PAPER *Special Section on Data Engineering*

Temporal Coalescing on Window Extents over Data Streams

Mohammed AL-KATEB[†a)], ***Student Member***, **Sasi Sekhar KUNTA**[†b)], ***and*** **Byung Suk LEE**[†c)], ***Nonmembers***

SUMMARY This paper focuses on the coalescing operator applied to the processing of continuous queries with temporal functions and predicates over windowed data streams. Coalescing is a key operation enabling the evaluation of interval predicates and functions on temporal tuples. Applying this operation for temporal query processing on windowed streams brings the challenge of coalescing tuples in a window extent each time the window slides over the data stream. This coalescing becomes even more involving when some tuples arrive out of order. This paper distinguishes between eager coalescing and lazy coalescing, the two known coalescing schemes. The former coalesces tuples during window extent update and the latter does it during window extent scan. With these two schemes, the paper first presents algorithms for updating a window extent for both tuple-based and time-based windows. Then, the problem of optimally selecting between eager and lazy coalescing for concurrent queries is formulated as a 0-1 integer programming problem. Through extensive performance study, the two schemes are compared and the optimal selection is demonstrated.

key words: *data streams, window extents, temporal coalescing*

1. Introduction

Time is a very common aspect of real-world phenomena and, indeed, numerous real-world applications are temporal in nature. A large class of these applications deal with continuous, unbounded, high-volume data streams (e.g., Internet traffic pattern study, stock ticker price monitoring, sensor networks monitoring). Thus, methods for temporal processing over data stream are important for those applications.

Existing work on temporal processing over data streams includes temporal data stream mining (e.g., [1] [2]), spatiotemporal data streams (e.g., [3] [4]), temporal aggregate computation over data stream (e.g., [5] [6]), and temporal event detection over data stream (e.g., [7] [8]).

Contrasted with the existing work, the area of our work is the processing of *continuous queries with temporal functions and predicates over windowed data streams*. Queries of this type are very useful in a wide range of stream applications. To the best of our knowledge, however, there has been no study conducted specifically targeting such queries.

We provide below two examples of windowed temporal stream queries, considering a wireless sensor network in which sensors are mounted with weather boards to collect timestamped topology information along with humidity, temperature, etc [9]. We further express the queries using syntax borrowed from CQL [10] and TSQL [11].

Manuscript received May 23, 2010.

[†]Department of Computer Science, The University of Vermont
a) E-mail: malkateb@cs.uvm.edu
b) E-mail: skunta@cs.uvm.edu
c) E-mail: bslee@cs.uvm.edu

DOI: 10.1587/transinf.E93.D.1

Example 1 (Windowed temporal stream join):
Assume a query that detects, at every minute, any two regions of similar humidity values lasting 4 minutes or longer in the past 60 minutes[*]. *The query processor outputs two region IDs and the associated time interval whenever it finds two regions satisfying the condition. Specifically, it performs a non-temporal join on the regionIDs and humidity values, a temporal join with the* OVERLAPS *predicate, and a temporal selection on the* INTERSECT *of two overlapping intervals.*

```
SELECT s1.regionId, s2.regionId,
       VALID INTERSECT(VALID(s1),VALID(s2))
FROM   Stream(regionId, humidity) as s1
       RANGE 60 MINUTE SLIDES 1 MINUTE,
       Stream(regionId, humidity) as s2
       RANGE 60 MINUTE SLIDES 1 MINUTE,
WHERE  s1.regionId != s2.regionId
  AND  s1.humidity ~= s2.humidity //approximately equal
  AND  VALID(s1) OVERLAPS VALID(s2)
  AND  CAST(VALID INTERSECT(VALID(s1),VALID(s2))
       AS INTERVAL MINUTE)(PERIOD) >= 4;
```

Example 2 (Windowed temporal stream aggregation):
The query below detects the longest period of high temperature in the past 24 hours for each region. The query processor outputs the region ID and the maximum duration of temperature being above 100 degrees in each region. The aggregations are computed over a 24-hour window sliding each hour.

```
SELECT s.regionId,
       MAX(CAST(VALID(s.temperature) AS INTERVAL HOUR))
FROM   Stream(regionId, temperature) as s
       RANGE 24 HOUR SLIDES 1 HOUR
WHERE  s.temperature > 100;
```

To answer windowed temporal stream queries like these, there should be a framework that supports the following three aspects: *modeling* the temporal dimension of a windowed data stream, *coalescing* tuples in a window extent, and evaluating *temporal predicates and functions* over coalesced tuples. The temporal dimension models the time at which a fact is true (i.e., valid) in the modeled reality. Coalescing [12] [13] is the process of merging adjacent or overlapping timestamps of value-equivalent tuples. It is a fundamental operation in the temporal data model, and is essential to temporal query processing since queries evaluated on uncoalesced data may generate incorrect answers [12] (see Section 3). A temporal predicate evaluates an interval comparison [14] (e.g., OVERLAPS, CONTAINS, BEFORE, AFTER), and a temporal function returns time points associated with an interval (e.g., VALID, INTERSECT). For temporal functions and predicates, coalescing is a key operation to support.

[*]This is a *self-join* query, but, from the query processing perspective, is equivalent to joining two duplicate streams.

The focus of this paper is on the problem of *coalescing* with respect to the update of a window extent and temporal query processing on the tuples in a window extent.

Two kinds of coalescing schemes exist for temporal database query processing: *eager* coalescing and *lazy* coalescing [15]. Eager coalescing performs coalescing at the time of updating a temporal table, whereas lazy coalescing defers it to query execution time [15]. The former saves time for coalescing during query execution while requiring one coalesced table to be materialized for each set of coalescing attributes specified in the query.

One question is whether and how these two coalescing schemes can be used for temporal *stream* query processing. Simply speaking, it only means substituting a window extent for a temporal table. Unlike database queries, however, stream tuples arrive unboundedly and stream queries run continuously, typically on an ever-changing subset of the stream referred to as a *window extent* [16]. This makes both coalescing schemes disable the basic window extent update algorithms. Thus, in this paper we present new algorithms working correctly with coalescing.

There are tradeoffs between eager coalescing and lazy coalescing in terms of the memory space required to store window extents and the time for updating and querying window extents (see Section 3.2). Given this tradeoff, we address the problem of making an optimal choice between the two schemes for given sets of queries with distinct coalescing attributes. For this purpose, we develop a cost model for estimating the total cost combining the three cost items.

We conduct two sets of performance study of the two coalescing schemes with respect to the total cost. The first set of experiments compares the relative costs between eager and lazy coalescing considering a single query, and the second set of experiments examines interesting cases of the costs when multiple queries are registered to the system.

The main contribution of this paper lies in the study of the coalescing to support continuous queries with temporal functions and predicates over windowed data streams. More specific contributions include defining a temporal data stream model with the coalescing in mind, presenting correct and efficient algorithms for updating window extents with the coalescing in place, and solving the problem of an optimal selection between eager and lazy coalescing. To the best of our knowledge, this is the first work that conducts an in-depth study of coalescing for windowed temporal query processing over data stream.

This rest of this paper is organized as follows. Section 2 describes the temporal data stream model assumed in the presented work. Section 3 provides some relevant discussions on coalescing. Section 4 presents the window extent update algorithms. Section 5 discusses the window extent scan algorithms. Section 6 develops a cost model for the three cost items (memory, update, scan) and formulates the optimal eager-lazy selection problem based on the cost model. Section 7 presents the experiments for performance study. Section 8 reviews related work. Section 9 summarizes the paper and suggests future work.

2. Temporal Data Stream Model

In this section, we present the temporal data stream model assumed in our work. Both the data stream model and the temporal data model are hinged on the common notion of timestamp. Thus, their integration is natural, although the exact modeling of temporal dimension is different – as a sequence of the time instants of (future) tuples arriving in data stream versus a sequence of the time intervals of (past) tuples archived in temporal database. In this section we first summarize these two models briefly, and then describe their integration to a temporal data stream model.

2.1 Data stream model

A data stream is an infinite sequence of tuples [17][18]. Typically, each tuple in a data stream is associated with a timestamp attribute. In many cases only tuples bounded by a window on a data stream are of interest at any given time [17]. A window may be tuple-based or time-based [17][18]. At any time instant t, a tuple-based window of size w (tuples) on a data stream contains tuples with the largest w timestamps not exceeding t and a time-based window of size w (e.g., seconds) contains tuples with the timestamps in the range of $t - w$ to t. The set of physical tuples contained in a window is referred to as a *window extent*, and the specification of a window extent is done through the *window operator* [16]. In other words, a window operator is like a “cookie cutter” and window extents are like “cookies cut” with it [16].

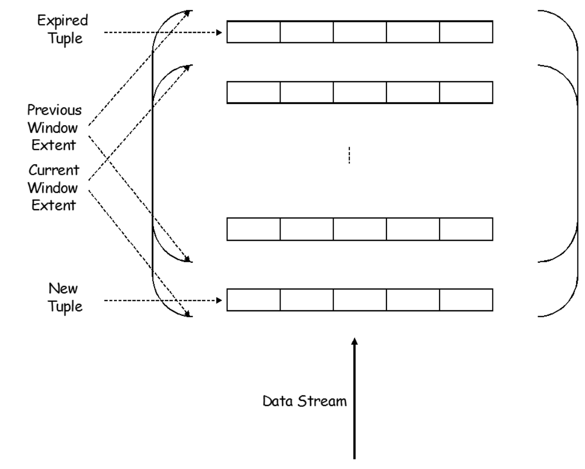
When a new tuple arrives from the data stream, the current window extent is updated by adding the new tuple to it and discarding any expired tuples from it (see Figure 1(a)). While this mechanism is a standard mechanism for handling the arrival of new tuples from an input stream (e.g., [16] [17] [19]), the need for modeling the temporal dimension in data stream tuples demands a different mechanism (see Section 2.3).

