Transformation of Continuous Aggregation Join Queries over Data Streams

Tri Minh Tran and Byung Suk Lee
Department of Computer Science, University of Vermont
Burlington VT 05405, USA
{ttran, bslee}@cems.uvm.edu

Received 2 October 2008; Accepted 11 February 2009

Aggregation join queries are an important class of queries over data streams. These queries involve both join and aggregation operations, with window-based joins followed by an aggregation on the join output. All existing research address join query optimization and aggregation query optimization as separate problems. We observe that, by putting them within the same scope of query optimization, more efficient query execution plans are possible through more versatile query transformations. The enabling idea is to perform aggregation before join so that the join execution time may be reduced. There has been some research done on such query transformations in relational databases, but none has been done in data streams. Doing it in data streams brings new challenges due to the incremental and continuous arrival of tuples. These challenges are addressed in this paper. Specifically, we first present a query processing model geared to facilitate query transformations and propose a query transformation rule specialized to work with streams. The rule is simple and yet covers all possible cases of transformation. Then we present a generic query processing algorithm that works with all alternative query execution plans possible with the transformation, and develop the cost formulas of the query execution plans. Based on the processing algorithm, we validate the rule theoretically by proving the equivalence of query execution plans. Finally, through extensive experiments, we validate the cost formulas and study the performances of alternative query execution plans.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems–Query processing

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Aggregation Join Query, Data stream, Query Transformation

1. INTRODUCTION

Aggregation join queries are common in continuous queries over data streams. Queries of this type involve both join operations and aggregation operations (the aggregation may be a grouped aggregation.) Aggregation join queries require windows on the join inputs because processing join over unbounded streams requires unbounded memory, which is impractical. A window, which restricts the number of tuples processed, is a common technique proposed in much existing research [Kang et al. 2003; Golab and Ozsu 2003; Das et al. 2003; Li et al. 2005; Ayad and Naughton 2004; Arasu and Widom 2004; Arasu and Manku 2004; Ding and Rundensteiner 2004; Ghanem et al. 2007; Babcock et al. 2002].

Window-based aggregation join queries (called simply "aggregation join queries" from now on) are needed in various data stream applications. For example, an online auction system -which has continuous streams of auction items registered, members (i.e., account holders) signing in, and bids made – may be monitored to build statistics of auction activities. In another example, a network traffic management application [Babcock et al. 2002], a network administrator may want to monitor packet data flow (i.e., number of packets transmitted) through links between different networks. Below, let us take a look at an example query of the online auction application.

Example 1 In an online auction application, we may pose a continuous query running on two data streams *Bid(ts, auctionID, bidderID, bidPrice)* and *Auction (ts, auctionID, sellerID, startPrice)* (based on schema used by Babu et al. [Babu et al. 2003]) and one relation *Person(personID, name, state)*; where the meanings of attributes are self-explanatory. Users may want to know the total number of bids made in the last one hour for each auction created up to now by a seller from Vermont. In this case, the query involves a three-way join (involving two stream windows and one relation) and a grouped aggregation, grouped by auctionID. The query can be expressed as an aggregation join query as follows:

```
SELECT A.auctionID, COUNT(B.*)
FROM Auction AS A [WINDOW UNTIL NOW], Bid AS B [WINDOW 1 HOUR],
        Person AS P
WHERE A.auctionID = B.auctionID AND A.sellerID = P.personID AND P.state = "VT"
GROUP BY A.auctionID;
```

Naturally, efficient processing of these aggregation join queries is very important. One premise in this paper is that, the queries can be processed more efficiently if the optimizations of join and aggregation are handled *as one problem*. Most of the existing research addresses them as separate problems: for example, joins in [Das et al. 2003; Golab and Ozsu 2003; Kang et al. 2003; Viglas et al. 2003; Urhan and Franklin 2000] and aggregations in [Dobra et al. 2002; Gehrke et al. 2001; Gilbert et al. 2001; Guha and Koudas 2002; Vitter and Wang 1999]. Two other existing studies [Dobra et al. 2002; Jiang et al. 2006] address the problem of efficiently processing aggregation join queries as one, but not as an optimization problem per se. Furthermore, their methods use sketching techniques [Dobra et al. 2002] and discrete cosine transform [Jiang et al. 2006], respectively; thus, they cannot be applied to our problem since they are not window-based and cannot handle grouped aggregations.

The premise mentioned above opens a door to generating a heuristically more efficient query execution plan (QEP) through *query transformations*, which is the focus of this paper. Query transformation produces alternative query execution plans (QEPs) for the same query so that the query optimizer may choose the QEP whose estimated execution cost is the lowest. In the initial QEP of an aggregation join query, joins are performed first and then aggregation follows. The key idea of query transformation adopted in this paper is to perform an aggregation before join – in other words, push aggregation down to a join input in a query execution tree. This transformation may reduce the join input cardinality and thus result in a more efficient QEP, although this is not guaranteed. In this paper we call the initial QEP a *late aggregation plan (LAP)* and the transformed QEP an *early aggregation plan (EAP)*, and call the pushed-down aggregation operator an *early aggregation operator*.[1]

There are such query transformation mechanisms proposed in the relational database [Chaudhuri and Shim 1994; Yan and Larson 1995]. These mechanisms, however, are not applicable to data streams due to the streaming nature of data that makes stream queries different from database queries. First, tuples arrive continuously in data streams, hence the query output must be updated *continuously* as well. Second, in many cases the arriving tuples must be processed on-line, which requires that the query must be processed *incrementally* as soon as tuples arrive.

In order to develop a working transformation mechanism for data streams, we introduce two key stream operators for query processing, called the *aggregation set update (AS update)* and the *aggregation set join (AS join)*, and the notion of a virtual window on the join output. The AS update operator is used to update aggregate values incrementally as new tuples are added to the input window and old tuples are removed from the window. The AS join operator is used to perform a join between a new tuple arriving at one stream and the output of an early aggregation operator (called an *aggregation set*) at another stream. Note its distinction from a window join, which uses a window of tuples instead of an aggregation set. To our knowledge, the AS join operator is a new operator introduced for the first time through this paper. The virtual window is a notion for enabling the AS update operation on the join output stream which becomes an input to the subsequent aggregation operator. Note that the query has no specification of a window on the join output, while a window is needed on the aggregation input.

In this paper we first formalize the notions of the aggregation set (AS) and the two associated operators, AS update and AS join, and the notion of the virtual window. Then, we propose a query transformation rule based on the approaches mentioned above, that is, supporting AS update and AS join operators and retaining a late aggregation operator. The rule is simple and yet general enough to be applicable to any input streams. Then we present a generic algorithm for executing all alternative QEPs (i.e., LAP and EAPs). Based on the algorithm, we validate the rule theoretically by proving the equivalence of LAP and EAP. Specifically, we use algebraic expressions to represent the stream operators and prove the equivalence of LAP and EAP by induction for each data item. We also validate the proposed transformation rule

[1]Join ordering is another important issue in query transformation. However, the issue of join ordering is independent of the issue of early aggregation, and is beyond the scope of this paper.

empirically. For this, we develop the cost functions for estimating the execution times of QEPs, and implement the algorithm in an operational prototype. We then compare the execution times of alternative QEPs estimated using the cost functions with those measured using the prototype.

To our knowledge, this is the first work done to address the query transformation on an aggregation join query over data streams. Main contributions of this paper include (1) proposing a formal query processing model that is suitable for a stream aggregation join query, (2) developing a query transformation rule that is simple and yet applicable to any input streams, (3) validating the rule through an inductive proof of the equivalence of alternative QEPs, (4) building analytical cost functions to estimate the execution time of a QEP, and (5) conducting extensive experiments to validate the cost functions and to examine the efficiencies of alternative QEPs.

This paper contains the result of a comprehensive study extended from our earlier work [Tran and Lee 2007]. The extensions made from the earlier work include validating the query transformation rule formally through the proof of the algebraic equivalence of an EAP and the corresponding LAP (Section 6.2), developing the cost functions for estimating the execution times of QEPs resulting from the query transformation (Section 6.3 and Section 7.1), and conducting more comprehensive experiments, including new experiments for validating the cost functions (using hash joins as well as nested loop joins[2] and using three-way as well as two-way joins) (Section 7.2). Besides, the presentation has been extended in several places including the related work section (Section 2) and the query processing model (Section 4).

The rest of the paper is organized as follows. Section 2 discusses related work, Section 3 provides some preliminary concepts, Section 4 presents the query processing model and the key operators, Section 5 proposes the query transformation rule, Section 6 describes the query processing algorithm and the cost functions and shows the equivalence of query transformations, Section 7 evaluates the cost functions and the efficiencies of QEPs through experiments, and Section 8 concludes the paper.

2. RELATED WORK

We find related work in two areas: (1) processing join queries and aggregation queries in data streams and (2) handling early aggregations databases.

2.1 Join Queries and Aggregation Queries on Data Streams

As far as we know, all of the existing data stream query processing systems – such as Aurora [Abadi et al. 2003], STREAM [Motwani et al. 2003], TelegraphCQ [Chandrasekaran et al. 2003], NiagaraCQ [Chen et al. 2000], Stream Mill [Bai et al. 2006], Nile [Hammad et al. 2004], Tribeca [Sullivan 1996] and GigaScope [Cranor et al. 2003]. – optimize aggregation join queries by considering the join and aggregation *separately*. In Aurora [Abadi et al. 2003] and STREAM [Motwani et al. 2003], query optimizers use query transformations by reordering filter operators (i.e., selection) and join operators in a QEP to generate equivalent QEPs. Their reordering, however, is

[2]The results from the nested loop joins are omitted in this paper due to space limit. The results can be found in [Tran and Lee 2007].

not applicable between a join operator and an aggregation operator. In the other systems, query optimizers do not even use query transformations. Thus, to our knowledge, our work is the first to allow for reordering the join and aggregation operators.

Aside from these comprehensive data stream management systems, join processing and aggregation processing have been researched quite extensively. A large number of studies focus on window join processing [Kang et al. 2003; Golab and Ozsu 2003; Viglas et al. 2003; Das et al. 2003; Srivastava and Widom 2004; Ding and Rundensteiner 2004; Hammad et al. 2003]. We briefly discuss these join techniques next.

In [Kang et al. 2003], Kang et al. propose sliding window two-way join algorithms and develop a unit-time cost model for evaluating the performances of their algorithms. The unit-time cost model is used to estimate the execution time to process tuples arriving in a unit time. In [Golab and Ozsu 2003], Golab et al. extend Kang et al.'s two-way window join algorithms to multi-way window join algorithms, and then propose join ordering heuristics to minimize the unit-time cost. A multi-way window join called the *MJoin* is also considered in [Viglas et al. 2003] by Viglas et al. based on the symmetric hash join. An MJoin assigns a join order for each input stream and generates join output without maintaining intermediate results. In contrast to MJoin, the *XJoin* proposed in [Urhan and Franklin 2000] is a multi-way join executed in a tree of two-way joins and maintains a fully-materialized join results for each intermediate two-way join.

For *window aggregation* processing, Li et al. [Li et al. 2005] propose a generic window concept and present an efficient window aggregation technique that computes the aggregate values in one pass. The key idea is to assign to each tuple a range of the identifiers of windows to which it belongs. Zhang et al. [Zhang et al. 2005] address the problem of processing multiple aggregation queries that differ only in grouping attributes. Their approach is to compute and maintain fine-granularity aggregations and use them to share computing resources among multiple queries. All this research is concerned only with an aggregation operation on a single input stream. There are other existing aggregation techniques [Considine et al. 2004; Manjhi et al. 2005; Tatbul and Zdonik 2006; Babcock et al. 2004; Vitter and Wang 1999; Gilbert et al. 2001] as well, but they are not window-based.

As mentioned in the Introduction, while there have been many efforts to address join query processing and aggregation query processing as separate problems, there has been very little research addressing them in combination as one optimization problem. There are two existing studies [Dobra et al. 2002; Jiang et al. 2006] done to address the problem of processing the same type of query as ours. Their approaches, however, are to use *approximation* techniques using sketching [Dobra et al. 2002] and discrete cosine transform [Jiang et al. 2006]. Specifically, Dobra et al. [Dobra et al. 2002] use random variables to construct the sketch of each stream and approximate the aggregate value as an expected value based on the random variables, and Jiang et al. [Jiang et al. 2006] use discrete cosine transform to represent the distribution of the join attribute value in each stream and use it to estimate the join output size. These techniques are not applicable to our problem which handles window-based queries and grouped aggregations. In summary, to the best of our knowledge our paper is the first to address the problem of optimizing aggregation join queries as one integral problem.

