# ストリームデータ処理におけるデータ生存期間管理方式

# Data Lifetime Management Method in Stream Data Processing

今木 常之♠ 西澤 格♠
Tsuneyuki IMAKI Itaru NISHIZAWA

**近年，RFID タグ，センサノードなどを情報源として生成される高レートのデータを，継続的にリアルタイム処理するストリームデータ処理の重要性が増している．ストリームデータ処理においては，無限に継続するデータ系列の中から処理対象を規定するため，データに生存期間の概念が必要となる．データの生存期間を管理する方式としては Negative Tuple 法と Direct 法が提案されており，Direct 法は処理スループットが Negative Tuple 法より優れる一方，実行可能な問合せが限定されていた．本稿では，Direct 法で実行可能な処理のクラスを Negative Tuple 法と同等に広げるためのデータ生存期間管理方式を提案した．さらに，本方式により Direct 法で実行可能となった基本クエリに対して評価実験を実施し，1.4～2.0 倍のスループット向上を確認した．**

**Emerging RFID technology and sensor network systems produce huge amount of data, and modern enterprise data processing systems must handle the huge amount of data in the real-time fashion. Stream data processing, which continuously processes the data in the real-time fashion, has proposed and accepted as a new data processing paradigm. Data lifetime notion is essential in the stream data processing because it requires extracting the target data from infinite data sequences using the notion. Negative Tuple Method (NT) and Direct Method (DT) have been proposed to manage the data lifetime. Although DT outperforms NT in the system throughput, it cannot be applied to some basic query processing operators due to its processing mechanism. This paper proposes a new data lifetime management method which extends DT to enable applying it to the basic query processing operators as the same as NT. Moreover, we have confirmed by experiments that our new data lifetime management method outperforms NT by 1.4-2.0 times in the system throughput.**

## 1. はじめに

近年，RFID タグの読取情報，センサノードの監視情報，ITS，プローブカーによる渋滞情報，株価情報など，絶え間なく連続的にデータを生成する情報源が増加している．これらのデータは即時性に意味がある場合が多いため，瞬時かつ継続的に処理することが求められている．このような要求が増大する中，高レートデータの連続処理を目的としたストリームデータ処理が注目されている．

ストリームデータ処理は，RDBMS で広く利用されてきた SQL に類似の言語で処理内容を定義することにより，C++, Java などの言語でアプリケーションを構築する場合に比べて開発コストを大幅に削減できる．一方で，SQL の計算モデルである関係代数は，有限のデータ集合を演算対象とするため，無限に連続するデータから処理対象を絞り込むことが不可欠となる．これに対し，連続データから時間範囲を区切ってデータを切り出す，スライディングウィンドウの概念が提案されている[1]．ウィンドウはデータの生存期間を定める演算と捉えることができる．ウィンドウ演算により生存期間の定められたデータは，関係代数に基づく別の演算(射影，選択，結合，集計など)の入力となり，その演算結果もやはり生存期間を持つデータとなる．結果データの生存期間の決まり方は演算の種類によって異なる．

ストリームデータ処理の実装におけるデータの生存期間の管理方式として，大別して Negative Tuple 法(以下 NT)と Direct 法(以下 DT)の二方式を挙げることができる．両方式とも，複数の演算をノードとする実行木を構築し，データキューを介して演算同士をパイプライン的に処理させる．NT は一つのデータを，生存期間の始まり(出現)と終わり(消滅)を表すそれぞれ別々のオブジェクトで表現する．そのため，一つのデータにつき二度の処理を必要とする点で非効率的である．DT は一つのデータを，出現と消滅の両時刻を含む一つのオブジェクトで表現することでこれを回避する．但し，一部の演算は結果データ生成時に消滅時刻を確定できないため，DT で実現できる処理は限定されていた．

