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

Abstract. We address continuously processing an *aggregation join* query over data streams. Queries of this type involve both join and aggregation operations, with windows specified on join input streams. To our knowledge, the existing researches address join query optimization and aggregation query optimization as *separate* problems. Our observation, however, is that by putting them within the *same* scope of query optimization we can generate more efficient query execution plans. This is through more versatile *query transformations*, the key idea of which is to perform aggregation before join so join execution time may be reduced. This idea itself is not new (already proposed in the database area), but developing the query transformation rules faces a completely new set of challenges. In this paper, we first propose a query processing model of an aggregation join query with two key stream operators: (1) aggregation set update, which produces an aggregation set of tuples (one tuple per group) and updates it incrementally as new tuples arrive, and (2) aggregation set join, i.e., join between a stream and an aggregation set of tuples. Then, we introduce the concrete query transformation rules specialized to work with streams. The rules are far more compact and yet more general than the rules proposed in the database area. Then, we present a query processing algorithm generic to all alternative query execution plans that can be generated through the transformations, and study the performances of alternative query execution plans through extensive experiments.

1 Introduction

In this paper, we consider the problem of processing continuous *aggregation join* queries over data streams. These queries involve both join and aggregation operations. (The aggregation may be a grouped aggregation.) Many aggregation join queries are *window-based* because joins are blocking operators (i.e., needing a finite set of tuples). A window, which restricts the number of tuples processed, is a common technique proposed in many existing researches [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

Window-based aggregation join queries (called simply "aggregation join queries" in this paper) are needed in various data stream applications. For example, in a telephone call tracking application [11], a telephony company may want to keep track of the monthly total calling time on international calls made from each telephone number (i.e., subscriber) within a specific area code [11]. As another example, in a network traffic management application [10], a network administrator may want to monitor packet data flow through links between different networks [10]. For another example, in an online

D. Papadias, D. Zhang, and G. Kollios (Eds.): SSTD 2007, LNCS 4605, pp. 330–347, 2007.

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 some statistics of auction activities. Below, let us take a look at an example query for this application.

Example 1. In an online auction application, we may have a continuous query running on two data streams Bid(ts, auctionID, bidderID, bidPrice) and Auction(ts, auctionID, sellerID, startPrice)[1] and one relation Person(personID, name, state). Users may want to know, for each auction created up to now by a seller from Vermont, the total number of bids made in the last one hour. 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 P.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 researches address them as separate problems: for example, joins in [3, 2, 1, 13, 14] and aggregations in [15, 11, 16, 17, 18]. Two other existing researches [19, 15] address the problem of efficiently processing aggregation join queries as one, but not as an optimization problem per se. Furthermore, their methods can only provide approximate answers using sketching techniques [15] and discrete cosine transform [19], 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*, and this is the focus of this paper. In the initial QEP of an aggregation join query, joins are performed first and then aggregation follows. The key idea of query transformation 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 generally reduces the join input cardinality and results in a more efficient QEP, although this may not be always 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*.

Similar query transformation mechanism has been proposed in [20] and [21]. Their mechanism, however, is for *database* aggregation join queries. Due to the streaming nature of data, stream queries are fundamentally different from database queries. First, tuples arrive continuously and hence the query output must be updated *continuously* as

[1] Based on the schema used by Babu et al. [12].

well. Second, in many cases arriving tuples must be processed on-line and this requires that the query must be processed *incrementally* as soon as tuples arrive. These differences make the transformation rules for database aggregation join queries inapplicable to stream aggregation join queries.

In order to handle this problem, we introduce two key stream operators for query processing: an *aggregation set update (AS update)* and an *aggregation set join (AS join)*. An AS update operator is used to update aggregate values incrementally as new tuples arrive on the input. This operator works the same way as the group-by operator mentioned in [9]. An 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 the first time through this paper. An AS update is preceded by a *window* join in an LAP, whereas preceded by an *AS* join 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. This side effect can be easily treated by retaining a late aggregation operator on the query output even after placing an early aggregation before joins. As a coincidental side benefit, this approach does not require any constraint between streams, unlike the database case in which either a foreign key join [20] or a functional dependency [21] is required.

In this paper we first formalize the notions of the aggregation set (AS) and the two associated operators, AS update and AS join. Then, we propose compact query transformation rules based on the approaches mentioned above, that is, supporting AS update and AS join operators and retaining a late aggregation operator. Additionally, we present a generic algorithm for executing all alternative QEPs (i.e., LAP and EAPs). Note that the algorithm works just as well for a stream-*relation* join as a stream-stream join, since a relation can be viewed as a window with no update of tuples. (We have also algebraically proven the equivalence of the generated QEPs, but we omit the details in this paper due to space limit. Interested readers are referred to [22].) Then, through experiments we study the efficiencies of alternative QEPs for varying key parameters (e.g., window size, stream rate, number of groups, join selectivity factor).