2.2 Early Aggregation in Relational Queries

The early aggregation idea is employed for database queries in the work by Chaudhuri and Shim [Chaudhuri and Shim, 1994] and by Yan and Larson [Yan and Larson, 1994; Yan and Larson, 1995]. In [Chaudhuri and Shim, 1994] the authors present query transformation rules for three cases. These cases are different in terms of the relation to which each of the aggregation, grouping, and join attributes belong. The second case subsumes the first case, and the third case subsumes the second case. Specifically, the first case specifies that the late aggregation operator is pushed down before join if the join is a foreign-key join. The second case specifies that an early aggregation operator can be inserted before join if both grouping attributes and aggregation attributes are in the same relation. In the third case, which is the most general, a new operator called the aggregate join is introduced. This operator is used to perform a join between one relation and the output of an early aggregation on the other relation. In their work, query transformation rules are restricted to apply to only the class of left-deep join trees.

In [Yan and Larson 1994; Yan and Larson 1995] the authors present more comprehensive transformation rules that apply to any class of join trees. Their rules are also proposed for different cases depending on which relation the aggregation, grouping, and join attributes belong to and cover all the cases considered in [Chaudhuri and Shim 1994]. In addition, instead of introducing a new operator like the aggregate join proposed in [Chaudhuri and Shim 1994], they use a "query re-writing" technique to generate an EAP. This re-writing technique involves inserting one or more early aggregation operators and modifying the aggregation functions specified in late aggregation operators.

The results of their work are not directly applicable to our problem which requires dealing with unbounded sequences of tuples in continuous data streams as opposed

Table I. Notations used in this paper.

Notation	Meaning
$S_i(TS_i, X_i, G_i, J_i, A_i)$	A stream S_i with a list of attributes, where TS_i is a timestamp attribute, G_i is a grouping attribute, J_i is a join attribute, A_i is an aggregation attribute, and X_i is the set of remaining attributes.
$W[T](t)$	A window of size T at time t.
$W^+(t_1, t_2)$	A window increment, i.e., the set of tuples added to the window during the time interval $[t_1, t_2]$.
$W^-(t_1, t_2)$	A window decrement, i.e., the set of tuples removed from the window during the time interval $[t_1, t_2]$.
${}_G\Im_{F(A)}(\cdot)$	An aggregation operator with the list of grouping attributes G and an aggregation function F on the aggregation attribute A.
${}_G\mathcal{U}_{F(A)}(\cdot)$	An aggregation set update operator with the list of grouping attributes G and an aggregation function F on the aggregation attribute A.
$S_i \overset{F(A)}{\bowtie}_{J_i=J_j} AS_j$	A one-way *AS join* from S_i to AS_j via join attributes S_i. J_i and AS_j. J_j.

to bounded sets of tuples in relations. Moreover, a new transformation rule is needed under the query processing model geared for data stream processing.

3. PRELIMINARIES

In this section, we present some key notations and concepts needed to understand the rest of the paper.
Table I summarizes the notations used in this paper.

Data streams.

We consider a data stream S, of an ordered sequence of tuples. Each tuple in the stream has the schema $S(TS, X_1, X_2,..., X_d)$, where TS is a timestamp attribute and $X_1, X_2,..., X_d$ are non-timestamp attributes. We denote a tuple of the above schema as $s(ts, x_1, x_2,..., x_d)$, where ts is the value of TS and x_i is the value of X_i for each i=1, 2,..., d. (We use an upper-case letter to denote an attribute and a lowercase letter to denote the value of an attribute.) We assume that the tuples arrive in the order of timestamp; handling out-of-order tuples is beyond the scope of this paper.

Windows.

Definition 1 (Window) A *window* W of size T, W[T], on stream S at time t is defined as a set of tuples whose timestamps are in the range of $[t - T, t)$. That is, $W[T](t)= \{s \mid t - T \leq s.ts < t\}$.

Definition 2 (Window increments and decrements) Given a window $W[T](t_1)$, a *window increment*, denoted as $W^+(t_1, t_2)$, is the set of tuples added to the window during a time interval $[t_1, t_2]$, i.e., $W^+(t_1, t_2) = \{s \mid t_1 \leq s.ts < t_2\}$. A *window decrement*, $W^-(t_1, t_2)$, is the set of tuples removed from the window during the same time interval, i.e., $W^-(t_1, t_2) = \{s \mid t_1 - T \leq s.ts < t_2 - T\}$.

Given a window $W[T](t_1)$ at time t_1, and a window increment $W^+(t_1, t_2)$ and decrement $W^-(t_1, t_2)$ between t_1 and t_2, the window $W[T](t_2)$ at time t_2 is computed as:

$$W[T](t_2) = W[T](t_1) \cup W^+(t_1, t_2) - W^-(t_1, t_2)$$

Given the above definitions of window increments and decrements, three types of windows – *sliding* window, *tumbling* window, and *landmark* window [Li et al. 2005] – are supported in our processing model. Figure 1 illustrates the three window types, with their corresponding increments and decrements. The tumbling window and the landmark window are special cases of the sliding window, and therefore we consider only the sliding window in this paper.

Window joins.

A two-way window join [Kang et al. 2003] between two streams S_1 and S_2 with windows W_1 and W_2, respectively, is computed as follows. For each new tuple s_1 in a window increment of S_1, s_1 is inserted into W_1 and any expired tuples are removed from W_1. Then, W_2 is probed for matching tuples of s_1 and matching tuples are

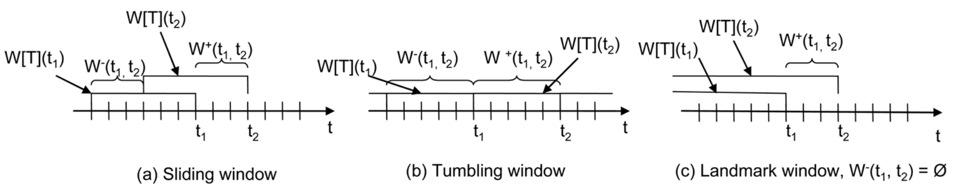


Figure 1. Windows of different types ($t_1 < t_2$).

appended to the join output stream. The computation is symmetric for each new tuple s_2 in a window increment of S_2. Generalized from this, in a multi-way join among m ($m > 2$) streams, for each new tuple s_k in a window increment of S_k, matching tuples are found from the other $m - 1$ windows and then appended to the output stream. We assume that the join computation is fast enough to finish before the other $m - 1$ windows are updated.

4. QUERY PROCESSING MODEL

In this section we present a model for continuous and incremental processing of aggregation join queries. Key components of the model include the *aggregation set*, the *aggregation set update (AS update)* operator, the *aggregation set join (AS join)* operator, and the virtual window. This model provides a basis for the query transformation rule and the query processing algorithm presented in Section 5 and Section 6 respectively.

The concepts of aggregation set and AS update operator are the same as the concepts of window aggregate and group-by operator mentioned in [Ghanem et al. 2007]. These concepts are refined and presented formally in this paper using the notions of window increment and window decrement. The AS join is a combination of the window join defined in Section 3 and the aggregate join proposed for database aggregation join queries in [Chaudhuri and Shim 1994].

Aggregation set.

Aggregation of the tuples in a *window* produces a set of tuples, one tuple for each group. We call this set of tuples an *aggregation set (AS).*

Definition 3 (Aggregation set) Consider a set of tuples in a window at time t, denoted as W[T](t). Additionally, consider an aggregation operator, denoted as ${}_{G}\Im_{F(A)}(W[T](t_1))$ where $G \equiv (G1,...,G_p)$ is a list of grouping attributes, A is an aggregation attribute, and F is an aggregation function on A. Then, an aggregation set is defined as a set of tuples $\{(g_1,..., g_p, v)\}$ where g_i is a value of G_i ($i = 1, 2,..., p$) and v is an aggregate value computed as $F(A)$ for the group ($g_1,..., g_p$) over $W[T](t)$. We denote the schema of an aggregation set as $AS(G, F(A))$; here, $F(A)$ denotes an attribute whose value is v.

Aggregation set update.

An aggregation set update operator is used to update the AS as the window content changes. This is done *incrementally* without re-evaluating the whole window content.

Definition 4 (Aggregation set update) Consider an aggregation set $AS \equiv {}_G\Im_{F(A)}(W[T](t_1))$ at time t_1, a window increment $W^+(t_1, t_2)$ and a window decrement $W^-(t_1, t_2)$ at time t_2 ($> t_1$). Then, an AS *update* operation, denoted by ${}_G\mathcal{U}_{F(A)}(AS, W^+(t_1, t_2), W^-(t_1, t_2))$, returns an updated aggregation set AS' resulting from the following updates on AS:

• For each tuple s in $W^+(t_1, t_2)$, if there exists a tuple l in AS such that $l.G = s.G$ (i.e., s belongs to a group in AS) then update the aggregate value $l.F(A)$ as follows.[3]

$$l.F(A) = \begin{cases} l.F(A) + 1 & \text{if } F \equiv \text{COUNT} \\ l.F(A) + s.A & \text{if } F \equiv \text{SUM} \\ s.A & (\text{if } F \equiv \text{MIN and } s.A < l.F(A)) \text{ or } (\text{if } F \equiv \text{MAX and } s.A > l.F(A)) \\ l.F(A) & (\text{if } F \equiv \text{MIN and } s.A \geq l.F(A)) \text{ or } (\text{if } F \equiv \text{MAX and } s.A \leq l.F(A)) \end{cases}$$

Otherwise, insert a new tuple l' whose $l'.G = s.G$ and whose $l'.F(A)$ is as follows.

$$l'.F(A) = \begin{cases} 1 & \text{if } F \equiv \text{COUNT} \\ s.A & \text{if } F \in \{\text{SUM, AVG, MIN, MAX}\} \end{cases}$$

• For each tuple r in $W^-(t_1, t_2)$, find a tuple l in AS such that $l.G = r.G$ (i.e., r belongs to a group in AS), and then update the aggregate value $l.F(A)$ as follows.

$$l.F(A) = \begin{cases} l.F(A) - 1 & \text{if } F \equiv \text{COUNT} \\ l.F(A) - r.A & \text{if } F \equiv \text{SUM} \\ l.F(A) & \text{if } F \in \{\text{MAX, MIN}\} \text{ and } r.A \neq l.F(A) \\ {}_G\Im_{F(A)}(W[T](t_1) - \{r\}) & \text{if } F \in \{\text{MAX, MIN}\} \text{ and } r.A = l.F(A) \end{cases}$$

The above definition shows that from the above definition, updating an aggregate value $l.F(A)$ for each tuple $r \in W^-(t_1, t_2)$ requires re-evaluating the whole window only if $F \equiv$ MIN or $F \equiv$ MAX and $r.A = l.F(A)$. Note that even this situation happens only with a sliding window and not with a tumbling or a landmark window. In the case of a tumbling window, a window decrement is discarded and a new aggregation set is generated using the new window increment only. In the case of a landmark window, there is no window decrement.

Aggregation set join.

We first present the *coalescing property* [Chaudhuri and Shim 1994] of an aggregation function. This property is used in Definition 6 to compute the aggregate value of each output tuple generated from the aggregation set join.

[3] $F \equiv$ AVG is computed by maintaining both COUNT and SUM.

Definition 5 (Coalescing property) Consider an aggregation function F on an attribute A. The aggregate of ctuples that have the same value, a, of A is computed using the following function $f(c, a)$ depending on the type of F.

$$f(c,a) = \begin{cases} a * c & \text{if F} \equiv \text{SUM} \\ c & \text{if F} \equiv \text{COUNT} \\ a & \text{if F} \in \{\text{AVG, MAX, MIN}\} \end{cases}$$

An AS join handles a join between a stream S and an aggregation set AS and computes the aggregate value of a join output tuple using the coalescing property.

Definition 6 (One-way aggregation set join) Consider two streams S_1 and S_2 with their window $W_1[T](t_1)$ and $W_2[T](t_1)$, respectively, at time t_1. Additionally, consider the window increment $W_1^+(t_1, t_2)$ and decrement $W_1^-(t_1, t_2)$ of S_1 at time t_2 ($> t_1$). Now, given an aggregation $F(A)$ specified in the query, let the aggregation set $AS_2(t_1)$ on stream S_2 be computed as follows depending on whether A is in the schema of S_2 or not.

$$AS_2(t_1) = \begin{cases} {}_{J_2}\Im_{F(A)}(W_2[T](t_1)) & \text{if } A \text{ belongs to } S_2. \text{ (See Definition 3.)} \\ {}_{J_2}\Im_{COUNT(*)}(W_2[T](t_1)) & \text{otherwise. } (J_2 \text{ is the join attribute in } S_2.) \end{cases}$$

Then, a one-way *AS join* from S_1 to AS_2 via join attributes $S_1.J_1$ and $AS_2.J_2$, denoted as $S_1 \overset{F(A)}{\bowtie}_{J_1=J_2} AS_2$, is computed as follows.

