書込み処理の遅延化と動的制御による ストリームデータの永続化

阿部 泰芽† 川島 英之†‡ 北川 博之†‡

†筑波大学大学院システム情報工学研究科 〒305-8573 茨城県つくば市天王台 1-1-1
‡筑波大学計算科学研究センター 〒305-8573 茨城県つくば市天王台 1-1-1
E-mail: †abe@kde.cs.tsukuba.ac.jp, ‡{kawasima, kitagawa}@cs.tsukuba.co.jp

あらまし 現在，各種センサや GPS，ネットワークカメラなどのセンシングデバイスの発展により，デバイスから自律的に発信されるストリームデータが増加してきている．ストリームデータはリアルタイムに処理されるだけでなく，処理結果のロギングや事後解析の為に永続化される．全て主記憶上で処理されるストリームデータ処理に比べ，ディスクアクセスの伴う永続化処理は低速であるため，ストリームデータの入力レートが高い場合や多数の永続化要求が存在する場合，ストリームデータの入力レートに追随してそれを永続化することは困難である．そこで我々は先行研究において，処理木を最適化し中間データを中間領域へ永続化することで効率良く永続化を行う手法を提案した．この手法は 1 台のマシン環境を想定していたため，よりストリームデータの入力レートが高い場合などには負荷が増大し，永続化処理が実行できなくなる問題があった．そこで本論文では，複数台のマシンを用いて中間領域を分散させることで，さらに効率よく永続化を行う手法を提案する．また，実験により提案手法の有効性を示す．

キーワード ストリームデータ処理，永続化，分散処理

An Efficient Data Archiving Method by Dynamic Thread Control

Taiga ABE† Hideyuki KAWASHIMA†‡ and Hiroyuki KITAGAWA†‡

† Graduate School of Systems and Information Engineering, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573 Japan
‡ Center for Computer Sciences, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573 Japan
E-mail: †abe@kde.cs.tsukuba.ac.jp, ‡{kawasima, kitagawa}@cs.tsukuba.ac.jp

1. はじめに

近年，センシングデバイスの技術と，ネットワークの技術の発展に伴い，デバイスから自立的・かつ継続的に発信されるストリームデータが増加してきている．例えば，温度や湿度,照度などを計測し，ネットワークを通じて配信するセンサデバイスや，ライブ放送に利用されるネットワークカメラ，データ放送やニュースなどの情報配信サービスが挙げられる．これらのストリームデータの処理基盤としてデータストリーム管理システム(DSMS : Data Stream Management System)が近年盛んに研究されている[6]．DSMS は要求を内部に登録し，ストリームデータが到着するたびにその要求を評価し，前回の処理結果との差分を出力する．また，全ての問合せ処理を主記憶上で処理を行うため高速な処理が可能である．

DSMS では主記憶領域確保のため，処理したデータは主記憶から削除される．そのため，ストリームデータのロギングや事後解析を行うためには，データベース管理システム(DBMS)などと連携し永続的デバイスへ永続化する必要がある．ここで，本論文ではこのような永続的デバイスへの書込を伴う処理要求を永続化要求と呼ぶ．

現在主流となっている永続的デバイスはハードディスクであるため，永続化要求処理速度は非常に低速となり，多くの永続化要求を処理する場合や，ストリームデータの入力レートが高い場合には，ストリームデータの入力に追随してそれを永続化する事が困難となってしまう．

そこで我々はこの問題に着目し，先行研究において永続化処理を効率的に処理可能とする手法を提案した[1][2]．先行研究の手法は処理木を最適化し，処理を次の 2 つのステップに分割する．**ステップ 1：**データを

中間領域へ永続化，**ステップ 2 :** データを中間領域から最終領域へ移動．これらのステップについて下記で述べる．

ステップ 1 では，処理木中で永続化データ量が準最小化する点においてデータを中間領域へ永続化する．ディスクアクセス削減の為，タプルをバッファリングし，複数のタプルを纏めて中間領域へ追記方式で書込処理を行う．追記処理はシーケンシャルに行われるため，書込処理の性能は高い．また，各ストリームに対して登録されている要求のうち，射影演算と選択演算を結合することによりデータの重複を避け永続化データ量を削減する．このようにすることで，永続化処理におけるデータ転送時間を削減する事が可能となる．

ステップ 2 は上記ステップ 1 で中間領域へ永続化したデータを最終領域へ移動する．中間領域の存在はユーザには不可視であるため，データをユーザへ見せるには，この移動処理が必要である．本論文ではこの処理を最終書込処理と呼ぶ．

上記の先行研究は 1 台のマシンを想定していた．この想定状況ではストリーム数の増加に伴い性能が急激に劣化する．また，先行研究では詳細な実験を行っていなかった．

そこで本論文では複数台のマシンを用いて中間領域を分散化することで，永続化処理の負荷を削減する手法を提案する．また，先行研究を含めた詳細な実験結果を示す．

本論文の構成は以下の通りである．2 節では本論文で扱う研究課題について述べる．3 節では先行研究で提案した永続化手法とそのアルゴリズムについて述べる．4 節では本論文で提案する分散化手法について述べる． 5 節では評価実験について述べる．6 節で関連研究について述べ，7 節でまとめと今後の課題について述べる．