To our knowledge, this is the first work addressing query transformation on aggregation join query over data streams. Main contributions include a formal query processing model that are suitable for an aggregation join query and transformation rules that are compact and yet general enough not to assume any constraint among input streams.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 discusses some preliminary concepts. Section 4 describes the query processing model and the key operators. Section 5 proposes the query transformation rules and

presents the generic query processing algorithm. Section 6 presents the experiments and the results. Section 7 concludes the paper.

2 Related Work

We discuss related work in two areas: (1) processing join queries and aggregation queries in data streams and (2) handling early aggregations in the *database*.

Processing Join and Aggregation in Data Streams

As far as we know, all of the existing data stream query processing systems – such as Aurora [23], STREAM [24], TelegraphCQ [25], NiagaraCQ [26], Stream Mill [27], Nile [28], Tribeca [29] and GigaScope [30] – process aggregation join queries by handling join and aggregation *separately*. Specifically, in all these systems an aggregation join query is always processed by first processing the join and then passing the output of the join onto the aggregation operator. In Aurora [23] and STREAM [24], 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 not applicable between a join operator and an aggregation operator. In the other systems query optimizers do not even use query transformation at all. Thus, to our knowledge, our work is the first to allow 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 them focus on *window join* processing [1, 2, 13, 14, 3, 31, 8, 32]. In [1], Kang et al. propose sliding window two-way join algorithms and develop a unit-time cost model that estimates the execution time of the join algorithms. Golab et al. [2] extend Kang et al.'s work to multi-way window join algorithms and propose join ordering heuristics to reduce the cost. Viglas et al. [13] propose a pipelined multi-way window join called MJoin. An MJoin assigns a join order for each input stream and generates join output without maintaining intermediate results. In contrast to MJoin, an XJoin proposed in [14] 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. Some other researches [3, 31, 33] address *approximate* window join processing in the case of limited system resources. None of these window join researches considers an aggregation following a join operator.

For *window aggregation* processing, Li et al. [4] propose a generic window concept and present an efficient window aggregation technique which 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. [34] address the problem of processing multiple aggregation queries that differ only in grouping attributes. Some other researches address computing approximate answers to an aggregation query using sampling [35, 36], wavelets [18, 16], histograms [11, 17], and sketching [15, 37].

As mentioned in Introduction, two researches [19, 15] address the problem of processing the same type of query as ours. Their approaches, however, are to use approximation techniques using sketching [15] and discrete cosine transform [19]. Moreover, they do not consider window-based and grouped aggregations in their problems.

Handling Early Aggregation in the Database

The early aggregation idea stems from the idea proposed for database queries [20, 38, 21]. The key idea of performing an early aggregation is to reduce the number of tuples participating in subsequent joins. In [20] the authors present query transformation rules for three cases depending on which relation the grouping, aggregation, and join attributes belong to. The authors also introduce a new operator called *aggregate join* that performs a join between one relation and the output of an early aggregation on the other relation. Yan et al. [38, 21] consider more general transformation cases in which the grouping attributes and the aggregation attributes may belong to more than one relation. In addition, instead of introducing a new operator as in [20], they use a "query rewriting" technique which involves reordering the join and aggregation operators and inserting an additional projection operator to compute the aggregate value of the reordered operators. As already mentioned, our work is fundamentally different from their works, as we deal with unbounded continuous data streams, not bounded finite set of tuples (i.e., relations).

3 Preliminaries

In this section, we present some key concepts needed to understand the rest of the paper.

Data Streams. We consider a data stream S, of an ordered sequence of tuples. Each tuple in the stream has the schema $S(TS, X_1, X_2, \ldots, X_d)$, where TS is a timestamp attribute and $X_1, X_2, \ldots, X_d$ are non-timestamp attributes. We denote a tuple of the above schema as $s(ts, x_1, x_2, \ldots, x_d)$, where ts is the value of TS and x_i is the value of X_i for each $i = 1, 2, \ldots, d$. (We use an upper-case letter to denote an attribute and a lower-case 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. Three types of window are considered in our processing model. They are classified as in [4] depending on how the tuples in the window are updated: *sliding* window, *tumbling* window, and *landmark* window. A sliding window partitions an incoming stream into overlapping blocks, and a tumbling window does that into disjoint consecutive blocks. A landmark window accumulates all tuples that have arrived since the start of the query.