これに対し本研究では，結果生成時に消滅時刻が未確定となる演算にも DT を適用可能とする実装方式を提案する．これにより，全てのクエリを DT で実行可能となった．また，本提案により DT で新たに実現されたクエリについてマイクロベンチマークを行い，基本クエリで NT に比べてスループット向上の効果が得られることを併せて確認した．

## 2. データモデル

本研究では，STREAM[1]におけるデータ処理定義言語である CQL[2]を利用する．但し，CQL では時刻を離散値(正の整数)として扱うのに対し，本研究では時刻の精度を高める目的で，時刻を連続値(例として，正の実数)として扱えるように，データモデルを改変する．

### 2.1 ストリームと期間リレーション

ストリームデータ処理で扱うデータモデルを定義する(以下，タイムスタンプと時刻は同義とする)．

**リレーション**：n 個の集合の直積集合の有限部分集合．即ち n 項組データ値の有限集合．

**ストリームタプル**： n 項組データ値 $v$ とタイムスタンプ $t$ の組 $\langle v, t \rangle$．ストリームタプル $s$ のデータ値を $v(s)$，タイムスタンプを $t(s)$と表す．

**ストリーム**：ストリームタプルを要素とする無限または有限の順序集合であり，$S$ と表す．またその要素を $s_i (1 \le i)$ と表す．$S$ 上の要素は $t(s_i) \le t(s_j)\ (i<j)$ を満たす．

**部分ストリーム**：$S$ の最初の $m$ 要素からなる有限順序集合で，$S(m)$と表す．$S(m) = \{s_i \mid 1 \le i \le m \wedge s_i \in S\}$．また，ある時刻 $t_o \ge t(s_m)$ に対し，$t(s_{m+1}) < t_o$ となる $m$+1 が存在しない場合(即ち，ストリーム $S$ において，ストリームタプル $s_m$ の後，時刻 $t_o$ より前には他のストリームタ

♠ 正会員 株式会社日立製作所 中央研究所
{tsuneyuki.imaki.nn, itaru.nishizawa.cw}@hitachi.com

プルが到来しない場合)，$t_o$ を $S$ における $m$ 延長時刻と呼ぶ．

**期間タプル：** n 項組データ値 $v$，出現時刻 $ta$，消滅時刻 $te$ の組 〈$v$, $ta$, $te$〉．期間タプル $r$ のデータ値を $v(r)$, 出現時刻を $ta(r)$, 消滅時刻を $te(r)$ と表す．

**期間リレーション：** 期間タプルを要素とする有限集合．

ストリームデータ処理では，部分ストリームをウィンドウ演算によって，期間リレーションに変換することで，処理対象となるデータを規定する．部分ストリーム $S(m)$，$m$ 延長時刻 $t_o$，およびウィンドウ演算 $W$ が与えられたときに，以下のような期間リレーションを得る関数 $Rw$ を定義する．

$$Rw(W, t_o, S(m)) = \{r \mid r = W(t_o, S(m), s) \wedge s \in S(m)\}$$

但し $W$ は，部分ストリーム $S(m)$上の一つのストリームタプル $s_i (1 \le i \le m)$ が与えられたときに，$v(s_i)$をデータ値，$t(s_i)$ を出現時刻，および $t(s_i)$ 以降のある時刻を消滅時刻とする一つの期間タプルを得る，以下のような関数とする．

**時間ウィンドウ:** ストリームタプルを，特定の生存期間 $T$ を持つ期間タプルに変換

$$W(t_o, S(m), s_i) = \langle v(s_i), t(s_i), t(s_i)+T \rangle$$

**行数ウィンドウ:** ストリームタプルを，その $N$ 番後ろのストリームタプルのタイムスタンプを消滅時刻とする期間タプルに変換

$$W(t_o, S(m), s_i) = \langle v(s_i), t(s_i), t(s_{i+N}) \rangle \quad (1 \le i \le m\text{-}N)$$

$$W(t_o, S(m), s_i) = \langle v(s_i), t(s_i), nft(t_o) \rangle \quad (m\text{-}N+1 \le i \le m)$$