For each tuple s1 in $W_1^+(t_1, t_2)$ and for each tuple r_1 in $W_1^-(t_1, t_2)$,

1. Find matching tuples from $AS_2(t_1)$. (Denote each tuple as l.)
2. Return a sequence of tuples where each tuple (u) is made of s_1 (or r_1) and each l and has the value of $F(A)$ set as follows.

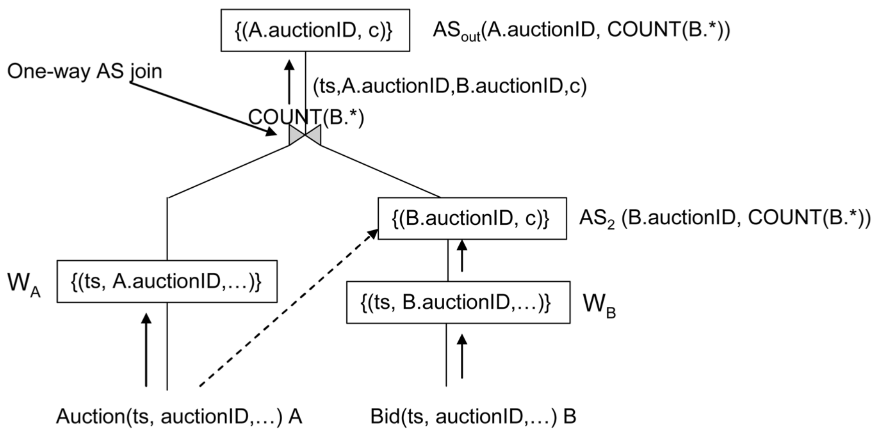
$$u.F(A) = \begin{cases} l.F(A) & \text{if } A \text{ belongs to } S_2 \\ f(c,a) & \text{otherwise} \end{cases}$$

where is the value of $s_1.A$(or $r_1.A$), c is the number of tuples aggregated to l in AS_2, and f is the function in the definition of the coalescing property (Definition 5).

An extension to a *multi-way* AS join is straightforward. That is, a one-way AS join is repeated from each stream S_k, ($k \in \{1, 2, ..., m\}$) to the aggregation sets AS_i on the other streams S_i, $i \neq k$.

Example 2 Given the query in Example 1, a one-way AS join between the stream Auction A and the aggregation set AS_2 on Bid B shown in Figure 2:

$$A \overset{COUNT(*)}{\bowtie}_{A.auctionID=B.auctionID} B$$



An arrow from the Auction stream to the aggregation set AS_2 denotes an AS join (Definition 6).

Figure 2. An example one-way AS join.

where $AS_2 \equiv {}_{B.auctionID}\Im_{COUNT(B.*)}(W_B(t))$. AS$_2$ is then a set of tuples, {(*B.auctionID, c*)}. For each tuple (*ts,A.auctionID,...*) in W_A^+, the one-way AS join from A to AS_2 produces a sequence of output tuples *u*(*ts,A.auctionID,B.auctionID,c*) where *A.auctionID* = *B.auctionID* and the aggregate value equals *c* (= *f*(*c, a*) in Definition 5). Similar steps are taken for each tuple in W_A^-.

Virtual window.

As mentioned in the definitions of the aggregation set and the aggregation set update (Definition 3 and Definition 4), an aggregation set is computed from a set of tuples in a window and is updated with the tuples in the window increment and the window decrement. However, there is no query window specified on the aggregation input which is a join output in an aggregation join query. We thus introduce the notion of a virtual window on the join output. The computation of virtual window extent depends on whether the join is a window join or an AS join, and is defined as shown below based on the definitions of the window join (Section 3) or the AS join (Definition 6 in this section).

Definition 7 (Virtual window on a window join output) Consider an aggregation join query on *m* input streams $S_1, S_2, ..., S_m$ with the corresponding windows $W_1[T](t_1)$,..., $W_m[T](t_1)$ at time t_1. Let $W_k^+(t_1, t_2)$, $W_k^-(t_1, t_2)$ be respectively the window increment and decrement on stream S_k during a time interval $[t_1, t_2]$, and let S_{out} be the *m*-way window join output stream. Then, the virtual window, $W_{out}[T](t_1)$, on S_{out} is defined as

$$W_{out}[T](t_1) = W_1[T](t_1) \bowtie W_2[T](t_1) \bowtie \cdots \bowtie W_m[T](t_1)$$

and the virtual window increment and decrement (respectively due to the window increment and window decrement on the stream S_k) are computed as follows.

$$\begin{aligned}
W^+_{out}(t_1,t_2) &= W_1[T](t_1) \bowtie W_2[T](t_1) \bowtie \cdots \bowtie W^+_k(t_1,t_2) \bowtie \cdots \bowtie W_m[T](t_1) \\
W^-_{out}(t_1,t_2) &= W_1[T](t_1) \bowtie W_2[T](t_1) \bowtie \cdots \bowtie W^-_k(t_1,t_2) \bowtie \cdots \bowtie W_m[T](t_1)
\end{aligned}$$

Definition 8 (Virtual window on an AS join output) Similar to Definition 7, the virtual window on an AS join output is defined as

$$W_{out}[T](t_1) = AS_1(t_1) \overset{F}{\bowtie} AS_2(t_1) \overset{F}{\bowtie} \cdots \overset{F}{\bowtie} AS_m(t_1)$$

and the corresponding virtual window increment and decrement are computed as follows.

$$\begin{aligned}
W^+_{out}(t_1,t_2) &= AS_1(t_1) \overset{F}{\bowtie} AS_2(t_1) \overset{F}{\bowtie} \cdots \overset{F}{\bowtie} W^+_k(t_1,t_2) \overset{F}{\bowtie} \cdots \overset{F}{\bowtie} AS_m(t_1) \\
W^-_{out}(t_1,t_2) &= AS_1(t_1) \overset{F}{\bowtie} AS_2(t_1) \overset{F}{\bowtie} \cdots \overset{F}{\bowtie} W^-_k(t_1,t_2) \overset{F}{\bowtie} \cdots \overset{F}{\bowtie} AS_m(t_1)
\end{aligned}$$

Given the definitions above, our query processing model handles an aggregation join query as follows. Windows (see Definition 1) are used on the join inputs, and the aggregation set update operator (see Definition 4) is used on the virtual window of the join output. Each window is updated incrementally with the tuples in the window increment and the window decrement, respectively. The query output is an aggregation set (see Definition 3), which is updated by the AS update operator on a virtual window for each tuple in the window increment and the window decrement, respectively. In Section 6 we present a query processing algorithm based on this model.

5. QUERY TRANSFORMATION RULE

In this section, we propose a query transformation rule developed for aggregation join queries on data streams. As mentioned in the Introduction, in order to make the query transformation rule work on data streams, the aggregation sets in a QEP should be updated incrementally and continuously, both before and after the transformation. To handle this problem, we use the AS update and AS join operators introduced in Section 4. Precisely, *only* the AS update operator is needed in an LAP and *both* operators are needed in an EAP.

There is a side effect of using the AS join operator. As mentioned earlier, we consider a window-based join in this paper. A window join is processed as multiple one-way window joins – that is, each new tuple arriving in one stream is matched with tuples in the windows of the other streams. By performing early aggregations in an EAP, one or more of these one-way window joins in an LAP is replaced by one-way AS joins in an EAP. This results in *different* join output schemas depending on which window joins are replaced because the join output schema of a one-way AS join is different from that of a window join or another one-way AS join. To handle this side effect, in the transformed plan we *always* keep a late aggregation (LA) operator in its original position. This LA operator guarantees that the schema of the aggregation join query output is the same even though the schemas of one-way join outputs are different. This guarantee is due to the fact that two different tuples with the same grouping attribute value are put into the same group.

Example 3 In Example 2, for each tuple (*ts, A.auctionID*, ...) in stream A, the one-way AS join from A to AS_2 produces a sequence of output tuples u with the schema (*ts,A.auctionID, B.auctionID, COUNT*(*B*.∗)). Continuing with this example, for each tuple (*ts, B.auctionID*, ...) in stream *B*, the window join from *B* to W_A produces a sequence of output tuples *v* with the schema (*ts, A.auctionID, B.auctionID*, ...). The schema of *u* is different from that of *v*. By retaining the LA operator on the join output, two tuples *u* and *v* that have the same *A.auctionID* are put into the same group and the query output always has the schema AS_{out}(*A.auctionID, COUNT*(*B*.∗)).

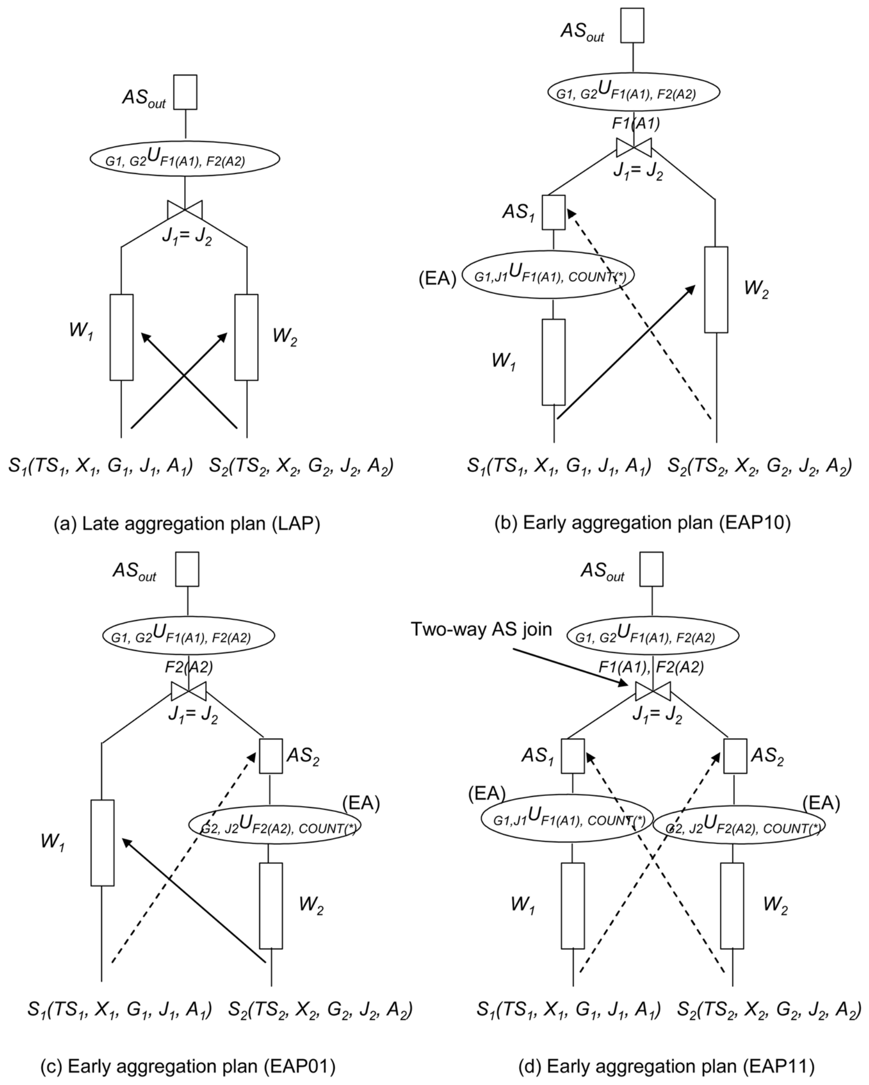
We now summarize the query transformation rule. This rule specifies how to construct an early aggregation operator which is specified in two parts, the grouping attributes and the aggregation functions. These two parts are determined based on the composition of the attributes of the input stream on which the early aggregation operator is placed.