Definition 1 (Window). A *window* W_S of size 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, $W_S(t) = \{s_i \,|\, t - T \leq s_i.ts < t\}$. □

Definition 2 (Window increments and decrements). Given a window $W_S(t_1)$ at time t_1, a *window increment*, denoted as $W_S^+(t_1, t_2)$, is the set of tuples added to the window during a time interval $[t_1, t_2]$, and a *window decrement*, $W_S^-(t_1, t_2)$, is the set of tuples removed from the window during the same time interval. □

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

$$W_S(t_2) = W_S(t_1) \cup W_S^+(t_1, t_2) - W_S^-(t_1, t_2)$$

Given the above definitions of window increments and decrements, the tumbling window and the landmark window can be considered as special cases of the sliding window. Figure 1 illustrates the three window types with their corresponding increments and decrements.

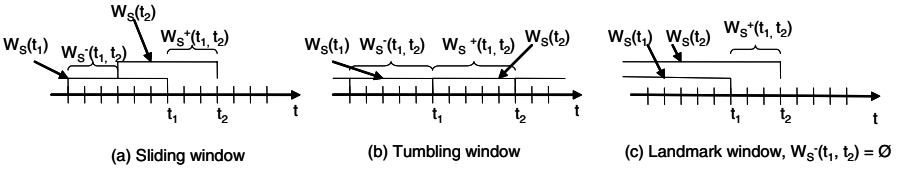


Fig. 1. Windows of different types $(t_1 < t_2)$

Window Joins. A two-way window join [1] between two streams S_1 and S_2 with windows W_{S_1} and W_{S_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_{S_1} and any expired tuples are removed from W_{S_1}. Then, W_{S_2} is probed for matching tuples of s_1 and matching tuples are 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 are the *aggregation set*, *aggregation set update (AS update)* operator and the *aggregation set join (AS join)* operator. This model provides a basis for the query transformation rules and the query processing algorithm presented in Section 5.

The concepts of aggregation set and AS update operator are the same as the concepts of window aggregate and group-by operator mentioned in [9]. 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 [20].

Aggregation Sets

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 $W_S(t)$. Additionally, consider an aggregation operator, denoted as ${}_G\mathcal{A}_{F(A)}(W_S(t))$ where $G \equiv (G_1, \ldots, G_p)$ 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 $\{(g_1, \ldots, g_p, v)\}$ where g_i is a value of G_i $(i = 1, 2, \ldots, p)$ and v is an aggregate value computed as $F(A)$ for the group $(g_1, \ldots, g_p)$ over $W_S(t)$. 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 $AS \equiv {}_G\mathcal{A}_{F(A)}(W_S(t_1))$ at time t_1, a window increment $W_S^+(t_1, t_2)$ and a window decrement $W_S^-(t_1, t_2)$ at time t_2 ($> t_1$). Then, an *AS update* operation, denoted by ${}_G\mathcal{U}_{F(A)}(AS, W_S^+(t_1, t_2), W_S^-(t_1, t_2))$, returns an updated aggregation set AS' resulting from the following updates on AS:

- For each tuple s in $W_S^+(t_1, t_2)$, 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: if F is COUNT then increase $l.F(A)$ by one; if F is SUM then increase $l.F(A)$ by $s.A$; if F is MIN and $s.A < l.F(A)$ or F is MAX and $s.A > l.F(A)$ then set $l.F(A)$ to $s.A$, otherwise no change; (if F is AVG then compute $l.F(A)$ by maintaining both COUNT and SUM). If there does not exist such a tuple l in AS, then insert a new tuple l' with $l'.G$ set to $s.G$ and $l'.F(A)$ set to 1 if F is COUNT or to $s.A$ if F is in {SUM, AVG, MIN, MAX}.
- For each tuple r in $W_S^-(t_1, t_2)$, 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: if F is COUNT then decrease $l.F(A)$ by one; if F is SUM then decrease $l.F(A)$ by $s.A$; if F is MIN or MAX and $r.A = l.F(A)$ then recompute $l.F(A)$ from the set $W_S(t_1) - \{r\}$, otherwise no change. □

As we see from the above definition, updating an aggregate value $l.F(A)$ for each tuple $r \in W_S^-(t_1, t_2)$ requires re-evaluating the whole window only if F is MIN or 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 Joins

We first present the *coalescing property* [20] of an aggregation function; this property will be used to define the aggregation set join in Definition 6.

