したトランザクションモデルは状態一貫性の保証に適用することができ，本稿とは相互補完的な関係にあると考えられる．

異なるアプローチとして，Wan らはパターンの検出中に状態が参照でき，パターンの検出後に状態を更新できる新たな複合イベント処理モデルとして Active CEP (Complex Event Processing) という枠組みを提案し，その上にトランザクションモデルを定義している [19]．彼女らは分離性の維持単位として，複合イベント処理におけるパターンを用いた．これは，ある問合せがパターンを検出している間，その問合せが参照する状態を他の問合せが更新することはできないということを意味する．Wan らは Active CEP の持つ特徴に注目し，SS2PL と状態の多版化を用いた効率的な同時実行制御手法を提案している．

また，Golab らは，ストリームウェアハウスにおける実体化ビューに対し，時間的一貫性（temporal consistency）を定義している [10]．彼らの定義した一貫性は分離性によるものとは大きく異なり，本稿の対象とする問題とは区別される．

6. ま と め

本稿では，外部データを参照しながらおこなわれるストリームデータ処理において，外部データが処理中に更新される場合に生じる処理結果の非一貫性に着目した．そして，継続的クエリの処理結果について状態一貫性という概念を導入し，継続的クエリの処理結果に対して状態一貫性を保証するストリームデータ処理方式としてロック方式とスナップショットの二つを提案した．また，プロトタイプシステムによる評価実験をおこない，継続的クエリ処理のスループットと更新処理のスループットとはトレードオフの関係にあり，継続的クエリ処理のスループットが優先される状況においてはロック方式を，更新処理のスループットが優先される状況においてはスナップショット方式を用いた方がよいという知見を得た．

今後，次のようにして研究を発展させていく．まず，3.4 節で述べた継続的クエリ内にオーバーラッピングウィンドウが含まれる場合の状態一貫性保証方式について検討する．そして，3.1 節で説明した Pull 型の継続的クエリ処理方式だけでなく，より一般的な Push 型の継続的クエリ処理方式についても提案方式を適用する．その後，複数の継続的クエリが同時に動作する際の効率的な状態一貫性保証方式について考察し，より綿密な評価実験により提案方式のさらなる分析をおこなっていく．

謝　　辞

本研究の一部は，科学研究費補助金基盤研究 (A)(#21240005)，科学研究費補助金若手研究 (B)(#22700090)，独立行政法人情報通信研究機構 (NICT) 委託研究「新世代ネットワークを支えるネットワーク仮想化基盤技術の研究開発」による．

文　　献

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[2] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In G. Lausen and D. Suciu, editors, *DBPL*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford InfoLab, 2003.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[5] M. A. Bornea, A. Deligiannakis, Y. Kotidis, and V. Vassalos. Semi-streamed index join for near-real time execution of etl transformations. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE*, pages 159–170. IEEE Computer Society, 2011.

[6] I. Botan, P. F. Bijay, and D. K. N. Tatbul. Transactional stream processing. In *EDBT*, 2012.

[7] A. Chakraborty and A. Singh. A partition-based approach to support streaming updates over persistent data in an active datawarehouse. In *IPDPS*, pages 1–11. IEEE, 2009.

[8] Coral8. http://coral8.com/.

[9] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*. www.crdrdb.org, 2009.

[10] L. Golab and T. Johnson. Consistency in a stream warehouse. In *CIDR*, pages 114–122. www.crdrdb.org, 2011.

[11] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *SIGMOD Conference*, pages 847–854. ACM, 2009.

[12] Jubatus. http://jubat.us/.

[13] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 international conference on Management of data*, pages 1081–1092, New York, NY, USA, 2010. ACM.

[14] D. Maier, J. Li, P. A. Tucker, K. Tufte, and V. Papadimos. Semantics of data streams and operators. In T. Eiter and L. Libkin, editors, *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2005.

[15] M. A. Naeem, G. Dobbie, G. Weber, and S. Alam. R-meshjoin for near-real-time data warehousing. In I.-Y. Song and C. Ordonez, editors, *DOLAP*, pages 53–60. ACM, 2010.

[16] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell. Supporting streaming updates in an active data warehouse. In *ICDE*, pages 476–485. IEEE, 2007.

[17] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. Knowl. Data Eng.*, 20(7):976–991, 2008.

[18] N. Tatbul. Streaming data integration: Challenges and opportunities. In *ICDE Workshops*, pages 155–158. IEEE, 2010.

[19] D. Wang, E. A. Rundensteiner, and R. T. Ellison. Active complex event processing over event streams. *PVLDB*, 4(10):634–645, 2011.