Rule 1 (Query transformation rule) Given an LAP of an aggregation join query, an equivalent EAP is obtained by placing one or more early aggregation (EA) operators on any of the input streams of the LAP. Once placed on a certain input stream, the operator generates an AS, and thus an AS join to the AS is used instead of the window join to the input stream window. For those EA operators placed, their grouping attributes and aggregation functions are determined as follows:

- **Grouping attributes in an EA operator:** If the EA operator is placed on a stream that has some or all of the grouping attributes in the query, then use these and the join attributes as the grouping attributes of the EA operator. Otherwise, use only the join attributes as the grouping attributes of the EA operator.
- **Aggregation functions in an EA operator:** If the EA operator is placed on a stream that has all the aggregation attributes in the query, then use the aggregation function in the query as the aggregation function of the EA operator. If the stream has only some (not all) aggregation attributes in the query, then use both the aggregation function in the query and COUNT(∗) as the aggregation function of the EA operator. Otherwise, use only COUNT(∗) as the aggregation function of the EA operator.

In the transformation rule, an EA operator may be placed on any of the input streams, thus, given a LAP with *m* input streams, there are $2^m - 1$ possible equivalent EAPs. Determining the input streams on which to place EA operators is based on the resulting EAPs' execution times as estimated using cost functions (see Section 6.3).

Figure 3 illustrates transformations of an aggregation join query with two input streams, obtained by applying the rule. It shows all four possible QEPs when the aggregation attributes are split into two streams. In Figure 3(b), an EA operator is placed on S_1. By the transformation rule, the grouping attributes of the EA operator are G_1 (grouping attribute of the query) and J_1 (join attribute) and the aggregation functions of the EA operator are $F_1(A_1)$ which is the aggregation function of the query



AS_out
G1, G2 U F1(A1), F2(A2)
J1 = J2
W1
W2
S1(TS1, X1, G1, J1, A1) S2(TS2, X2, G2, J2, A2)
(a) Late aggregation plan (LAP)
F1(A1)
AS1
(EA)
G1,J1 U F1(A1), COUNT(*)
(b) Early aggregation plan (EAP10)
F2(A2)
AS2
G2, J2 U F2(A2), COUNT(*)
(c) Early aggregation plan (EAP01)
Two-way AS join
F1(A1), F2(A2)
(d) Early aggregation plan (EAP11)

An arrow from an input stream to the window of another stream denotes a window join (see Section 3) and an arrow from an input stream to the aggregation set of the window on another stream denotes an AS join (Definition 6).

Figure 3. Transformations of aggregation (two-way) join QEPs on data streams. (The aggregation attributes are split into two streams.)

and COUNT(∗). The EA operator includes these two aggregation functions because the input stream S_1 contains *some* aggregation attributes of the query. In Figure 3(c), an EA operator placed on S_2 is constructed similarly to the EA operator placed on S_1. Note that in these two figures, since the EA operator is placed on only one input stream, a one-way AS join is used (instead of a one-way window join) between an input stream and an AS. In Figure 3(d), two EA operators are placed on both input streams and a two-way AS join is used instead of a two-way window join.

With the window join and AS join in place, the four QEPs in Figure 3 are equivalent. The following example illustrates the equivalence of two QEPs shown in Figures 3a and 3c. (We will show the equivalence of the transformed QEPs algebraically in Section 6.2.)

Example 4 (LAP vs. EAP) Consider the QEPs shown in Figures 3a and 3c, and assume that both aggregation function F_1 and F_2 are SUM. Then, the query output AS_{out} is updated in each QEP as follows:

In LAP (Figure 3a), a window join is performed from S_1 to W_2. Assume that, for each tuple $s_1(ts_1, x_1, g_1, j_1, a_1) \in W_1^+$, the tuple matches c tuples, $\{s_2(ts_{2i}, x_{2i}, g_{2i}, j_2, a_2), i = 1, 2, ..., c\}$ where $s_2.j_2 = s_1.j_1$, in W_2. Then, the window join generates c output tuples, $\{u(ts_1, x_1, g_1, j_1, a_1, ts_{2i}, x_{2i}, g_{2i}, j_2, a_{2i}) | i = 1, 2, ..., c, j_2 = j_1\}$. Further assume that, among these c output tuples, c_g tuples have the same value, g_2, for g_{2i}, and hence the same value, (g_1, g_2), for (g_1, g_{2i}). Then, for a tuple in $AS(G_1, G_2, SUM(A_1), SUM(A_2))$ whose value of (G_1, G_2) equals (g_1, g_2), the value of $SUM(A_1)$ is increased by $a_1 * c_g$ and the value of $SUM(A_2)$ is increased by $v_2 = \Sigma\, a_{2i}$, $i = 1, 2, ..., c_g$.

In EAP (Figure 3c), an AS join is performed from S_1 to AS_2. Assume that, for each tuple $s_1(ts_1, x_1, g_1, j_1, a_1)$, it matches one tuple, $l_2(g_2, j_2, v_2, c_g)$ where $j_2 = j_1$ and $\Sigma\, a_{2i}$, $i = 1, 2, ..., c_g$, in $AS_2(G_2, J_2, SUM(A_2), COUNT(*))$. Then, the AS join generates an output tuple $u(ts_1, x_1, g_1, j_1, a_1 * c_g, g_2, j_1, v_2, c_g)$ (see Definition 5 for the coalesced value $a_1 * c_g$). This tuple is input to the AS update operator, which then makes the same update (i.e., $a_1 * c_g$ and v_2) on the aggregation set AS.

6. QUERY PROCESSING ALGORITHM, TRANSFORMATION EQUIVALENCE, AND COST FUNCTIONS

In this section, we first present a generic algorithm for executing a QEP, i.e. a late aggregation plan (LAP) or an early aggregation plan (EAP). Then, based on the algorithm, we show the equivalence of the LAP and EAPs, and build generic cost functions of them.

6.1 Generic Algorithms for Query Processing

Algorithm 1 outlines a high-level algorithm for processing tuples with a multi-way join among m ($m \geq 2$) streams $S_1, S_2, ..., S_m$[4]. The algorithm is generic enough to cover any of the possible QEPs. It updates the output aggregation set AS_{out} for each tuple S_k in the window increment W_k^+ and each tuple r_k in the window decrement W_k^-. The algorithm performs (1) AS updates on the output of an EA operator in lines 3 and 9 if there exists an EA operator on S_k, (2) window updates in lines 4 and 10, (3) either AS joins or window joins in lines 5 and 11 depending on whether an EA operator is placed on S_k, and (4) AS updates on the query output AS_{out} in lines 6 and 12.

As mentioned in the Introduction, our query processing algorithm can be used to execute a *stream-relation* join as well. In this case, a relation can be viewed as a window with no update of tuples, and therefore an aggregation set produced by an EA

[4]This algorithm processes tuples in pipelined fashion, but it may be queue-based as well. The query transformation works well with both types of algorithms.

operator on a relation is fixed.

```
Input:
  • $W_1, W_2, \cdots, W_m$;
  • $AS_{i_1}, AS_{i_2}, \cdots, AS_{i_p}$: EA output aggregation sets ($p \leq m$);
  • $W_k^+$; $W_k^-$; $AS_{out}$
Output: $AS_{out}$: updated query output aggregation set.
1  begin
2      for each tuple $s_k$ in $W_k^+$ do
3          If there exists an EA operator on $S_k$, then with $s_k$ find its group in $AS_k$ and update the
           aggregate value. (AS update on EA output);
4          Add $s_k$ to $W_k$. (Window update);
5          With $s_k$, find matching tuples in either $AS_j$ or $W_j$ for each $j = 1, 2, ..., k-1, k+1, ..., m$,
           depending on whether an EA operator is placed on $S_k$ (then $AS_j$) or not (then $W_j$). (Window
           join or AS join);
6          For each tuple produced in line 5, find its group in $AS_{out}$ and update the aggregate value.
           (AS update on query output);
7      end
8      for each tuple $r_k$ in $W_k^-$ do
9          If there exists an EA operator on $S_k$, then with $r_k$ find its group in $AS_k$ and update the
           aggregate value. (AS update on EA output);
10         Remove $r_k$ from $W_k$. (Window update);
11         With $r_k$, find matching tuples in either $W_j$ or $AS_j$ for each $j = 1, 2, ..., k-1, k+1, ..., m$,
           depending on whether an EA operator is placed on $S_k$ (then $AS_j$) or not (then $W_k$). (Window
           join or AS join);
12         For each tuple produced in line 11, find its group in $AS_{early}$ and update the aggregate value.
           (AS update on query output);
13     end
14 end
```

Algorithm 1. A generic QEP-execution algorithm.

6.2 Equivalence of Query Transformations

In this subsection, we show the equivalence between an LAP and EAPs generated using the query transformation rule presented in Section 5. We first prove, in Theorem 1, the equivalence considering the case in which *both* input streams have grouping attributes and aggregation attributes of the query (see Figure 3d), as this is the most general case. Then, in Corollaries 1 and 2, we prove the equivalence for more special cases (see Figures 3b, and c). A two-way join case is considered first for simplicity (in Theorem 1, Corollary 1, and Corollary 2), and then it is extended to a multi-way join case in Theorem 2.

Theorem 1 Consider an aggregation two-way join query, and assume that there are grouping attributes in *both* streams and aggregate attributes in *both* streams. Additionally, consider an EAP that can be generated by placing EA operators on *both* input streams in an LAP according to the Rule 1 (see EAP11 in Figure 3d). Then, the aggregation set produced by the LAP (see Figure 3a) is always the same as the aggregation set produced by the EAP.

Proof. The LAP (Figure 3a) is executed in the following steps based on the QEP execution algorithm (Algorithm 1).

$$W_1 \leftarrow W_1 \cup \{s_1\} \quad \text{: Window update} \tag{1}$$

$$T^{+}_{late} \leftarrow \{s_1\} \bowtie_{J_1=J_2} W_2 \quad \text{: Window join} \tag{2}$$

$$AS_{late} \leftarrow {}_{G_1,G_2}\mathcal{U}_{F_1(A_1),F_2(A_2)}(AS_{late}, T^{+}_{late}, \emptyset) \quad \text{: AS update on } AS_{out} \text{ (Def. 4)} \tag{3}$$

$$W_1 \leftarrow W_1 - \{r_1\} \quad \text{: Window update} \tag{4}$$

$$T^{-}_{late} \leftarrow \{r_1\} \bowtie_{J_1=J_2} W_2 \quad \text{: Window join} \tag{5}$$

$$AS_{late} \leftarrow {}_{G_1,G_2}\mathcal{U}_{F_1(A_1),F_2(A_2)}(AS_{late}, \emptyset, T^{-}_{late}) \quad \text{: AS update on } AS_{out} \tag{6}$$

where T_{late}^{+} is a virtual window increment due to the tuple s_1 in $W_1^{+}(t_1, t_2)$, and T_{late}^{-} is a virtual window decrement due to the tuple r_1 in $W_1^{-}(t_1, t_2)$. The EAP (Figure 3d) is executed in the following steps based on the QEP execution algorithm.

$$AS_1 \leftarrow {}_{G_1,J_1}\mathcal{U}_{F_1(A_1),COUNT(*)}(AS_1, \{s_1\}, \emptyset) \quad \text{: AS update on } AS_{out} \tag{7}$$

$$W_1 \leftarrow W_1 \cup \{s_1\} \quad \text{: Window update} \tag{8}$$

$$T^{+}_{early} \leftarrow \{s_1\} \overset{F_1(A_1)}{\bowtie}_{J_1=J_2} AS_2 \quad \text{: AS join} \tag{9}$$

$$AS_{early} \leftarrow {}_{G_1,G_2}\mathcal{U}_{F_1(A_1),F_2(A_2)}(AS_{early}, T^{+}_{early}, \emptyset) \quad \text{: } AS_{out} \text{ update} \tag{10}$$

$$AS_1 \leftarrow {}_{G_1,J_1}\mathcal{U}_{F_1(A_1),COUNT(*)}(AS_1, \emptyset, \{r_1\}) \quad \text{: EA-output AS update} \tag{11}$$

$$W_1 \leftarrow W_1 - \{r_1\} \quad \text{: Window update} \tag{12}$$

$$T^{-}_{early} \leftarrow \{r_1\} \overset{F_1(A_1)}{\bowtie}_{J_1=J_2} AS_2 \quad \text{: AS join} \tag{13}$$

$$AS_{early} \leftarrow {}_{G_1,G_2}\mathcal{U}_{F_1(A_1),F_2(A_2)}(AS_{early}, \emptyset, T^{-}_{early}) \quad \text{: AS update on } AS_{out} \tag{14}$$

T_{early}^{+} and T_{early}^{-} are the counterparts of T_{late}^{+} and T_{late}^{-}, respectively.

The LAP (Figure 3a) and the EAP (EAP11 in Figure 3d) are both symmetric. Thus, it suffices to prove the equivalence for one-way joins only. Let us arbitrarily choose the one-way window join from S_1 to W_2 for LAP and the one-way AS join from S_1 to AS_2 for EAP. We can prove the equivalence using induction.