We assume tuples may arrive out of order. Our processing model uses the timestamp of a tuple to detect if the tuple has arrived out of order. Though simpler than the processing model of [20] (exploiting punctuation semantics [21]), our model is adequate enough to support the coalescing of tuples arriving out of order. Additionally, we assume that tuples in a window extent are maintained in an increasing order of timestamp with no duplicate timestamp between any pair of tuples in the same stream.

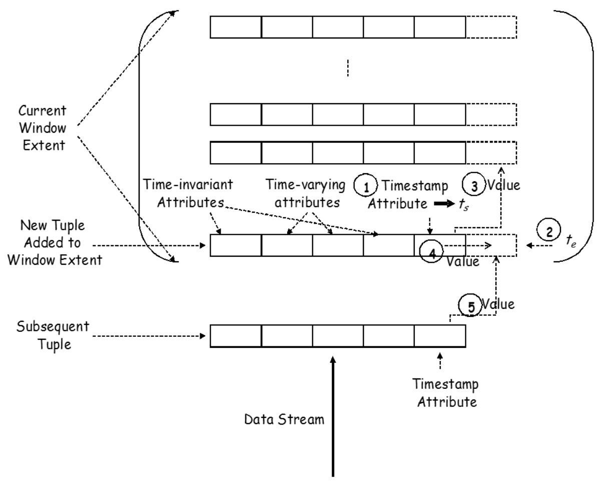
2.2 Temporal data model

In a temporal data model, two orthogonal temporal dimensions are considered [22]: valid time and transaction time. The valid time of a fact is the time at which the fact is true in the modeled reality. The transaction time is the time at which the fact is actually present in the system. We consider the modeling of valid time, which is adequate enough for the purpose of this paper since coalescing is commonly performed on valid time. In addition, a temporal data model may support either attribute timestamping or tuple timestamping [22] depending on whether each timestamp is associated with an attribute or the entire tuple. This paper assumes *tuple timestamping*. Using attribute timestamping is more complex [23], and can be sought as an interesting future work.

2.3 Integration to a temporal data stream model



(a) With Conventional Data Stream Model



(b) With Temporal Data Stream Model

Fig. 1 Updating a Window Extent

In order to support temporal queries over data streams, we propose to model temporal dimension in the data stream model. The model is straightforward, since time is an essential component in both data stream model and temporal data model. Specifically, each tuple arriving from a data stream is timestamped with a left-closed/right-open interval $[t_s, t_e)$, where t_s is the starting instant and t_e is the ending instant of the interval during which the data value of the tuple is valid.

Each tuple arriving in the raw data stream consists of a set of time-invariant attributes (optional), a set of time-varying attributes, and a timestamp attribute. When a new tuple is added to a window extent, the temporal dimension in that tuple is modeled as follows (see Figure 1(b)): (1) its timestamp attribute is used to represent its starting instant, (2) a new attribute is attached to the tuple to represent its ending instant, (3) its starting instant value is assigned to the ending instant of its preceding tuple, (4) its ending instant takes the value of its starting instant until a subsequent tuple arrives, and (5) its ending instant is assigned a value equal to the value of the timestamp attribute of the subsequent tuple. Whenever necessary, we will call the resulting tuples *temporal tuples* to distinguish them from the raw stream tuples. Note that, as mentioned in Section 1, modeling temporal dimension is essential to support temporal query processing and is needed particularly for the coalescing operation.

3. Coalescing

In this section, we discuss some relevant issues on coalescing, with a focus on the importance of coalescing in temporal query processing and the contrast between eager coalescing and lazy coalescing.

The introduction of coalescing for temporal query processing dates back to Böhlen [24]. Generally speaking, coalescing [12] is an operation of reorganizing temporal data for temporal query processing. Specifically, it is the process of merging adjacent or overlapping time intervals of value-equivalent tuples in order to capture the maximal temporal extent of an instance in a temporal relation.

3.1 Importance of coalescing

As shown in several existing work (e.g., [12] [15] [24]), coalescing is a central operation in the temporal data model, and, without it, the correctness of a temporal query processing result is not guaranteed. We provide below two examples which demonstrate this point by showing that temporal queries evaluated on uncoalesced data generate incorrect answers. The first example is for temporal database applications, and the second example is for data stream applications.

Example 3 (Coalescing over Temporal Databases):
Consider the snapshot of a temporal table of employees' data as shown in Table 1. Assume that the manager needs to know the history of Andy's salary. There are three tuples for Andy. The first tuple represents Andy when he was earning $100k, whereas the second and third tuples reflect the time during which Andy's salary has been $120k. Without coalescing, the history of Andy's salary is presented as shown in Table 2 which incorrectly represents the salary of $120k in two separate tuples. With coalescing, however, (see Table 3) the two tuples with the salary of $120k are merged into one single tuple because the timestamps of the two original tuples are adjacent to each other.

Table 1 Temporal table Employee.

Name	Dept	Salary	Start	End
Andy	Development	100k	2000	2004
Andy	Development	120k	2004	2008
Andy	R&D	120k	2008	NOW

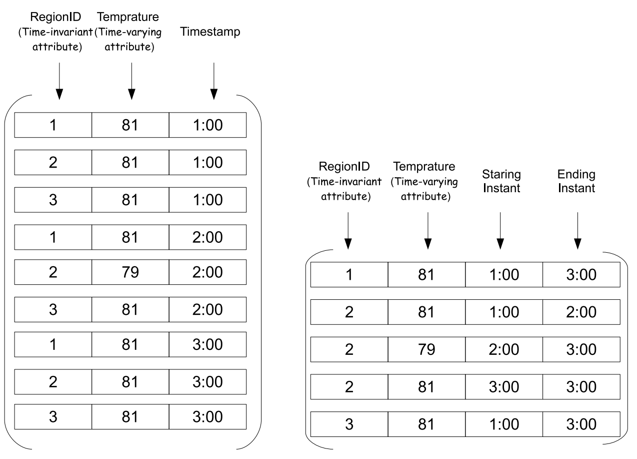
Table 2 History of Andy's Salary without Coalescing.

Name	Salary	Start	End
Andy	100k	2000	2004
Andy	120k	2004	2008
Andy	120k	2008	NOW

Table 3 History of Andy's with Coalescing.

Name	Salary	Start	End
Andy	100k	2000	2004
Andy	120k	2004	NOW

Example 4 (Coalescing over Windowed Data Streams): *Consider the application of wireless sensor networks assumed in Section 1. Assume a query that monitors the change in temperature of different regions with the goal of reporting regions whose temperature did not change for two or more hours. Figure 2 shows the results of coalescing tuples in a window currently holding nine tuples of temperature readings, and contrasts the coalesced window to its counterpart in the un-coalesced window. We can see from the figure that while the un-coalesced window has nine tuples, the coalesced window represents its content with only five tuples. The first tuple represents readings from region* 1 *with temperature value of* 81 *from 1:00 p.m. to 3:00 p.m.; the next three tuples represent readings from region* 2 *and show that temperature in this region fluctuated from* 81 *at 1:00 p.m., to* 79 *at 2:00 p.m., and back up to* 81 *at 3:00 p.m.; the third tuple represents readings from region* 3 *with temperature value of* 81 *from 1:00 p.m. to 3:00 p.m.*


RegionID (Time-invariant attribute)
Temprature (Time-varying attribute)
Timestamp
Staring Instant
Ending Instant

RegionID (Time-invariant attribute)	Temprature (Time-varying attribute)	Timestamp
1	81	1:00
2	81	1:00
3	81	1:00
1	81	2:00
2	79	2:00
3	81	2:00
1	81	3:00
2	81	3:00
3	81	3:00

RegionID (Time-invariant attribute)	Temprature (Time-varying attribute)	Staring Instant	Ending Instant
1	81	1:00	3:00
2	81	1:00	2:00
2	79	2:00	3:00
2	81	3:00	3:00
3	81	1:00	3:00

(a) Un-coalesced (b) Coalesced

Fig. 2 Coalescing over Windowed Data Streams

3.2 Eager versus lazy coalescing

As mentioned in Section 1, there are pros and cons between eager and lazy coalescing schemes.

One the pros side, eager coalescing offers the following advantages. First, it obviates repeated coalescing of the window extent during query processing, specifically window-based join processing, as a join window extent should be scanned repeatedly every time a new tuple arrives at the other stream. Second, eager coalescing during window-extent updates pays off if coalescing is a relatively expensive operation. Third, if the available window buffer space is limited, then eager coalescing is a useful mechanism for reducing the required buffer space.

On the cons side, eager coalescing physically alters the tuples in a window extent and, therefore, if there are two or more temporal queries that require coalescing on different time-varying attributes, then there must be separate coalesced window extents maintained. The storage overhead and the window extent update overhead in this case may make the eager coalescing not worthwhile.

4. Updating a Window Extent

In this section, we present algorithms for updating a window extent in the presence of eager or lazy coalescing while preserving the correct window semantics.

The algorithms designed make the following assumptions. First, tuples may arrive out of the order of timestamp, which may necessitate retracting some of the coalesced tuples. Second, different tuples may belong to different groups, and only tuples belonging to the same group are coalesced (e.g., humidity readings are grouped by regionID). Third, tuples are always memory-resident (with no overflow to disk), thus demanding frugal consumption of memory space. Fourth, no index is used to access tuples in a window extent, which lifts the overhead of updating an index as tuples arrive (and possibly coalesced) but brings the need to linear-scan the window extent for query processing.

