論文 / 著書情報
Article / Book Information

論題(和文)	低遅延ストリーム処理のための結合演算並列実行方式
Title(English)	Parallel Join Execution Methods for Low Delay Stream Processing
著者(和文)	渡辺陽介, 横田治夫
Authors(English)	Yousuke Watanabe, Haruo Yokota
出典(和文)	情報処理学会研究報告, Vol. 2012-DBS-154, No. 10, pp. 1-8
Citation(English)	IPSJ SIG Technical Report, Vol. 2012-DBS-154, No. 10, pp. 1-8
発行日 / Pub. date	2012, 8
権利情報 / Copyright	

Vol.2012-DBS-154 No.10
Vol.2012-IFAT-107 No.10
2012/8/2

低遅延ストリーム処理のための結合演算並列実行方式

渡辺 陽介[1,a)]　横田 治夫[1,b)]

概要：ストリームデータの増加に共に伴い，ストリームデータに対する問合せ処理が重要となってきている．情報源の種類も多様化し，複数の情報源から到着するストリームデータ同士の統合処理は特に需要が高くなっている．ストリームの統合を実現する演算はウインドウ結合と呼ばれており，時間幅や列数で定義されたスライディングウインドウにより処理対象を最新のデータに限定した結合処理を行う．結合処理を高速に実行するための既存のアプローチとして，分散環境を用いた処理の並列化が提案されており，スループットの改善が行われている．しかし，処理遅延の改善に関しては，分散環境がネットワーク遅延の影響を受けやすいということもあって，これまであまり考慮されてこなかった．本研究では，分散環境の代わりにマルチコア環境を用いて，複数ストリームに対するウインドウ結合演算の処理遅延を改善するような並列化方式を提案する．現在，N 個のストリームに対する結合を実現する方式は，2 入力の結合演算をツリー状に多段接続するものと，N 入力の結合演算を用いるものの 2 種類に大別される．本研究は，両方の方式に対応する並列結合ツリー方式と並列 MJoin 方式の 2 つを提案する．並列結合ツリー方式では，N 個の入力ストリームに対し，それぞれのストリームの出力を最速で出力することに特化したプランを N 個導出し，それらを N 並列に走らせるというものである．並列 MJoin 方式は，既存の MJoin のアルゴリズムがシーケンシャルに行っていたプローブ処理を N 並列化したものである．本研究では，提案手法 2 つとオリジナルの方式 2 つについて，評価実験を行いそれぞれの性能の比較を行う．

1. はじめに

近年，時間と共に次々と生み出されるストリーム型のデータに対して注目が集まっている．カメラ・センサーからの計測データや，マイクロブログ [1] からのつぶやきなど，ストリームデータの種類は豊富である．そのため，ストリームデータに対する選択・蓄積・統合・分析などのデータ処理の要望が高まっている．特に複数の情報源から到着するストリームデータ同士の統合は需要が高い．

ストレージに格納された RDB データなどの静的な情報とは異なり，次々と新しいデータが得られるストリームでは，従来型の DBMS によるデータ処理は十分なパフォーマンスが得られない．そこで，ストリームデータに対する連続的な問合せ処理を実現するストリーム処理エンジンの開発が行われてきている [2], [3], [4], [5], [6], [7]．ストリーム処理エンジンでは，問合せ処理要求として SQL ベースの問合せ文またはデータの流れを表すフロー等を登録すると，新規データが到着する度に対応する演算評価を繰り返し行い，処理結果を逐次生成する．ストリーム処理エンジンの扱うデータ処理演算のうち，複数の情報源のストリームデータ同士を統合する処理算はウインドウ結合 (Window-Join)[8] と呼ばれている．ウインドウ結合演算は，リレーショナル代数演算の中の結合演算の拡張であり，無限に到着し続けるストリームデータを扱うために，処理対象となるデータの範囲を制限するためのスライディングウインドウの概念が導入されたものである．

ストリーム処理エンジンは主記憶上でのデータ処理を前提としているため，DBMS 等よりもパフォーマンスは高かったが，ストリームデータのさらなる増加に伴い，より少ない遅延でより大量のデータを処理できる，効率的な問合せ処理技術が求められるようになってきた．問合せ処理効率化の一つの方向として，複数台のノードを用いた分散処理があり，これまで多くの研究において，ストリームに対する問合せ処理を分散化する手法が提案されている [9], [10], [11], [12], [13], [14].