- *Base case*: Initially, $AS_{late} \equiv AS_{early} = \emptyset$.

- *Inductive case*: If $AS_{late} \equiv AS_{early}$ holds at time t_1, then $AS_{late} \equiv AS_{early}$ holds at time $t_2 > t_1$ after being updated for each tuple s_1 in $W_1^{+}(t_1, t_2)$ and for each tuple r_1 in $W_1^{-}(t_1, t_2)$. To prove this inductive case, we need to show that AS_{late} in Equation 6 is equal to AS_{early} in Equation 14 after executing the algorithm for an arbitrary newly added tuple s_1 and an arbitrary removed tuple r_1. Here, since the equations for the decrement are parallel to the equations for the increment, it suffices to show the equivalence only for the increment, that is, for s_1 only. Let us show this now.

LAP case (AS_{late}): Assume the tuple $s_1(ts_1, x_1, g_1, j_1, a_1)$ in Equation 2 matches c

tuples $\{s_2(ts_{2_i}, x_{2_i}, g_{2_i}, j_2, a_{2_i}) | i = 1, 2, ..., c\}$ (where $s_1.j_1 = s_2.j_2$) in W_2. Then, the window join in Equation 2 generates c output tuples $\{u(ts_1, x_1, g_1, j_1, a_1, ts_{2_i}, x_{2_i}, g_{2_i}, j_2, a_{2_i}) | j_1 = j_2, i = 1, 2, ..., c\}$, which are inserted into T_{late}^{+}. Further assume that among the c matching tuples in W_2, c_{g_2} tuples have the same value of g_{2_i} (let us denote the same value as g_2). Then, there are c_{g_2} output tuples (each denoted as u) in T_{late}^{+} with the same grouping attribute value (g_1, g_2). In addition, all the c_{g_2} tuples in T_{late}^{+} have the same aggregate value a_1 and, therefore, the aggregation $F_1(A_1)$ of the c_{g_2} tuples is computed as $f(c_{g_2}, a_1)$ (see Definition 5 for $f(\cdot)$). For $F_2(A_2)$, since A_2 belongs to stream S_2, the aggregation is computed straightforwardly by applying F_2 on the a_2 values of the c_{g_2} tuples.

EAP case (AS_{early}): Consider Equation 9, which is an AS join for finding, from AS_2, tuples that match the tuple $s_1(ts_1, x_1, g_1, j_1, a_1)$. Note that AS_2 is the output of an aggregation operator with the schema of four attributes – two from grouping attributes (G_2, J_2) and two from aggregation functions $F_2(A_2)$ and *COUNT*. Thus, given the above tuple s_1, it is matched with the tuples in AS_2 via the join condition $S_1.J_1 = AS_2.J_2$. There exists only one matching tuple $l(g_2, j_2, v_2, c_{g_2})$ where $v_2 = F_2(A_2)$ and $c_{g_2} = COUNT(*)$ because c_{g_2} tuples with the same grouping attribute value (g_2, j_2) in LAP case are grouped into one tuple in AS_2 with count c_{g_2}. Then, by the definition of AS join (Definition 6), a single join output tuple (denoted as $u(ts_1, x_1, g_1, j_1, v_1, g_2, j_2, v_2, c_{g_2})$ where $j_1 = j_2$) is generated and inserted into T_{early}^{+}; in the tuple u, the aggregate value v_1 (of the attribute A_1) is computed as $v_1 = f(c_{g_2}, a_1)$, and the aggregate value v_2 (of the attribute A_2) is computed using the query aggregation function as $v_2 = F_2(A_2)$.

From the LAP case and the EAP case above, we conclude that AS update using T_{late}^{+} (in Equation 3) and AS update using T_{early}^{+} (in Equation 10) both update the tuple whose grouping attribute value equals (g_1, g_2) by the same value computed using $f(c_{g_2}, a_1)$ and $F_2(A_2)$.

Corollary 1 Consider an aggregation two-way join query, and assume that there are grouping attributes in *both* streams and aggregate attributes in *both* streams. Additionally, consider an EAP that can be generated by placing an EA operator on *only one* input stream in an LAP according to the Rule 1 (see EAP10 and EAP01 in Figures 3b and c). Then, the aggregation set produced by the LAP (see Figure 3a) is always the same as the aggregation set produced by the EAP.

Proof. EAP10 in Figure 3b and EAP01 in Figure 3c each have one window join and one AS join. Since these two EAPs are symmetric to each other, it suffices to show the equivalence for only one. Let us arbitrarily choose EAP01. First, the window join from S_2 to W_1 in EAP01 is identical to that in LAP. Second, the AS join (from S_1 to AS_2) in EAP01 is equivalent to the other window join (from S_1 to W_2) in LAP, as already proven in Theorem 1 (see the LAP case (AS_{late}) and the EAP case (AS_{early}) in the theorem). Hence, LAP and EAP10 are equivalent.

Corollary 2 Consider an aggregation two-way join query, and assume that there are grouping attributes in *both* streams but aggregation attributes in *only one* stream. Additionally, consider an EAP that can be generated by placing an EA operator on either *only one* input stream according to the Rule 1 or *both* input streams. Then, the aggregation set produced by the LAP is always the same as the aggregation set produced by the EAP.

Proof. The query considered in this corollary is a special case of the query considered in Theorem 1 and Corollary 1 in that the set of aggregation attributes in one stream is empty. Since the four QEPs in Figure 3 are equivalent by Theorem 1 and Corollary 1, we need only show that the QEPs in Figure 3 are reduced to the QEPs in this corollary in the special case. Since EAP10 and EAP01 in Figure 3 are symmetric to each other, we can arbitrarily choose either one of A_1 and A_2 to be empty. Let us assume A_2 is empty. In this case, first, there is no aggregation function $F_2(A_2)$ in any of the QEPs, since A_2 does not exist. Second, there is no aggregation function *COUNT*(∗) in the EA operator that is placed on the stream with aggregation attributes. This is because the aggregate value of output tuples is equal to $F_1(A_1)$ only, which is calculated by the EA operator. When we apply these two changes, the QEPs in Figure 3 become identical to the QEPs in this corollary.

Theorem 2 Consider an aggregation *multi-way* join query. Additionally, consider an EAP that can be generated by placing EA operators on one or more input streams in an LAP according to the Rule 1. Then, the aggregation set produced by the LAP is always the same as the aggregation set produced by the EAP.

Proof. Let $m(>2)$ be the arity of a multi-way join. Then, the m-way join can be executed as a sequence of $m-1$ two-way joins or, equivalently, $m-1$ pairs of one-way joins. The LAP has a sequence of $m-1$ pairs of one-way window joins and one aggregation operator at the output of the join sequence. The alternative QEPs (i.e., EAPs) generated by the rules differ in where EA operators are placed among the m input streams and, depending on whether an EA operator has been placed on the joined stream or not, each one-way join is either a window join or an AS join (with the associated AS update). In other words, an EAP has a sequence of $m-1$ pairs of one-way window or AS joins and one aggregation operator at the output of the join sequence. Note that, by Theorem 1 and Corollary 1, the results from intermediate two-way joins, generated in the LAP and any of the EAPs, are always equivalent. Therefore, it naturally holds that LAP and EAPs are equivalent for a multi-way join.

6.3 Generic Cost Functions

We use a *unit-time* cost model, proposed by Kang et al. in [Kang et al., 2003], as the cost metric. Given a QEP, it estimates the time to process tuples arriving in unit time. We consider two-way joins for simplicity; it is straightforward to extend it for multi-way joins. Table 2 summarizes the notations used in the formulas (m_i and n_i are derived parameters. Their derivation will be explained in Section 7.1.1).

Generic unit-time cost functions of four possible QEPs are formulated as follows. We believe the terms in the formulas are evident from the algorithm (Algorithm 1).

Generic cost function of an LAP:

$$\begin{aligned} C_{LA} &= \lambda_1[C_{uw}(W_1) + C_{fmt}(W_2) + m_2 C_{iau}(AS)] + \\ &\quad \lambda_2[C_{uw}(W_2) + C_{fmt}(W_1) + m_1 C_{iau}(AS)] \end{aligned} \tag{15}$$

Generic cost functions of EAPs:

$$\begin{aligned} C_{EA10} &= \lambda_1[C_{uw}(W_1) + C_{fmt}(W_2) + m_2 C_{iau}(AS) + C_{iau}(AS_1)] + \\ &\quad \lambda_2[C_{uw}(W_2) + C_{fmt}(AS_1) + n_1 C_{iau}(AS)] \end{aligned} \tag{16}$$

$$\begin{aligned} C_{EA01} &= \lambda_1[C_{uw}(W_1) + C_{fmt}(AS_2) + n_2 C_{iau}(AS)] + \\ &\quad \lambda_2[C_{uw}(W_2) + C_{fmt}(W_1) + m_1 C_{iau}(AS) + C_{iau}(AS_2)] \end{aligned} \tag{17}$$

$$\begin{aligned} C_{EA11} &= \lambda_1[C_{uw}(W_1) + C_{fmt}(AS_2) + n_2 C_{iau}(AS) + C_{iau}(AS_1)] + \\ &\quad \lambda_2[C_{uw}(W_2) + C_{fmt}(AS_1) + n_1 C_{iau}(AS) + C_{iau}(AS_2)] \end{aligned} \tag{18}$$

Using these cost functions, the query optimizer can estimate the costs of the QEPs (i.e., LAP and all possible EAPs) and then select the one that has the smallest estimated execution time. By the design of an early aggregation in an EAP, the EAP costs (especially C_{EA11}) are typically lower than the LAP cost C_{LA}. But this is not guaranteed, and any of the QEPs may be the "winner" depending on the input stream statistics (e.g., stream rates, join selectivity factors, number of groups). For instance, the performance benefit of an EAP diminishes as the number of groups increases. Thus, the selection of the most efficient QEP can be changed over time if the input stream statistics change.

Table II. Notations in generic cost formulas.

Cost terms

Notation	Meaning
C_{LA}	Cost of an LAP.
C_{EA01}	Cost of an EAP when an EA operator is placed on S_2 only.
C_{EA10}	Cost of an EAP when an EA operator is placed on S_1 only.
C_{EA11}	Cost of an EAP when EA operators are placed on both S_1 and S_2.
$C_{fmt}(W_i)$	Cost of finding matching tuples in W_i for each new arrival tuple.
$C_{uw}(W_i)$	Cost of updating W_i for each new arrival tuple.
$C_{iau}(AS_i)$	Cost of AS update on the aggregation set AS_i for each new arrival tuple.

Input stream and query statistics

Notation	Meaning
λ_i	Stream rate of S_i, i.e., the average number of tuples arriving at stream S_i in unit time ($i = 1, 2$).
m_i	Join cardinality of W_i, i.e., the average number of matching tuples found in W_i for each tuple in stream $S_{\bar{i}}$ ($i = 1, 2$). (derived)
n_i	Join cardinality of AS_i, i.e., the average number of matching tuples found in AS_i for each tuple in stream $S_{\bar{i}}$ ($i = 1, 2$). (derived)

$S_{\bar{i}}$ denotes the stream on the opposite side.

7. EVALUATIONS

In this section, we validate the cost functions and then study the performance of the proposed query transformations with a focus on the QEP efficiencies. There are three objectives of our experiments: (1) validate the cost functions by comparing the execution times of QEPs between the cost functions and the prototype program, (2) examine the performance trends of the alternative QEPs for varying key parameter values and (3) show cases of each alternative QEP being the most efficient one in relation to the parameter values. The first objective is important since a valid cost function enables a query optimizer to choose the right QEP in a set of equivalent QEPs. The last two objectives are to confirm the need for a (transformation-based) query optimizer.

We first instantiate the generic cost functions for different implementation choices and describe a setup for the experiments in Section 7.1. Then, we present the experiments and their results in Section 7.2.

