オペレータスケジューリングへの制約付与による トランザクショナルデータストリーム処理の効率化

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 発展的なデータストリーム処理では，継続的に実行されるクエリからデータベースやクラス分類器をはじめとする様々な外部リソースが参照される．継続的クエリは長時間にわたり実行されるため，クエリの実行期間中に参照する外部リソースが他のシステムによって更新され，クエリの処理結果に不整合が生じうるという問題があった．この問題を解決するため，継続的クエリ内でのリソース参照を外部リソースに対する連続的な参照トランザクションとして実行するトランザクショナルデータストリーム処理が近年提案されている．トランザクショナルデータストリーム処理では処理結果の不整合を防ぐためにオペレータの再実行処理がおこなわれ，性能低下の要因となる．本稿では，オペレータの再実行回数がオペレータの実行順序に依存することを明らかにし，オペレータスケジューリングへ制約を付与することでオペレータの再実行回数を削減する方式を提案する．そして実験により，オペレータスケジューリングに提案する制約を付与した場合，オペレータの再実行回数が削減され性能向上がみられることを示す．

キーワード データストリーム処理，トランザクション，オペレータスケジューリング，同時実行制御，継続的クエリ，ウィンドウ演算

1. 序 論

リアルタイムデータの増加にともない，多量のデータストリームを DSMS（Data Stream Management System）によりオンライン処理するデータストリーム処理への注目が集まっている．近年のデータストリーム処理における動向の一つに，リレーショナルデータベースや機械学習モデルなどの外部リソースを参照した処理の隆盛がある．例えば，日立はデジタルサイネージをつかった広告配信システムにおいて，データストリームとしてセンサから流れてくる人々の移動・滞留情報をショッピングサイトにおける人々の購買履歴やアクセスログを用いて分析し広告配信に役立てるシステムを提案している(注1)．また，NTT と PFI により開発されているオンライン機械学習向け分散処理フレームワーク Jubatus では，機械学習モデルを参照してデータストリームの分類・回帰・推薦処理などをおこなうことができる(注2)．さらに，Twitter の開発する分散リアルタイム処理基盤 Storm では，Trident と呼ばれる機構により memcached や Cassandra をはじめとするデータストア内の外部リソースを参照した処理がおこなえる(注3)．

外部リソースを参照したデータストリーム処理においては，継続的に実行される継続的クエリ（Continuous Query）の処理中に外部リソースが更新された場合に不適切な処理結果やシステム障害が発生しうる．例えば，継続的クエリの処理中に外部リソースが更新される例として，図 1 のような状況を考える．図 1 において，DSMS は「データストリーム e_1, e_2, e_3, e_4 を外部

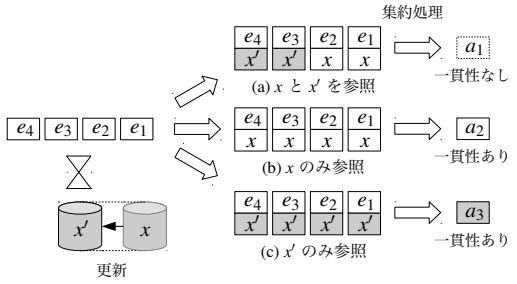


図 1 データストリームとリレーショナルデータの結合処理

のリレーショナルデータ x と結合し，その結果を四つ毎に集約する」という継続的クエリを実行している．このとき，DSMS が継続的クエリを実行している途中に，他のシステムによって x が更新され x' へと変化したとする．すると，更新処理のタイミングによって，DSMS が出力する継続的クエリの処理結果も変化する．図中，(a) は e_2 の到着後に x が更新された場合，(b) は e_4 の到着後に x が更新された場合，(c) は e_1 が到着する前に x が更新された場合，にそれぞれ対応するが，このとき (a) では最終的な集約結果 a_1 を出力する際に，その入力として更新前のリレーショナルデータ x と更新後のリレーショナルデータ x' が混在してしまっている．すなわち，(a) においては継続的クエリによるリレーショナルデータの参照処理が一貫していない．このような状況は，例 1.1 にみられるような不適切な処理結果やシステムの障害などの幅広い問題を引き起こす可能性がある．

例 1.1 (単位の不一致からくる誤った処理結果の発生) 図 *1* のなかで，x にヤード・ポンド法で記録された位置情報が含まれており，それらがメートル法へと変換されて x' になったとする．このとき，集約処理で四つの位置データの中心点を計算す

(注1)：http://www.hitachi.co.jp/Prod/comp/soft1/spcon/itbnavi_1105/
(注2)：http://jubat.us/
(注3)：http://engineering.twitter.com/2012/08/trident-high-level-abstraction-for.html

る処理がおこなわれた場合，*(a)* ではヤード・ポンド法で記録された位置情報とメートル法で記録された位置情報が混在しており，誤った中心点が算出されてしまう．

我々は，例 1.1 のような問題を防ぐための処理パラダイムとして，継続的クエリ内の外部リソース参照処理をトランザクションとみなし同時実行制御をおこなう**トランザクショナルデータストリーム処理**を提案してきた [15]．トランザクショナルデータストリーム処理は，処理結果一つ一つを出力する過程で一貫して外部リソースを参照できるよう，処理木を解析したうえで参照演算処理群を継続的クエリ由来トランザクションと呼ばれるトランザクションへと割り当てる．そして，継続的クエリ処理の中で同時実行制御をおこなうことで，それらのトランザクションと外部リソースを更新するトランザクションが直列可能に実行されることを保証し，例 1.1 のような問題を防ぐ．

トランザクショナルデータストリーム処理では，直列可能性を保証するために処理木内の特定オペレータ群からなる部分木を再実行することがあり，性能低下の要因となっていた．そこで，本稿では従来のトランザクショナルデータストリーム処理と同じ出力結果を保ったうえで，オペレータの再実行処理を減らすことで効率化することを目的とする．この実現のため，はじめにオペレータの再実行回数がオペレータの実行順序によって変化することを示したうえで，再実行回数の最小化に必要なルールをあきらかにする．そして，このルールに基づきオペレータのスケジューリングへ制約を付与することでオペレータの再実行回数が削減されることを示す．