4.1 Influence of coalescing on the basic window extent update algorithms

The basic algorithms for updating a window extent work by simply adding new tuples to a window extent and discarding expired tuples from it. However, in the presence of eager coalescing, these basic algorithms no longer work. The reason is that with eager coalescing one or more tuples may be coalesced to form a single tuple in the window extent. In this case, for example if some but not all of the coalesced original tuples expire, then there is no correct way the basic algorithms can handle it. If they discard the coalesced tuple, they have removed the tuples that have not expired yet. If they do not, then they have retained the tuples that have expired. In either case, the query results are incorrect. Careful changes of the basic algorithms are required to ensure the correctness.

On the other hand with lazy coalescing, the impact is smaller. Each time a new tuple arrives, the structure of the tuple is altered with an additional attribute attached to represent its ending instant and the value of its timestamp is assigned to the ending instant of the previous tuple. Changes required for this alteration are less involving.

4.2 Window extent update algorithms

We present the algorithms targeting tuple-based windows and discuss the modifications necessary for time-based windows. (See Appendix A for basic functions needed by these algorithms.).

4.2.1 Update with lazy coalescing

In the lazy coalescing scheme, the algorithm of updating a window extent is a direct extension of the basic window extent update algorithm. It employs only the idea of updating the starting and ending instants of tuples, and defers any coalescing to the query execution time.

Updating a tuple-based window extent

The algorithm works in two phases upon the arrival of a new tuple (see Algorithm 1). The first phase concerns discarding an expired tuple from the window extent, and the second phase concerns adding the new tuple into the window extent. In the first phase, the algorithm simply removes the oldest tuple from the window extent and updates s_{oldest} to the new oldest tuple (Lines 1–3). In the second phase, if the new tuple has arrived too late (i.e., has arrived out of order and should have expired), it is discarded (Line 7). If the new tuple has arrived out of order but still should be added to the window extent, then the algorithm finds where to insert the tuple in the window extent and adds it (Lines 9–10). If the new tuple has arrived in order, then the algorithm adds the new tuple at the end of the window extent and updates s_{latest} to the new latest tuple (Lines 14–15).

Updating a time-based window extent

The only differences from updating a tuple-based window are in the way old tuples are identified for deletion and in the way the belatedness of an out-of-order arrival tuple is determined. Thus, we only need to replace Lines 1–3 of Algorithm 1 by Lines 1–4 shown below and replace the if-condition in Line 6 of Algorithm 1 by $s_{new}.ts < s_{new}.ts - T$, where T is the time-based window size.

```
1: while ($s_{oldest}.ts < s_{new}.ts - T$) do
2:     $s_{oldest} = s_{oldest}.next()$
3:     remove $s_{oldest}.prev()$
4: end while
```

4.2.2 Update with eager coalescing

In the eager coalescing scheme, the algorithm of updating a window extent is based on three key ideas. First, each tuple is attached with a timestamp vector (denoted as **v**) which comprise the timestamps of all subsequent tuples coalesced with that tuple. Remembering the timestamps of the coalesced tuples is important in order to correctly discard tuples when they expire. Second, updating a window extent is achieved primarily by manipulating the timestamp vectors. That is, adding a new tuple to the current window extent may not necessarily result in an actual insertion of the new tuple. Instead, it may result in merging the new tuple with an existing tuple and updating the timestamp vector of that existing tuple. Similarly, discarding an expired tuple from the current window extent may not necessarily result in an actual removal of the tuple. Instead, it may result in only updating the timestamp vector of an exiting tuple. Third, if a new tuple arrives out of order, it may either be merged with an existing tuple or cause an existing tuple to split into two tuples.

Algorithm 1 *tupleBasedLazyUpdate(s_{new})*

```
Inputs: $s_{new}$ // a new arrival tuple
// Global variables:
// $s_{oldest}$: the tuple with the smallest $t_s$ in the window extent
// $s_{latest}$: the tuple with the largest $t_e$ in the window extent
    // Discard the oldest tuple
1: $s_{next\_oldest} = s_{oldest}.next()$
2: remove $s_{oldest}$
3: $s_{oldest} = s_{next\_oldest}$
    // Add the new tuple
4: if $s_{new}.ts < s_{latest}.t_s$ then
5:     // $s_{new}$ has arrived out of order
6:     if $s_{new}.ts < s_{oldest}.t_s$ then
7:         discard $s_{new}$    // arrived too late
8:     else
9:         find the tuple $s_{prev}$ next to which $s_{new}$ should be added, i.e.,
           $s_{prev}.t_s < s_{new}.ts < s_{prev}.next().t_e$
10:        $addTuple(s_{new}, s_{prev})$
11:    end if
12: else
13:    // $s_{new}$ has arrived in order
14:    $addTuple(s_{new}, s_{latest})$
15:    $s_{latest} = s_{new}$
16: end if
```

Updating a tuple-based window extent

The algorithm works in two phases upon the arrival of a new tuple (see Algorithm 2). The first phase of the algorithm concerns discarding the expired tuple from the window extent. If the oldest tuple in the window is currently coalesced with other tuples, then the expired tuple is removed by updating the starting instant of the oldest tuple with the smallest timestamp value in its timestamp vector (Lines 2–3). Otherwise it is directly removed from the window extent and s_{oldest} is updated to the new oldest tuple (Lines 5–7).

The second phase concerns adding the new tuple into a window extent. If the new tuple has arrived too late, that is, has arrived out of order and should have expired, then it is discarded (Line 12). If it has arrived out of order but still

Algorithm 2 *tupleBasedEagerUpdate*(s_{new})

```
Inputs: $s_{new}$ // a new arrival tuple
// Global variables:
// $s_{oldest}$: the tuple with the smallest $t_s$ in the window extent
// $s_{latest}$: the tuple with the largest $t_e$ in the window extent
    // Discard the oldest tuple
 1: if $s_{oldest}.\mathbf{v}.size() > 0$ then
 2:     $s_{oldest}.t_s = s_{oldest}.\mathbf{v}.first()$
 3:     remove $s_{oldest}.\mathbf{v}.first()$ from $s_{oldest}.\mathbf{v}$
 4: else
 5:     $s_{next\_oldest} = s_{oldest}.next()$
 6:     remove $s_{oldest}$
 7:     $s_{oldest} = s_{next\_oldest}$
 8: end if
    // Add the new tuple
 9: if $s_{new}.ts < s_{latest}.t_s$ then
10:     // $s_{new}$ has arrived out of order
11:     if $s_{new}.ts < s_{oldest}.t_s$ then
12:         discard $s_{new}$    // arrived too late
13:     else
14:         find the tuple $s_{target}$ that overlaps $s_{new}$, i.e., $s_{target}.t_s < s_{new}.ts <$
            $s_{target}.t_e$, and belongs to the same group as $s_{new}$
15:         if $s_{new}$ is value-equivalent with $s_{target}$ then
16:             $mergeTuples(s_{new}, s_{target})$
17:         else
18:             $s_{prev} = splitTuple(s_{target}, s_{new}.ts)$
19:             $addTuple(s_{new}, s_{prev})$
20:         end if
21:     end if
22: else
23:     // $s_{new}$ has arrived in order
24:     find the latest tuple $s_{latest}$ that belongs to the same group as $s_{new}$
25:     if $s_{new}$ is value-equivalent with $s_{latest}$ then
26:         $mergeTuples(s_{new}, s_{latest})$
27:     else
28:         $addTuple(s_{new}, s_{latest})$
29:         $s_{latest} = s_{new}$
30:     end if
31: end if
```

should be in the window extent, then the algorithm scans the window extent to find a tuple that overlaps the new tuple (that is, a tuple whose time interval contains the timestamp of the new tuple) and belongs to the same group (Line 14). If the two tuples should be coalesced, then they are merged (Line 16). Otherwise, the overlapping tuple is split (Line 18) and the new tuple is inserted between the two split tuples (Line 19). If the new tuple has arrived in order, then it scans the window extent to find the latest tuple that belongs to the same group (Line 24). If the two tuples should be coalesced, then they are merged (Line 26). Otherwise the new tuple is added at the end of the window extent (Line 28) and s_{latest} is updated to the new latest tuple (Line 29).

Updating a time-based window extent

In the same manner as the case of lazy coalescing, the only modifications needed are to replace Lines 1–8 of Algorithm 2 by Lines 1–7 shown below and replace the if-condition in Line 11 of Algorithm 2 by $s_{new}.ts < s_{new}.ts - T$, where T is the time-window size.

```
1: while ($s_{oldest}.ts < s_{new}.ts - T$) do
2:     if ($s_{oldest}.v.size() > 0$ and $s_{oldest}.t_e > s_{new}.ts - T$) then
3:         $s_{oldest}$ = splitTuple($s_{oldest}, s_{new}.ts - T$)
4:     end if
5:     $s_{oldest} = s_{oldest}.next()$
6:     remove $s_{oldest}.prev()$
7: end while
```

5. Scanning a Window Extent

We use window extent scan as an operation that generally represents the window-based temporal query processing over data streams. Temporal join queries are of particular interest in terms of using a window. Temporal predicates are typically defined on intervals, and joins on these interval predicates are essentially non-equijoins. For example, the temporal join predicate `VALID(s1) OVERLAPS VALID(s2)` in Example 1 is equivalent to the following non-equijoin predicate `s1.`t_s < `s2.`t_e `AND s2.`t_s < `s1.`t_e. A window scan is a common operation in non-equijoin processing like this. Temporal aggregation queries may as well require window extent scan, especially for selective aggregation functions like MAX, MIN, and MEDIAN. For example, the windowed aggregation in Example 2 requires scanning the entire window whenever a tuple with the maximum temperature value is discarded from the window extent.