これらの分散処理手法が重点を置いていることはスループットの向上，すなわちノードごとに異なる入力データを同時に処理させることで，システム全体として単位時間当たりに処理可能なデータ量を増加させることである．これまでの並列化手法はスループット向上については貢献してきたが，一方で，入力されたデータをより少ない遅延で出

[1] 東京工業大学
Tokyo Institute of Technology
[a)] watanabe@de.cs.titech.ac.jp
[b)] yokota@cs.titech.ac.jp

力することに関してはあまり考慮してきてはいない．処理遅延は，データが演算を通過する際に発生してしまう不可避のものであるが，応答性の高さを求められるアプリケーションにおいては非常に重要な要素である．しかし，分散環境ではノード間のデータ転送のための通信オーバヘッドが必要なために，処理遅延を削減する目的で並列化を行うことは難しかった．

分散化とは別の方向で，近年のマルチコア CPU の登場により，1 台のノードの中でもデータ処理を並列化することが容易になってきている．1 台に数十個のコアを有するノードも珍しくなくなった．マルチコア環境は，一般的な分散環境よりも並列化にかかるオーバヘッドがはるかに小さいため，従来の分散ストリーム処理とは異なる戦略が可能である．例えば，複数個の CPU コアをスループット向上だけでなく，処理遅延を削減する目的で用いることができる．このようなマルチコアを前提とした並列ストリームデータ処理については，まだ十分な検討が行われていない．

本研究は，マルチコア環境における処理遅延の削減が可能な並列ストリーム処理の実現を目標とし，特に本稿では，複数ストリーム上でのウインドウ結合演算を対象に，マルチコアを活かした並列処理方式を検討する．現在，N 個のストリームに対する結合を実現する方式は，2 入力の結合演算をツリー状に多段接続するものと，N 入力の結合演算を用いるものの 2 種類に大別される．本稿では，両方の方式に対応する並列結合ツリー方式と並列 MJoin 方式の 2 つを提案する．並列結合ツリー方式では，N 個の入力ストリームに対し，それぞれのストリームの出力を最速で出力することに特化したプランを N 個導出し，それらを N 並列に走らせるというものである．並列 MJoin 方式は，既存の MJoin [15] のアルゴリズムがシーケンシャルに行っていたプローブ処理を N 並列化したものである．また，本稿では，提案手法 2 つとオリジナルの方式 2 つについて，評価実験を行いそれぞれの性能の比較を行う．

本稿の構成は以下の通りである．まず，2 節において本研究の提案手法を理解するうえで必要なウインドウ結合について紹介する．次に 3 節において，提案手法である並列結合ツリー方式と並列 MJoin 方式を説明する．4 節では，本研究で行った評価実験について述べる．5 節で関連研究について紹介したのち，6 節にてまとめと今後の課題を述べる．

2. 前提知識: ウインドウ結合

ここでは，これまでに提案されてきている，ウインドウ結合について紹介する．最初にスライディングウインドウについて説明し，2 入力の結合アルゴリズムについて述べ，3 入力以上の結合方式について述べる．

2.1 スライディングウインドウ

本稿では，リレーションのタプルをベースにした，タプルストリームを前提とする．各タプル t には情報源からシステムに到着した時刻を表すタイムスタンプ $t.TS$ がつけられているものとする．本研究の目的は処理遅延の削減であるので，新規にタプルが到着した場合，可能な限り短い時間でそのタプルから生成できる処理結果を出力へ送るものとする．データ処理に関しては，本稿はウインドウ結合の処理遅延の削減が目的のため，ここでは結合演算のみを扱う．

ウインドウ結合演算は，リレーショナル代数演算の中の結合演算の拡張であり，無限に到着し続けるストリームデータを扱うために，処理対象となるデータの範囲を制限するための**スライディングウインドウ (以下ウインドウ)** の概念が導入されたものである．ウインドウは，時間幅で指定されるものと，列数で定義されるものの 2 種類がある．時間幅指定のウインドウでは，新規到着したタプルのタイムスタンプを基準に，w 単位時間 (e.g. w ミリ秒) 以内に届いた他の情報源のタプルを処理の対象とする．列数指定のウインドウでは，あるタプルが到着した時点で，最新の k 個のタプルが処理の対象となる．本稿では，時間幅のもののみを対象とし，情報源ごとに異なる時間幅のウインドウを指定可能とする．

各タプルは，ウインドウの範囲に含まれている間，システム内のメモリ領域に保持されていなければならない．本稿では，ストリーム S からの到着タプル t を一定期間保持するための領域を**ウインドウ領域** W_S と表記する．

2.2 2 入力のウインドウ結合

まず最初に 2 つのストリームを入力として扱うウインドウ結合の処理アルゴリズムについて紹介する [8]．

ここでは 2 つのストリーム S_1, S_2 に対するウインドウ結合 $S_1 \bowtie_{\theta,W_1,W_2} S_2$ を考える．ただし，θ は結合条件，W_1, W_2 はそれぞれ S_1, S_2 のウインドウ領域とする．

結合のアルゴリズムとしては，入れ子ループ結合 (Nested-loop join) ベースのものとハッシュ結合 (Hash join) ベースのものがある．ハッシュ結合は θ の条件が等結合 (=) の場合のみしか使えないが，大量データの処理に適している．本稿で述べる並列化手法はどちらにもアルゴリズムの場合も適用可能であるが，本稿の実験ではハッシュ結合の方を実装した．

2.2.1 入れ子ループ結合

$S_1 \bowtie_{\theta,W_1,W_2} S_2$ を入れ子ループ結合で処理する場合について説明する (図 1)．

入れ子ループ結合では，ストリーム S_1 からの新規到着タプル t_1 が与えられた際に，S_2 のウインドウ領域 W_2 内に含まれる全てのタプル $w_2 \in W_2$ とのマッチングが行われる．もし，マッチング候補 (t_1, w_2) が条件 θ を満たす場