本稿は次の構成をとる．まず，2. 節にてデータストリーム処理の基礎概念を説明し，3. 節にてトランザクショナルデータストリーム処理の考え方を示した後，4. 節でその同時実行制御について説明する．そして，5. 節で問題を定義したあと，6. 節で制約付与方式を提案し，7. 節でその評価をおこなう．その後，8. 節にて関連研究を紹介したあと，9. 節でまとめる．

2. 準　　備

本節では，データストリーム処理に関する基礎概念を説明する．

2.1 データストリーム

本稿では，センサデータ，ネットワークパケットなどの継続的に到着するデータを総称してデータストリームと呼び，これを次のようなタイムスタンプ付きタプルの集合として扱う [3]．

定義 2.1 (データストリーム)　データストリーム S は，e: $\langle s,t\rangle$ なるタプル e の集合である．ここで $s \in S_{schema}$ は S のスキーマ S_{schema} に属すタプルを，$t \in T$ は e のシステム到着時刻（タイムスタンプ）を意味する．また，T はシステムが扱う時刻のドメインとなる順序集合をあらわす．

2.2 DSMS

データストリームをオンライン処理するためのシステムとして，数々の *DSMS (Data Stream Mangement System)* が研究開発されている [2], [3], [9], [12]．これらのシステムは，問合せ言語 [3], [8] により記述した**継続的クエリ** *(Continuous Query)* が与えられると，それを内部表現に変換したうえでシステムの定めるセマンティクスに基づき継続的に実行する．

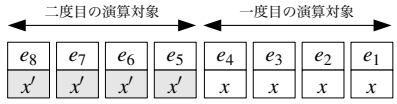


図 2　$r = s = 4$ のときのウィンドウ推移（演算対象に重複なし）

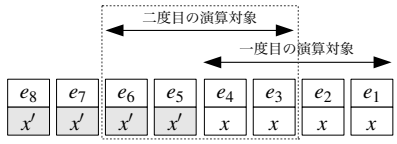


図 3　$r = 4, s = 2$ のときのウィンドウ推移（演算対象に重複あり）

2.3 処　理　木

本稿では継続的クエリの内部表現として，関係演算をノード，エッジをデータストリームとする**処理木**を用いる．処理木内の個々の関係演算を**オペレータ**と呼ぶ．オペレータは一つ以上の入力データストリームを持ち，それらに関係演算を適用したうえで，処理結果を自身の出力データストリームへと逐次出力する．この処理を各オペレータが繰り返すことにより，入力データストリームは処理木を伝搬し，継続的クエリの処理結果として継続的に出力される．

本稿では，データストリーム中の各タプルに対し，そのタプルの計算に用いられたタプル群を次のようにあらわす．

定義 2.2 (依存タプル群)　$\mathbb{S}$ を処理木内にあらわれる全てのデータストリームからなる集合としたとき，タプル $e \in S \in \mathbb{S}$ に対する依存タプル群 $DEP : S \to 2^{\mathbb{S}}$ は，以下のようになる．

（*1*）S があるオペレータの出力データストリームであるとき，そのオペレータの入力データストリーム群 $S_1^{in},...,S_n^{in}$ に対して，e が $S_1^{dep} \subset S_1^{in},...,S_n^{dep} \subset S_n^{in}$ を使って計算されたならば，$DEP(e) := \bigcup_{k=1}^{n} S_k^{dep}$ となる．

（*2*）S が処理木の葉オペレータに対する入力データストリームであるとき，$DEP(e) := \emptyset$ となる．

2.3.1 ウィンドウオペレータ

データストリームに対して，最大値・平均値・頻度分布の集計処理や結合演算など複数個のタプルが必要となる関係演算をおこなうオペレータを，ウィンドウオペレータと呼ぶ．ウィンドウオペレータは，理論上無限のタプルからなるデータストリームより**ウィンドウ**と呼ばれる有限個のタプルを切り取ったうえで処理をおこなう．ウィンドウは，一定数のタプルを演算対象とするタプルベースウィンドウと，一定時間内にシステムに到着したタプルを演算対象とするタイムベースウィンドウの二つに大別される．

ウィンドウオペレータがどのようにデータストリームからタプル群を切り出すかは，ウィンドウ幅 r とスライド幅 s という二つのパラメータによって決まる [3], [10], [14]．r は一度の演算で演算対象となるタプル群の範囲を，s はウィンドウ演算の開始位置の移動幅をあらわす(注4)．図 2 と図 3 に，ウィンドウオペレータによる演算対象の推移例を示す．

2.3.2 参照オペレータ

関係演算の中で外部リソースを参照するオペレータを**参照オペレータ** *(Reference Operator)* と呼ぶ．参照オペレータの例には，外部リレーションとデータストリームの結合処理や，線形

(注4)：タプルベースウィンドウならタプル数，タイムベースウィンドウなら時間により指定される．

分類器を用いたデータストリームの分類処理などがある．

本稿では，参照オペレータの処理結果タプルに対し，その処理に際しておこなわれた全ての参照処理を次のようにあらわす．

定義 2.3 (依存参照群) X を処理木内で参照されうる全ての外部リソースの集合としたとき，$e \in S \in \mathbb{S}$ に対する依存参照群 $REF : S \to 2^{X \times T}$ は，以下のようになる．

（1） S が参照オペレータの出力データストリームであるとき，e の処理に際して外部リソース $x_k \in X$ $(1 \leqq k \leqq n)$ が時刻 $t_k \in T$ で参照されたならば，$REF(e) := \{\langle x_1, t_1\rangle, ..., \langle x_n, t_n\rangle\}$ となる．

（2） それ以外のとき，$REF(e) := \emptyset$ となる．

3. 継続的クエリ由来トランザクション

本節では，継続的クエリ内の参照演算群をトランザクションとしてとらえる**継続的クエリ由来トランザクション**の考え方を説明し，コミット処理・中断処理などのセマンティクスを定める．