Thus, we use the window extent scan cost as an objective counterpart of the window extent update cost in the performance study (Section 7)

The window extent scan algorithms are simple. They involve a linear scan of all tuples in the window extent for both window types regardless of the coalescing approach. (The scan could be an index-based scan if an index were available, but in the current work an index is not considered on a window extent which is memory-resident.) The only distinction in our work is that, in the case of lazily-coalesced window extent, there is an additional overhead of coalescing tuples during query execution. This overhead increases linearly with the number of tuples in the window extent and the coalescing probability.

6. Optimal Selection between Eager and Lazy Coalescing

From the tradeoffs observed in Section 3.2, it is evident that the costs of storing, updating, and scanning window extents are the key cost items affecting the choice between eager coalescing and laze coalescing. In this section we formulate the problem of choosing between the two coalescing schemes to optimize an objective defined as a combination of the three cost items. We first develop a cost model of the three cost items and then formulate the optimal selection problem based on the model. The cost model presented here assumes a tuple-based window. Only the cost model needs to be replaced for the same optimization framework to work with a time-based window. Table 4 summarizes the notations used in the cost model.

Table 4 Notations used in the cost model.

Symbol	Description
r^{eager}	tuple size in eagerly-coalesced window extent (bytes)
r^{lazy}	tuple size in a lazily-coalesced window extent (bytes)
m	timestamp size (bytes)
w	window extent size (number of tuples)
i	tuple index ($i \geq 1$)
cp_i	probability that the i^{th} tuple coalesces with the $(i-1)$th tuple of the same group ($i \geq 2$)
oop_i	probability that the i^{th} tuple arrives out of order ($i \geq 1$)
k	the number of tuples that have arrived so far ($k \geq 1$)
C_{read}	cost of reading a specific tuple in the window extent
C_{insert}	cost of inserting a tuple at a specific position in the window extent
C_{delete}	cost of deleting the oldest tuple from the window extent
C_{attach}	cost of attaching a timestamp to a specific tuple
C_{detach}	cost of detaching the first timestamp from the timestamp vector of the oldest tuple
C_{merge}	sum of the cost of finding the target tuple and the cost of *mergeTuples*, i.e., merging the timestamp of the new tuple into the timestamp vector of the target tuple
C_{split}	sum of the cost of finding the target tuple and the cost of *splitTuple*, i.e., splitting the target tuple and its timestamp vector and inserting the new tuple
$M^{eager}(k)$	memory consumption for storing a window extent of k eagerly-coalesced tuples
$M^{lazy}(k)$	memory consumption for storing a window extent of k lazily-coalesced tuples
$U^{eager}(k)$	time for updating a window extent with the first k tuples using eager coalescing
$U^{lazy}(k)$	time for updating a window extent with the first k tuples using lazy coalescing
$S^{eager}(k)$	time for scanning a window extent containing k tuples stored with eager coalescing
$S^{lazy}(k)$	time for scanning a window extent containing k tuples stored with lazy coalescing

(C_{merge} and C_{split} include extra costs for setting up and following up the operations specified in the functions *mergeTuples* and *splitTuple* (Section Appendix A).)

6.1 Cost model

6.1.1 Window extent memory consumption

In the case of eager coalescing (Equation 1), for each new tuple, the memory consumption increases by the timestamp size if coalescing occurs and by the tuples size if not. For each expiring tuple, the memory consumption decreases by the timestamp size if the oldest tuple was coalesced with its subsequent tuples and by the tuple size otherwise. In the case of lazy coalescing (Equation 2), the memory consumption increases linearly with the number of new tuples until the window becomes full and then remains constant.

$$\begin{aligned} M^{eager}(k) \;=\; & \sum_{i=1}^{k}((cp_i\, m + (1-cp_i)\, r^{eager}) \\ & -(cp_{i-w}\, m + (1-cp_{i-w})\, r^{eager})) \end{aligned} \quad (1)$$

where $k \geq 1$ and $cp_{i-w} = 0$ for $k \leq w$.

$$M^{lazy}(k) = \min(k, w)\, r^{lazy} \text{ where } k \geq 1 \quad (2)$$

Note that a tuple in a lazy-coalesced window extent has all the attributes in an incoming tuple of the stream, whereas in an eager-coalesced window extent, it has only the attributes required by the query. Thus, r^{eager} is no larger than r^{lazy}.

6.1.2 Window extent update time

In the case of eager coalescing (Equation 3), for each new tuple the update time comprises the costs of either inserting it uncoalesced (if it arrives in order and does not coalesce with an existing tuple), merging it with an existing tuple (if it arrives in order and it does coalesce with that existing tuple), inserting it and splitting an existing tuple (if it arrives out of order and it does not coalesce with that existing tuple), or merging it with an existing tuple (if it arrives out of order and it does coalesce with that existing tuple). In addition, once the number of tuples exceeds the window size, then there is the additional cost of either uncoalescing the oldest tuple and detaching the timestamp from its timestamp vector (if it was coalesced) or simply discarding the oldest tuple (if not). In the case of lazy coalescing (Equation 4), the update time comprises the costs for inserting a new tuple and, if the number of tuples exceeds the window size, then the cost of deleting the oldest tuple.

$$\begin{aligned} U^{eager}(k) \;=\; & \sum_{i=1}^{k}((1-oop_i)(1-cp_i)\, C_{insert} + (1-oop_i)\, cp_i\, C_{merge} \\ & + oop_i\,(1-cp_i)\,(C_{insert} + C_{split}) + oop_i\, cp_i\, C_{merge} \\ & + (cp_{i-w})\, C_{detach} + (1 - cp_{i-w}\, C_{delete})) \end{aligned} \quad (3)$$

where $k \geq 1$ and $cp_{i-w} = 0$ for $k \leq w$.

$$U^{lazy}(k) = \begin{cases} C_{insert} & \text{if } k \leq w \\ C_{insert} + C_{delete} & \text{if } k > w \end{cases} \quad (4)$$

6.1.3 Window extent scan time

The window extent scan involves reading all tuples in the window extent[†]. In the case of eager coalescing (Equation 5), all tuples in the current window extent are simply scanned linearly. The number of tuples can be computed by dividing the current window extent's memory consumption by the tuple size. In the case of lazy coalescing (Equation 6), there is the overhead of coalescing tuples during query execution. The time for reading each tuple during the scan depends on whether tuple coalesces with the previous tuple or not.

$$S^{eager}(k) = C_{read}\, \frac{M^{eager}(k)}{r^{eager}} \quad (5)$$

$$S^{lazy}(k) = \sum_{i=1}^{k}(cp_i\,(C_{coalesc} + C_{read}) + (1-cp_i)\, C_{read}) \quad (6)$$

[†]In the current work, an index is not considered on a window extent which is memory-resident.

6.2 Optimal selection problem formulation

Given multiple concurrent temporal queries running against the same window, we can consider a set of all queries, each specifying time-variant coalescing attributes and time-invariant grouping attributes. We call it a *distinct coalescing query (DCQ) set*. For example, if one query Q_1 specifies coalescing on attributes $\{A, B\}$ grouped by $\{G_{11}, G_{12}\}$, two other queries Q_2 and Q_3 specify coalescing on attributes $\{B, C\}$ grouped by $\{G_2\}$, and another query Q_4 specifies coalescing on the same attributes $\{B, C\}$ but grouped by $\{G_3\}$, then $\{(\{A, B\}, \{G_{11}, G_{12}\}), (\{B, C\}, \{G_2\}), (\{B, C\}, \{G_3\})\}$ is the DCQ set from the four queries.

There is one window extent needed for each DCQ that is subject to eager coalescing, as the eager coalescing physically alter the tuples in it, and one lazily-coalesced window extent is needed for all the other DCQs. Note that the number of window extents that should be maintained is thus no more than the cardinality of the DCQ set. (They are equal if and only if eager coalescing is used for all the queries.)

The window extent size has a different unit from the other two cost items. Thus, we normalize them to values in the range of [0,1] by dividing them by the maximum possible values (M_{max}, U_{max}, and S_{max}). M_{max} is the window extent size needed when no coalescing is done, U_{max} is the window extent update time when all tuples are coalesced (then there is only one tuple in the window extent), and S_{max} is the window extent scan time when all tuples are coalesced during the scan.

Let us denote the normalized window extent size, update time, and scan time as μ, υ, and σ, respectively. Then, they are expressed as follows for the jth DCQ depending on the coalescing scheme.

$$\mu_j^{eager}(k) = \frac{M_j^{eager}(k)}{M_{max}};\ \upsilon_j^{eager}(k) = \frac{U_j^{eager}(k)}{U_{max}};\quad \sigma_j^{eager}(k) = \frac{S_j^{eager}(k)}{S_{max}} \text{ for } j \in \mathcal{S}' \tag{7}$$

$$\mu^{lazy}(k) = \frac{M^{lazy}(k)}{M_{max}};\ \upsilon^{lazy}(k) = \frac{U^{lazy}(k)}{U_{max}};\quad \sigma_j^{lazy}(k) = \frac{S_j^{lazy}(k)}{S_{max}} \text{ for } j \in \mathcal{S} - \mathcal{S}' \tag{8}$$

where $\mathcal{S}$ is the DCQ set and $\mathcal{S}'$ is the set of DCQs whose window extents are coalesced eagerly.