ここで，$nft(t_o)$ は時刻 $t_o$ において消滅時刻が**未確定**であることを表す．期間リレーション $Rw(W, t_o, S(m))$ は，一般に時刻 $t_o$ によって異なる(但し $m$ は $t_o$ を延長時刻とする任意の値)．従って，以降では期間リレーションを時刻 $t_o$ の関数として $R(t_o)$ のように記す．

期間リレーション $R(t_o)$，および時刻 $t$ $(0<t<t_o)$ が与えられたときに，以下のようなリレーションを得る関数 $Rv$ を定義する．

$$Rv(t, R(t_o)) = \{v(r) \mid r \in R(t_o) \wedge ta(r) \le t \wedge t < te(r)\} \quad (0<t<t_o)$$

$Rv(t, R(t_o))$ を，期間リレーション $R(t_o)$上の時刻 $t$ におけるリレーションと呼ぶ．

以上の直感的な意味付けを図 1 に示す．ストリームタプルは時間軸上における点データであり，ウィンドウ演算によって生存期間を持つ線分である期間タプルに変換される．時刻 $t$ に交わる期間タプルのデータ値の集合が，時刻 $t$ におけるリレーションとなる．関数 $Rv$ の定義より，期間タプルの生存期間は左閉右開区間となる．即ち，生存期間に出現時刻を含み，消滅時刻を含まない．

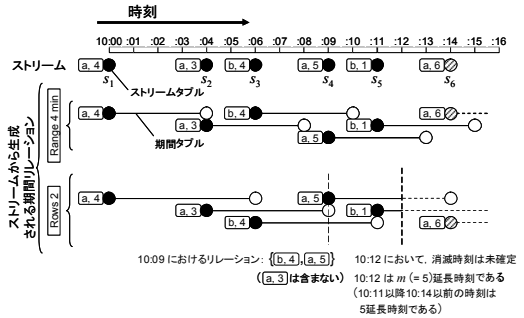


図 1: ウィンドウ演算による期間リレーションの生成
Fig.1: Conversion of a stream to time-limited relations by window operators

## 2.2 期間リレーションの関係演算

リレーションに対して定義される従来型の関係演算 $Q$ (二項演算とする)および二つの期間リレーション $R_1(t_o)$，$R_2(t_o)$ が与えられたときに，以下のような期間リレーションを得る関数 $Rq$ を定義する．

全ての時刻 $t$ $(0<t<t_o)$ において以下が成り立つ：

$$Rv(t, Rq(Q, R_1(t_o), R_2(t_o))) = Q(Rv(t, R_1(t_o)), Rv(t, R_2(t_o)))$$

このような期間リレーション $Rq(Q, R_1(t_o), R_2(t_o))$ を得る処理を，期間リレーションの関係演算と定義する．$Q$ が単項演算である場合も同様に定義できる．このような期間リレーションは一意には定まらないが，ここで定義した関係演算に対しては，その何れも全て等価である(2.3 節参照)．

## 2.3 期間リレーションの等価性

[定義]

二つの期間リレーション $R(t_o)$ と $R'(t_o)$ に関し，全ての時刻 $t$ $(0<t<t_o)$ において $Rv(t, R(t_o)) = Rv(t, R'(t_o))$ が成り立つことを，$R(t_o)$ と $R'(t_o)$ が**等価**であると呼ぶ．

[定理]

期間リレーションの関係演算においては，異なる二つの期間リレーションが等価であるならば，それぞれを入力とした関係演算の結果も等価となる．

[証明]

$Q$ を従来型の関係演算(単項演算)とする．また，二つの期間リレーション $R(t_o)$と $R'(t_o)$が等価であるとする．

2.2 節に示した期間リレーションの関係演算の定義より，

$$Rv(t, Rq(Q, R(t_o))) = Q(Rv(t, R(t_o))) \quad (0<^{\forall}t<t_o)$$
$$Rv(t, Rq(Q, R'(t_o))) = Q(Rv(t, R'(t_o))) \quad (0<^{\forall}t<t_o)$$