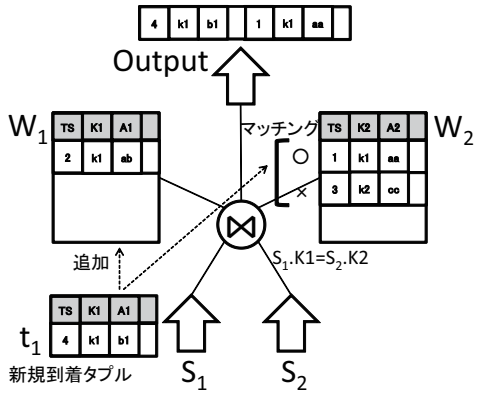

図 1 入れ子ループ結合

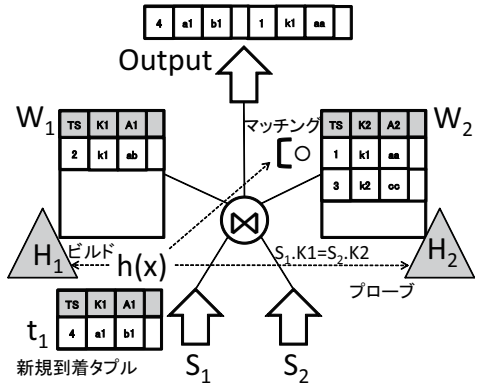

図 2 ハッシュ結合

合は，タプル $(t_1 \times w_2)$ が出力される．マッチング処理の終わった t_1 は後続のタプルの到着に備えて W_1 へ追加され，ウインドウの幅指定の条件を満たす間システム内に留まり続ける．S_2 からの新規到着タプル t_2 が与えられた場合も同様な処理が行われる．

2.2.2 ハッシュ結合

次に $S_1 \bowtie_{\theta,W_1,W_2} S_2$ をハッシュ結合で処理する場合について説明する (図 2)．入れ子ループ結合では，結合相手のウインドウ領域内に含まれる全てのタプルとのマッチングが必要だったが，ハッシュ結合ではウインドウ領域内のタプルに対するハッシュ表 H_S を用意することでマッチングのコストを削減する．

ハッシュ表は，結合条件 θ で結合キーに指定された属性に対し，ハッシュ関数 $h(x)$ を適用することで構築される．ストリーム S_1 からの新規到着タプル t_1 が与えられた際に，ハッシュ値 $h(t_1)$ を計算し，S_2 のハッシュ表 H_2 から $h(t_1) = h(w_2)$ となるタプル $w_2 \in W_2$ とのマッチングが行われる．マッチング処理の終わった t_1 は後続のタプルの到着に備えて W_1 へ追加され，さらに S_1 のハッシュ表 H_1 には t_1 の情報が追加される．

以降では，統合相手のストリームのハッシュ表を検索してマッチングを行う部分の処理を**プローブ**，自分のストリームのハッシュ表に新規到着タプルの情報を反映する処理を**ビルド**と呼ぶものとする．

2.3 N 入力のウインドウ結合

2.2 節は 2 つのストリームが入力として与えられた場合の話であるが，次に N 個のストリームが与えられた場合の結合処理について述べる．主なアプローチは 2 つあり，2 入力の結合演算をツリー状に多段に組み合わせて実現するものと，N 入力の結合演算として実現するものがある．

2.3.1 結合ツリー

この方式では，N 個の入力ストリーム $S_1, \ldots, S_N$ が与えられた場合，N-1 個の 2 入力のウインドウ結合演算を多段に組み合わせて目的の結合処理を実現する (図 3)．

ウインドウ結合の組み合わせ方は，**結合ツリー (Join tree)** と呼ばれる木構造で表現される．結合ツリーは，クエリの出力を根，各入力ストリームを葉とした 2 分木であり，枝分かれしている部分が 2.2 節で説明した 2 入力の結合演算に対応する．

結合ツリーの構造として left-deep ツリー, bushy ツリー等どの構造を選ぶか，また各ストリームをツリー内のどの位置に配置していくかについては，問合せ最適化技術 [16] の長年のテーマでもある．一般に，各ストリームの入力レート (#tuples/second) と各結合条件の選択率 (結合結果のタプル数/直積結果のタプル数) が完全に既知で，しかもそれらが時間変化しないという前提であれば，最適なツリーを決めることが可能である (近似解でない場合は入力ストリーム数 N に対して多項式時間では解けないが)．最適なツリーをどのように生成するかについては，本稿の範囲を超えるのでここでは触れない．

本研究の目的である，処理遅延の削減という観点からこのアプローチをみたときに重要なのは，ストリーム S_i のタプル t_i が届いた時に，S_i がツリー内のどの位置にあるかに応じて，出力までに通過しなければならない結合演算の数が変わってくるという点である．例えば，図 3 のような left deep な結合ツリーの場合，根から最も深い位置になったストリーム (e.g. S_1) のタプルの処理結果は N-1 個の演算を通過しなければ出力できないのに対して，根から最も浅い位置になったストリーム (e.g. S_N) のタプルの処理結果は 1 個の演算を通過するだけで出力にたどり着ける．ツリー内の位置に応じて処理遅延が変わるということが本研究が今回着目した性質であり，3 節の提案手法では，マルチコアによる並列化によりこの部分を補っている．