2. 研究課題

本節では本研究の目的を研究課題へ定式化する．本研究の目的はストリームデータの永続化処理を効率化することである．処理性能を評価する一般的な指標には，スループットとレイテンシがある．また，リアルタイムシステムのようにデータ生成が周期的である場合には，デッドラインミス率も評価指標となる．リアルタイムシステムはストリーム情報源であるから，これは本研究では重要な評価指標である．評価指標を概観すると，スループットはマクロ的であり，残りの 2 つはミクロ的である．そこで本研究ではスループット極大化とデッドラインミス率極小化を研究課題とする．

2.1. デッドラインミス率

監視カメラの映像や犯罪者の位置情報などのストリームデータは欠損なく永続化される必要がある．なぜなら，事件・事故の発生後，発生時のデータが永続化されていないと，事件・事故の原因特定が困難となるためである．

データを欠損なく永続化するには，情報源が一定時間データを保持する必要がある．なぜならデータ永続化処理途中でシステムが故障してしまえば，そのデータはシステムから消失してしまうからである.ただし，情報源が保持すべきデータ量は少ない方が良い．そこで本研究では情報源が保持すべきデータ量を 1 データユニットとする．

このとき，情報源 S が時刻 t に生成したデータ d(t) のデッドラインを，S が次のデータ d(t+1)を生成する時刻(t+1)+C とする. C はシステムから情報源へのミス情報転送時間である．デッドラインまでに永続化が完了しない，あるいはシステムが停止した場合には，C 以内にミス情報が伝達されるとモデル化する．

この時，デッドラインミス率は下式で表される．

$$\text{デッドラインミス率} = \frac{\text{成功した永続化要求数}}{\text{全ての永続化要求数}}$$

2.2. 書込スループット

本研究では永続化デバイスとしてディスクを用いる．そのため，ディスクへの書込スループットをマクロ的性能指標とする．

3. 先行研究[1,2]

ここでは，我々が先行研究で行ったストリームデータ永続化処理の高速化手法について述べる．

3.1. 演算子

本研究で扱う演算子について述べる．本研究ではまず，リレーショナル演算である射影演算子，選択演算子，結合演算子，直積演算子を扱う．これに加えて，本研究では中間書込演算子，中間読出演算子，最終書込演算子の 3 つの演算子を新たに定義する．即ち，本研究が扱う演算子の数は 7 である．

中間書込演算子は，タプルを入力として受け取り，中間領域バッファへと書き込む演算子である．中間領域バッファについては，3.3 節にて説明する．中間読出演算子は，中間領域で永続化されたタプルを読み出す演算子である．最終書込演算子は，タプルを入力とし受け取り，最終領域へと書き込む演算子である．

3.2. 処理木の最適化

先行研究では，処理木を最適化し，永続化データ量が準最小化する位置を求め，そこで中間データを纏めて中間領域へと永続化する．ここでは，その最適化手順について説明する．例として，図 3-1~図 3~3 に示すクエリが登録された場合を考える．これらのクエリは登録されると処理木へと変換され，図 3-4 のようにな

る．ここで S_F は最終書込演算子を表す．さらに，従来の複数問合せ最適化を適用すると図 3-5 のようになる．先行研究では，図 3-5 の処理木に対してさらに最適化を施す．

3.2.1. 永続化位置の決定

本研究で扱う演算のうち，データ量に関係する演算子は，選択演算子，射影演算子，直積演算子，結合演算子の 4 つである．これらの出力データ量について検討する．選択演算子はタプル数を増やさない．射影演算子はタプルサイズを増やさない．従って，両者は永続化前に実行されるべきである．

直積演算子はタプル数とタプルサイズを共に増加させるため，永続化処理後に実行されるべきである．結合演算子はタプルサイズを増加させるがタプル数は結合選択率によって増減する．従って結合選択率に依存して適切な配置場所は変動する．結合選択率は一定でないし，その推定は困難である．そこで本研究では悪い場合を想定し，結合演算子は永続化後に実行する．