7.1 Setup

7.1.1 Implementation-specific cost functions

In the implementation, we consider the nested loop join and the hash join for AS and window joins. We consider hash-based grouping for all aggregation sets. Equations 19 through 22 show the cost functions instantiated from the generic models (Equations 15 through 18) when using a nested loop join, and Equations 23 through 26 show those when using a hash join. Notations used in the formulas are summarized in Table 3. We assume the grouping attributes, aggregation attributes, and join attributes are independent and that each of them has the uniform distribution. In the experiments we use the query illustrated in Figure 3, with SUM used as the aggregation function for both F_1 and F_2.

$$CN_{LA} = \lambda_1(2w_2P_n+2U_n+2m_2\frac{\alpha}{B_g}U_g) + \lambda_2(2w_1P_n+2U_n+2m_1\frac{\alpha}{B_g}U_g) \quad (19)$$

$$CN_{EA10} = \lambda_1(2w_2P_n+2U_n+2\frac{\alpha_1}{B_g}U_g+2m_2\frac{\alpha}{B_2}U_g) + \lambda_2(2\alpha_1P_n+2U_n+2n_1\frac{\alpha}{B_g}U_g) \quad (20)$$

$$CN_{EA01} = \lambda_1(2\alpha_2P_n+2U_n+2n_2\frac{\alpha}{B_g}U_g) + \lambda_2(2w_1P_n+2U_n+2m_1\frac{\alpha}{B_g}U_g+2\frac{\alpha_2}{B_2}U_g) \quad (21)$$

$$CN_{EA11} = \lambda_1(2\alpha_2P_n+2U_n+2\frac{\alpha_1}{B_1}U_g+2n_2\frac{\alpha}{B_g}U_g) + \lambda_2(2\alpha_1P_n+2U_n+2\frac{\alpha_2}{B_2}U_g+2n_1\frac{\alpha}{B_g}U_g) \quad (22)$$

$$CH_{LA} = \lambda_1(2\frac{w_2}{B_2}P_h+2U_h+2m_2\frac{\alpha}{B_g}U_g) + \lambda_2(2\frac{w_1}{B_1}P_h+2U_h+2m_1\frac{\alpha}{B_g}U_g) \quad (23)$$

$$CH_{EA10} = \lambda_1(2\frac{w_2}{B_2}P_h+2U_h+2\frac{\alpha_1}{B_1}U_g+2m_2\frac{\alpha}{B_g}U_g) + \lambda_2(2\frac{\alpha_1}{B_1}P_h+2U_h+2n_1\frac{\alpha}{B_g}U_g) \quad (24)$$

$$CH_{EA01} = \lambda_1(2\frac{\alpha_2}{B_2}P_h+2U_h+2n_2\frac{\alpha}{B_g}U_g) + \lambda_2(2\frac{w_1}{B_1}P_h+2U_h+2\frac{\alpha_2}{B_2}U_g+2m_1\frac{\alpha}{B_g}U_g) \quad (25)$$

$$CH_{EA11} = \lambda_1(2\frac{\alpha_2}{B_2}P_h+2U_h+2\frac{\alpha_1}{B_1}U_g+2n_2\frac{\alpha}{B_g}U_g) + \lambda_2(2\frac{\alpha_1}{B_1}P_h+2U_h+2\frac{\alpha_2}{B_2}U_g+2n_1\frac{\alpha}{B_g}U_g) \quad (26)$$

In these cost formulas, the parameters α, α_i, m_i, n_i, B_i and B_g are derived from w_i, σ_i, and g_i ($i = 1, 2$). Details of the derivations appear in Appendix A. The model tuning parameters P_n, U_n, P_h, U_h and U_g are constant processing costs per tuple. Their values are obtained by measuring the execution time for processing 10,000 tuples and computing the average per tuple. Note that each term in the cost functions has a factor of two, since the cost is measured for one tuple in the window increment and one tuple in the window decrement.

7.1.2 Prototype

The prototype is a program that implements the QEP algorithm shown in Algorithm 1. It is programmed to execute a multi-way join, but we use only two-way joins in most

Table III. Notations used in the cost formulas.

Cost terms

Notation	Meaning
CN_{LA}, CH_{LA}	C_{LA} when using a nested loop join and a hash join, respectively.
CN_{EA01}, CH_{EA01}	C_{EA01} when using a nested loop join and a hash join, respectively.
CN_{EA10}, CH_{EA10}	C_{EA10} when using a nested loop join and a hash join, respectively.
CN_{EA11}, CH_{EA11}	C_{EA11} when using a nested loop join and a hash join, respectively.

Input stream and query statistics

Notation	Meaning
σ_i	Average join selectivity factor of tuples in window WSi for each tuple arriving at stream $S_{\bar{i}}$ $(i=1, 2)$.
g_i	Maximum number of groups in stream S_i.
w_i	Size (i.e., number of tuples) of window W_i $(i=1, 2)$.
α	Size (i.e., number of tuples) of aggregation set AS (derived).
α_i	Size (i.e., number of tuples) of aggregation set AS_i $(i=1, 2)$ (derived).

System parameters

Notation	Meaning
B_i	Number of buckets in the hash table created on the join attribute in stream S_i $(i=1, 2)$ (derived).
B_g	Number of buckets in the hash table created on the grouping attribute (de rived).
B	Maximum size (i.e., number of buckets) allowed for a hash table.

Cost function tuning parameters

Notation	Meaning
P_n, P_h	Per-tuple cost of probing a window or an aggregation set in a nested loop join and a hash join, respectively.
U_n, U_h	Per-tuple cost of updating a window in a nested loop join and a hash join, respectively.
U_g	Per-tuple cost of finding and updating a group (tuple) within an aggregation set.

$S_{\bar{i}}$ denotes the stream on the opposite side.

experiments; we add one experiment later using a three-way join to show the consistency of the results regardless of the join arity. The prototype uses the same join methods and grouping methods as those assumed in the cost functions (see Section 7.1.1). Additionally, it executes a join using *sliding* windows, of which tumbling and landmark windows are only special types (see Section 3).

Inputs to the prototype program are data streams generated using a data generator[5] (described below), the join arity (i.e., number of data streams) (m), the size of each join window (w_1, w_2), and the QEP case number (0 for LAP, 1, 2, 3, ... $2^m - 1$ for EAPs). It then processes the input stream data according to the specified QEP and reports the execution time. This task is performed by two processes running concurrently: one process reads new arrival tuples from the input stream data files and feeds them to the other process for join execution. The program has been written in Java 2 SDK 1.4.2, and runs on a Linux PC with Pentium IV 1.6GHz processor and 512MB RAM.

The data generator generates stream data sets as a sequence of tuples. Inputs to the data generator are the number of tuples in the data set, the number of attributes in the stream schema, the stream rate (i.e., number of tuples per second), the number of groups in the stream, and the number of distinct values of the join attribute. (A join selectivity factor equals the reciprocal of the number of distinct values of the join attribute.) Each tuple has a timestamp attribute, whose value is determined based on the stream rate. It also has other attributes such as join attribute, grouping attribute, and aggregation attribute. As mentioned in Section 7.1.1, values of these attributes are independent and assigned randomly with the uniform distribution. We use the string data type for grouping and join attributes and the integer data type for aggregation attribute.

7.2 Experiments and Results

In this section, we first validate the cost functions in Section 7.2.1. Then, in Section 7.2.3, we build showcases of different alternative QEPs being the most efficient ones. In all the experiments, the execution time of a QEP is reported per time-unit (second). For this, we measure the execution time for tuples arriving in 1000 milliseconds. We run each experiment three times, for one time-unit at each run, and compute the average execution time in seconds.

7.2.1 Cost function validations

A valid cost function enables a query optimizer to choose the right QEP that is the most efficient among all alternative QEPs considered. In this regard, we compare the relative efficiencies among alternative QEPs (generated by the query transformation) between those obtained using the cost function and those obtained using the prototype. In each set of experiments, we measure the execution time of QEPs by varying one of the four pairs of parameters: (1) window size (w_1, w_2), (2) number of groups (g_1, g_2), (3) stream rate (λ_1, λ_2), and (4) join selectivity factor (σ_1, σ_2). Furthermore, for each

[5]The data generator allows us to vary the input stream statistics so that we can evaluate the efficiencies of alternative QEPs with different input parameters.

pair of parameters we vary only the parameters of stream S_1 (i.e., w_1, g_1, λ_1 and σ_1), since the QEPs are symmetric.

Figure 4 shows a comparison between the cost function and the prototype. (As mentioned in the Introduction section, we show only the results obtained using the hash join, and refer the readers to [Tran and Lee 2007] for the results obtained using the nested loop join.) The cost function tuning parameters (P_n, P_h, U_n, U_h, U_g in Table 3) are tuned separately in each set of experiments. The same setting has been used for other parameters (λ_i, w_i, g_i, and σ_i for $i = 1, 2$) across the four sets of experiments in each case of varying the parameters.

In the figure, the shapes of performance curves are very similar between the cost function and the prototype. Moreover, the ranking of efficiencies among alternative QEPs is the same between them most of the time. This confirms the precision of the cost functions as usable by a query optimizer.

7.2.2 Query execution costs for varying stream statistics

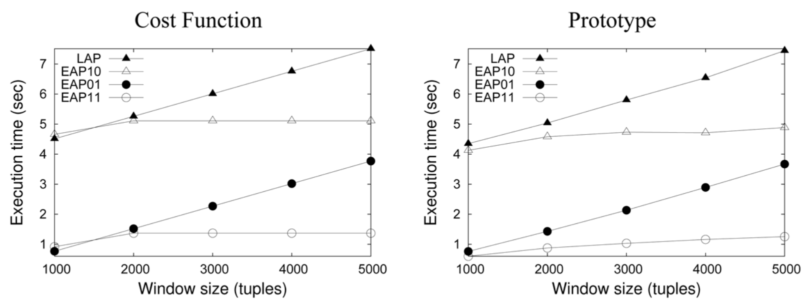
Figure 4 also shows the performance of four alternative QEPs (i.e., LAP, EAP01, EAP10, EAP11) depending on each of the four parameters. Let us now examine the results of each set of experiments to each varying parameter (i.e., window size, number of groups, stream rate, join selectivity factor).

Varying window size: We vary w_1 from 1000 to 5000 tuples at the increment of 1000 while fixing w_2 at 5000. In the figure, as w_1 increases, LAP and EAP01 increase linearly. In contrast, EAP10 and EAP11 initially increase linearly but then stay constant as w_1 exceeds 2000. The reason for this is as follows: In LAP and EAP01, there is no EA operator placed on S_1 and, therefore, the execution time depends on w_1 only. Unlike this, in EAP10 and EAP11 which have an EA operator placed on S_1, the cost stops depending on w_1 but starts depending on α_1 (which is fixed) when w_1 is greater than 2000 (see α_1 in Appendix A). Additionally, EAPs are always better than LAP because in these experiments, α_1 and α_2 are set smaller than window size w_1 and w_2.

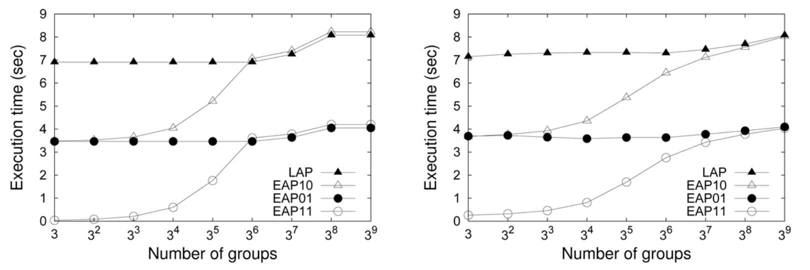
Varying the number of groups: We vary g_1 from 3 to 19683 (= 3^9) by a factor of 3. In the figure, as g_1 increases, EAP10 increases and approaches LAP and, likewise, EAP11 increases and approaches EAP01. The initial increase of EAP10 and EAP11 is caused by the increase of the aggregation set size (α_1). But, as g_1 becomes large enough ($g_1 = 3^8$), α_1 stops depending on $g_1\frac{1}{\sigma_1}$ and starts depending on w_1 (see α_1 in Appendix A). As a result, EAP10 and EAP11 loses the advantage of placing an EA operator on S_1.

Varying stream rate: We vary λ_1 from 100 to 900 tuples/second while fixing λ_2 at 500. In the figure, as λ_1 increases, the costs of all four QEPs increase linearly but LAP and EAP10 increase faster than EAP01 and EAP11. The reason is that the per-tuple processing time for each tuple from S_1 in EAP01 and EAP11 is shorter than that in LAP and EAP10, as it takes less time to find matching tuples in an aggregation set AS2 instead of W_2.