もう 1 点，次の 2 つ目のアプローチとの対比で重要な特徴がある．それは，1 つの結合演算の出力が次の結合演算の入力となり，次の結合演算のウインドウ領域にはその中間結果が保持されるという点である．これらの中間結果は別のタプルが到着した際の処理に再利用できるという利点があるが，中間結果のためのウインドウ領域やハッシュ表の管理も必要になるという欠点も同時に存在する．

2.3.2 MJoin

N 入力を扱うことが可能な結合演算 (MJoin [15]) のアプ

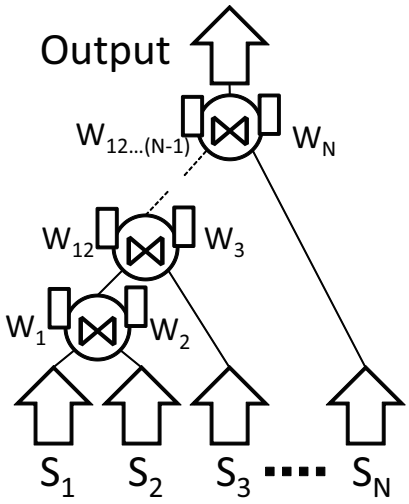

図 3 結合ツリー

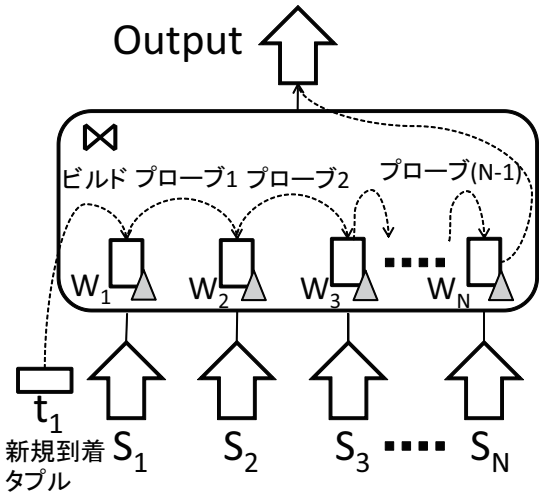

図 4 MJoin

ローチについて説明する (図 4).

MJoin では，N 個の入力ストリーム $S_1, \ldots, S_N$ に対応するウインドウ領域 $W_1, \ldots, W_N$ とハッシュ表 $H_1, \ldots, H_N$ を保持している．あるストリーム S_i から新規タプル t_i が到着した場合，t_i を W_i と H_i へ反映させるビルドの処理を行い，その後で他のストリーム $S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_N$ の各ハッシュ表を用いたプローブ処理を行う．1 つのハッシュ表のプローブ処理が終わったら，それを中間結果として次のハッシュ表のプローブ処理を行うということを N-1 回繰り返し，最後まで残ったものを出力する．

図 4 では左側からプローブ処理が行われているが，一般的にはプローブ処理を行うストリームの順番は，結合条件の選択率等に応じて中間結果がより少なくなるようなものが選ばれる．また，このプローブの順番は各タプルの属するストリームごとに決めることができ，選択率が変化した場合は動的に順番を変更してもよい．本研究の目的である，処理遅延の削減という観点からこのアプローチをみたときに重要なのは，プローブ処理をシーケンシャルに行っているという点である．この N-1 回のプローブ処理を並列化できれば処理遅延が減るのではないか，というところが 3 節で述べる 2 つ目の提案手法のアイデアである．

また，前述の結合ツリーを使うアプローチとの対比で重要な特徴は，MJoin では中間結果が保持されないということである．必要なウインドウ領域は少なくて済む半面，新規データが届くたびに N-1 回のプローブ処理が必要になる．なお，MJoin の改良手法の一つとして中間結果をキャッシュするものも提案されている [17] が，本研究ではキャッシュについては考慮しない．

3. 提案内容: 並列ウインドウ結合

本節では，本研究で提案する処理遅延を削減するための並列結合演算処理方式 2 種類について述べる．2 節で述べた通り，N 入力のストリームを統合する方式は，結合ツリーを用いるものと N 入力の結合演算を用いるものがある．ここではそれぞれについてマルチコア向けの並列化を行った方式を 1 つずつ提案する．

3.1 並列結合ツリー方式

2.3.1 で述べた通り，結合ツリーを用いて 2 入力のウインドウ結合演算をつなぎ合わせた場合，どのようなツリーの構造になるにしろ，ツリーの根から浅いところのストリームと深いところのストリームでは処理遅延に差が生じてしまう．しかし，逆に言えば処理遅延を減らしたいストリームをあえてツリーの浅いところに配置すれば，そのストリームからのタプルについては短い処理遅延で結合処理が可能になるということである．

ここでマルチコアの特性を生かして，複数の異なる結合ツリーを並列に実行することを考える．このとき，N 個のストリームに対して N 個のツリーを用意し，i 番目のツリーではストリーム S_i が根の最も浅い箇所で結合処理が行われるようにする．すなわち，以下のような N 個の結合ツリーができるようにする (図 5).