即ち，本研究では選択・射影演算子を永続化前に実行し，結合・直積演算子を永続化後に実行する．図 3-5 では，破線位置で永続化処理を実行する．

3.2.2. 演算子の併合・生成

次に，演算子の併合・生成処理について述べる．図 3-5 の破線より下にある射影演算子および選択演算子に着目する．図中の Query 1 における射影演算子，及び選択演算子と，Query 2 の Sensor A を入力とする選択演算子，及び射影演算子について，Query 1 の選択演算子は“Temp > 30”のタプルを選択し，Query 2 の選択演算子は”Temp > 40”のタプルを選択する．そのため，“Temp > 40”以上のタプルが Sensor A のストリームから到着した場合，重複して永続化を行ってしまう．同様に，Query 1 の射影演算子は属性“Temp, Humid”を射影し，Query 2 の射影演算子は属性“ID, Temp, Illuminate”を射影するため，属性“Temp”を重複して永続化してしまう．また，Query 3 における射影演算子，及び選択演算子と，Query 2 の Stream 2 を入力とする選択演算子，及び射影演算子についても同様である．そこで，図 3-6 のように，これらの演算子同士を併合し，元の演算子の出力を全て持つ演算子を生成し，その演算子を処理木の入力ストリーム側に挿入する．さらに，図 3-7 のように中間書込演算子と中間読出演算子を 1 つずつ生成し，先に生成した射影演算子、選択演算子の上に挿入する．ここで，S_I は中間書込演算子を表し，R は中間読出演算子を表す．このようにすることで，図 3-5 の破線の位置における中間データを全て永続化する．また同じデータを重複して永続化することがなくなるため，総永続化データ量を削減し，永続化処理におけるデータ転送時間を短縮できる．

```
INSERT INTO   Table1
SELECT        Temp, Humid
FROM          SensorA
WHERE         Temp > 30
```

図 3-1　Query1

```
INSERT INTO   Table2
SELECT        S1.ID, S1.Temp, S1.Illuminate, S2.ID, S2.Accel
FROM          SensorA as S1, SensorB as S2
WHERE         S1.ID = S2.ID and S1.Temp > 40 and S2.Accel > 50
```

図 3-2　Query2

```
INSERT INTO   Table3
SELECT        Temp, Accel
FROM          SensorB
WHERE         Temp > 60
```

図 3-3　Query3

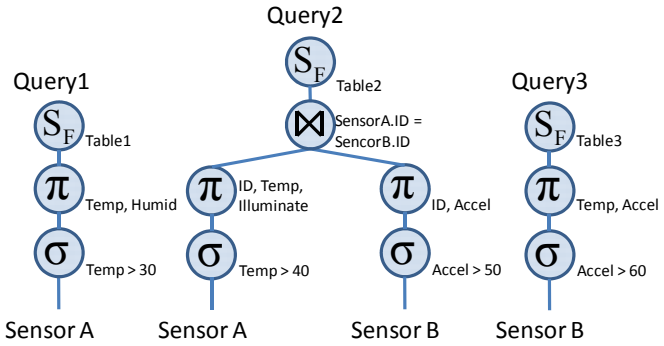

図 3-4　処理木の例

3.3. 中間領域に対する書込・読出処理の制御

中間書込処理をする際に 1 タプルごとに永続化を行うと，そのたびにディスクアクセスを発生させるため，非効率的である．そこで我々は図 3-7 に示すようにバッファリングを行い，複数のタプルを纏めて永続化を行う．この時，通常はなるべく多くのタプルをバッファリングした方が効率は良くなると考えられるが，先に述べたように，本研究では各タプルにはデッドラインを設けているため，中間バッファに含まれるタプルのデッドラインを守るように中間領域書込処理を行う必要がある．また，本研究では永続化処理を「データを中間領域へ永続化するステップ」と，「データを中間領域から読み出し，最終領域へ書き込むステップ」の，2 ステップと分けている．これにより中間書込処理と最終書込処理における中間領域への書込・読出処理が衝突する可能性がある．そこで，我々は中間書込処理を実行するタイミングを制御しつつ，中間領域読出処理を制御することで，デッドラインミス率を低減する技法を提案した[1,2].