Definition 5 (Coalescing property). Consider an aggregation function F on an attribute A. The aggregate of c tuples 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 $W_{S_1}(t_1)$ and $W_{S_2}(t_1)$, respectively, at time t_1. Additionally, consider the window increment $W^+_{S_1}(t_1, t_2)$ and decrement $W^-_{S_1}(t_1, t_2)$ 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.

$$AS_2(t_1) = \begin{cases} {}_G\mathcal{A}_{F(A)}(W_{S_2}(t_1)) & \text{if } A \text{ belongs to } S_2. \text{ (See Definition 3.)} \\ {}_G\mathcal{A}_{COUNT(A)}(W_{S_2}(t_1)) & \text{otherwise} \end{cases}$$

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 s_1 in $W^+_{S_1}(t_1, t_2)$ and for each tuple r_1 in $W^-_{S_1}(t_1, t_2)$,

1. Find matching tuples from $AS_2(t_1)$. (Denote each tuple as l.)
2. Return a sequence of tuples where for each tuple (u) the value of $F(A)$ is 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 a 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, one-way AS join is repeated from each stream (S_k, ($k \in \{1, 2, \cdots, m\}$) to the aggregation sets AS_i on the other streams S_i, $i \neq k$.

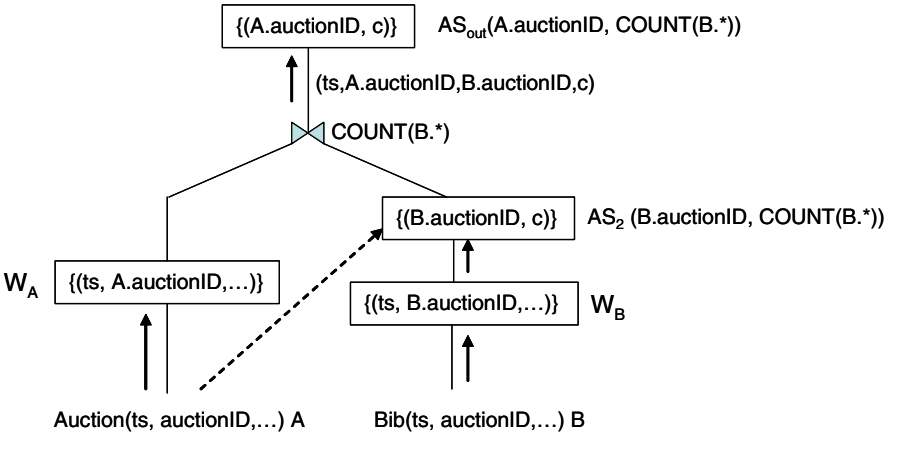


Fig. 2. An example one-way AS join

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:

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

where $AS_2 \equiv {}_{B.auctionID}\mathcal{A}_{COUNT(B.*)}(W_B(t))$. 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^-. □

5 Query Transformations

In this section, we first propose transformation rules for generating EAPs. We then present a generic algorithm for executing a query execution plan (QEP), i.e., a late aggregation plan (LAP) or an early aggregation plan (EAP).

5.1 Transformation Rules

In this section, we propose query transformation rules developed for aggregation join queries on data streams. As mentioned in the Introduction, there are two technical problems in order to make query transformation rules work on data streams. First, the aggregation sets in a QEP should be updated incrementally and continuously, both before and after the transformation. Second, the transformation should cope with the different schemas of one-way join outputs in an EAP, as the join output schema of one-way AS join in an EAP differs according to the schema of the aggregation set generated by an early aggregation operator. Since a join output is a union of multiple one-way (AS or window) join outputs but join output schema of a one-way AS join is different from that of a one-way window- or another AS join, the different schemas of one-way join outputs hinder the union.

To handle the first 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. To handle the second problem, 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.

In an EAP, early aggregation (EA) operators may be placed on any of the input streams. Once placed on a certain input stream, the operator generates an AS and, thus, allows for using an AS join to the AS instead of the window join to the input stream window. Determining the input streams to place EA operators on is based on the resulting EAPs' execution times as estimated using cost models[2]. For those EA operators inserted, their grouping attributes and aggregation functions are determined using the following *EA operator construction* rules.

[2] In this paper, we focus on query transformations only, cost models are presented in [22].

Rule 1 (Grouping attribute 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. □