$$
\begin{aligned}
&S_1 \bowtie (S_2 \bowtie, \ldots, \bowtie S_N) \\
&S_i \bowtie (S_1 \bowtie, \ldots, \bowtie S_{i-1} \bowtie S_{i+1} \bowtie, \ldots, \bowtie S_N) \\
&S_N \bowtie (S_1 \bowtie, \ldots, \bowtie S_{N-1})
\end{aligned} \tag{1}
$$

根に一番近い場所の結合演算に配置するストリームだけ上記方針で決定し，残りの部分は従来通りの問合せ最適化手法を用いて最適な木の構造を決定する．このようにすることで，i 番目の結合ツリーは各ストリーム S_i の処理遅延の最小化に特化したものとなる．

また，N 個のツリーからの出力をすべて受け取ってしまうと処理結果に重複が発生してしまうので，ストリーム S_i のタプル t_i が到着した場合には，i 番目の結合ツリーからの処理結果だけを受け取るようにする．i 番目以外の結合ツリーは結合処理はするが処理結果を最後の出力へは送ら

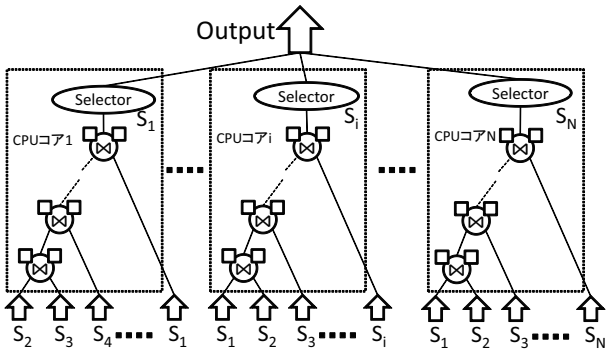

図 5 並列結合ツリー方式

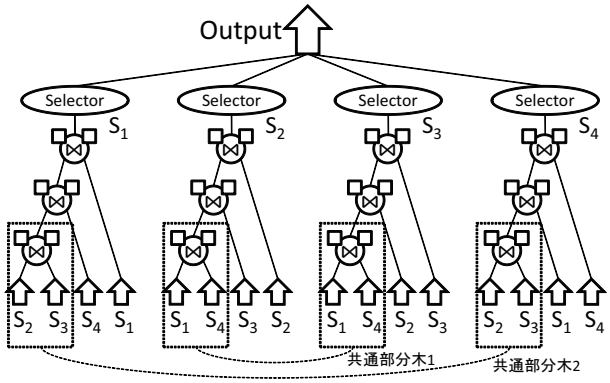

図 6 並列結合ツリー方式における共通部分木の共有

ずに破棄する．図 5 における Selector 演算がその役割を行う．このようにしておくことで，どのストリームのタプルが到着しても最も少ない処理を行うツリーから処理結果を受け取ることができるため，処理遅延を最小化できる．

3.1.1 メモリ使用量の削減について

N 個の結合ツリーを完全に独立で実行することも可能ではあるが，実際にはウインドウ領域などに必要なメモリ使用量などが N 倍になってしまい，都合が悪い場合も考えられる．そのような場合は，N 個の結合ツリーの間での共通部分木の共有化を行う．最も浅い結合演算以外の部分は，N 個のツリーの間で共通部分が存在するため，重複した中間結果を作らずに済ますことが可能である．

図 6 は，4 つの入力ストリーム S_1,S_2,S_3,S_4 に対して，並列結合ツリー方式を適用した場合である．ここでは $S_2 \bowtie S_3$ と $S_1 \bowtie S_4$ が共通部分木として 2 回ずつ出現しており，これらは共有化が可能である．

3.2 並列 MJoin 方式

ここではマルチコア環境での並列実行により，MJoin の遅延時間を削減することを考える．2.3.2 で述べた通り，MJoin では N-1 回のプローブ処理を順番に行っていた．これを N 個のコアを用いて 1 回のビルド処理と N-1 回プローブ処理を並列に実行し，最後に一か所に処理結果を集めて統合するというアプローチを考える．理想的にはプローブ処理 1 回分の時間ですべてのプローブ処理が完了でき，処

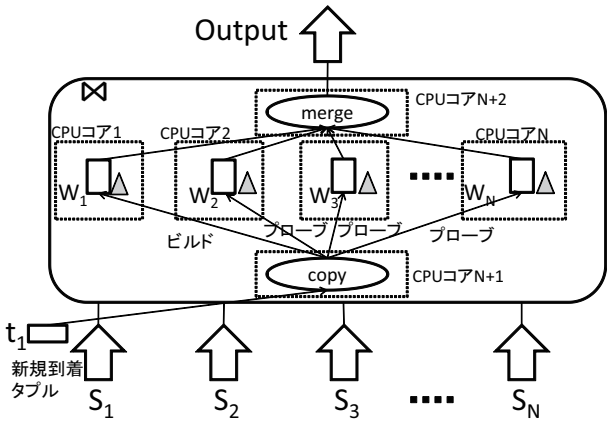

図 7 並列 MJoin 結合

理遅延を大幅に減らすことができる．並列 MJoin 方式はすべての結合要求に対して適用できるわけではないが，制限については 3.2.1 節で述べる．

並列 MJoin 方式の処理手順について述べる (図 7)．ストリーム S_i からタプル t_i が新規到着した場合，まず t_i を N 個のスレッドの処理キューへ追加する (Copy 処理)．各スレッドでは以下の処理を実行する．