我々の提案手法を，図 3-8 を用いて説明する．時刻 t1 にタプル 1 が到着し，また，時刻 t2 でタプル 2 が到

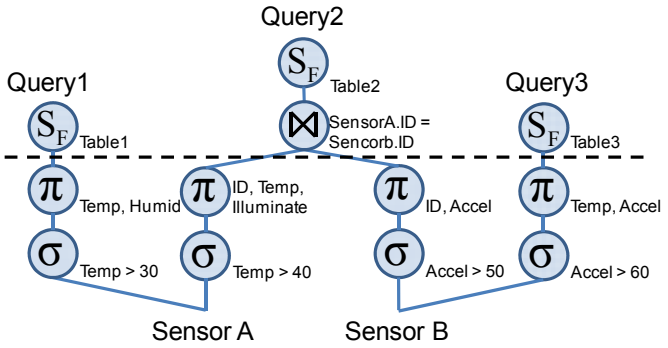

図 3-5　単純な複数問合せ最適化

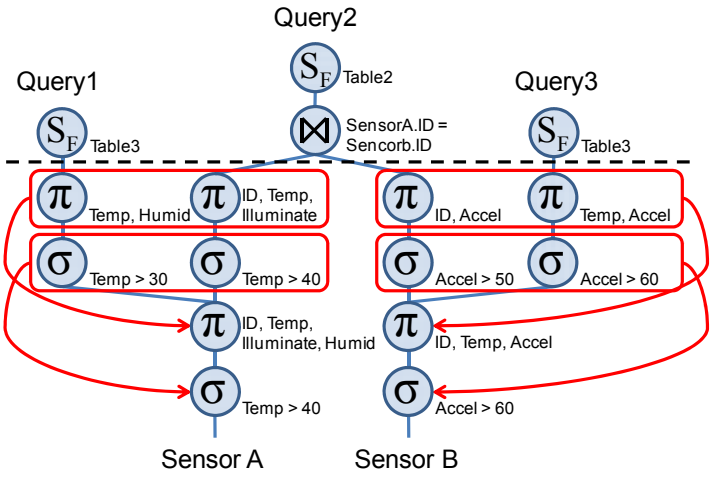

図 3-6　選択・射影演算子の併合・生成

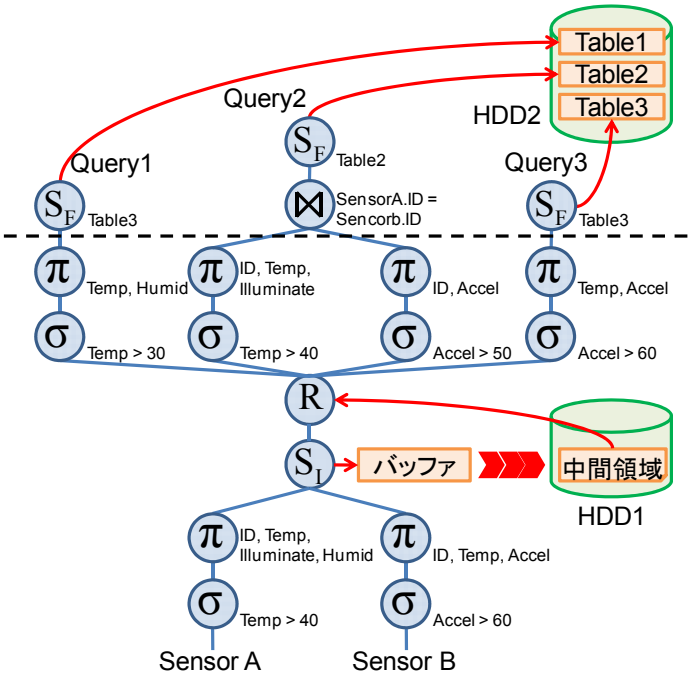

図 3-7　中間書込・中間読出演算子の生成

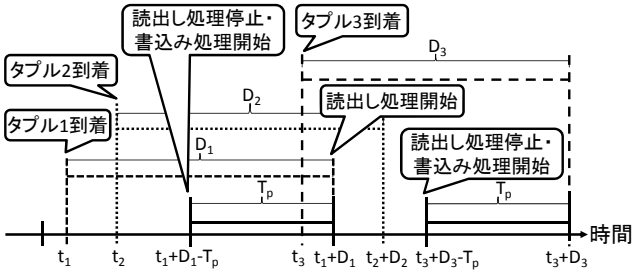

図 3-8　中間領域書込処理・読出処理の制御

着し，それぞれ中間バッファへ格納されているとする．ここで，D_1，D_2 はそれぞれタプル 1，タプル 2 の到着間隔を表す．そのため，タプル 1 とタプル 2 のデッドラインはそれぞれ t_1+D_1 と t_2+D_2 となる．ここで，中間バッファに格納されているタプルのデッドラインのうち，現在時刻から一番近いデッドラインを守るように永続化処理を実行すれば，中間バッファに格納されている全てのタプルのデッドラインを守る事が可能となる．

そのため，我々の手法では，前回の永続化処理が完了した時点で，その時刻から一番近いデッドラインを確認し，その時刻から予想ディスクアクセス時間を引いた時刻に次の永続化処理を開始する．ここで，T_p を予想ディスクアクセス時間とし，本研究では定数を与える．

例として，図 3-8 の t_2 において前回の永続化処理が完了した場合を考える．その時点で中間バッファに格納されているタプルのデッドラインを確認すると，タプル 1 のデッドラインが一番近いため，そのタプル 1 のデッドライン t_1+D_1 から T_P 前の $t_1+D_1-T_p$ まで中間書込処理を待機する．そして，中間書込処理が待機している間のみ，中間読出処理を実行する．

中間書込処理中の時刻 t_3 にタプル 3 が到着し，中間バッファへ格納されたとする．ここで D_3 をタプル 3 の到着間隔，実際にディスクアクセスに要した時間が T_p だったとする．このとき，中間書込処理が完了した時点で中間バッファに格納されているタプルのデッドラインを確認すると，中間書込処理の完了時刻 t_1+D_1 に一番近いデッドラインはタプル 3 のデッドラインである t_3+D_3 となるため，次回の中間書込処理の開始時刻は $t_3+D_3-T_p$ となる．この時刻まで中間書込処理を待機し，その間，中間読出処理が実行される．以上をまとめると，我々の手法は次のように動作する：(1)中間書込処理完了時に中間バッファ内タプルのデッドラインを確認し，最近デッドラインから予想ディスクアクセス時間を引いた時間まで中間書込処理を待機．(2)中間書込処理の待機時のみ，最終書込処理に要する中間読込処理を実行．