が成り立つ．一方，等価性の定義から，

$$Rv(t, R(t_o)) = Rv(t, R'(t_o)) \quad (0<^{\forall}t<t_o)$$

であるので，演算 $Q$ の性質から

$$Q(Rv(t, R(t_o))) = Q(Rv(t, R'(t_o))) \quad (0<^{\forall}t<t_o)$$

が成り立つ．従って，

$$Rv(t, Rq(Q, R(t_o))) = Rv(t, Rq(Q, R'(t_o))) \quad (0<^{\forall}t<t_o)$$

が成り立つので，$Rq(Q, R(t_o))$ と $Rq(Q, R'(t_o))$ は等価である．$Q$ が二項演算の場合も同様に示せる． □

なお，$ta(r) = te(r)$ であるような期間タプル $r$ のデータ値 $v(r)$ は，関数 $Rv$ の定義よりいかなる $t$ $(0<t<t_o)$ における $Rv(t, R(t_o))$ にも含まれないので，期間リレーションの等価性に影響を与えない．本稿では，このような $r$ を**ゴースト**と呼ぶ．

本節に示した等価性に基づくことで，実装において，期間タプルの生成方法に関する自由度を保証できる．

# 3. 提案手法

## 3.1 未確定消滅時刻の再評価法

1 節に示したように DT では，行数ウィンドウのように演算実行の際に結果タプルの消滅時刻が確定しない演算を実現できない．SUM, AVG などの集計演算，INTERSECT, EXCEPT などの集合演算もこのような演算に分類される．

上記に分類される演算 A について，演算実行の際には結果タプルの消滅時刻が未確定でも，A がさらに別の入力タプルを処理することで確定する可能性がある．そこで，結果タプルに付与する消滅時刻として，具体的な時刻値の代わりに**消滅時刻ホルダ**を付加しておく．これは，のちに結果タプルの消滅時刻が確定したタイミングでその時刻を代入すること

が可能なホルダオブジェクトである．

一方，実行木上で演算 A の親ノードに位置する演算 B では，受理したタプルに消滅時刻ホルダが付いている場合は，同タプルを演算 B の**消滅時刻未確定集合**に保持し，消滅時刻の確定状況を監視する Observer を割当てる．のちに演算 A によってホルダに確定時刻が代入された際に同 Observer がコールバックされ，その処理において同タプルを演算 B の**消滅時刻確定集合**に移動する．

例として図 2 では，状態 1 において演算 A(行数ウィンドウ)にストリームタプル 1 が入力され，その結果である期間タプルを演算 B が消滅時刻未確定集合に格納し，さらに状態 2 において，ストリームタプル 2 が入力されて期間タプル 1 の消滅時刻が確定する様子を示している．

但し，結果タプルの生存期間が入力タプルと全く同じとなる演算では，入力の消滅時刻が未確定であっても，消滅時刻ホルダへの参照を結果タプルにコピーするのみで構わない．選択，射影，和集合の三つがこのような演算に分類される．

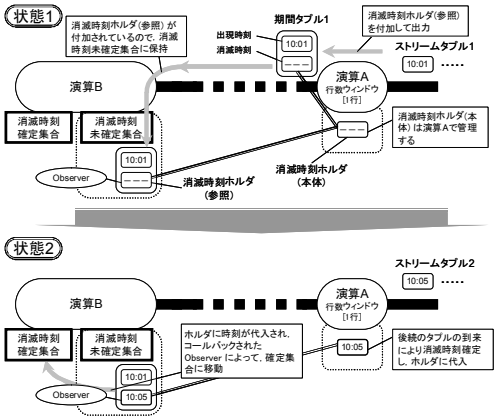


図 2: 消滅時刻ホルダによる消滅時刻の確定

Fig. 2: Determining expiration time by using expiration time holder

### 3.2 消滅時刻順の遅延整列法

消滅時刻確定集合にある期間タプルのうち，入力タプルの出現時刻(以下，入力時刻)より早い消滅時刻を持つタプル群は，当該入力タプルを処理する前にその演算での処理対象から除く必要がある．DT では，この操作を効率的に行うために，消滅時刻確定集合にあるタプルを消滅時刻順に整列しておく．この整列を単純な挿入ソートで行うと，タプルを入力する度に集合のサイズに比例するコストがかかる．