- ストリーム S_i のウインドウ領域とハッシュ表に対するビルド処理 (1 スレッド)．ビルドが終わったら，t_i 単体からなる集合 M_i を返す．
- S_i 以外のストリーム S_j $(i \neq j)$ のハッシュ表とウインドウ領域に対するプローブ処理 (N-1 スレッド)．ただし，プローブ処理の結果として，通常の統合結果 $(t_i \times w_j)$ ではなく，t_i とマッチした w_j の集合を返す．すなわち半結合 (semi-join) として動作する．処理結果は M_j と表記する．

N 並列の実行の結果，N 個のタプル集合 $M_1, \ldots, M_N$ が得られる．最後にこれらを統合 $(M_1 \times \ldots \times M_N)$ して結合演算の処理結果の生成を行う (Merge 処理)．

並列 MJoin では，入力ストリーム数 N に対して，ビルド処理に 1 スレッド，プローブ処理に N-1 スレッド，Copy 処理に 1 スレッド，Merge 処理に 1 スレッドが必要なため，合計 N+2 スレッドを同時実行できるだけのコア数が必要となる．

3.2.1 並列 MJoin 方式の制限事項

ここでは並列 MJoin 方式が適用可能なクエリについて述べる．

並列 MJoin 手法が適用できるクエリの前提として，すべてのストリームの間に結合条件があり，かつ，それらは同じ結合キーを用いていることが必要となる．すなわち以下のような結合条件のときとなる．

$$S_1.k_1 = S_2.k_2 = \ldots = S_N.k_N \quad (2)$$

ただし，クエリ内で明示的に記述されている結合条件だけでなく，推移規則を使って導出可能なものも含めたときに，

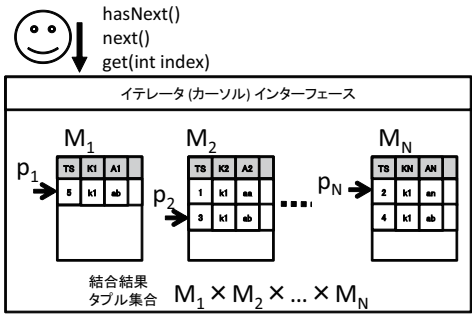

図 8 マージ結果タプル集合の実装

上記が成り立っていればよい．

それでも，ユーザから与えられる結合要求が，そのまま上記の制約を満すことを期待するのは難しい．しかし，制約を満たさないストリームを一部除いた部分集合において，上記制約を満たすことは十分にありえる．そのようなクエリを含む場合の一般的な処理手順は以下のようになる．

(1) まず与えられた入力ストリームを分析し，上記制約を満たすストリームの集合とそれ以外のストリームの集合に分ける．
(2) 制約を満たすストリーム集合のプローブ処理だけを並列 MJoin 方式によって並列実行する．
(3) 並列 MJoin の結果を使って，残りのストリームへのプローブ処理を従来の MJoin 同様に逐次実行する．

3.2.2 Merge 処理の効率化

最後の Merge 処理は，論理的には N 個のビルド・プローブの結果のタプル集合の直積 $M_1 \times \ldots \times M_N$ を生成する作業に相当するが，実装上はこれをマテリアライズするコストは高い．そこで，実装上の工夫として，直積結果を実際には生成せずに，内部で N 個のタプル集合のまま保持し，N 個のポインタを動かして，外側からは見かけ上直積結果があるように振る舞わせるようにした (図 8)．

直積結果に対する操作は，イテレータを用いて間接的に行わせるようにしており，イテレータの next() メソッドで次のタプルのデータが要求された場合，内部のポインタを移動させて，次の直積結果のタプルがあるように見せかける．

4. 評価実験

提案手法の有効性を検証するために評価実験を行った．実験環境は表 1 の通りであり，48 コアを持つマシンを用いた．

実験データは，ストリームデータ，クエリともに人工的に生成したものを用いた．ストリームデータは 10 個の数値属性からなるタプル列で，入力レートは各ストリームともに毎秒 20 タプル (到着間隔 50ms) とした．結合条件のためのキーとなる属性の値は，ストリームごとにカウンタを用いて 1 ずつカウントアップして値を生成し，それ以外の属性の値は乱数を用いた．実験に用いたクエリは N 個のストリームに対する結合処理である．N は 5, 10, 15, 20 とした．ウインドウの時間指定は全ストリームとも同じ長さとし，今回は 1〜5 秒の長さのウインドウを用いた．結合条件は 3.2.1 節の制約を満たすものを用いた．

表 1 実験環境

CPU	AMD Opteron 6174 (2.2GHz, 12 コア) x 4
RAM	32GB
OS	Linux Kernel 2.6.35
Java	JDK(Oracle) 1.7.0_5