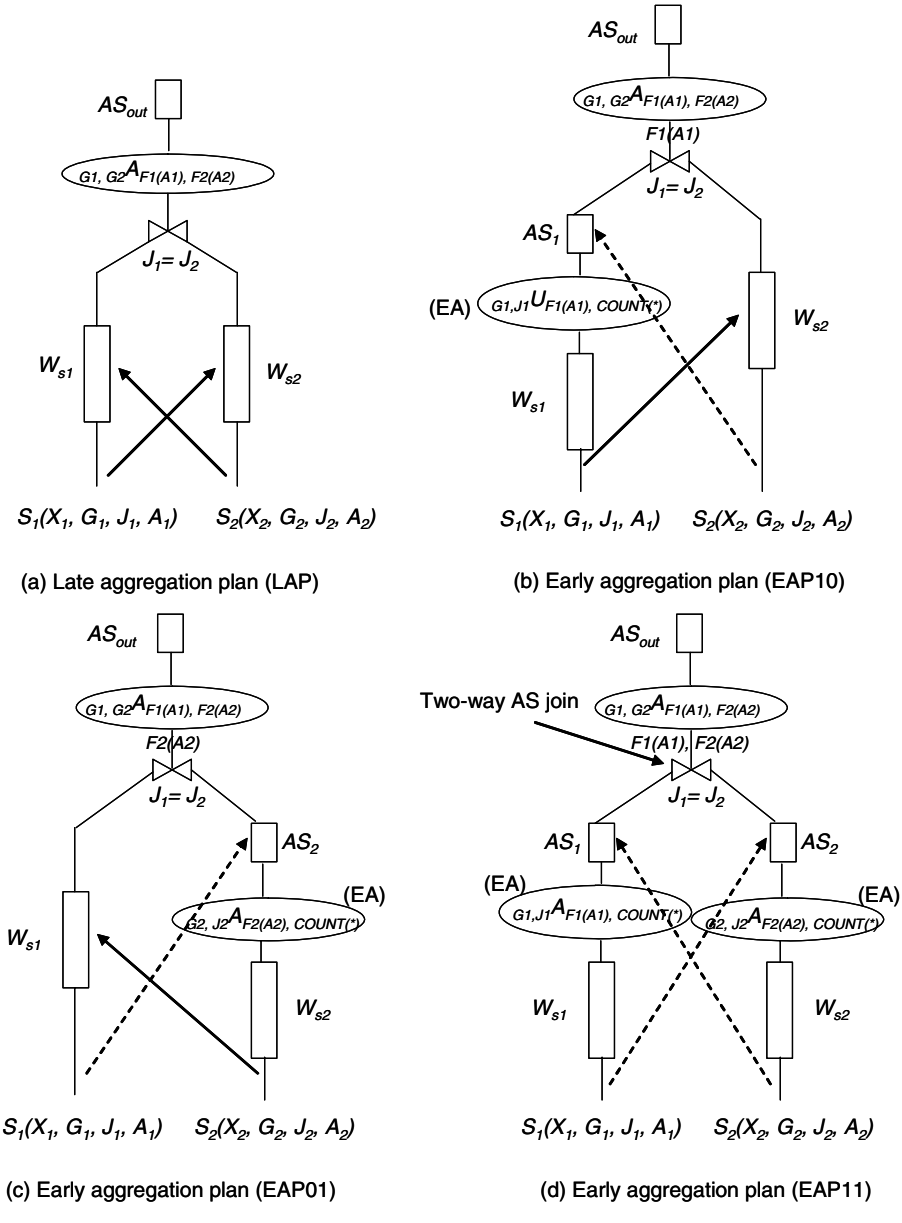
Rule 2 (Aggregation function in an EA operator)
If the EA operator is placed on a stream which has all the aggregation attributes in the query, then use the aggregation function in the query as the aggregation function of the EA operator. If the stream has only some (not all) aggregation attributes in 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. □

Note that these transformation rules are far more compact and yet more general than the transformation rules presented in the database case [20, 21]. For instance, our rules are applied to each stream without regard to other streams, whereas the database rules are applicable only if certain constraints hold among relations, such as a referential integrity [20] or a functional dependency [21]. Moreover, our rules do not depend on which streams the grouping attributes, join attributes and aggregation attributes belong to. This thus covers all the cases considered in [21].

Figure 3 illustrates transformations of an aggregation join query considering the most general case of a two-way join, i.e., both streams have grouping attributes and aggregation attributes. (This case can be reduced to special cases as considered in [20], in which only one stream has grouping (or aggregation) attributes, by setting one of the grouping (or aggregation) attributes empty.) The figure shows all four possible QEPs. The reader is asked to verify that these illustrated transformations conform to the rule for constructing an EA operator.

With the window join and AS join in place, the four QEPs in Figure 3 are equivalent. (See [22] for a proof of the equivalence.) The following example illustrates the equivalence of two QEPs shown in Figures 3a and 3c.