These three normalized quantities (i.e., μ, υ, and σ) participate in the optimal selection problem formulation as follows. For each DCQ in $\mathcal{S}'$, there exists one window extent eagerly coalesced on the attributes of that query and, thus, the costs of storing, updating, and scanning tuples are incurred for each window extent. We express this cost as a weighted sum of the three normalized quantities, that is,

$$Cost_j^{eager}(k) = \lambda_\mu \mu_j^{eager}(k) + \lambda_\upsilon u_j^{eager}(k) + \lambda_\sigma \sigma_j^{eager}(k) \tag{9}$$

In contrast, there exists one window extent for all DCQs in $\mathcal{S} - \mathcal{S}'$ and, thus, the costs of storing and updating tuples are incurred for the single window extent, but the cost of scanning tuples is incurred for each DCQ. The total cost for all DCQs in $\mathcal{S} - \mathcal{S}'$ is also expressed as a weighted sum as well, that is,

$$Cost^{lazy}(k) = \lambda_\mu \mu^{lazy}(k) + \lambda_\upsilon \upsilon^{lazy}(k) + \sum_{j \in \mathcal{S}-\mathcal{S}'} Scan\_cost_j^{lazy}(k) \tag{10}$$

where

$$Scan\_cost_j^{lazy}(k) = \lambda_\sigma \sigma_j^{lazy}(k)$$

Based on these cost expressions, the optimal selection problem of determining which DCQs are subject to eager coalescing (leaving the rest to lazy coalescing) is stated as follows.

> **Optimal selection problem**: Given a DCQ set $\mathcal{S}$ and the number k of tuples that have arrived so far, find a subset $\mathcal{S}'$ of $\mathcal{S}$ that minimizes the cost computed as
>
> $$\sum_{j \in \mathcal{S}'} Cost_j^{eager}(k) + \sum_{j \in \mathcal{S}-\mathcal{S}'} Scan\_cost_j^{lazy}(k)$$

Note that the cost terms of lazy coalescing (i.e., $\lambda_\mu \mu^{lazy}(k)$ and $\lambda_\upsilon \upsilon^{lazy}(k)$) are independent of the DCQ set j and, thus, are not part of the cost objective function. Besides, their values are zero if there is no DCQ subject to lazy coalescing.

Theorem 1: The optimal selection between lazy and eager coalescing for multiple concurrent queries is a 0-1 integer programming problem.

Proof Sketch: The proof can be easily seen by defining $x_j = 1$ if $j \in \mathcal{S}$ and 0 otherwise (i.e., $j \in \mathcal{S} - \mathcal{S}'$). Then, the problem can be rewritten to that of minimizing $\sum_{j=1}^{n}(c_j x_j + d)$ where $n = |\mathcal{S}|$, $x_j \in [0, 1]$, $c_j = Cost_j^{eager}(k) - Scan\_cost_j^{lazy}(k)$, and $d = \sum_{j=1}^{n} Scan\_cost_j^{lazy}(k)$.

7. Performance Study

We study the performances of eager and lazy coalescing with respect to the costs formulated as a weighted sum of the three normalized cost items (see Equations 9 and 10).

The performance study consists of two sets of experiments. In the first set of experiments, we compare the two coalescing mechanisms for different coalescing probabilities and out-of-order probabilities under different memory, update, and scan cost weights (i.e., $\lambda_\mu, \lambda_\upsilon, \lambda_\sigma$) when a single query is registered. We assume that coalescing and out-of-order probabilities are learned and captured during the processing of input data streams. In the second set of experiments, we study interesting cases in the optimal selection problem by comparing the costs resulting from different selections when multiple queries are registered: eager coalescing for all queries, lazy coalescing for all queries, and an optimal selection as described in the previous section.

All experiments are implemented in C++ on a laptop with Windows Vista running on Intel Core2 Duo 2 GHz CPU and 3 GB RAM.

In this section, we describe the setup of data sets used in the experiments in Section 7.1 and present the two sets of experiments and their results in Section 7.2 and Section 7.3, respectively, summarize the results in Section 7.4, and share some practicality considerations in Section 7.5.

7.1 Data sets

Synthetic data sets

We generate synthetic data streams of 200, 000 timestamped tuples. The timestamp size is eight bytes. Each tuple has three time-varying attributes and three time-invariant attributes, each of size four bytes. Time-invariant attributes are used for grouping.

As mentioned in Section 1, we use the coalescing probability (*cp*) and the out-of-order probability (*oop*) as the control parameters in the experiments.[†] Thus, we generate multiple data streams with different coalescing probabilities and out-of-order probabilities (each ranging from 0 to 1 with an interval of 0.1). In contrast, the number of groups has no effect on the relative costs between eager and lazy coalescing, and thus we set the number of groups to single values 3, 5, and 7 for the three time-invariant attributes, respectively.

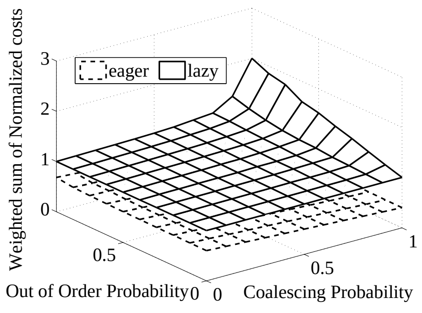
Real data set

We use a real data set collected from sensors deployed in the Intel Berkeley Research lab between February 28th and Aprils 5th, 2004 [9]. Sensors mounted with weather boards collected timestamped topology information, along with humidity, temperature, light and voltage values once every 31 seconds. Collection of data was done using the TinyDB query processing system, built on the TinyOS platform. The resulting data file includes a log of about 2.3 million readings collected from these sensors. The schema of records is ⟨date: yyyy-mm-dd, time: hh:mm:ss.xxx, epoch: int, moteid: int, temperature: real, humidity: real, light:real, voltage:real⟩. In this schema, epoch is a monotonically increasing sequence number unique for each mote. Temperature is in degrees Celsius. Humidity is temperature-corrected relative humidity, ranging from 0 to 100%. Light is in Lux (1 Lux is about moonlight, 400 Lux about a bright office, and 100,000 Lux about full sunlight.) Voltage is expressed in volts, ranging from 2.0 to 3.0 volts.

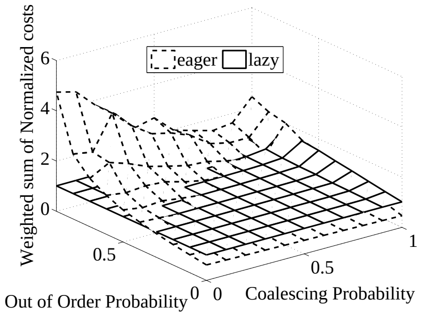
[†]Four other parameters were considered as well – window extent size (i.e., number of tuples), stream data set size (i.e., number of tuples), stream rate, and the number of groups. Among them, the window extent size shows insignificant effect as costs calculated are normalized, and the stream data set size and stream rate do not affect the costs at all as the cost is measured *per tuple* in a window extent. In the case of the number of groups, it affects only the coalescing probability and, since the coalescing probability is already considered a tuning parameter, it loses significance.

7.2 Experiments 1: comparison between eager and lazy coalescing costs

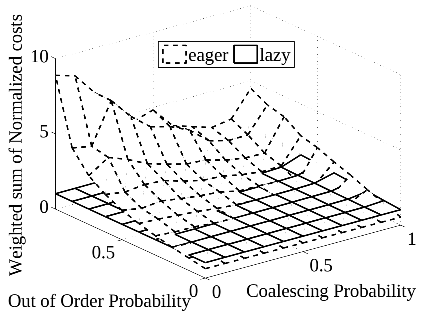
To visualize the expected performances of the algorithms, we plot the costs of eager coalescing and lazy coalescing obtained using the cost model (in Section 6.1) while varying the coalescing and out-of-order probabilities for different weights of memory cost(λ_μ), update cost(λ_υ) and scan cost(λ_σ). These three costs are the actual values measured by running the algorithms (see Sections 4 and 5) on synthetic data sets.



(a) Weights:(1,1,1)



(b) Weights:(1,50,1)



(c) Weights:(1,100,1)

Fig. 3 Costs of eager and lazy coalescing for an increasing update cost weight.

Before we present the experimental results, let us first discuss how the costs of different per-tuple atomic operations (Table 4) vary with respect to the coalescing and out-of-order probabilities. (We implement a window extent as an array data structure in this performance study.)

- C_{read}: This cost depends only on the size of a specific tuple. Since the tuple size is constant in lazy coalescing but increases linearly with the coalescing probability in eager coalescing as more timestamps are added to the time stamp vector of a tuple, C_{read} varies in the same manner with the coalescing probability. Evidently, the out-of-order probability has no effect on this cost.
- C_{insert} and C_{delete}: These two operations are limited to a specific single tuple and, thus, are irrelevant to coalescing and out-of-order probabilities.
- C_{detach} and C_{attach} : These costs are applicable only in the case of eager coalescing and remains constant for the same reason as C_{insert} and C_{delete}.
- C_{merge}: If the new tuple to be merged is not arriving in order, then we have to search the window for the appropriate position in the window extent and also search the timestamp vector of that tuple for finding the appropriate position to insert the timestamp of the new tuple. Thus, this cost increases with the out-of-order probability. Additionally, if the coalescing probability is higher, then the timestamp vector will be larger and so the cost of merging is higher. Thus, C_{merge} increases monotonously with $oop \times cp$.
- C_{split}: This cost is applicable only for out-of-order tuples and increases monotonously with the out-of-order probability. On the other hand, the cost decreases monotonously with the coalescing probability because the number of tuples to be moved to create a room in the window extent (with the array data structure) is smaller when more tuples have been coalesced. Thus, C_{split} increases monotonously with oop / cp.