3.1 継続的クエリ由来トランザクションの導出

トランザクショナルデータストリーム処理では，継続的クエリの処理結果一つ一つがリソースを一貫して参照して処理されるたものとなるように，各処理結果に対しその計算過程でおこなわれた参照演算群を一つのトランザクションとして扱う．例えば，図 4(a) の処理木に五つのタプルが入力され，結合演算がリソース x を五回参照し，その参照結果が図 4(b) に示すようにウィンドウ演算により a_1, a_2 へ集約されたとする．トランザクショナルデータストリーム処理では a_1 と a_2 の入力で使われた参照処理群をそれぞれ一つのトランザクションとして考え，それらを継続的クエリ由来トランザクションと呼ぶ．

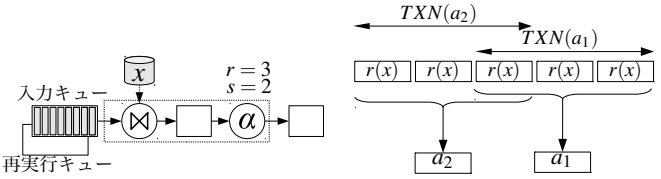
継続的クエリ由来トランザクションをより形式的にあらわすため，継続的クエリのある処理結果からその処理結果の計算過程でおこなわれた全ての参照処理群を導出する関数を次のように定義する．

定義 3.1 (トランザクション導出関数) S をデータストリーム，X を継続的クエリ内で処理されうる全ての外部リソースからなる集合としたとき，$e \in S$ に対するトランザクション導出関数 $TXN : S \to 2^{X \times T}$ は，次のように再帰的にあらわされる．

$$TXN(e) := \left(\bigcup_{e^{dep} \in DEP(e)} TXN(e^{dep}) \right) \bigcup REF(e)$$

トランザクション導出関数を用いると，継続的クエリの処理結果 e に対する継続的クエリトランザクションを $TXN(e)$ と記すことができる．また，以降で継続的クエリ由来トランザクション $TXN(e)$ の依存するタプル群という場合，これは e に対する依存タプル群である $DEP(e)$ を意味する．

継続的クエリ由来トランザクションの考え方を用いると，処理結果ストリームが $e_1, \ldots, e_n$ である継続的クエリの実行処理は，継続的クエリ由来トランザクション $TXN(e_1), \ldots, TXN(e_n)$ の実行処理を内包すると解釈できる．トランザクショナルデータストリーム処理ではこれらの継続的クエリ由来トランザクションとリソース更新トランザクションに対し同時実行制御をおこなうことで，処理結果毎にリソースを一貫して参照することを保証する．



(a) 参照処理結果をウィンドウ演算にかける処理木の例　(b) 図 4(a) の処理結果 a_1, a_2 に対する継続的クエリ由来トランザクション

図 4　継続的クエリ由来トランザクションの導出例

3.2 継続的クエリ由来トランザクションのコミット処理

トランザクショナルデータストリーム処理では，以下に定義を示すコミットオペレータによる処理結果 e の出力処理を，その処理結果に対する継続的クエリ由来トランザクション $TXN(e)$ のコミット処理とみなす．

定義 3.2 (コミットオペレータ) 処理木における最上位のウィンドウオペレータないし参照オペレータをコミットオペレータと呼ぶ．

コミットオペレータは，処理木における以降のオペレータの処理でそれ以上の参照処理がおこなわれない，すなわち継続的クエリ由来トランザクションにそれ以上の参照処理が含まれないことを保証する．

3.3 継続的クエリ由来トランザクションの中断処理

継続的クエリ由来トランザクション内でおこなわれる処理は参照演算に限定され，外部のリソースへ変更を及ぼすことがない．そのため，継続的クエリ由来トランザクション $TXN(e)$ の中断処理は，コミット処理のキャンセル，すなわち処理結果 e を出力しないことと定める．

3.4 継続的クエリ由来トランザクションの再実行処理

継続的クエリ由来トランザクション $TXN(e)$ の再実行処理を，$TXN(e)$ の依存するタプル群 $DEP(e)$ をつかって，もう一度 $TXN(e)$ の参照処理を実行しなおすことと定める．これは $DEP(e)$ を再び処理木の葉オペレータ側から流すことで実現できるが，このとき処理結果に重複が生じてしまったりタプルの順番が入れ替わってしまうことがないよう，注意しなければならない．以下で，こうした問題の生じない継続的クエリ由来トランザクションの再実行処理について述べる．

3.4.1 再実行キュー

継続的クエリ由来トランザクションの再実行に必要なタプルをバックアップするために，入力データストリームに対して**再実行キュー**と呼ばれるキューを配置し，新たなタプルがそのデータストリームに到着するたびに，再実行キューにもそのタプルを追加する．これにより，再実行時にキュー内から適切なタプル群を取り出し対応するデータストリームに流すことができる．再実行キューの配置について，我々は継続的クエリ由来トランザクション内の参照演算が正しく実行され，さらに再実行されるオペレータ数が最小となるような配置方式とその効率的な探索アルゴリズムを提案している [15]．その方式において，再実行キューは次の二種類のデータストリームに配置される．

（1） 下流にウィンドウオペレータを持ち，上流に参照オペレータを持たないような参照オペレータの入力データストリーム．

（2） 参照オペレータを上流に持つウィンドウオペレータの入力データストリームのうち，上流に参照オペレータを持たな

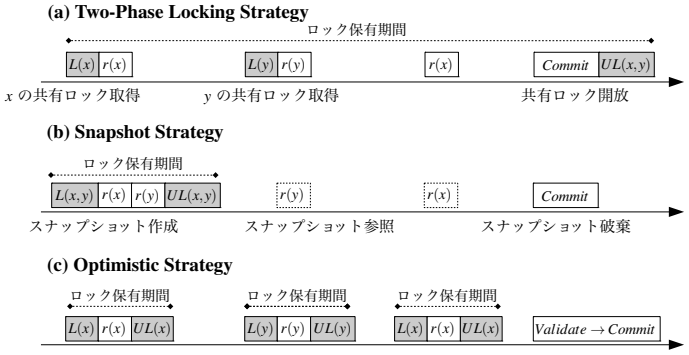


図 5　各制御方式における $r(x), r(y), r(x), commit$ の実行処理

いもの．

コミットオペレータを根とし，全入力データストリームに再実行キューが配置されたオペレータを葉とする部分木を**再実行領域** *(Redo Area)* と呼ぶ．例えば図 4(a) では集約演算 ⓐ がコミットオペレータとなり，破線で囲まれた領域が再実行領域になる．