Example 3 (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_{S_2}. Assume that, for each tuple $s_1(x_1, g_1, j_1, a_1) \in W_{S_1}^+$, the tuple matches c tuples, $\{s_2(x_{2_i}, g_{2_i}, j_2, a_2), i = 1, 2, ..., c\}$ where $s_2.j_2 = s_1.j_1$, in W_{S_2}. Then, the window join generates c output tuples, $\{u(x_1, g_1, j_1, a_1, x_{2_i}, g_{2_i}, j_2, a_{2_i}) | i = 1, 2, ..., c, j_2 = j_1\}$. Further assume that, among these c output tuples, c_g tuples have the same value, g_2, for g_{2_i}, and hence the same value, (g_1, g_2), for (g_1, g_{2_i}). 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 $v_2 = \sum a_{2_i}, i = 1, 2, ..., c_g$. In the second QEP (EAP in Figure 3c), an AS join is performed from S_1 to AS_2. Assume that, for each tuple $s_1(x_1, g_1, j_1, a_1)$, it matches one tuple, $l_2(g_2, j_2, v_2, c_g)$ where $j_2 = j_1$ and $\sum a_{2_i}, i = 1, 2, ..., c_g$, in $AS_2(G_2, J_2, SUM(A_2), COUNT(*))$. Then, the AS join generates an output tuple $u(x_1, g_1, j_1, a_1 * c_g, g_2, j_1, v_2, c_g)$ (see Definition 5 for the coalesced value $a_1 * c_g$).



G_i: grouping attributes; J_i: join attributes; A_i: aggregation attributes; X_i: the other attributes; AS_{out}, AS_1, AS_2: aggregation sets.

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).

Fig. 3. Transformations of aggregation (two-way) join QEPs on data streams

Algorithm: QEP_Execution
Inputs:

- $W_{S_1}, W_{S_2}, \cdots, W_{S_m}$: join windows.
- $AS_{i_1}, AS_{i_2}, \cdots, AS_{i_p}$: EA output aggregation sets ($p \leq m$).
- $W^+_{S_k}$: window increment on S_k.
- $W^-_{S_k}$: window decrement on S_k.
- AS_{out}: query output aggregation set.

Output:

- AS_{out}: updated query output aggregation set.

Procedure:
Begin

For each tuple s_k in $W^+_{S_k}$ {

1. 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)
2. Add s_k to W_{S_k}. (Window update)
3. With s_k, find matching tuples in either AS_j or W_{S_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_{S_j}). (Window join or AS join)
4. For each tuple produced in step 3, find its group in AS_{out} and update the aggregate value. (AS update on query output)

}
For each tuple r_k in $W^-_{S_k}$ {

5. 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)
6. Remove r_k from W_{S_k}. (Window update)
7. With r_k, find matching tuples in either W_{S_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_{S_k}). (Window join or AS join)
8. For each tuple produced in step 7, find its group in AS_{early} and update the aggregate value. (AS update on query output)

}
End

Fig. 4. A generic QEP-execution algorithm

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. □

5.2 Generic Algorithm for Query Executions

Figure 4 outlines a high level algorithm for processing tuples with a multi-way join among m ($m \geq 2$) streams $S_1, S_2, \ldots, S_m$.[3] 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 $W^+_{S_k}$ and each tuple r_k in the window decrement $W^-_{S_k}$. The algorithm performs (1) AS updates on the output of an EA operator in steps 1 and 5 if there exists an EA operator on S_k, (2) window updates in steps 2 and 6, (3) either AS

[3] 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.

joins or window joins in steps 3 and 7 depending on whether an EA operator is placed on S_k, and (4) AS updates on the query output AS_{out} in steps 4 and 8.

6 Performance Study

In this section, we study the performance of the proposed query transformations, with a focus on the QEP efficiencies. There are two objectives of the experiments: (1) examine the performance trends of the alternative QEPs for varying key parameter values; (2) show the cases of each alternative QEP being the most efficient one in relation to the parameter values. Section 6.1 describes the experimental setup, and Section 6.2 present the experiments conducted and their results.

6.1 Experimental Setup

We have built an operational prototype that implements the QEP execution algorithm (see Figure 4). The prototype 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. For a join method, it supports hash join and nested loop join and for an aggregation method, it supports hash-based grouping. Additionally, it executes a join using *sliding* windows, of which tumbling and landmark windows are only special types (see Section 3).