Now, we present the experimental results. We have conducted this set of experiments with a wide range of different weight combinations of memory, update and scan costs. We show a few interesting cases here.

Among the three cost weights ($\lambda_\mu, \lambda_\upsilon, \lambda_\sigma$), increasing λ_υ gives an advantage to lazy coalescing whereas increasing λ_μ or λ_σ gives an advantage to eager coalecing. Thus, in this experiment, we increase λ_υ relative to λ_μ and λ_σ, and observe the performance trend. Figures 3(a) through 3(c) show the costs of eager coalescing and lazy coalescing when $\langle\lambda_\mu, \lambda_\upsilon, \lambda_\sigma\rangle$ is $\langle 1, 1, 1\rangle$, $\langle 1, 50, 1\rangle$, and $\langle 1, 100, 1\rangle$, respectively. These weight combinations refer to the cases of an increasingly higher weight given to the update cost.

¿From these figures we make two observations. First, when the update cost weight (λ_υ) is comparable to the memory cost weight (λ_μ) and the scan cost weight (λ_σ) (see Figure 3(a)), eager coalescing outperforms lazy coalescing in the entire ranges of *oop* and *cp*. The reason is that eager coalescing costs less than lazy coalescing in two (i.e., memory and scan) of the three cost items. Note that all three costs are normalized (to maximum 1.0).

Second, the three figures show a trend of the relative performance between eager coalescing and lazy coalescing. The closer the (*cp*, *oop*) pair is to $(0.0, 0.0)$, the lower the cost of eager coalescing compared to lazy coalescing. In addition, the crossover line moves toward the point $(1.0, 0.0)$ as the update cost weight increases. This indicates that the cost of eager coalescing increases faster at $(0.0, 1.0)$ than any other point. To understand the reason for this trend, let us study the behavior of the graphs shown in these figures in detail. From Equation 3 and our discussion of the costs of the atomic operations above, we see that *mergeTuples* and *splitTuples* are the only operations contributing to the change in cost when the update weight is changed. Specifically, the following two observations hold. First, for a given *cp*, the update cost of eager coalescing, U^{eager}, increases with the increase of *oop*. The reason for this is quite clear from the fact that, as mentioned above, C_{merge} and C_{split} increase monotonously with $cp \times oop$ and oop/cp, respectively. Second, for a given *oop*, U^{eager} decreases with the increase of *cp*. The reason for this can be explained as follows. With the increase of *cp*, C_{merge} increases but C_{split} decreases. In this performance study, C_{split} is a more dominant factor than C_{merge} in the cost of eager coalescing because of the current *array* implementation of a window extent. We can deduce from these two observations that whatever the update weight may be, eager coalescing cost is the highest at (*cp*,*oop*) = (0.0, 1.0) and the lowest at (1.0, 0.0), and as the update weight increases the crossover line moves toward (1.0, 0.0) as visualized in Figure 3(a) through Figure 3(c).

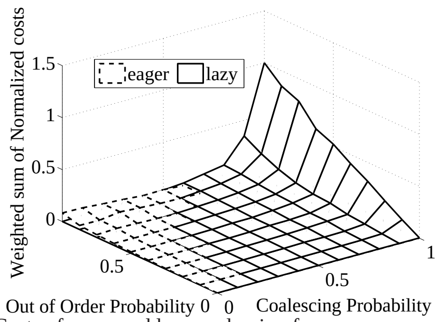


Fig. 4 Costs of eager and lazy coalescing for no memory cost weight.

As a special case, Figure 4 shows the costs of eager coalescing and lazy coalescing when λ_μ=0, λ_υ=1 and λ_σ=1. This combination of weights refers to the case in which the system has sufficient memory and update and scan costs are given equal priority. As expected, eager coalescing and lazy coalescing show more or less the same costs, with slight differences in a region represented by lower *cp* and higher *oop*. This is because the difference in normalized scan costs (for which lazy > eager) is countered well by the difference in normalized update costs (for which eager > lazy) in this region.

7.3 Experiments 2: optimal selection between eager and lazy coalescing

For this set of experiments, we consider a scenario of six queries with the following DCQ setup: { ({h}, {ep}), ({t}, {md}), ({v}, {lt}), ({h, t}, {ep, md}), ({t, v}, {md, lt}), ({h, t, v},

Table 5 Scenarios used for the real data set

	$cp_1, cp_2, cp_3, cp_4, cp_5, cp_6$	**Result**	**Selection between eager and lazy**
Case 1	0.1, 0.1, 0.1, 0.1, 0.1, 0.1	all lazy = optimal < all eager	lazy on all queries
Case 2	0.8, 0.8, 0.8, 0.8, 0.8, 0.8	all eager = optimal < all lazy	eager on all queries
Case 3	0.5, 0.5, 0.5, 0.1, 0.1, 0.1	optimal < all lazy < all eager	eager on Q_1, Q_2, Q_3 and lazy on Q_4, Q_5, Q_6
Case 4	0.8, 0.8, 0.8, 0.1, 0.1, 0.1	optimal < all eager < all lazy	eager on Q_1, Q_2, Q_3 and lazy on Q_4, Q_5, Q_6

(cp_i denotes the coalescing probability for query Q_i.)

$\{ep, md, lt\})$ } where h, t, v, ep, md, and lt refer to humidity, temperature, voltage, epoch, mote ID, and light, respectively.[†] We attain the required coalescing probabilities for these experiments by quantifying the coalescing attribute values with an appropriate quantum size. All tuples are in order in the real data set and, thus, out-of-order probability is zero. $\langle \lambda_\mu, \lambda_\upsilon, \lambda_\sigma \rangle$ are considered to be $\langle 1, 1, 2 \rangle$. When multiple queries are registered in the system, memory and update costs give favor to lazy coalescing while scan cost gives favor to eager coalescing. Additionally, in the case of lazy coalescing, at least two scans are needed for any operation on the window extent, one for coalescing the window and the other for performing the requisite operation according to the query. This justifies assigning weights $\langle 1, 1, 2 \rangle$.

We measure the costs of the per-tuple atomic operations C_{read}, C_{insert}, C_{delete}, C_{merge}, C_{split}, and C_{detach} by repeating each operation 1000 times and averaging it over all the tuples of the stream data set.

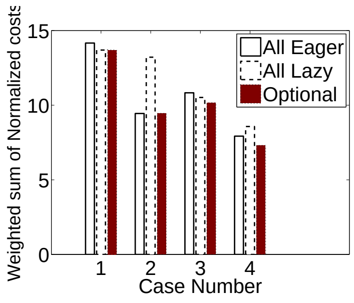
The scenario shown in Table 5 are used for the real data set. In this scenario setup, the six queries are divided into two groups (Q_1, Q_2, Q_3 and Q_4, Q_5, Q_6) and high (= 0.8), medium (=0.5), or low (=0.1) coalescing probability is assigned to each group with different cases of combination. Figure 5(a) summarizes the resulting costs of all-lazy, all-eager, and optimal selections in each case.

We can see from the selection decisions in Table 5 that the optimal solution for the given set of queries suggests eager coalescing for queries having higher coalescing probability and lazy coalescing for queries having lower coalescing probability. This phenomenon is mainly due to the fact that when the coalescing probabilities are high the normalized memory costs and scan costs for eager coalescing are very low and hence can negate the extra update cost for performing eager coalescing.

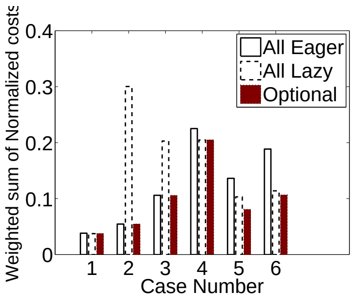
Now, let us see some scenarios where oop plays a role. For this we consider the synthetic data sets, as they allow us to change the out-of-order probability. We consider the same scenario of six queries except that the attributes are integers. Weights on the update and scan costs are considered to be equal while the weight on the memory cost is assumed to be zero in order to show the impact of oop more clearly. (Note that oop does not affect the memory cost while cp does.) Like the cases in Table 5, all the cases shown in Table 6 compare the costs of performing queries with coalescing all lazy, all eager, and the optimal selections.

In this scenario setup, the queries are tested with three different sets of cases. In the first set (comprising Cases 1 and 2), we assign low $cp(= 0.2)$ in Case 1 and high $cp(= 0.7)$ in Case 2 and vary oop from 0.0 to 1.0. Due to space constraint and insignificant difference in the selection results, we show only the results for two oop values (= 0.3, 0.6). In the second set (comprising Cases 3 and 4), we set cp to a medium value (= 0.5) and vary oop from 0.0 to 1.0. We observe that at oop = 0.7 there is a change in the selection result from eager to lazy. We examine this behavior further in the third set of cases (comprising Cases 5 and 6), where we set cp to a value similar to the value used in Case 3 of Table 5 and vary the oop values of queries Q2 and Q3. Figure 5(b) summarizes the resulting costs of all-lazy, all-eager, and optimal selections in each case.

We can see from Table 6 that the optimal solution for the cases suggests eager coalescing for queries having lower oops and lazy coalescing for queries having higher oops (for given cps). This phenomenon is mainly due to the fact that, for the current array implementation of the window extent, the difference in update costs between eager and lazy is larger than the difference in scan costs at higher oops (where eager coalescing cost > lazy coalescing cost) and the difference in scan costs is larger than the difference in update costs at lower oops (where lazy coalescing cost > eager coalescing cost).



(a) Different cp cases.



(b) Different $\langle cp, oop \rangle$ cases.

Fig. 5 All lazy, all eager, and optimal costs.

[†]We use light as a grouping attribute because its shows a limited number of constant values in the data set.