今回はストリーム処理エンジンを用いずに，結合処理だけを実行可能な独立のものを Java によって開発した．評価対象は以下の 4 つである．

- 結合ツリー方式: 2.3.1 節で述べた方式で，シングルスレッドで動作する．ツリーの生成は，クエリに記述されたストリームの出現順序のまま left deep ツリーを生成する．今回はストリームごとに入力レートや選択率の違いがないので，結合順序による差はほぼ生じない．
- 並列結合ツリー方式: 3.1 節で述べた提案方式で，ストリーム数 N に対して N 個の結合ツリーを作り，N スレッドで動作する．今回の実装では共通部分木の共有化は行っていない．
- MJoin 方式: 2.3.2 節で述べた方式で，シングルスレッドで動作する．今回の実装では，プローブ処理の順番はクエリで与えられたときのストリームの順番のまま行うものとした．
- 並列 MJoin 方式: 3.2 節で述べた提案方式で，N+2 スレッド (ストリーム数 N+copy+merge) で動作する．

4 つともハッシュ結合をベースに実装されている．

処理遅延の計測は，タプルが生成されてから結合処理を終えて出力結果として出てくるまでの時間とした．タプルを生成する専用のスレッドを用意し，タプルを生成した時刻をそのタプルのタイムスタンプとして登録する．結合結果を受け取る側も別スレッドで動作し，受け取った時刻とタプル内のタイムスタンプを比べて，処理遅延を計算する．現在時刻の取得には Java の System.nanoTime() メソッドを使用した．今回はデータを 20 秒間流し続けたときに，各処理結果生成時に発生した処理遅延の平均を測定した．

4.1 実験結果

ウインドウサイズを 1 秒に固定し，入力ストリームの数を 5, 10, 15, 20 と変えた時の処理遅延の平均値を図 9 に示す．横軸は入力ストリーム数，縦軸は処理遅延 (ナノ秒) である．並列結合ツリー方式は，入力ストリーム数 5 から 15 までは最も遅延が少なくなった．並列 MJoin 方式は，今回の実験では残念ながらどの場合でも最も遅い結果となった．これらの原因については次節で考察する．

次に，入力ストリームの数を 10 に固定し，ウインドウ

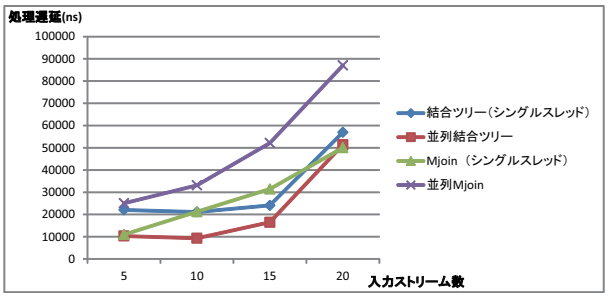

図 9 入力ストリーム数に対する処理遅延

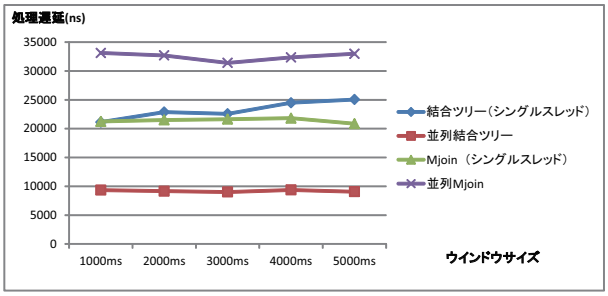

図 10 ウインドウサイズに対する処理遅延

サイズを 1〜5 秒に変えた時の処理遅延の平均値を図 9 に示す．ただし，今回の実験ではウインドウサイズを変えたことによる影響はほぼ見られなかった．これはハッシュ結合の場合，ウインドウに含まれるすべてのタプルとマッチングを行わず，ハッシュ値が一致するものとだけマッチングが行われるため，今回のキー値の設定ではウインドウサイズ設定の影響を受けなかったための考えられる．

4.2 考察

並列結合ツリー方式は，図 9 でストリーム数 5 から 15 までは最も遅延が少ないが，20 の時にシングルスレッドの MJoin に追い抜かれ，シングルスレッドの結合ツリー方式にも追いつかれている．これは並列結合ツリー方式の場合，同じクエリを N 個同時に処理することとほぼ同じリソースを消費するため，他の手法に比べて並列度に応じて動的にオブジェクトを生成・破棄する回数が増え，ガベージコレクションにかかるコストが増加したことが原因と考えられる．改善策の一つとして，3.1.1 節で述べた共通部分木の共有機能を実装することが挙げられる．

並列 MJoin 方式は，今回の実験では残念ながら最も遅い結果となった．原因としては，データの受け渡しに用いているキューの管理方式の問題点が考えられる．今回の実装では，copy 処理のスレッドから各プローブ処理のスレッドへのデータの受け渡しと，各プローブ処理のスレッドから merge 処理のスレッドへのデータの受け渡しに 2N 個のキューを用いている．今回はこれらのキューに新しいデータが追加されていないかを監視するアルゴリズムにラウンドロビン方式を採用したが，しかしこの監視方法では，キューの数が増えるほどキューの中身を確認する順番が回ってくる周期が長くなってしまうため，処理遅延の増加につながってしまったと考えられる．この部分の実装改善が今後の課題であることが分かった．