3.4.2　再実行処理

再実行キューを用い，継続的クエリ由来トランザクション $TXN(e)$ の再実行処理は次のような流れでおこなわれる．まず，処理結果の重複を防ぐために再実行領域内のキューやオペレータの状態をリセットする．その後，再実行キューから継続的クエリ由来トランザクションの依存するタプル群 $DEP(e)$ を取り出し，対応するデータストリームの先頭に追加する．これにより $TXN(e)$ 実行前の状態が復元され，処理を再びおこなうことで再実行処理が実現される．

3.5　実システムにおける実行処理

本稿では，一つの処理木内では同時に一つの継続的クエリ由来トランザクション τ しか実行されないシステムを想定する．処理木内で参照演算をおこなう際には，(1) もし τ があればその参照演算を τ の一部として扱い，(2) もし τ がなければ新たな継続的クエリ由来トランザクションを開始し τ とする．コミットオペレータが結果を出力すると，τ は終了するため，処理木の実行処理内では「参照演算により継続的クエリ由来トランザクションが開始し，コミット処理よりその継続的クエリ由来トランザクションが終了する」という処理が繰り返されることとなる．このとき，本来 τ の一部ではない参照演算が τ の一部として実行され，その参照結果が使われず無駄になってしまうことがある．この問題については 5. 節で詳しく説明し，6. 節でその解決方式を提案する．

4.　同時実行制御に基づく継続的クエリ処理

本節では，3. 節で述べた継続的クエリ由来トランザクションとリソース更新トランザクションが直列可能となるような継続的クエリの実行方式について説明する．

4.1　悲観的方式

はじめに，悲観的同時実行制御に基づく継続的クエリ実行方式である**二相ロック方式** *(Two-Phase Locking Strategy)* と**スナップショット方式** *(Snapshot Strategy)* の二つについて説明する．

4.1.1　二相ロック方式

二相ロック方式は，参照オペレータにおける参照処理を**二相ロックプロトコル** *(Two-Phase Locking Protocol, 2PL)* [16] に基づいておこなう方式である．参照オペレータはリソース x を参照する前に，x に対する共有ロックを取得する．そして，参照処理終了後も取得したロックは手放さず，コミットオペレータが結果を出力したあと，取得していたロックを全て手放す．

図 5(a) に見られるように，二相ロック方式において継続的クエリ由来トランザクションは実行期間中リソースに対する共有ロックを保持し続ける．そのため，データストリームの到着レートが低く，タプルがウィンドウ内に蓄積される——すなわちトランザクションが終了するまでに長大な時間を要す——場合，二相ロック方式ではその間のリソース更新を阻害することとなってしまう．実システムによる実験では，ウィンドウの r が大きくなるにつれて，更新トランザクションのスループットが減少していくことが確認されている [15]．ゆえに，頻繁にリソースの更新が必要とされる状況に二相ロック方式は適さないといえる．

4.1.2　スナップショット方式

スナップショット方式は，ロックの保有期間を最小にすることをねらった方式であり，保守的二相ロックプロトコル (Conservative Two-Phase Locking Protocol, C2PL) に基づく．この方式では，継続的クエリ由来トランザクションの中で参照されるリソースが事前に全て判明しているという前提のもと，継続的クエリ由来トランザクション内で参照処理がおこなわれる前にそのトランザクション内で参照されうる全てのリソースに共有ロックを取得する．そして，全てのリソースを参照し参照結果をスナップショットとして DSMS にキャッシュしたうえで，ロックを開放する．そのトランザクション内で以後おこなわれる参照処理では，実際のリソースを参照するかわりにスナップショットを用いることで，一貫したリソースを参照することを保証する．

実システムにおける実験により，スナップショット方式は提案する継続的クエリ処理方式の中でもっとも更新トランザクションのスループットに与える影響が少ないことが判明している [15]．これは，図 5(b) に見られるようにスナップショット方式においてはロックがまとめて取得され，さらにその保有期間が短いためである．

4.1.3　重なりのあるウィンドウへの対応

重なりのあるウィンドウ演算では連続する演算間で同じタプルが用いられるため，前回用いられたタプルをオペレータにキャッシュしておくことで，無駄な再演算を防ぐことができる．このようにキャッシュされたタプル群は**シノプシス** *(Synopsis)* [2] と呼ばれ，多くの DSMS 実装で用いられている．しかし，シノプシスを用いた場合，上記の悲観的同時実行制御にもとづく二方式を用いた場合であっても，トランザクションが直列可能でなくなることがある．そのような状況の例を図 3 に示す．図 3 において，一度目の演算対象 $\langle e_1, x\rangle, ..., \langle e_4, x\rangle$ は同一のトランザクションに属すため，悲観的方式によって同じ x を参照すると保証される．しかし，そのトランザクションの終了後に更新トランザクションがリソース x を x' へと更新した場合，二つめのトランザクションではシノプシス内の $\langle e_3, x\rangle, \langle e_4, x\rangle$ とその後の参照結果 $\langle e_5, x'\rangle, \langle e_6, x'\rangle$ が混在することとなる．すなわち，トランザクション内でリソースが一貫して参照されておらず，直列可能ではなくなっている．

この問題を防ぐため，再実行領域内にウィンドウパラメータが $r > s$ となる——すなわちウィンドウ演算の重なる——オペレータが存在するとき，我々は 3.4 節で説明した継続的クエリ由来トランザクションの再実行処理に基づいた対応をとる．コミットオペレータでコミット処理がおこなわれると，再実行領域内のキューやシノプシスをリセットし，再実行キューから後続するトランザクションで使われるタプル群を取り出して対応するデータストリームに入力することで，後続するトランザクションをはじめからやりなおす．これにより，各トランザクションが一貫してリソースを参照することが保証される．