Table 6 Scenarios used for the synthetic data sets

	$cp_1, cp_2, cp_3, cp_4, cp_5, cp_6$	$oop_1, oop_2, oop_3, oop_4, oop_5, oop_6$	**Selection between eager and lazy**
Case 1	0.2, 0.2, 0.2, 0.2, 0.2, 0.2	0.3, 0.3, 0.3, 0.3, 0.3, 0.3	lazy on all queries
Case 2	0.7, 0.7, 0.7, 0.7, 0.7, 0.7	0.6, 0.6, 0.6, 0.6, 0.6, 0.6	eager on all queries
Case 3	0.5, 0.5, 0.5, 0.5, 0.5, 0.5	0.3, 0.3, 0.3, 0.3, 0.3, 0.3	eager on all queries
Case 4	0.5, 0.5, 0.5, 0.5, 0.5, 0.5	0.7, 0.7, 0.7, 0.7, 0.7, 0.7	lazy on all queries
Case 5	0.4, 0.4, 0.4, 0.2, 0.2, 0.2	0.6, 0.6, 0.6, 0.7, 0.7, 0.7	eager on Q_1, Q_2, Q_3 and lazy on Q_4, Q_5, Q_6
Case 6	0.4, 0.4, 0.4, 0.2, 0.2, 0.2	0.6, 0.8, 0.8, 0.7, 0.7, 0.7	eager on Q_1 and lazy on Q_2, Q_3, Q_4, Q_5, Q_6

(cp_i denotes the coalescing probability for query Q_i.)

7.4 Summary of experiment results

The experimental results suggest the following conclusions.

- Eager coalescing outperforms lazy coalescing when the update cost weight (λ_υ) is comparable to the memory cost weight (λ_μ) and scan cost weight (λ_σ).
- The cost of eager coalescing becomes increasingly lower than that of lazy coalescing as (*cp*, *oop*) approaches (0.0, 0.0) and, as the weight of update cost increases, becomes increasingly higher than that of lazy coalescing as (*cp*, *oop*) approaches (0.0, 1.0).
- If all tuples arrive in order, then the cost of eager coalescing increases with the increase in coalescing probability. If some tuples arrive out of order, then the cost of eager coalescing decreases with the increase in coalescing probability.
- When multiple queries are registered in the system, there exists an optimal set of queries, some coalesced eagerly and others lazily, whose cost is no greater than both the cost of performing eager coalescing on all the queries and the cost of performing lazy coalescing on all the queries.
- When multiple queries are registered in the system with different coalescing probabilities and the weights of update scan and memory are equal, optimal selection problem suggests us to do eager coalescing on queries with higher coalescing probabilities and lazy coalescing on the rest.
- When multiple queries are registered in the system with different coalescing probabilities, the probability for a query to be selected to coalesce lazily increases with an increase in the update cost weight and the memory cost weight and the probability for a query to be selected to coalesce eagerly increases with an increase in the scan cost weight.

7.5 Practicality considerations on the cost objective

The cost objective, expressed as a weighted sum of normalized cost items, carries some practical implications with respect to the weighting scheme.

The weighting scheme accommodates different situations of available computing resources in real-world streaming applications. For many applications, a stream processing system is limited in either or both of memory and processing power [17]. When more limited in memory, it can be reflected through a larger weight on the memory cost. When more limited in the processing power, it can be reflected through larger weights on the update and scan costs. Further, for streaming applications with high arrival rates [17], assigning a larger weight to the update cost accommodates the situation adequately.

While normalization is generally a good mechanism to ensure fairness when considering cost items of different units and scales, it is possible to have a deviation among the three normalized cost items. In such a case, the weighting scheme can be used to compensate for the effects of the deviation. If any normalized cost item is excessively large compared with others, then the optimization will be particularly sensitive to the weight assigned to the large item. So, prior knowledge of the deviations among the normalized cost items would be helpful in determining the appropriate weights on them.

8. Related Work

The areas of related work are temporal data management and data stream processing. Extensive research efforts have been directed to develop concepts, tools, and techniques that better support the management of temporal data; see [22] and [25]. Data stream processing has also received a great deal of research attention in recent years; see [17] and [18] for comprehensive overviews. The uniqueness of our work lies in being the first in-depth work about temporal coalescing to support continuous queries with temporal functions and predicates applied to windowed data streams. Therefore, in this section we discuss other work on coalescing in temporal databases and coalescing over data streams.

Coalescing in temporal databases

Böhlen [24] is the first to emphasize the importance of coalescing for temporal databases with respect to query semantics. Then, Böhlen et al. [12] investigated the performance of three approaches to implement coalescing in temporal databases. The first approach requires modifying the underlying DBMS internals, which is time-consuming and expensive. The second approach works by reading a temporal table into main memory, coalescing it, and then writing the table back to the database. This approach is not applicable in many situations in which temporal tables are too large to be loaded in main memory. The third approach defines coalescing operation as a set of pure SQL statements. In this approach, however, a coalescing query is usually complex.

Zhou et al. [26] proposed and compared two approaches to address the complexity of supporting coalescing in RDBMS through SQL implementation. The first approach utilizes new functionality of SQL 2003, which can be used to support a built-in coalescing function that is transparent to the users. The second approach proposes to integrate the coalescing functionality in RDBMS as a user-defined aggregates.

Dyreson [15] addressed the problem of temporal coalescing in temporal databases for the specific situations in which tuples contain incomplete temporal information and model different temporal granularities

Coalescing over data streams

Barga et al. [7] introduced a framework for complex event processing over data streams based on a temporal data model that uses the concept of coalescing. In Barga's work, coalescing is simply used to represent two events as one single event if the valid-time intervals of the two events overlap.

RayChaudhuri et al. [27] employed coalescing in sensor networks applications to obscure the temporality of data collected by local sensor nodes before the readings are streamed out. Coalescing in their work, however, is different from the coalescing in temporal database - it is simply an accumulation of sensor readings made at local sensor nodes during each time interval.

Recently, Zaniolo [28] demonstrated that temporal coalescing can be expressed using Kleene-closure constructs, which are extensions of that SQL standards proposed for finding patterns in a sequence of data (e.g., data streams and ordered sequences of events). Zaniolo's work, however, considers a data stream simply as a sequence of data. That is, it does not take into account the unique model of data stream query processing itself. In contrast to Zaniolo's work, our work pertains to the coalescing assuming the sliding window model, which is typically an integral part of the query processing over data streams.

In an effort to propose a formal foundation for continuous queries over data streams [29], Krämer et al. considered coalescing as a physical operator that compacts the representation of a data stream by merging tuples with identical values and consecutive timestamps into a single tuple. Our work has the following fundamental differences from their work. First, they coalesce tuples over a data stream itself as a means to controlling the data stream rate, while we coalesce tuples over a sliding window as a means to evaluating temporal functions and predicates over data streams. Second, coalescing operator in their work does not have an impact on the semantics of a query, while in our work it is a mandatory preprocessing step that should be done before evaluating temporal functions and predicates to guarantee the correctness of query results.

9. Summary and Future Work

In this paper, we studied the coalescing operation for supporting temporal query processing over data streams.

In view of eager and lazy coalescing, we developed window extent update algorithms for tuple- and time-based windows. With eager coalescing, the basic algorithm does not work, so we designed a correct algorithm (achieving almost the same efficiency). The algorithm manipulates tuple timestamps to merge uncoalesced tuples when a new tuple is added and to split already coalesced tuples when an existing tuple is discarded. With lazy coalescing, the timestamp of a newly added tuple must be modified to correctly model the validity of the tuple.

Additionally, we addressed the problem of optimally selecting eager and lazy coalescing for multiple temporal queries running concurrently over the same data stream. Given the tradeoff between lazy and eager coalescing schemes, the optimization minimizes the total cost incurred on all window extents needed by those queries. (The total cost is a weighted sum of the normalized costs of storing, updating, and scanning all window extents.) For this purpose, we developed a cost model of the individual cost items in eager and lazy coalescing and used it to formulate the optimization problem as a 0-1 integer programming problem.

We then conducted a performance study of the two coalescing schemes with respect to the total cost. One set of experiments compared the relative costs between eager and lazy coalescing considering a single query. The comparison was done for different combinations of the weights on the three cost items. The other set of experiments observed interesting cases of selecting between the two coalescing schemes for different groups of multiple concurrent queries. Two parameters affect the costs – the coalescing probability and the out-of-order probability. Thus, the performance study was centered on these two control parameters.

The presented research opens an avenue for several future work. First, the data model needs to be extended. For example, attribute timestamping should be supported as well as tuple timestamping. Second, the query language should be fully developed. For this, temporal database query language and data stream query language should be integrated to produce a temporal stream query language. Third, various system issues pertinent separately to temporal databases and data streams should be revisited. Example issues are temporal stream query optimization, temporal stream indexing, and temporal stream load shedding.

Acknowledgments

The authors would like to thank the members of Intel Berkeley Research lab for graciously granting the permission to use their sensor datasets in the experiments.

This research is based upon work supported by the National Science Foundation under Grant No. IIS-0415023.

References

[1] C.J. Chu, V.S. Tseng, and T. Liang, "An efficient algorithm for mining temporal high utility itemsets from data streams," Journal of Systems and Software, vol.81, no.7, pp.1105–1117, 2008.

[2] L. Harada, "Detection of complex temporal patterns over data streams," Inf. Syst., vol.29, no.6, pp.439–459, 2004.

[3] M. Hadjieleftheriou, N. Mamoulis, and Y. Tao, "Continuous constraint query evaluation for spatiotemporal streams," SSTD'07, pp.348–365.