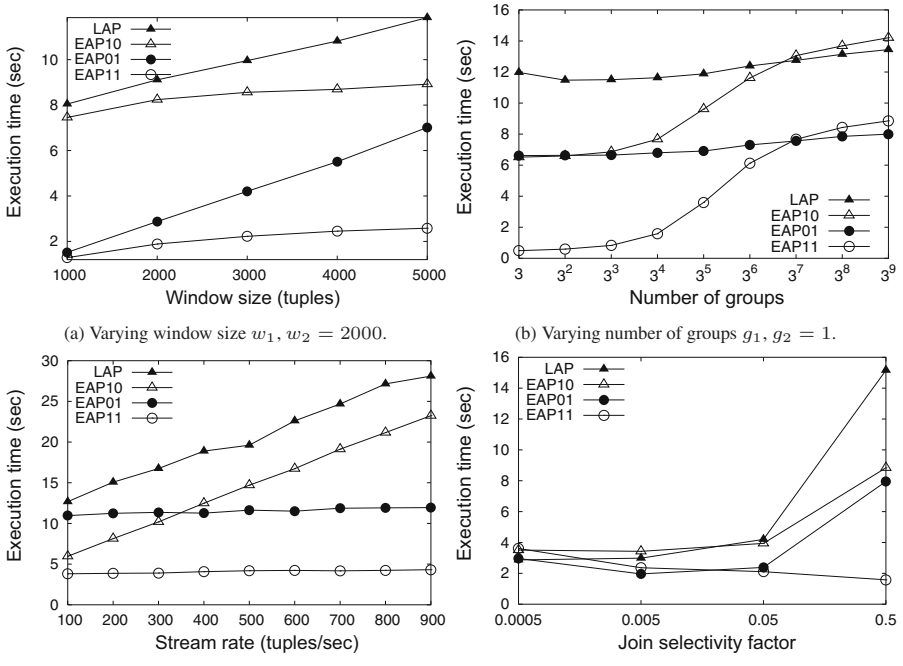
Inputs to the prototype are data streams generated using a data generator[4] (described below), the join arity (i.e., number of data streams) (m), the size of each join window (w_1, w_2), and the QEP case number (0 for LAP, $1, 2, 3, ...2^m - 1$ for EAPs). It then processes the input stream data according to the specified QEP and reports the execution time.

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. Values of each of these attributes are 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.

6.2 Experiments and Results

In this section, we first investigate the efficiencies of alternative QEPs by varying streams statistics (i.e. stream rates, join selectivities, number of groups and window sizes). Then, 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).

[4] 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.



(a) Varying window size $w_1, w_2 = 2000$.

(b) Varying number of groups $g_1, g_2 = 1$.

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

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

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

Fig. 5. Execution times of QEPs (using two-way nested loop join)

Experiment 1: Query Execution Costs for Varying Stream Statistics

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 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 5 shows the results from the four sets of experiments. The curves in each graph represents the execution times of four alternative QEPs (i.e., LAP, EAP01, EAP10, EAP11). Due to space limit, we present the results for two-way nested-loop joins only; the results from using hash joins and three-way joins show the same trends. Interested readers are referred to [22] for the results of more comprehensive experiments.

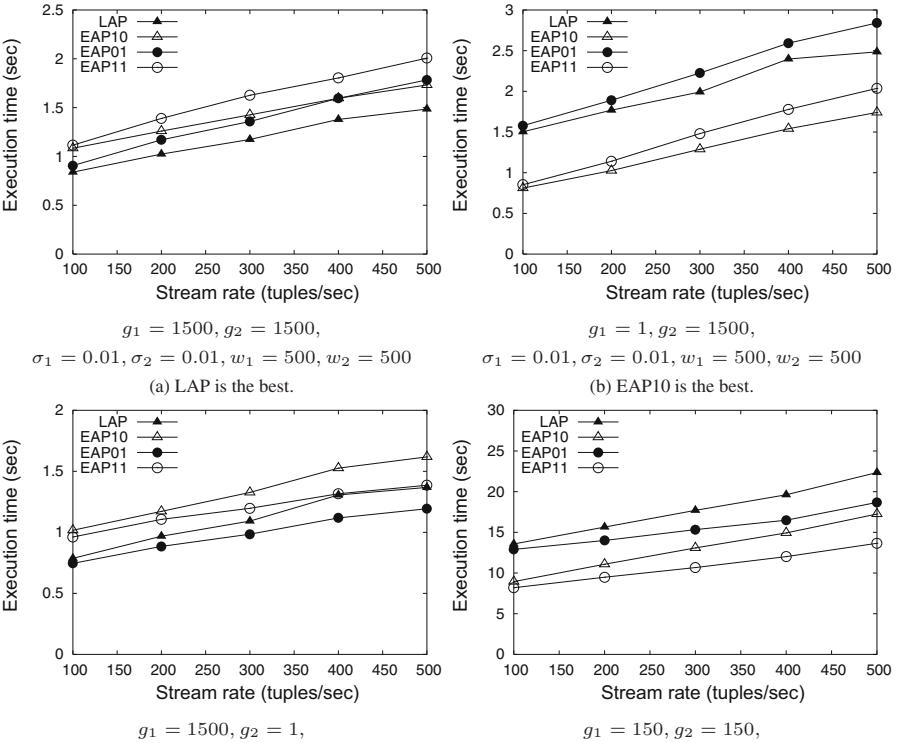
Let us now examine the results of each set of experiments for varying each of the four parameters (i.e., window size, number of groups, stream rate, join selectivity factor). In the following discussion, we use the name of a QEP (i.e., LAP, EAP01, EAP10, EAP11) to refer to the cost of executing the QEP.