一方，入力時刻以降の消滅時刻を持つタプルは処理対象として残るので，整列されている必要はない．このことを利用し，整列動作を必要になるまで遅延させる方式を提案する．具体的には以下のような動作とする．

消滅時刻確定集合を整列済み集合と非整列集合に分ける．消滅時刻確定集合に移動されたタプルは非整列集合に保存し，同集合内で最早の消滅時刻を記憶しておく．入力時刻が同時刻を超えた時にはじめて整列し，整列済み集合とマージする．

### 3.3 提案方式の正当性

以上説明した動作において，消滅時刻未確定のタプルは無条件に処理対象として残すことになる．このような処理の正当性は以下のように説明できる．

演算 A が演算 B の孫ノードであるとする．A で生成したタプル T が消滅時刻未確定で B に保持されており，その時点での B の入力時刻を $t_B$ とする．一方，A がその時点で処理を完了している最新のタプルの出現時刻を $t_A$ とする．A がその時点で T の消滅時刻を確定できていないということは，T の消滅時刻が定まるとすれば，その時刻 $t_X$ について $t_X \geq t_A$ が成り立つ．一方，パイプライン型処理に従えば，B は A より後段に処理される演算なので，$t_A \geq t_B$ が成り立つ．従って $t_X \geq t_B$ が成り立つため，タプル T は演算 B において，その入力タプルと生存期間が重なり，同時に処理対象となることが保証される．

但し，丁度 $t_X = t_B$ である場合は余分なゴーストが生成されることになるが，2.3 節で示したようにゴーストは結果の等価性に影響しないので，方式の正当性は保たれる．

## 4. 実験

### 4.1 クエリとデータ

図 3 に示す 6 つの基本クエリとデータでマイクロベンチマークを実施した．Java を実装言語とし，1GB のメモリを搭載した PC で Java VM に 756MB のメモリを割り当てた．

クエリ

```
選択        select * from STRu[WINu] where cb > 3
射影        select ca, cb from STRu[WINu]
和集合      select * from STRb0[WINb]
              union all
            select * from STRb1[WINb]
結合        select * from STRb0[WINb], STRb1[WINb]
              where STRb0.ca = STRb1.ca
差集合      select * from STRb0[WINb]
              except
            select * from STRb1[WINb]
集計(平均)  select avg(cb) from STRu[WINu] group by ca, cc
```

データ

| STRu<br>TimeStamp[msec] ca, cb, cc | STRb0<br>TimeStamp[msec] ca, cb, cc | STRb1 |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 14390.0: (a, 0, v) | 14390.0: (535, a, 0) | 14390.5: (535, a, 0) |
| 14391.0: (a, 1, u) | 14391.0: (536, b, 1) | 14391.5: (536, b, 1) |
| 14392.0: (b, 2, u) | 14392.0: (537, c, 2) | 14392.5: (537, c, 2) |
| 14393.0: (b, 3, v) | 14393.0: (538, d, 3) | 14393.5: (538, d, 3) |
| 14394.0: (a, 4, v) | 14394.0: (539, e, 4) | 14394.5: (539, e, 4) |
| 14395.0: (a, 5, u) | 14395.0: (540, a, 5) | 14395.5: (540, a, 5) |
| ⋮ | ⋮ | ⋮ |