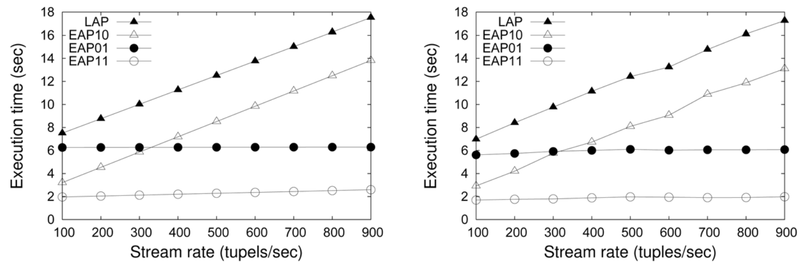
Varying join selectivity factor: We vary σ_1 and σ_2 equally from 0.0005 to 0.5 by a factor of 10. As $\sigma(\equiv \sigma_1 = \sigma_2)$ increases, the costs of all four QEPs increase when a hash join is used (see Figure 4d). The reason for the cost increase is that a higher join



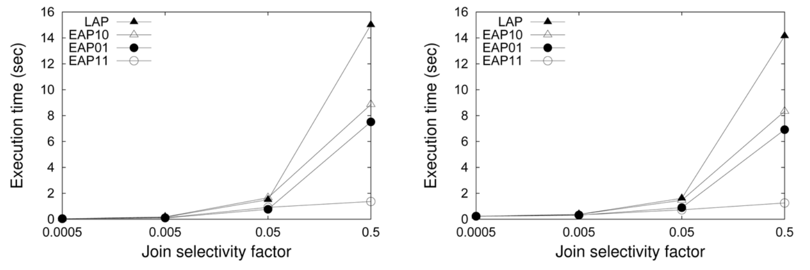
(a) Varying window size ($1000 \le w_1 \le 5000$, $w_2 = 2000$).



(b) Varying number of groups ($3 \le g_1 \le 19683 (= 3^9)$, $g_2 = 1$ (no grouping attribute in S_2)).



(c) Varying input stream rate ($100 \le \lambda_1 \le 900$, $\lambda_2 = 500$).



(d) Varying join selectivity factor ($0.0005 \le \sigma_1 = \sigma_2 \le 0.5$).

EAP01: an EA operator on S_2 only, EAP10: an EA operator on S_1 only, EAP11: EA operators on both S_1 and S_2.

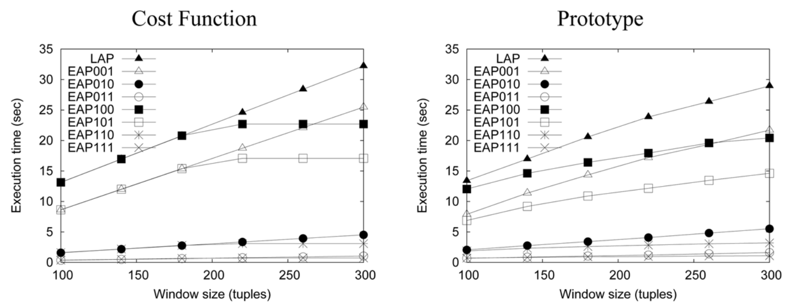
Default setting: $\lambda_1 = 300, \lambda_2 = 300, w_1 = 5000, w_2 = 5000, g_1 = 150, g_2 = 1, \sigma_1 = 0.1, \sigma_2 = 0.1$.

Figure 4. Execution times of QEPs between cost function and prototype (two-way hash join, $B = 1021$).

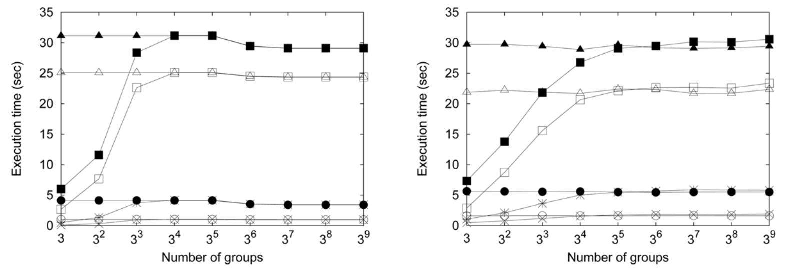
selectivity factor leads to more matching tuples from a join, which in turn leads to more tuples updating the query aggregation set (*AS*).

Extension to a three-way join

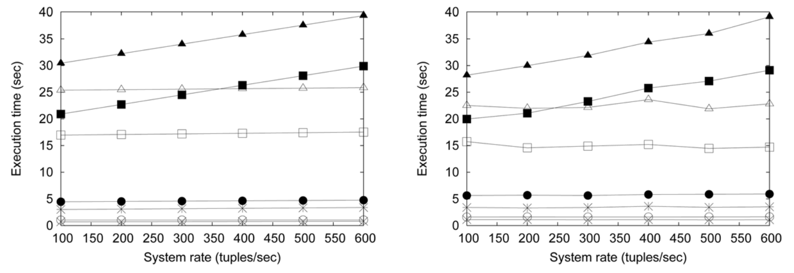
We have extended the experiments shown in Figures 4 to use three-way joins. Similar to the two-way join experiments, we vary the values of parameters in stream S_1 and fix those in the other streams (S_2 and S_3). The results are shown in Figures 5. There are *eight* alternative QEPs for a three-way join. The performance curves show the



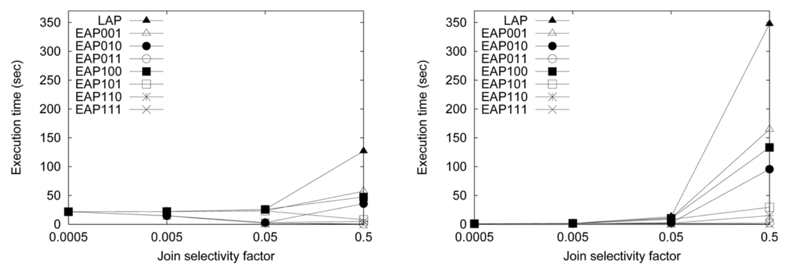
(a) Varying window size ($100 \le w_1 \le 300$, $w_2 = 300$, $w_3 = 300$).



(b) Varying number of groups ($3 \le g_1 \le 3^9$, $g_2 = 1$, $g_3 = 1$ (no grouping attribute in S_2 and S_3)).



(c) Varying input stream rate ($100 \le \lambda_1 \le 600$, $\lambda_2 = 200$, $\lambda_3 = 200$).



(d) Varying join selectivity factor ($0.0005 \le \sigma_1 = \sigma_2 = \sigma_3 \le 0.5$).

Default setting: $\lambda_1 = 200, \lambda_2 = 200, \lambda_3 = 200, w_1 = 300, w_2 = 300, w_3 = 300, g_1 = 20, g_2 = 1, g_3 = 1, \sigma_1 = 0.1, \sigma_2 = 0.1, \sigma_3 = 0.1$

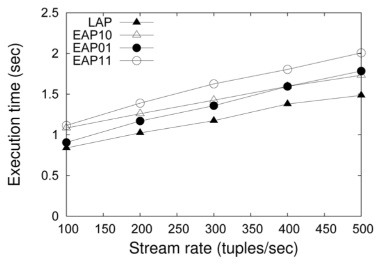
Figure 5. Execution times of QEPs between cost function and prototype (three-way hash join).

same trends as in the two-way join experiments. That is, there is a similarity between the cost function and the prototype, and the rankings of efficiencies among alternative QEPs are the same between them. Interestingly, the performance advantage of an early aggregation is bigger than the two-way join case. This is because the reduction of join cardinality is magnified as the arity of a join increases.

7.2.3 Showcases of different best QEPs

Intuitively, the advantage of an early aggregation is more highlighted when the number of groups (g_i) is smaller or the join selectivity factor (σ_i) is larger or the window size (w_i) is larger. Specifically, a decrease in the number of groups leads to a decrease of an EA output aggregation set size in an EAP, thus enhancing the benefit of join reduction due to early aggregation. On the other hand, an increase in the join selectivity factor or an increase in the window size leads to an increase of join output tuples in an LAP, thus increasing the penalty of late aggregation.

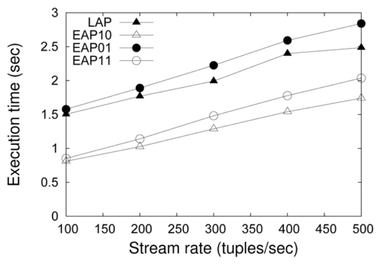
Figure 6 shows the cases where different QEPs are chosen as the most efficient one. The result confirms the intuition. That is, EAP11 is the best when both g_1 and g_2 are low, EAP10 is the best when g_1 is low and g_2 is high, EAP01 is the best when g_1 is high and g_2 is low, and LAP (or, "EAP00") is the best when both g_1 and g_2 are high. In Figure 6d, the scale of the graph is larger than those in the other figures (Figure 6a, b and c). This is because the execution times are much longer due to the higher join selectivity factors and larger window sizes used to generate the showcase.



$g_1 = 1500, g_2 = 1500, \sigma_1 = 0.01, \sigma_2 = 0.01,$

$w_1 = 500, w_2 = 500$

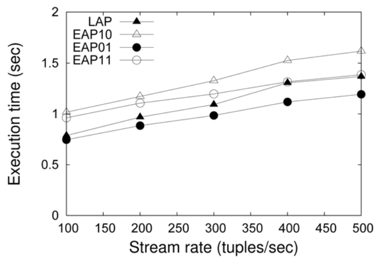
(a) LAP is the best.



$g_1 = 1, g_2 = 1500, \sigma_1 = 0.01, \sigma_2 = 0.01,$

$w_1 = 500, w_2 = 500$

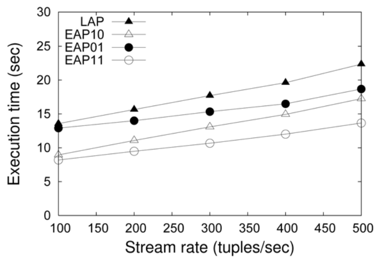
(b) EAP10 is the best.



$g_1 = 1500, g_2 = 1, \sigma_1 = 0.01, \sigma_2 = 0.01,$

$w_1 = 500, w_2 = 500$

(c) EAP01 is the best.



$g_1 = 150, g_2 = 150, \sigma_1 = 0.1, \sigma_2 = 0.1,$

$w_1 = w_2 = 2000$

(d) EAP11 is the best.

EAP01: an EA operator on S_2, EAP10: an EA operator on S_1, EAP11: EA operators on both S_1 and S_2.

Default setting: $100 \leq \lambda_1 \leq 500, \lambda_2 = 500$

Figure 6. Showcases of different best QEPs (using a two-way nested loop join).

8. CONCLUSION

In this paper, we addressed the problem of continuously processing aggregation join queries on data streams using query transformations. We proposed an incremental query processing model with two key stream operators, aggregation set update and aggregation set join. Based on the processing model, we presented a query transformation rule to generate alternative query execution plans depending on which input streams early aggregations are applied to. We then developed an algorithm for executing the query execution plans and built a prototype that implemented the algorithm. Based on the algorithm, we validated the rule theoretically through an inductive proof of the equivalence of alternative QEPs. Given alternative QEPs, a query optimizer needs the cost functions of individual QEPs to choose the best one with the minimum estimated cost. In this regard, we developed analytical cost functions and validated them through experiments. We also empirically studied the efficiencies of alternative QEPs and showed the cases of different best QEPs with respect to stream statistics.

Query transformation has been studied extensively in databases but not in data streams. To our knowledge, this is the first work addressing query transformation on aggregation join queries over data streams. Our query transformation rule is simple and yet generic to be applicable to any input streams. The results of our experiments indicate that the query transformation indeed generates alternative QEPs of which the efficiencies are distinct enough to influence a stream query optimizer.

In our work, the cost functions consider the execution time as the cost. Ideally, however, the space overhead should be part of the cost as well. The primary space overhead would be for storing the aggregation sets, and this overhead would depend on the number of distinct values of either or both of the grouping attributes and the join attributes in each aggregation set. Besides, the cost functions consider the execution time of only basic arithmetic aggregation functions such as SUM, COUNT, AVG, MIN, and MAX. The extension of the cost model to consider the storage space overhead and to work with more complex aggregation functions (e.g., nested-grouped aggregations, expensive statistical aggregations) is an interesting problem for future work.

We considered only the query transformation as an essential part of the query optimizer. Broader future work includes developing a comprehensive framework that integrates other components such as a stream statistic monitor and an efficient search algorithm for finding an optimal QEP. Regarding the latter, query optimization is done typically online in a data stream processing environment and thus its complexity may better be polynomial. In this case, one possible polynomial algorithm idea is based on a greedy heuristic. Specifically, an early aggregation is inserted on an input stream if the cost reduction resulting from using a one-way aggregation set join instead of a one-way window join is greater than the added cost of updating the aggregation set of the early aggregation.