3.4. アルゴリズム

上述した処理手順をアルゴリズム 3-1 に示す．先行研究の手法は，処理を 3 つのスレッドへ割り当てる．処理木の中間書込演算子以降の処理を LowerThread，中間バッファを中間領域へ書き込む処理（中間書込処理）を MiddleThread，処理木の中間読出演算子以降の処理(最終書込処理)を UpperThread へ割り当てる．

LowerThread は入力ストリームごとに入力キューからタプルを読み込み，併合された各演算子の処理を行い，中間バッファへタプルを書き込む処理を繰り返す．

MiddleThread は，中間バッファに格納されたタプルを一括して中間領域へ追記処理により書き込む．また，3.3 節で説明したとおり，中間書込処理が完了した時点で，MiddleThread は中間バッファに格納されている全てのタプルのデッドラインを確認し，それらの最近時刻から予測ディスクアクセス時間を引いた時刻まで次の中間書込処理を待機する．アルゴリズム 3-1 では，次の中間書込処理の開始時間を求める関数 getSleepTime を呼び出す．この関数が返却した値だけ Sleep 関数により停止し，その際に，UpperThread を再開させることで 3.3 の処理を実現している(14~17 行目)．また，UpperThread は自身の実行時間を逐次計測しておき，次の中間書込処理の開始時間を超えそうになれば，自律的に動作を停止する(27 行目)．そのための関数として Check 関数を設けている(アルゴリズム 41~47 行目)．Check 関数内では，UpperThread の処理時間の推定値を用いている．この値は，本研究では事前に得られているとし，定数として与えている．

アルゴリズム 3-1　演算子併合化と動的アクセス制御

```
1.  Begin LowerThread
2.    While true do
3.      For each input stream
4.        入力キューからタプルを読込;
5.        各演算子の処理を実行;
6.        タプルを中間バッファへ書込;
7.      End for
8.    End while
9.  End LowerThread

10. Begin MiddleThread
11.   While true do
12.     中間バッファを中間領域へ書込;
13.     ts = getSleepTime;
14.     If ts > 0 do
15.       UpperThread 再開;
16.       Sleep(ts)
17.     End if
18.   End while
19. End MiddleThread

20. Begin UpperThread
21.   While true do
22.     ログ領域からタプルを読み込む;
23.     For each query
24.       各演算子の処理を行う;
25.       最終領域へ書き込む;
26.     End for
27.     Check;
28.   End while
29. End UpperThread

30. tp = 予想ディスクアクセス時間;
31. Begin getSpeepTime
32.   tn = 現在時刻;
33.   td = 中間バッファ内タプルのデッドラインの内，現在時刻に一番近い値;
34.   ts = td − tn − tp;
35.   If ts < 0 do
36.     ts = 0;
37.   End if
38.   Return ts;
39. End
40.
41. tex := UpperThread の推定実行時間;
42. Begin Check
43.   tre = UpperThread が
          実行開始してからの経過時間;
44.   If tre + tex > ts do
45.     UpperThread の実行停止
46.   End if
47. End Check
```

4. 中間領域の分散化

3 節で述べた技法は，より多数の永続化要求処理を効率よく処理可能とする．しかし，ハードディスクの書き込み性能には限界があり，1 台のマシンに搭載可能なハードディスクの数には上限があるため，3 節で想定した以上に多数の入力ストリームを扱う場合や，入力レートが高い場合には，性能が劣化する．

そこで本節では，複数台のマシンを利用して中間領域を分散させることで，デッドラインミス率を減少させ，かつ書込スループットを高くする手法を提案する．提案手法では，まず先行研究と同じく登録された処理木を最適化する．

この時，N 台のマシンが存在する場合，中間書込演算子と中間読出演算子のペアを N-1 個作るように最適化する．そして，N 台あるマシンの中の 1 台を最終領域用マシンと設定し，全ての最終書込演算子をそのマシンに配置する．また，その他の N-1 台のマシンを中間領域用マシンとし，中間書込演算子と中間読出演算子のペアをそれぞれに配置する．そして，その他の演算子を適宜分散配置する．

例として図 3-1 の処理木を分散配置し，Node 1, Node 2，Node 3 の 3 台のマシンで分散処理を行う場合を述べる．まず，マシンが 3 台あるため，中間書込演算子と中間読出演算子のペアを 2 つ(N-1)生成するよう処理木を変形する．そして，Node 1 を最終領域用マシン