4.2 楽観的方式

次に，楽観的同時実行制御[13] に基づいた継続的クエリ処理方式である**楽観的方式** *(Optimistic Strategy)* について述べる．トランザクションが直列可能となるようにルールを定めて参照演算をおこなう悲観的同時実行制御とは異なり，この方式ではルールを定めずとも直列可能になるという楽観的な姿勢で参照演算をおこない，コミット時になって実際に直列可能かどうかを判定する．代表的な楽観的同時実行制御 [13] は (1) 参照処理，(2) コミット可否判定，(3) 書き込み反映処理，という三つのフェーズからなるが，継続的クエリ由来トランザクションは参照演算限定であるため，(3) を省くことができる．そのため，楽観的方式は以下に示す二つのフェーズで構成される．

（1） **参照フェーズ (Read phase)**: 参照オペレータは参照処理時に共有ロックを取得し，参照処理後に取得した共有ロックを開放する．

（2） **コミット可否判定フェーズ (Validate phase)**: コミットオペレータはコミットの準備ができると，それまでにおこなわれた参照処理をさかのぼって確認し，トランザクションが直列可能となるかどうかによってコミットの可否を判定する．コミット可能と判定された場合，コミットオペレータは処理結果を出力することでコミット処理をおこないトランザクションを終了する．コミット不可能と判定された場合，トランザクションは中断されたのちに再実行される．

本小節の残りでは，楽観的方式の実装にあたり必要となる (1) トランザクションが直列可能となるかどうか，すなわちコミットの可否をいかに判定するか，(2) いかにトランザクションの中断と再実行をおこなうか，について説明する．

4.2.1 コミット可否の判定

継続的クエリ由来トランザクションのコミット可否は，そのトランザクションの開始時刻 t_0 と各参照処理時点でのリソースの更新時刻 $t_1,...,t_n$ を比較することでおこなわれる(注5)．ここで，もし $t_k > t_0$ $(1 \leqq k \leqq n)$ なる k が存在すれば，その継続的クエリ由来トランザクションの途中で他の更新トランザクションがリソースを更新しているため継続的クエリ由来トランザクションはコミット不可能であると判定される．それ以外の場合，トランザクションは競合直列可能となるためコミット可能であると判定される．

4.2.2 トランザクションの中断と再実行

楽観的方式における継続的クエリ由来トランザクションの中断処理と再実行処理には，3.3 節と 3.4 節で述べた仕組みをそのまま用いる．

(注5)：本稿では，トランザクションが直列可能となるかどうかを判定するために，楽観的方式を適用する環境においては各リソースが最後に更新された時刻を持つと仮定する．

5. オペレータスケジューリング問題

これまでに見てきたように，同時実行制御に基づく継続的クエリ処理では必要に応じて継続的クエリ由来トランザクションの中断処理と再実行処理，すなわち処理木内の部分木の再実行処理がおこなわれる．本節では，再実行されるオペレータ数がオペレータの実行順序によって変化することを明らかにしたうえで，本稿で取り組む問題を定義する．

5.1 オペレータスケジューリング

はじめに，DSMS における処理木のオペレータスケジューリングについて簡単に説明する．本稿で扱う関係代数に基づいた継続的クエリ処理では，処理木の実行において各オペレータは自律的に動作でき，どのオペレータがどのような順番で実行されても最終的な処理結果は変化しない．一方，オペレータの実行順序の定め方によってメモリ使用量・スループット・レイテンシといった性能特性が変動することが知られており，特定のポリシーに基づいてオペレータの実行順序を定める種々のオペレータスケジューリング方式が提案されている [4], [7].

オペレータがプロセッサから実行権限を得て，入力データストリームよりタプルを一つ取り出し，処理結果を出力データストリームへ出力することを，オペレータの**一度の評価**と呼ぶ．また，オペレータの評価をおこなうことができるとき——すなわちオペレータの入力データストリームにタプルが一つ以上存在するとき——そのオペレータは**評価可能**であると表現する．また，本稿では処理木の実行処理を，単一のスレッドによる以下の 2 フェーズの繰り返しとする．

（1） **オペレータ選択フェーズ**: スケジューラが評価可能なオペレータを一つ選択する．

（2） **オペレータ評価フェーズ**: 選択されたオペレータが評価される．

このとき，選択されたオペレータの列を**スケジュール**と呼ぶ．

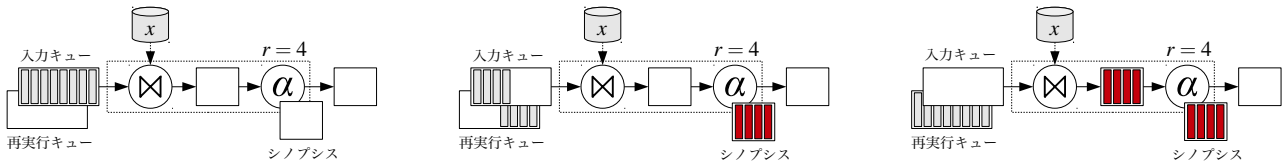
5.1.1 バッチスケジューリング方式

本稿では，あるオペレータが評価可能なうちは連続して何度もそのオペレータを評価するようなスケジューリング方式を**バッチスケジューリング方式**と呼ぶ．多くのスケジューリング方式 [4], [7] がバッチスケジューリング方式に分類される．バッチスケジューリングでは同じオペレータが連続して評価されるため，CPU キャッシュミスの低減によるパフォーマンスの向上がのぞめる．また，複数回の評価をまとめておこなうことにより，オペレータ毎の最適化も可能となる．例えば外部リレーションとの結合演算においては，外部リレーションのスキャン処理を一度にまとめることで，I/O コストを大幅に削減することができる．

5.2 オペレータ実行順序とリセットされるタプル数の関係

次に，具体例を用いてトランザクショナルデータストリーム処理におけるオペレータのスケジュールとリセットされるタプル数の関係について説明する．

図 6(a) は「外部リソース x とデータストリームの結合処理をおこなったあと，結果を 4 タプル毎に集約する」という処理木にタプルが 8 個入力されたときの内部状態をあらわしている．