[4] M.F. Mokbel and W.G. Aref, "SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams," The VLDB Journal, vol.17, no.5, pp.971–995, 2008.

[5] D. Zhang, D. Gunopulos, V.J. Tsotras, and B. Seeger, "Temporal aggregation over data streams using multiple granularities," EDBT '02, pp.646–663.

[6] D. Zhang, D. Gunopulos, V.J. Tsotras, and B. Seeger, "Temporal and spatio-temporal aggregations over data streams using multiple time granularities," Inf. Syst., vol.28, no.1-2, pp.61–84, 2003.

[7] R.S. Barga, J. Goldstein, M.H. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," CIDR '07, pp.363–374.

[8] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," SIGMOD '06, New York, NY, USA, pp.407–418, ACM.

[9] "Intel lab data. http://berkeley.intel-research.net/labdata/.."

[10] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," The VLDB Journal, vol.15, no.2, pp.121–142, 2006.

[11] R. T. Snodgrass et al., "TSQL2 language specification," SIGMOD Rec., vol.23, no.1, pp.65–86, 1994.

[12] M.H. Böhlen, R.T. Snodgrass, and M.D. Soo, "Coalescing in temporal databases," VLDB '96, pp.180–191.

[13] C. Vassilakis, "An optimisation scheme for coalesce/valid time selection operator sequences.," SIGMOD Record, vol.29, no.1, pp.38–43, 2000.

[14] J.F. Allen, "Maintaining knowledge about temporal intervals," Commun. ACM, vol.26, no.11, pp.832–843, 1983.

[15] C.E. Dyreson, "Temporal coalescing with now granularity, and incomplete information," SIGMOD '03, pp.169–180.

[16] J. Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," SIGMOD '05, pp.311–322.

[17] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," PODS '02, pp.1–16.

[18] L. Golab and M.T. Ozsu, "Issues in data stream management," SIGMOD Rec., vol.32, no.2, pp.5–14, 2003.

[19] 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 Engineering Bulletin, vol.26, no.1, pp.19–26, 2003.

[20] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," Proc. VLDB Endow., vol.1, no.1, pp.274–288, 2008.

[21] P.A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," IEEE Trans. on Knowl. and Data Eng., vol.15, no.3, pp.555–568, 2003.

[22] C. Date and H. Darwen, Temporal Data and the Relational Model, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[23] A.U. Tansel, "Temporal relational data model," IEEE TKDE, vol.09, no.3, pp.464–479, 1997.

[24] M.H. Böhlen, The Temporal Deductive Database System ChronoLog., Ph.D. thesis, Departement Informatik, ETH Ziirich, 1994.

[25] C. Dyreson et al., "A consensus glossary of temporal database concepts," SIGMOD Rec., vol.23, no.1, pp.52–64, 1994.

[26] X. Zhou, F. Wang, and C. Zaniolo, "Efficient temporal coalescing query support in relational database systems," DEXA '06, pp.676–686.

[27] A. RayChaudhuri, U.K. Chinthala, and A. Bhattacharya, "Obfuscating temporal context of sensor data by coalescing at source," Mobile Computing and Communications Review, vol.11, no.2, pp.41–42, 2007.

[28] C. Zaniolo, "Event-oriented data models and temporal queries in transaction-time databases," TIME, pp.47–53, 2009.

[29] J. Krämer and B. Seeger, "A temporal foundation for continuous queries over data streams.," COMAD '05, pp.70–82.

[30] T.T.L.D. Committee, "An evaluation of tsql2," A TSQL2 Commentary, year = 1994,.

Mohammed Al-Kateb is a Ph.D. student at the Department of Computer Science in The University of Vermont, U.S.A.. He received BS and MS degrees in Information Systems from Cairo University, Egypt. His research interests include data streams processing and temporal data management. He is a student member of IEICE.

Sasi Sekhar Kunta is a Ph.D. student at the Department of Computer Science in The University of Vermont, U.S.A.. He received a BS degree in Computer Science from Acharya Nagarjuna University, India, and a MS degree in Computer Science from BITS at Pilani, India. His research interests include complex event processing and data stream processing.

Byung Suk Lee is an Associate Professor at the Department of Computer Science in The University of Vermont, U.S.A.. He received a BS degree from Seoul National University, South Korea, a MS degree from KAIST, South Korea, and a Ph.D. degree from Stanford University, U.S.A.. His research interests include database systems, data stream processing, and event processing.

Appendix A: Basic functions

In this section, we describe in further details the basic functions used in the window extent update algorithms. The discussion in this section assumes the tuple-based window model. Basic functions implemented differently for the time-based window model can be described in the same way.

There are two basic functions, *prev*() and *next*(), which respectively return the immediately preceding and succeeding tuples of a given tuple. In addition, the algorithms use three basic operations: *addTuple*, *mergeTuples*, and *splitTuple*. The *addTuple* is common to both lazy coalescing and eager coalescing, whereas the *mergeTuples* and *splitTuple* are used in eager coalescing only. Let us describe these operations further now.

Function 1: *addTuple(s_{new}, s_{prev})*

```
Inputs: $s_{new}$ // a new tuple to be added
Inputs: $s_{prev}$ // the tuple next to which $s_{new}$ is added
1: rename $s_{new}.ts$ to $s_{new}.t_s$
2: attach a new attribute $t_e$ to $s_{new}$
3: $s_{prev}.t_e = s_{new}.t_s$
4: if $s_{prev.next()}$ == null then
5:     $s_{new}.t_e = s_{new}.t_s$
6: else
7:     $s_{new}.t_e = s_{prev}.next().t_s$
8: end if
9: add $s_{new}$ next to $s_{prev}$ in the window extent
```

The *addTuple* (see Function 1) adds a new tuple (s_{new}) to the current window extent as a temporal tuple (see Section 2.3). The timestamp attribute *ts* is used to represent the starting instant of the tuple (Line 1), and a new attribute is attached to the tuple to represent the ending instant (Line 2). The starting instant value of the new tuple is assigned to the ending instant of its preceding tuple (Line 3). If the new tuple has arrived in order (thus s_{new} is the last tuple), then its ending instant takes the value of its starting instant (Line 5). Otherwise, its ending instant is assigned the value of the starting instant of its subsequent tuple (Line 7). Eventually, the new tuple is added next to the previous tuple (s_{prev}) in the window extent (Line 9).

Function 2: *mergeTuples(s_{new}, s)*

```
Inputs: $s_{new}$ // a new tuple
        $s$ // an existing temporal tuple
1: insert the value of $s_{new}.ts$ into $s.\mathbf{v}$ in the sorted order of timestamp
2: if $s.t_e < s.\mathbf{v}.last()$ then
3:     $s.t_e = s.\mathbf{v}.last()$
4: end if
5: discard $s_{new}$
```

The *mergeTuples* (see Function 2) is called when a new tuple (s_{new}) is coalesced with an existing temporal tuple (s). It merges the two tuples by inserting the timestamp of the new tuple ($s_{new}.ts$) into the timestamp vector of the existing tuple ($s.\mathbf{v}$) while maintaining the sorted order of timestamp (Line 1). If the new tuple has arrived in order, then $s_{new}.ts$ becomes the new last timestamp value in $s.\mathbf{v}$. In this case, the ending instant (t_e) of the existing tuple is updated to the new value (Line 3). After merging the two tuples, the original new tuple is discarded from the window extent (Line 5).

Function 3: *splitTuple(s, ts)*

```
Inputs: $s$ // temporal tuple to be split
        $ts$ // timestamp value of the late arrival tuple
1: allocate a new temporal tuple $s_2$
2: copy $s$ to $s_2$
3: remove from $s.\mathbf{v}$ all timestamp values greater than $ts$
4: remove from $s_2.\mathbf{v}$ all timestamp values less than $ts$
5: add $s_2$ next to $s$ in the window extent
6: return $s$
```

The *splitTuple* (see Function 3) is called when a new tuple arrives out of order and causes a gap in an existing temporal tuple. It splits the existing tuple into two tuples and returns the first tuple after the split. The split is achieved by duplicating the existing tuple (Line 2), removing from the timestamp vector of the original tuple (s) all timestamp values greater than the given timestamp value (*ts*) (Line 3), and removing from the timestamp vector of the duplicate tuple (s_2) all timestamp values less than the given timestamp value (Line 4). Then, the duplicate tuple is added next to the original tuple in the window extent (Line 5), and the original tuple is returned (Line 6).

Appendix B: Extension of the coalescing operator

In this section, we discuss two issues relevant to our proposed window extent update algorithms.

Grouped coalescing: A user query may naturally require tuples to be coalesced separately for different groups. This grouped coalescing is very common in temporal database queries. For instance, most of query examples in the TSQL Commentary [30] use grouped coalescing. The same is common in real-world data *stream* applications. For instance, recall Example 1 in which a wireless sensor network application monitors the changes in the humidity of two different regions, grouped by the region. The algorithms support grouped coalescing in this paper.

General coalescing: Traditionally temporal database applications perform transactions on simple data (e.g., employee records). For these applications, the notion of value equivalence for coalescing has been limited to equality comparison. In contrast, data stream applications usually monitor a trend continuously over time. For these application, the notion of value equivalence needs to be generalized to use any arbitrary comparison operations which can be used to support a certain trend analysis. Example trends are "increasing" (e.g., the value is larger than the pervious one) and "bounded" (e.g., the value is within a certain bound from the previous one, the relative change from the previous value is within a certain bound). Moreover, the values compared may be the instances of any data type. Data stream applications can deal with a variety of data types ranging from sensor readings (numbers) to XML documents (texts) and to video frames (images). For those applications, the value comparison involves more expensive operations such as text search or image analysis.