図 3: 実験に用いたクエリとデータ

Fig. 3: Queries and data for experiments

ここで，WINu, WINb はウィンドウ演算のサイズ指定で，WINu は行数ウィンドウ 10 行～1,000,000 行，時間ウィンドウ 10 msec～1,000 sec で変化させ，WINb は行数ウィンドウ 5 行～500,000 行，時間ウィンドウ 5 msec～500 sec で変化させた．また，ストリームタプルには 1 msec 間隔のタイムスタンプを付けており，カラム STRb0.ca およびカラム STRb1.ca は 1 ずつ増える順番で並ぶ．

STRb0 と STRb1 のストリームタプルが揃って流れるため，結合処理のクエリではウィンドウサイズに係わらず，二つのタプルから一つの結果が生成される．また，STRb0 と STRb1 のタイムスタンプが 0.5 msec ずれているため，差集合クエリではウィンドウサイズに係わらず，二つのタプルから 0.5 msec の期間だけ生存する一つの結果が生成される．

なお，N msec の時間ウィンドウは，N 行の行数ウィンドウと同じく常に N 個のタプルを切り出す．

### 4.2 測定結果

図 4 に，NT と DT の性能比較をクエリ別に示す．縦軸はスループット，横軸はウィンドウが切出すタプル数を表す．なお，スループットは 10 行の行数ウィンドウについて NT

で処理した場合の結果を 1 とした相対値で示す.

選択，射影，和集合の各クエリにおいては，10,000 行までの行数ウィンドウについて, DT は NT に対し 1.4 倍～2.0 倍のスループット改善の効果があった．一方，ウィンドウサイズが 10,000 行を超えると急激に性能が低下し，1,000,000 行においては DT と NT の性能はほぼ等しくなった．この原因を調べるため，全体処理時間に対し Java VM のガベージコレクション(GC)が占める割合を測定した．その結果，ウィンドウサイズ 10,000 行の処理においては 1～2%と非常に小さい割合であったが，1,000,000 行の処理では 14～22%と GC のコストが大幅に増大していることが確認された. また, 10,000,000 行ではメモリオーバとなり実行不可能であった．

これに対し, 時間ウィンドウではこのような傾向は確認されず性能が安定していた．また，NT での 1,000,000 行の処理における GC のコストも，4～9%であり，DT に比べて小さく性能も安定していた．このことから，行数ウィンドウで発生する消滅時刻ホルダの生成･回収コストが大きいことが予想される．本研究の提案手法は空間コストの観点で改善の余地があることが示された.

結合クエリにおいては, 時間ウィンドウで DT が NT に対して 1.6～2.0 倍程度，差集合，集約クエリにおいては 1.2～1.4 倍程度の性能向上を示した．一方，行数ウィンドウについては，各クエリとも全てのウィンドウサイズに渡って DT と NT の性能はほぼ同じであった．これらのクエリの行数ウィンドウにおいては，消滅時刻ホルダを生成・付加するオーバヘッドがかかる．このコストが，NT でのウィンドウ演算におけるタプル管理コストと同程度になっていると予想される.

また，遅延整列を行なわない場合の結果(グラフ上での“DT:挿入”のライン)は，ウィンドウのタプル切り出し数が 100 の段階から性能が急激に悪化した．これに対し，遅延整列を行なう DT は NT と同じ性能の傾向を維持しており，遅延整列の効果が確認された.

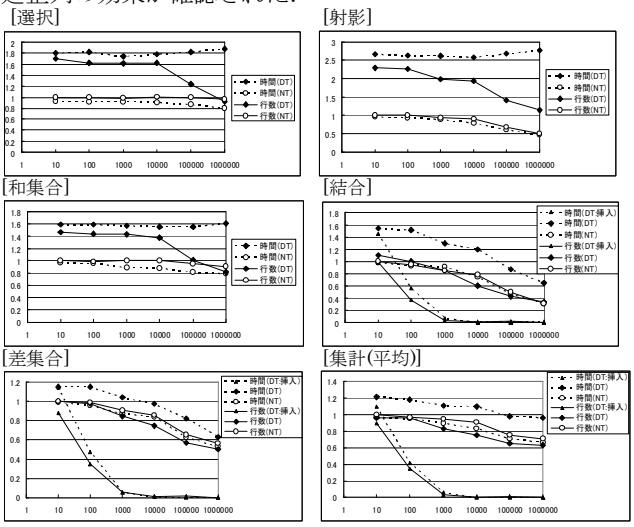


図 4: 基本クエリのスループット(相対値)