(a) 8 個のタプルが入力されている処理木内部

(b) 結合演算 4 回 → 集約演算 4 回というスケジュール後の処理木内部（リセットされるタプル数 = 4）

(c) 結合演算 8 回 → 集約演算 4 回というスケジュール後の処理木内部（リセットされるタプル数 = 8）

図 6　オペレータ評価順序の違いからくるリセットされるタプル数の差

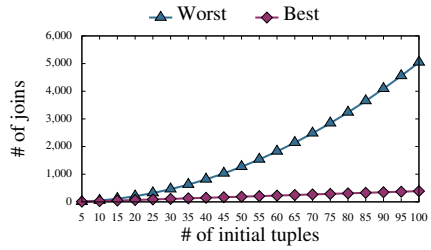


図 7　初期タプル数 n を変化させた際の $N_{\bowtie}^{best}$ と $N_{\bowtie}^{worst}$

図中，点線で囲まれた部分木が再実行領域に対応し，集約演算 α がコミットオペレータにあたる．また，結合演算 $\bowtie$ の結合選択率は 1 であり，タプルを一つ入力されると必ず処理結果タプルを一つ出力する．

ここで，図 6(a) の状態からコミットオペレータである集約演算 α にタプルを一つ出力させる——すなわち，コミット処理をおこなう——までのオペレータ評価順序を考える．α は幅 4 タプルのウィンドウ演算をおこなうため，コミット処理に至るまでには α が計 4 回評価される必要がある．また，α の入力が $\bowtie$ となっているため，α を n 回目に評価する際には，$\bowtie$ が n 回評価されている必要がある．

いま，これらの条件を満たすオペレータの評価順序例として，「結合演算 $\bowtie$ 4 回 → 集約演算 α 4 回」と「結合演算 $\bowtie$ 8 回 → 集約演算 α 4 回」という二つのスケジュールを考える．これらの順番でオペレータを評価した際の処理木内部状態を図 6(b) と図 6(c) に示す．ここで，4. 節で述べた同時実行制御に基づく継続的クエリ処理では，コミットオペレータのコミット時に再実行領域内のタプルがリセットされたのち再実行キュー内のタプルを使った再実行処理がおこなわれうる，ということを思い返す．すると，再実行時にリセットされるタプル数の観点からいえば，図 6(b) ではシノプシス内のタプル 4 個のみがリセットされるのに対し，図 6(c) ではシノプシス内のタプル 4 個に加えて結合演算 $\bowtie$ の出力結果 4 個がリセットされ無駄になっていることがわかる．どちらも得られる結果は同じであるため，リセットされるタプル数の少ない図 6(b) が望ましい．

5.3　スケジューリング方式の違いによる性能差

スケジューリング方式の違いによって生じる性能差を見るため，リセットされるタプル数が少ないほど良いという観点から，図 6(a) の処理木に対する最良・最悪なオペレータのスケジューリング方式を考える．最良のスケジューリング方式では集約演算 α が最低限必要とするタプルの数だけ結合演算 $\bowtie$ が評価されれば良く，この性質を持つスケジューリング方式としてアルゴリズム 1 が考えられる．一方，最悪なスケジューリング方式では集約演算 α が実行される時点で可能な限り結合演算 $\bowtie$ が評価されていれば良く，この性質を持つスケジューリング方式としてアルゴリズム 2 が考えられる．このとき，アルゴリズム 2 はバッチスケジューリング方式になっている．

Algorithm 1: 図 6(a) の処理木に対する最良のスケジューリング方式

```
1 while true do
2     ⋈ を選択し評価
3     α を選択し評価
```

Algorithm 2: 図 6(a) の処理木に対する最悪のスケジューリング方式

```
1 while true do
2     while ⋈ が評価可能 do
3         ⋈ を選択し評価
4     α を選択し評価
```

アルゴリズム 1 とアルゴリズム 2 の生成する最良・最悪なスケジュール内に結合演算がいくつ含まれるか，すなわち結合演算が評価される回数は，定量的に算出することができる．入力キュー内の初期タプル数を n，集約演算 α のウィンドウ幅とスライド幅をそれぞれ r,s としたとき，処理木が評価可能でなくなるまでに結合演算 $\bowtie$ が評価される回数を，最良のスケジュールについて $N_{\bowtie}^{best}$，最悪なスケジュールについて $N_{\bowtie}^{worst}$ と表記する．このとき，$N_{\bowtie}^{best}$ と $N_{\bowtie}^{worst}$ はそれぞれ式 1 と式 2 のようにあらわされる．

$$N_{\bowtie}^{best} = r(\lfloor \frac{n-r}{s} \rfloor + 1) \tag{1}$$

$$N_{\bowtie}^{worst} = \sum_{k=0}^{\lfloor \frac{n-r}{s} \rfloor} (n-ks) \tag{2}$$

式 1 と式 2 を用い，図 6(a) の処理木実行において最良のスケジュールと最悪のスケジュールを用いた場合の結合演算の評価回数を計算する．結合演算のウィンドウ幅 $r=4$，スライド幅 $s=1$ とし，入力キュー内の初期タプル数 n を 5 から 100 まで変化させたところ，最良・最悪なスケジュールにおける結合演算の評価回数は図 7 のようになった．図 7 は，同時実行制御に基づく継続的クエリ処理においてはスケジューリング方式の違いによって大きな性能差がつくことをあらわしている．また，実際のシステムにおける入力キュー内のタプル数はデータストリームの到着レートが高くなるほど増加するため，最良のスケジュールと最悪なスケジュールの性能差はデータストリームの到着レートが高くなるほど広がっていくと推測することができる．

5.4 問題定義

以上のことから，同時実行制御に基づく継続的クエリ処理においてはオペレータの実行順番がオペレータの評価回数に影響を与え，最良のスケジュールと最悪なスケジュールでは大きな性能差がつくことが分かった．そのため，トランザクショナルデータストリーム処理ではリセット処理を考慮したオペレータのスケジューリングが重要な課題であるといえる．これらを鑑み，本稿の残りでは次の問題に取り組む．

問題　コミットオペレータのコミット処理時に再実行領域がリセット・再評価されるという状況下で，処理木の全オペレータがそれ以上評価できなくなるまでのオペレータの合計評価回数を最小化する．