ACKNOWLEDGMENTS

We thank Professor X. Sean Wang at University of Vermont for his comments on several technical aspects of the research. This research has been supported by

Vermont EPSCoR Grant No. 524376 and US National Science Foundation through Grant No. IIS-0415023. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

APPENDIX

A Derivations of Parameters in Implementation-Specific Cost Functions

In this section, the derivation of the parameters α, α_i, m_i, n_i, B_i and B_g, used in the cost formulas (Equations 19 to 26), are described in detail. We consider the case of the experimental query (Figure 3), in which *both* streams have grouping attributes and aggregation attributes.

• $\alpha = \min(w_1, g_1) * \min(w_2, g_2)$: AS is the result of grouping tuples produced from the window join of W_1 and W_2 on the grouping attributes G_1 and G_2. Thus, its size is capped to the product of the number of possible value of G_1 in W_1 and the number of possible values of G_2 in W_2. Let us denote these two numbers as k_1 and k_2. Then, $\alpha \leq k_1 \times k_2$. In addition, k_1 is the smaller between the maximum number of groups in the stream S_1 and the number of tuples (i.e., size) in the window W_1, that is, $k_1 = \min(g_1, w_1)$ and, likewise, $k_2 = \min(g_2, w_2)$. Hence, $\alpha \leq \min(g_1, w_1) \times \min(g_2, w_2)$. We use the upper bound as the value of α so $\alpha = \min(g_1, w_1) \times \min(g_2, w_2)$.

• $\alpha_1 = \min(w_1, g_1\frac{1}{\sigma_1})$: AS_1 is the output of an EA operator that groups on both grouping attribute G_1 and join attribute J_1 (Figure 3(d)). Therefore, the size of AS_1 (α_1) is no larger than the maximum number of possible groups (on G_1 and J_1) in the stream S_1. This maximum number is the product of g_1 and the number of distinct values of join attribute (i.e., join cardinality $= \frac{1}{\sigma_1}$) in S1. Thus, $\alpha \leq g_1 \times \frac{1}{\sigma_1}$. Moreover, the size of AS_1 is limited by the window size w_1 as well. Hence, α_1 is estimated as the smaller of the two (i.e., $g_1\frac{1}{\sigma_1}$ and w_1).

• $\alpha_2 = \min(w_2, g_2\frac{1}{\sigma_2})$: This is symmetric to the case of α_1.

• $m_1 = \alpha_1 w_1$: m_1 is the average number of matching tuples found in W_1. Thus, it can be estimated by the product of join selectivity factor σ_1 and window size w_1.

• $m_2 = \sigma_2 w_2$: This is symmetric to the case of m_1.

• $n_1 = \min(w_1\sigma_1, g_1)$: Since AS_1 is the output of an EA operator that groups on both grouping attribute G_1 and join attribute J_1, n_1 (the average number of matching tuples for each join attribute value) is either the number of groups g_1 or the number of matching tuples found in W_1 ($= w_1\sigma_1$), whichever is smaller.

• $n_2 = \min(w_2\sigma_2, g_2)$: This is symmetric to the case of n_1.

• $B_1 = \min(1/\sigma_1, B)$: This equation estimates hash table size on join attribute in stream S_1. We set the hash table size equal to the join cardinality in stream S_1 ($= \frac{1}{\sigma_1}$), capped by the maximum hash table size B.

• $B_2 = \min(1/\sigma_2, B)$: This is symmetric to the case of B_1.

• $B_g = \min(g_1, B)$: The hash table of ASis constructed on grouping attribute G_1. Therefore, the number of buckets equals the number of groups g_1, capped by B.

REFERENCES

ABADI, D. J., D. CARNEY, U. ÇETINTEMEL, M. CHERNIACK, C. CONVEY, S. LEE, M. STONEBRAKER,

N. Tatbul, and S. B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12:120–139.

Arasu, A. and G. S. Manku. 2004. Approximate counts and quantiles over sliding windows. In: *Proceedings of the 23th Symposium on Principles of Database Systems*. ACM Press 286–296.

Arasu, A. and J. Widom. 2004. Resource sharing in continuous sliding-window aggregates. In: *Proceedings of the 30th International Conference on Very Large Data Bases*. Morgan Kaufmann 336–347.

Ayad, A. and J. F. Naughton. 2004. Static optimization of conjunctive queries with sliding windows over infinite streams. In: *Proceedings of the 23rd International Conference on Management of Data*. ACM Press 419–430.

Babcock, B., S. Babu, M. Datar, R. Motwani, and J. Widom. 2002. Models and issues in data stream systems. In: *Proceedings of the 21st ACM Symposium on Principles of Database Systems*. ACM Press 1–16.

Babcock, B., M. Datar, and R. Motwani. 2004. Load shedding for aggregation queries over data streams. In: *Proceedings of the 20th International Conference on Data Engineering*, IEEE Computer Society 350.

Babu, S., A. Arasu, and J. Widom. 2003. CQL: A language for continuous queries over streams and relations. In: *Proceedings of the 8th International Symposium on Database Programming Languages*. Springer 1–19.

BAI, Y., H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. 2006. A data stream language and system designed for power and extensibility. In: *Proceedings of the 15th International Conference on Information and Knowledge Management*. ACM Press 337–346.

Chandrasekaran, S., O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, M. A. Shah, and C. Q. Telegraph. 2003. continuous dataflow processing. In: *Proceedings of the 22nd International Conference on Management of Data*. ACM Press 668–668.

Chaudhuri, S. and K. Shim. 1994. Including group-by in query optimization. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. Morgan Kaufmann 354–366.

Chen, J., D. J. Dewitt, F. Tian, Y. Wang, and C. Q. Niagara. 2000. a scalable continuous query system for internet databases. In: *Proceedings of the 19th International Conference on Management of Data*. ACM Press 379–390.

Considine, J., F. Li, G. Kollios, and J. W. Byers. 2004. Approximate aggregation techniques for sensor databases. In: *Proceedings of the 20th International Conference on Data Engineering*. IEEE Computer Society 449–460.

Cranor, C., T. Johnson, O. Spataschek, and V. Shkapenyuk. 2003. Gigascope: a stream database for network applications. In: *Proceedings of the 22nd International Conference on Management of Data*. ACM Press 647–651.

Das, A., J. Gehrke, and M. Riedewald. 2003 Approximate join processing over data streams. In: *Proceedings of the 22nd International Conference on Management of Data*. ACM Press 40–51.

Ding, L. and E. A. Rundensteiner. 2004. Evaluating window joins over punctuated streams. In: *Proceedings of the 13rd International Conference on Information and Knowledge Management*. ACM Press 98–107.

Dobra, A., M. Garofalakis, J. Gehrke, and R. Rastogi. 2002. Processing complex aggregate queries over data streams. In: *Proceedings of the 21st International Conference on Management of Data*. ACM Press 61–72.

Gehrke, J., F. Korn, and D. Srivastava. 2001. On computing correlated aggregates over continual data streams. SIGMOD Record 30:13–24.

Ghanem, T. M., M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. 2007. Incremental evaluation of sliding-window queries over data streams. *IEEE Transactions on Knowledge and Data Engineering* 19:57–72.

Gilbert, A. C., Y. Kotidis, S. Muthukrishnan, and M. Strauss. 2001. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In: *Proceedings of the 27th*

International Conference on Very Large Data Bases. Morgan Kaufmann 79–88.

GOLAB, L. AND M. T. OZSU. 2003. Processing sliding window multi-joins in continuous queries over data streams. In: *Proceedings of the 29th International Conference on Very Large Data Bases*. ACM Press 500–511.

GUHA, S. AND N. KOUDAS. 2002. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In: *Proceedings of the 18th International Conference on Data Engineering*. IEEE Computer Society 567–579.

HAMMAD, M. A., W. G. AREF, AND A. K. ELMAGARMID. 2003. Stream window join: Tracking moving objects in sensor-network databases. In: *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*. 75–84.

HAMMAD, M. A., M. F. MOKBEL, M. H. ALI, W. G. AREF, A. C. CATLIN, A. K. ELMAGARMID, M. ELTABAKH, M. G. ELFEKY, T. M. GHANEM, R. GWADERA, I. F. ILYAS, M. S. MARZOUK, AND X. XIONG. 2004. Nile: A query processing engine for data streams. In: *Proceedings of the 20th International Conference on Data Engineering*. IEEE Computer Society 851–863.

JIANG, Z., C. LUO, W. C. HOU, F. YAN, AND Q. ZHU. 2006. Estimating aggregate join queries over data streams using discrete cosine transform. In: *Proceedings of the 17th International Conference on Database and Expert Systems Applications*. 182–192.

KANG, J., J. F. NAUGHTON, AND S. D. VIGLAS. 2003. Evaluating window joins over unbounded streams. In: *Proceedings of the 19th International Conference on Data Engineering*. IEEE Computer Society 341–352.

LI, J., D. MAIER, K. TUFTE, V. PAPADIMOS, AND P. A. TUCKER. 2005. Semantics and evaluation techniques for window aggregates in data streams. In: *Proceedings of the 24th International Conference on Management of Data*. ACM Press 311–322.

MANJHI, A., S. NATH, AND P. B. GIBBONS. 2005. Tributaries and deltas: efficient and robust aggregation in sensor network streams. In: *Proceedings of the 24th International Conference on Management of Data*. ACM Press 287–298.

MOTWANI, R., J. WIDOM, A. ARASU, B. BABCOCK, S. BABU, M. DATAR, G. S. MANKU, C. OLSTON, J. ROSENSTEIN, AND R. VARMA. 2003. Query processing, approximation, and resource management in a data stream management system. In: *Proceedings of the 1st International Conference on Innovative Data Systems Research*. 22–34.

SRIVASTAVA, U. AND J. WIDOM. 2004. Memory-limited execution of windowed stream joins. In: *Proceedings of the 13th International Conference on Very Large Data Bases*. Morgan Kaufmann 324–335.

SULLIVAN, M. 1996. Tribeca: A stream database manager for network traffic analysis. In: *Proceedings of 22th International Conference on Very Large Data Bases*. Morgan Kaufmann 594–606.

TATBUL, N. AND S. B. ZDONIK. 2006. Window-aware load shedding for aggregation queries over data streams. In: *Proceedings of the 15th International Conference on Very Large Data Bases*. ACM Press 799–810.

TRAN, T. M. AND B. S. LEE. 2007. Transformation of continuous aggregation join queries over data streams. In: *Proceedings of the 10th International Symposium on Advances in Spatial and Temporal Databases*. 330–347.

URHAN, T. AND M. J. FRANKLIN. 2000. Xjoin: A reactively-scheduled pipelined join operator. In: *IEEE Data Engineering Bulletin*. 27–33.

VIGLAS, S., J. F. NAUGHTON, AND J. BURGER. 2003. Maximizing the output rate of multi-way join queries over streaming information sources. In: *Proceedings of the 29th International Conference on Very Large Data Bases*. ACM Press 285–296.

VITTER, J. S. AND M. WANG. 1999. Approximate computation of multidimensional aggregates of sparse data using wavelets. In: *Proceedings of the 18th International Conference on Management of Data*. ACM Press 193–204.

YAN, W. P. AND P. Å. LARSON. 1994. Performing group-by before join. In: *Proceedings of the 10th International Conference on Data Engineering*. IEEE Computer Society 89:100.

YAN, W. P. AND P. Å. LARSON. 1995. Eager aggregation and lazy aggregation. In: *Proceedings of the 21st International Conference on Very Large Data Bases.* Morgan Kaufmann 345–357.

ZHANG, R., N. KOUDAS, B. C. OOI, AND D. SRIVASTAVA. 2005. Multiple aggregations over data streams. In: *Proceedings of the 24th International Conference on Management of Data.* ACM Press 299–310.

Tri Minh Tran is a Ph.D. candidate in Computer Science at the University of Vermont. He received the B.Eng. degree in Information Technology from Hanoi University of Technology in 2001 and the MS degree in Mathematics from Ohio University in 2003. He is currently working as a Software Engineer in the Query Optimizer group at Teradata Corporation. His main research interests include database systems, data management and query processing.

Byung Suk Lee is Associate Professor of Computer Science at the University of Vermont. His main research interests are database systems, data management, and query processing. He held several positions in industry and academia: previously at Gold Star Electric, Bell Communications Research, Datacom Global Communications, and University of St. Thomas, and currently at the University of Vermont. He was also a visiting professor at Dartmouth College and a participating guest at Lawrence Livermore National Laboratory. He served on international conferences as a program committee member, a publicity chair, a special session organizer, and a workshop organizer, and also on the review panels of US federal funding agencies. He holds a B.S. degree from Seoul National University, M.S. from KAIST, and Ph.D. from Stanford University.