Fig.4: Query processing throughput

## 5. 関連研究

[3]は，時間ウィンドウを含む実行木の構成法として，NT と Time Probing 法のハイブリッド方式を提案している．Time Probing 法は，各演算が実行木上での子ノードとなる演算に対して，処理が完了した時刻を確認し，自ノード上に保持するデータの中から消滅させて良いデータを確定する方式である．但し，子ノードの処理完了時刻から消滅時刻を計算するため，扱えるのは時間ウィンドウのみである.

[4]は，NT と DT のハイブリッド方式を提案している．演算実行の際に消滅時刻が確定されない演算を実行木上で pop up，確定される演算を push down し，ルート側を NT，リーフ側を DT で構成することで, DT の適用範囲を最大化する．但し，行数ウィンドウはリーフノードに位置しないとクエリの意味が変わる可能性があるので，この方式では実行木全体を NT で構成しなくてはならない.

以上のように，[3][4]両方式ともに NT と他の方式とのハイブリッドであるが，どちらも行数ウィンドウを入力とするクエリには適用できない．さらに，ハイブリッドを効果的に機能させるために，実行木の最適化が必要となる．本研究では行数ウィンドウを含む全ての演算を DT で実現するため，このような最適化処理は不要である.

[5]は，multi-way の結合演算の実現方式を示している．本方式には NT も DT も適用可能である．但し，ウィンドウ演算の結果が結合演算に直接入力されることが前提となるため，結合演算は実行木のリーフに位置する必要がある．本研究では，全ての演算を DT で実現するため，結合演算の実行木上での存在位置を自由に定めることが可能である.

## 6. おわりに

本稿では，ストリームデータ処理における DT の拡張方式として，消滅時刻が未確定であるデータの再評価法を提案した．同方式により，DT で全てのクエリを処理することが可能となった．さらに，DT で新たに実行可能となった基本クエリに関して，処理スループット改善の効果を確認した．以上により，NT の性能改善方式として DT を単独で利用する可能性を示した．課題として，巨大ウィンドウサイズにおける空間コストの削減が挙げられる.

## [文献]

[1] B. Babcock, S. Babu, M. Datar, R. Motwani and J. Widom, “Models and issues in data stream systems”, In Proc. of PODS 2002, pp. 1-16. (2002).

[2] A. Arasu, S. Babu and J. Widom, “CQL: A Language for Continuous Queries over Streams and Relations”, In Proc. of DBPL 2003, pp. 1-19 (2003).

[3] M. Hammad, W. Aref, M. Franklin, M. Mokbel and A. Elmagarmid, “Efficient Execution of Sliding-Window Queries Over Data Streams”, Technical Report, Purdue University (2003).

[4] L. Golab, M. Özsu, “Update-Pattern-Aware Modeling and Processing of Continuous Queries”, In Proc. of SIGMOD 2005, pp. 658-669 (2005).

[5] L. Golab, M. Özsu, “Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams”, In Proc. of VLDB 2003, pp. 500-511 (2003).

**今木 常之 Tsuneyuki IMAKI**

1996 年東京大学大学院工学系研究科修士課程修了．同年，日立製作所入社．中央研究所研究員．情報処理学会会員.

**西澤 格 Itaru NISHIZAWA**

1996 年東京大学大学院工学系研究科博士課程修了．同年，日立製作所入社．中央研究所主任研究員．2002～2003 年米スタンフォード大客員研究員．ACM，情報処理学会各会員.