6. オペレータスケジューリングへの制約付与

処理木の全オペレータがそれ以上評価できなくなるまでのオペレータの合計評価回数を最小にするためには，各コミット処理時においてコミットオペレータが必要とするだけのタプルのみが再実行領域内に存在すると保証すればよい．ここで，再実行領域内にタプルが追加されるのは再実行キューの配置されたデータストリームを入力に持つオペレータが評価されるときのみであるという性質に注目し，これらのオペレータを**門番オペレータ** *(Gatekeeper Operator)* と呼ぶ．もし，再実行領域内の門番でないオペレータが全て評価不可能であるならば，門番オペレータを評価しない限りそれ以降の処理は進まない．反対に，再実行領域内の門番でないオペレータが一つでも評価可能であれば，コミットオペレータがコミットする可能性がある．この性質を考慮し，オペレータスケジューリングへ次の制約を与える．

制約　再実行領域内の門番でないオペレータが一つでも評価可能であるとき，門番オペレータを評価してはならない．

上記の制約をオペレータのスケジューリングへ付与することにより，処理木の全オペレータがそれ以上評価できなくなるまでのオペレータの合計評価回数を最小化することができる．

7. 実　　験

本節では提案する制約の有効性を確認するため，実際の DSMS のスケジューリングアルゴリズムに提案する制約を付与し，制約付与前との比較実験をおこなう．

7.1 実験環境

実験に用いた計算機は，プロセッサが Intel Core2 Quad Q9400 2.66 GHz，メモリが 4GB，OS が Linux 3.2.0 となっている．システムは同時実行制御に基づいた継続的クエリ処理が可能な DSMS 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．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．また，以下で述べる実験では，DSMS における継続的クエリ処理方式に二相ロック方式を採用した．

7.2 バッチ・ラウンドロビン型スケジューラへの制約付与

DSMS にバッチ・ラウンドロビン型スケジューラを実装し，制約の有無によりどれほど性能が変化するかを計測した．バッチ・ラウンドロビン型スケジューラは次の 2 フェーズを繰り返すことにより処理木の処理をおこなう．

（1） **オペレータ選択フェーズ**: 評価可能なオペレータのリストから，オペレータを一つラウンドロビン形式で選択する．

（2） **オペレータ評価フェーズ**: 選択されたオペレータを最大で n 回評価する．

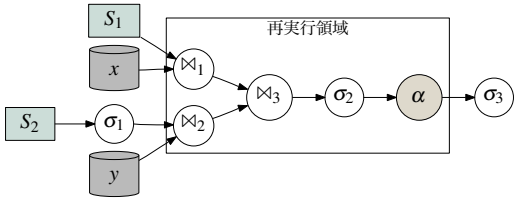
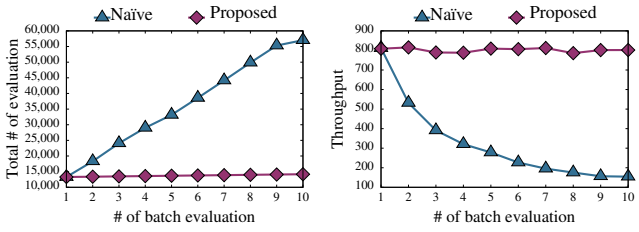
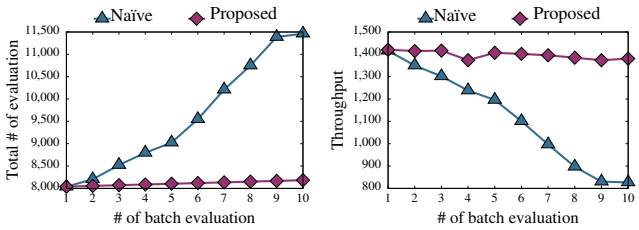


図 8　実験に用いた処理木



(a) オペレータ評価回数合計値　　(b) スループット

図 9　バッチ・ラウンドロビン型スケジューラにおいてバッチ処理サイズ n を変化させた際の性能変化（$\theta_{\bowtie_3} = 0.5$）



(a) オペレータ評価回数合計値　　(b) スループット

図 10　バッチ・ラウンドロビン型スケジューラにおいてバッチ処理サイズ n を変化させた際の性能変化（$\theta_{\bowtie_3} = 0.15$）

実験に用いた処理木を図 8 に示す．この処理木は，データストリーム S_1, S_2 をそれぞれ外部リソース x, y と結合したうえでウィンドウ結合演算によりそれらを併合し，条件を満たす併合結果を一定区間ごとに集約演算にかけて，一つの属性に対する平均値を算出したあと，その中から条件を満たすもののみを取得する．x, y はそれぞれ 1,000 のタプルからなるリレーションとし，結合はネステッドループ結合によりおこなう．$\bowtie_3$ を除くオペレータの選択率は，$\theta_{\bowtie_1} = 1/|x|, \theta_{\bowtie_2} = 1/|y|$, $\theta_{\sigma_1} = \theta_{\sigma_2} = \theta_{\sigma_3} = 0.5$ とした．また，ウィンドウパラメータは $\bowtie_3$ について $r = s = 10$，α について $r = 10, s = 1$ とした．

処理木の S_1, S_2 にそれぞれに 1,000 個のタプルを入力し，バッチ・ラウンドロビン型スケジューラにおいてバッチ処理可能な最大タプル数 n の値を 1〜10 まで変動させたときのオペレータの合計評価回数とスループットを計測した．ウィンドウ結合演算 $\bowtie_3$ の結合選択率 $\theta_{\bowtie_3} = 0.5$ のときの結果を図 9 に，$\theta_{\bowtie_3} = 0.15$ のときの結果を図 10 に，それぞれ示す．これらの図において，Naïve は制約を設けない通常のスケジューリング方式を，Proposed は提案する制約を設けたスケジューリング方式をそれぞれ意味する．

図 9 と図 10 から，提案する制約を付与することによりオペレータの評価回数が削減され，その結果スループットが大幅に向上していることが確認できる．制約を設けていない従来方式