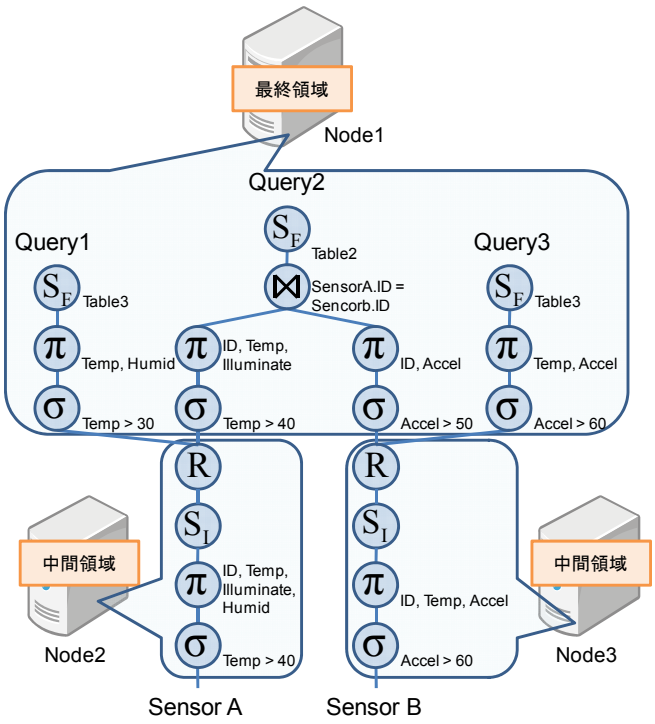

図 4-1　中間領域の分散化例

表 5-1　実験環境

OS	Ubuntu 10.04
カーネル	Linux 2.6.32-26-generic
CPU	Intel(R)Core(TM)i7-880
メモリ	2.9GiB
ファイルシステム	Ext3
ハードディスク	500GB×2

に指定し，Node 2，Node 3 の 2 台を中間領域用マシンに指定したとすると，全ての最終書込演算子を Node1 に配置する．そして他の 2 台に中間書込演算子と中間読込演算子のペアを配置する．最後に，その他の演算子を分散配置する．Sensor A の中間読込演算子までの部分処理木は Node 2 が受け持ち，Sensor B の中間読込演算子までの部分処理木は Node 3 が受け持つように配置し，中間読込演算子より上側にある演算子は全て Node 1 に配置したとすると，演算子の配置は図 4-1 のようになる．

5. 評価実験

ここでは，本論文で提案した中間領域の分散化手法の有効性の検証の為に行った評価実験について述べる．

5.1. 実験内容

分散化を行わない場合と，分散化を行った場合について性能評価実験を行った．実験内容は，タプルの入力レートを変化させた場合，入力ストリーム数を変化させた場合，永続要求数を変化させた場合について，それぞれデッドラインミス率とディスクへの書込スループットを計測した．各ストリームのスキーマは全て(id int, value int)とした．また，予想ディスクアクセス時間は 5(ms)に設定した．実験環境は表 5-1 の通りである．ハードディスクは 2 台あり，片方を中間領域用ディスク，もう片方を最終領域用ディスクとした．

5.2. 実験結果

実験結果を図 5-1~5-6 に示す．各図において“単一”は 1 台のマシンで処理を行った場合の結果を示し，“複数”は 3 台のマシンで分散処理を行った場合の結果を示す．また，“中間”は中間領域への書込スループットを，“最終”は最終領域への書込スループットを示す．

ストリームの入力レートを変化させた場合の実験結果を図 5-1 と図 5-2 に示す．入力ストリームは 2 本であり，各ストリームに 2 つずつ永続化要求数を登録した．実験結果より，デッドラインミス率は分散処理を行っても改善されない事がわかる．書込スループットについては，中間書込処理では大きな差が見られないものの，最終書処理込では分散処理による性能改善が観察される．

入力ストリーム数を変化させた場合の実験結果を図 5-3 と図 5-4 に示す．各ストリームの入力レートは 50(tuples/sec)であり，各ストリームに 2 つずつ永続化要求数を登録した．実験結果より，分散処理によりデッドラインミス率が低下する事がわかる．また，書込スループットについては，中間書込処理では大きな変化が見られないが，最終書込処理では，分散処理が優れた結果を示している．

永続化要求数を変化させた場合の実験結果を図 5-5 と図 5-6 に示す．入力ストリームは 2 本であり，各ストリームの入力レートは 50(tuples/sec)とした．実験結果より，分散処理を行うとデッドラインミス率が低下する事がわかる．また，デッドラインミス率は永続化要求数の影響を受けていない事がわかる．書込スループットについては，中間書込処理では大きな変化が見られないが，最終書込処理では分散処理を行うほうが高い結果となっている．

5.3. 議論

実験結果より，殆どの場合においてデッドラインミス率は分散化を行う場合の方が分散化を行わない場合よりも良い結果を示した．これは，分散処理を行うことでマシンが処理するタプル数が減少し，永続化処理負荷が削減されたためであると考えられる．また，最終領域への書込スループットも，多くの場合で分散処理を行なったほうが高い値を示した．これは，2 台のマシンに中間領域を分散したことで，中間領域からの読出処理を並列に実行でき，その分高速に最終書込処理を実行できたためであると考えられる．しかし，中間書込処理のスループットは分散処理を行っても大き