5. 関連研究

多入力のウインドウ結合についてはいくつかの改良手法が提案されている．Adaptive Caching [17] は，MJoin を改良するための手法の一つで，MJoin のプローブ処理の過程で生成された中間結果をキャッシュする．GrubJoin [18] も多入力を扱う結合演算の 1 つであり，特に高負荷時に必要性の低いデータの処理を間引く Load shedding のアイデアが入っている．これらの改良された手法は，本研究の提案手法と対立するものではなく，組み合わせて使用することも可能である．

分散ストリーム環境の前提にした研究も数多くされている．MJoin を分散環境で実行するものとして，PMJoin[19] が提案されている．PMJoin では，各ストリームのウインドウとハッシュ表を別々のノードで管理しており，到着タプルがプローブ処理のために各ノードを巡回する順序を，タプルのハッシュ値ごとに別々に設定することができる．この研究は，不要な中間結果の削減によるスループットの向上を目的としているが，分散環境ではノードを巡回する際にネットワーク通信コストが発生してしまうため，処理遅延の削減の目的にはあまり適していない．

Chase execution[20] は，Active stanby 方式で冗長化された分散ストリーム環境において同じクエリを異なる配置プランで複数実行し，より早く生成された方の処理結果を採用するという実行方式である．本研究の並列結合ツリー方式も Chase Execution と同じ発想に基づいているが，本研究はマルチコア環境を前提としており，処理の冗長化は考慮されない代わりに，複数プランの間での共有化などが可能な点が異なっている．

我々の既存研究では，マルチコア環境を想定した低遅延ストリーム処理方式 [21] を提案しているが，この中は結合演算は並列実行の対象とはなっていない．

6. おわりに

本研究では，分散環境の代わりにマルチコア環境を用いて，複数ストリームに対するウインドウ結合演算の処理遅延を改善するような並列化方式を 2 つ提案した．評価実験により，並列結合ツリー方式については処理遅延の削減を確認できたが，並列 MJoin 方式は逆に最も遅くなるという結果になった．ただし，両手法ともまだ実装の問題で理論上の性能を出し切れていない部分があり，それについては今後の課題となった．またより多様な状況下での評価実験は必要であり，それについても今後の課題である．

謝辞 本研究の一部は日本学術振興会科学研究費補助金基盤研究 A（#22240005），若手研究 B（#24700087）の助

成により行われた.

参考文献

[1] Twitter, http://twitter.com/.

[2] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[3] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[4] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[5] Mohamed H. Ali, Ciprian Gerea, Balan Sethu Raman, Beysim Sezgin, Tiho Tarnavski, Tomer Verona, Ping Wang, Peter Zabback, Anton Kirilov, Asvin Ananthanarayan, Ming Lu, Alex Raizman, Ramkumar Krishnan, Roman Schindlauer, Torsten Grabs, Sharon Bjeletich, Badrish Chandramouli, Jonathan Goldstein, Sudin Bhat, Ying Li, Vincenzo Di Nicola, Xianfang Wang, David Maier, Ivo Santos, Olivier Nano, and Stephan Grell. Microsoft cep server and online behavioral targeting. *PVLDB*, 2(2):1558–1561, 2009.

[6] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In Wei Fan, Wynne Hsu, Geoffrey I. Webb, Bing Liu, Chengqi Zhang, Dimitrios Gunopulos, and Xindong Wu, editors, *ICDM Workshops*, pages 170–177. IEEE Computer Society, 2010.

[7] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In Jason Tsong-Li Wang, editor, *SIGMOD Conference*, pages 1123–1134. ACM, 2008.

[8] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[9] Ying Xing, Jeong-Hyon Hwang, Ugur Çetintemel, and Stanley B. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.

[10] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802. IEEE Computer Society, 2005.

[11] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 25–36. IEEE Computer Society, 2003.

[12] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, pages 313–328. USENIX Association, 2010.

[13] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE*, page 49. IEEE Computer Society, 2006.

[14] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter Pietzuch. Sqpr: Stream query planning with reuse. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE*, pages 840–851. IEEE Computer Society, 2011.

[15] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[16] Ahmed Ayad and Jeffrey F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *SIGMOD Conference*, pages 419–430. ACM, 2004.

[17] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.

[18] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding. *IEEE Trans. Knowl. Data Eng.*, 19(10):1363–1380, 2007.

[19] Yongluan Zhou, Ying Yan, Feng Yu, and Aoying Zhou. Pmjoin: Optimizing distributed multi-way stream joins by stream partitioning. In *DASFAA*, pages 325–341, 2006.

[20] 上田高徳, 打田研二, 秋岡明香, and 山名早人.
データストリーム処理におけるレイテンシ削減と高可用性のためのオペレータ実行方法. 日本データベース学会論文誌, 10(3):1–6, 2012.

[21] 渡辺陽介 and 横田治夫.
マルチコア環境における可換演算群の
並列評価による低遅延ストリーム処理方式. In WebDB フォーラム, 2011.