ではバッチサイズの増加と共にオペレータの評価回数が増えてスループットが減少してしまっているのに対し，提案方式では性能の低下は見られない．制約の付与による性能向上はバッチサイズ $n = 10$ のとき最も大きく，$\theta_{\bowtie_3} = 0.5$ の場合にはスループットがおよそ 5.2 倍に，$\theta_{\bowtie_3} = 0.15$ の場合にはおよそ 1.7 倍になっている．$\theta_{\bowtie_3}$ が小さくなると性能向上も小さくなっているが，これは $\theta_{\bowtie_3}$ が小さくなることで再実行領域内に入るタプル数が減少し，オペレータの評価回数も同時に減少したためである．

7.3 考　　察

トランザクショナルデータストリーム処理においてバッチスケジューリングをおこなう場合，提案する制約を付与することでオペレータの評価回数が削減され，制約を付与しなかった場合と比べスループットに大きな差がつくことが分かった．一方で，バッチサイズを増やしていった際に期待される性能向上が見られていない．これは，処理木の実行時間の中で CPU キャッシュミスの占める割合が支配的ではなかったためであると考えられる．今後，バッチ処理におけるオペレータ固有の最適化処理を施し，バッチサイズを増加させることによる性能向上を示すことで，提案する制約の妥当性をさらに高める予定である．

8. 関 連 研 究

データストリーム処理におけるトランザクションについては，これまでにいくつかの考察がおこなわれている．Gurgen らは継続的クエリのパラメータが実行中に変更された場合に不適切な処理結果が発生しうることに注目し，継続的クエリを一定期間毎にネステッドトランザクションへと区切ることによって，各区間内で問題が発生しないことを保証した [11]．一定の期間内で分離性を保とうという彼らとは異なり，我々は処理結果毎に一貫してリソースを参照するように，分離性を保つ区間を決めている．また，Botan らは，データストリームが更新可能なリソースとしてみなせるとの考えに基づき，データストリームとリレーションの上に伝統的なトランザクションモデル [16] を再定義している [6]．本研究はデータストリームをリソースとみなしておらず，その点で彼らの研究と異なる．また，本研究が具体的な継続的クエリ処理モデルに対して同時実行制御方式を述べたのに対し，彼らは一般的なモデルの定義に主眼をおいている．

DSMS におけるオペレータのスケジューリング問題については，活発に研究がおこなわれてきた．初期の著名な DSMS である *Aurora* [1] ではオペレータのスケジューリングにおいて，接続関係にある複数のオペレータをグループとし，グループ毎にバッチスケジューリングをおこなうことによってスケジューリングのオーバーヘッドを低減している．この方式はバッチスケジューリングに基づくため，本稿で提案した制約付与が有効になると考えられる．Babcock らは，メモリ使用量を最小化するスケジュールの決定問題が NP 完全であることを示したうえでメモリ使用量を準最小化する *Chain* **スケジューラ**を提案している [4]．また，彼らの試みは処理木のメモリ使用量を見積もるために利用することもでき，クエリ最適化において最適な処理木を選択する際の一つの指標にメモリ使用量を用いることを可能とした．Bai らは Babcock らの提案した Chain スケジューラをレイテンシの最小化に応用する試みをおこなっている [5]．これらの方式ではバッチスケジューリングがおこなわれうるため，本稿で提案した制約付与が有効となる場合があると考えられる．

9. 結　　論

本稿では，トランザクショナルデータストリーム処理で処理結果の不整合を防ぐためにオペレータの再実行処理がおこなわれることに注目し，オペレータの再実行回数の最小化に取り組んだ．オペレータの再実行回数がオペレータの実行順序に依存することを明らかにしたうえで，その数を最小化する制約について述べ，評価実験によって提案する制約を付与した場合，付与しなかった場合と比べスループットが最大で 5.2 倍になることを確認した．今後は，制約についてのさらなる考察，バッチスケジューリング方式におけるオペレータ固有の最適化，再実行領域を考慮したクエリの最適化，などの課題に取り組んでいく予定である．

謝　　辞

本研究の一部は，科学研究費補助金基盤研究 (C)(#24500106)，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] D. J. Abadi *et al*. Aurora: A New Model and Architecture for Data Stream Management. *VLDB J.*, 12(2):120–139, 2003.

[2] A. Arasu *et al*. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[3] A. Arasu, S. Babu, J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB J.*, 15(2):121–142, 2006.

[4] B. Babcock *et al*. Operator Scheduling in Data Stream Systems. *VLDB J.*, 13(4):333–353, December 2004.

[5] Y. Bai, C. Zaniolo. Minimizing Latency and Memory in DSMS: A Unified Approach to Quasi-Optimal Scheduling. In *SSPS*, pages 58–67. ACM, 2008.

[6] I. Botan, P. F. Bijay, D. K. N. Tatbul. Transactional Stream Processing. In *EDBT*. ACM, 2012.

[7] D. Carney *et al*. Operator Scheduling in a Data Stream Manager. In *VLDB*, pages 838–849. VLDB Endowment, 2003.

[8] B. Gedik *et al*. SPADE: The System S Declarative Stream Processing Engine. In *SIGMOD*, pages 1123–1134. ACM, 2008.

[9] L. Golab *et al*. Stream Warehousing with DataDepot. In *SIGMOD*, pages 847–854. ACM, 2009.

[10] S. Guirguis *et al*. Three-level Processing of Multiple Aggregate Continuous Queries. In *ICDE*, pages 929–940. IEEE, 2012.

[11] L. Gürgen *et al*. Transactional Issues In Sensor Data Management. In *DMSN*, pages 27–32. ACM, 2006.

[12] S. Krishnamurthy *et al*. Continuous Analytics Over Discontinuous Streams. In *SIGMOD*, pages 1081–1092, 2010.

[13] H. T. Kung, J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[14] J. Li *et al*. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates Over Data Streams. *SIGMOD Record*, 34(1):39–44, 2005.

[15] M. Oyamada, H. Kawashima, H. Kitagawa. Continuous Query Processing with Concurrency Control: Reading Updatable Resources Consistently. In *SAC*, pages 788–794. ACM, 2013.

[16] G. Weikum, G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.