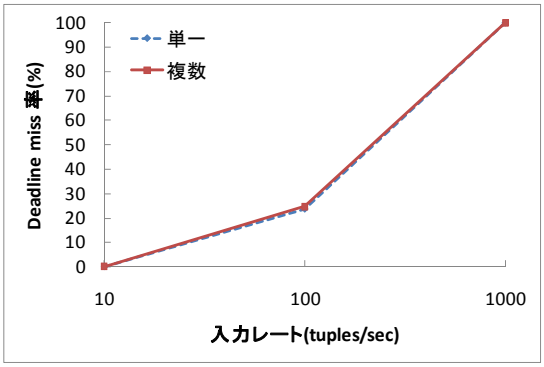

図 5-1
入力レートによるデッドラインミス率の変化

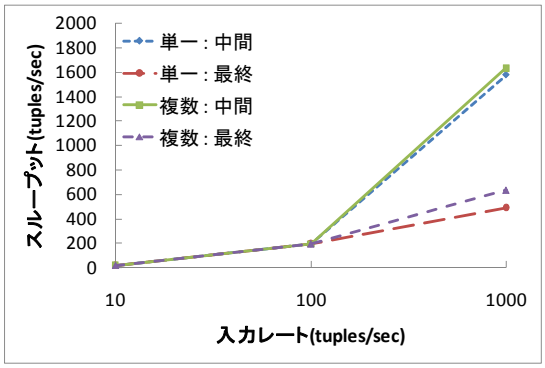

図 5-2
入力レートによる書込スループットの変化

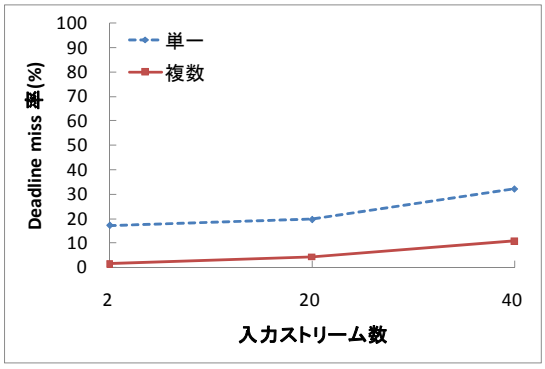

図 5-3
入力ストリーム数によるデッドラインミス率の変化

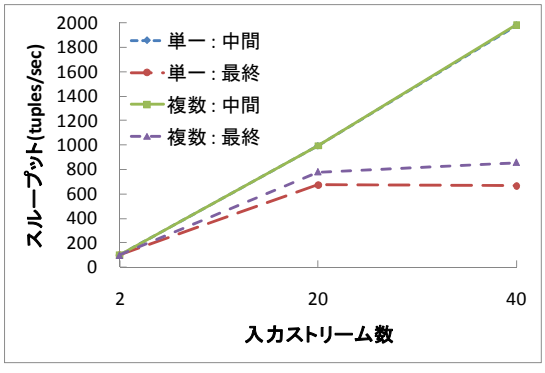

図 5-4
入力ストリーム数による書込スループットの変化

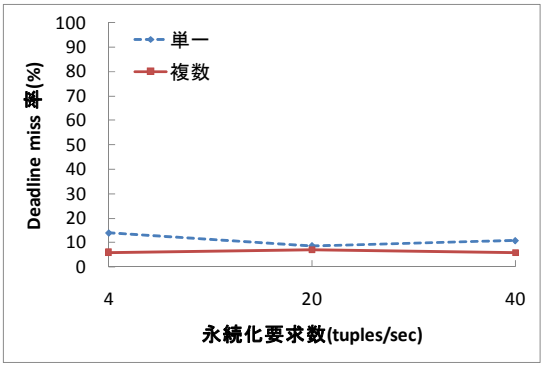

図 5-5
永続化要求数によるデッドラインミス率の変化

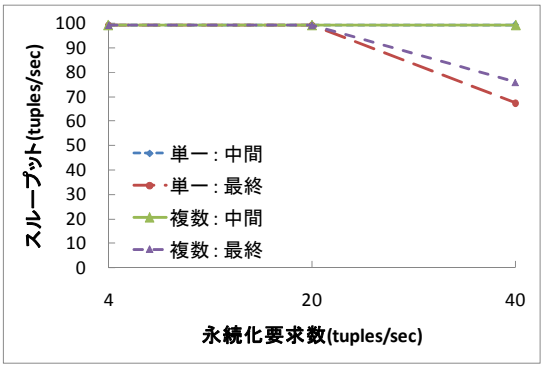

図 5-6
永続化要求数による書込スループットの変化

な改善は見られなかった．これは，1 台のマシンで処理を行った場合でも，マシンの能力にはまだ余裕があることが示唆される．そのため，より高負荷となる状況で処理を行えば，分散処理を行う場合の方が良い結果が得られると考えられる．また，分散処理を行わない場合，行う場合共にデッドラインミス率は永続化要求数の影響を受けない事がわかった．これは，処理木を最適化する際に，射影演算子と選択演算子を併合し，重複したデータの永続化を避けるため，永続化要求数が増加しても永続化するデータ量には変化が無いためであると考えられる．以上より，分散化によりデッドラインミス率，スループット共に改善することが可能である事が示された．

6. 関連研究

6.1. Harmonica

山田らの研究[3]では，ストリームに対する各種処理要求を容易に実現可能な実世界ストリーム管理基盤の開発を目指し，複数永続化要求最適化手法を開発した．

本研究では，複数永続化要求に関する性能改善を追及した．ただし，永続的デバイスとして本研究は DBMS ではなくファイルシステムを直接扱った．ファイルシステム直接扱う場合には，山田方式の場合とは異なり，

中間領域と最終領域を効率的に管理する必要性が生じる．なぜなら山田方式の場合には DBMS が中間領域と最終領域を隠蔽するからである．本研究ではこれらの領域について,デッドラインを考慮したバッファ制御，最終書込制御，そして中間領域の分散化に関する研究を行った．そして大きな性能改善を実現した．

6.2. DataDepot

DataDepot[5]はストリーム用データウェアハウスを作成するツールであり，数百 TB のデータを格納するために開発された．DataDepot は AT&T 内で複数の非常に巨大なウェアハウス化計画に使われている．

DataDepot は優れたウェアハウス作成システムであるが，本研究で扱っているような効率的なデータ永続化技法については検討がされていない．従って，DataDepot と本研究の成果は相互補完的であると考えられる．DataDepot に本研究成果を導入することで，一層高速なストリームをウェアハウス化できるようになる可能性がある．

6.3. NET-Fli

NET-Fli[4]はネットワークトラフィックを取り込むために開発された専用ストレージである．NET-Fli は急速圧縮技術と急速索引スキームにより，優れた性能を示す．

NET-Fli と本研究の目的は類似しており，いずれも高速なデータ永続化である．しかしながら細かい点で両者の目的は異なる．NET-Fli は上記のように索引付と永続化スループット極大化を主目的にしており，漏れの無いデータ保存とリアルタイム処理を主眼にしていない．NET-Fli の方式では，システム故障時に多数のデータが消失する可能性がある．なぜなら NET-Fli は多数のデータをメモリ上で圧縮処理してから永続化デバイスに書き込むからである．一方，本研究の目的は漏れの無いデータ保存とリアルタイム処理であり，スループット極大化を主目的にしていない．それゆえ両研究は異なる目的を有する．

6.4. Hyperion

Hyperion[7]はネットワークトラフィックを高速に永続化すると同時に，後で行われる問合せのためにデータをオンラインで索引化するシステムである．Hyperion は複数層索引と StreamFS と呼ばれる専用ファイルシステムから構成される

Hyperion は NET-Fli 同様にオンラインでトラフィックストリームに索引付をしながら高スループットでの永続化デバイスへの書込処理を行うシステムである．従って NET-Fli 同様に，Hyperion と本研究は研究目的が異なる．

7. まとめと今後の課題

本研究では，データストリーム管理システムにおいて永続化要求数が多い場合や，ストリームデータの入力レートが高い場合などにおいて，ストリームデータの入力に追随してそれを永続化することが困難となる問題に対し，これまで提案した手法を拡張し，中間領域を分散することで一層効率的に永続化処理を実行可能とする手法を提案した．そして，包括的な実験により，分散処理を行うことでデッドラインミス率と最終領域への書込スループットを改善する事が可能であることがわかった．

今後の課題は，提案手法をストレージマネジャとして実装し，それを汎用ストリーム管理エンジンと接続することが挙げられる．また，データを圧縮して永続化することが挙げられる．

謝辞

本研究の一部は，科研費 基盤研究 A(#21240005)，科研費 若手研究 B(#22700090)による．

参 考 文 献

[1] 阿部泰芽, 川島英之, 北川博之, “ストリーム処理エンジンにおける複数書き込み最適化の提案”, 情報処理学会創立 50 周年記念（第 72 回）全国大会，2010

[2] 阿部泰芽, 川島英之, 北川博之, “データストリーム処理の適応的最適化”, 情報処理学会ユビキタスコンピューティングシステム研究会(UBI)第 27 回研究発表会，2010

[3] 山田真一, 渡辺陽介, 北川博之, 天笠俊之. “データストリーム管理システム Harmonica の設計と実装”, 情報処理学会論文誌：データベース, Vol.48, No.SIG14 (TOD35), pp. 91-106, 2007 年 9 月.

[4] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos, “NET-FLi: on-the-fly compression, archiving and indexing of streaming network traffic”, In Proc. of VLDB2010.

[5] Lukasz Golab, Theodore Johnson, J. Spencer Seidel, and Vladislav Shkapenyuk, “Stream warehousing with DataDepot”, In Proc. of SIGMOD2009.

[6] Yosuke Watanabe and Hiroyuki Kitagawa, “Query Result Caching for Multiple Event-driven Continuous Queries”, Information Systems, Vol. 35, No. 1, pp. 94-110, 2010.

[7] Peter J. Desnoyers and Prashant Shenoy, “Hyperion: high volume stream archival for retrospective querying”, In Proc. of USENIX Annual Technical Conference 2007.

[8] A. Arasu, S. Babu, J. Widom, “The CQL Continuous Query Language: Semantic Foundations and Query Execution”, VLDB Journal, 2005.

[9] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, “Monitoring Streams: A New Class of Data Management Applications”, In Proc. of VLDB2002.