In Figure 5a, as window size 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 aggregation set size (i.e., $|AS_1|$) (which is fixed) when w_1 is greater than 2000. Additionally, EAPs are always better than LAP because in the experiment, aggregation set sizes $|AS_1|$ and $|AS_2|$ are set smaller than window size w_1 and w_2.

Figure 5b shows the results of varying the number of groups on stream S_1 by a factor of 3. In this experiment, there is no grouping attribute in stream S_2 and, thus, g_2 equals 1. In the figures, as the number of groups 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 ($|AS_1|$). But, as g_1 becomes large enough ($g_1 = 3^8$), $|AS_1|$ stops depending on g_1 and starts depending on $|W_1|$. As a result, EAP10 and EAP11 lose the advantage of placing an EA operator on S_1.

Figure 5c shows the results of varying stream rate of S_1 while fixing the stream rate of S_2. In the figures, 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



$g_1 = 1500, g_2 = 1500,$
$\sigma_1 = 0.01, \sigma_2 = 0.01, w_1 = 500, w_2 = 500$
(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: $100 \leq \lambda_1 \leq 500, \lambda_2 = 500$

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

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 shorter to find matching tuples in an aggregation set AS_2 instead of W_2.

The results in Figure 5d is for the case of varying join selectivity factors σ_1 and σ_2 ($\sigma_1 = \sigma_2$). As the join selectivity factors increase, the costs of all QEPs increase except for EAP11. The reason is that the cost of window joins in LAP, EAP01 and EAP10 depends on the join selectivity factors but this is not the case for the aggregation set join in EAP11. Moreover, as the join selectivity factors increase, $|AS_1|$ and $|AS_2|$ decrease, thus the cost of EAP11 decreases.

Experiment 2: 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 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.

7 Conclusion

In this paper, we focused on the problem of continuously processing an 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 query transformation rules to generate an early aggregation plan equivalent to a late aggregation plan. We then developed an algorithm for executing the query execution plans. Finally, we conducted a set of experiments to study the performances of alternative QEPs.

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. Our query transformation is compact and yet generic to be applicable to each stream separately. 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.

Query transformation is one step in query optimization. Thus, the future work includes to develop a comprehensive framework that integrates other components such as cost models for alternative QEPs and efficient search algorithms for finding an optimal QEP. The optimizer can run adaptively to switch to a more efficient QEP when input statistics (e.g., stream rates, number of groups, join selectivity) change significantly.

References

1. Kang, J., Naughton, J.F., Viglas, S.D.: Evaluating window joins over unbounded streams. In: Proceedings of ICDE, Bangalore, India, pp. 341–352. IEEE Computer Society Press, Los Alamitos (2003)
2. Golab, L., Ozsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: Proceedings of VLDB, pp. 500–511. ACM Press, New York (2003)
3. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: Proceedings of ACM SIGMOD, San Diego, California, pp. 40–51. ACM Press, New York (2003)
4. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: Semantics and evaluation techniques for window aggregates in data streams. In: Proceedings of SIGMOD, pp. 311–322. ACM Press, New York (2005)
5. Ayad, A., Naughton, J.F.: Static optimization of conjunctive queries with sliding windows over infinite streams. In: Proceedings of ACM SIGMOD, pp. 419–430. ACM Press, New York (2004)
6. Arasu, A., Widom, J.: Resource sharing in continuous sliding-window aggregates. In: Proceedings of VLDB, pp. 336–347. Morgan Kaufmann, San Francisco (2004)
7. Arasu, A., Manku, G.S.: Approximate counts and quantiles over sliding windows. In: Proceedings of PODS, pp. 286–296. ACM Press, New York (2004)
8. Ding, L., Rundensteiner, E.A.: Evaluating window joins over punctuated streams. In: Proceedings of CIKM, pp. 98–107. ACM Press, New York (2004)
9. Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G., Elmagarmid, A.K.: Incremental evaluation of sliding-window queries over data streams. IEEE TKDE 19(1), 57–72 (2007)
10. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of ACM SIGMOD, Madison, Wisconsin, pp. 1–16. ACM Press, New York (2002), doi:10.1145/543613.543615
11. Gehrke, J., Korn, F., Srivastava, D.: On computing correlated aggregates over continual data streams. SIGMOD Record 30(2), 13–24 (2001), doi:10.1145/376284.375665
12. Babu, S., Arasu, A., Widom, J.: CQL: A language for continuous queries over streams and relations. In: Lausen, G., Suciu, D. (eds.) DBPL 2003. LNCS, vol. 2921, pp. 1–19. Springer, Heidelberg (2004)
13. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of VLDB, pp. 285–296 (2003)
14. Urhan, T., Franklin, M.J.: Xjoin: A reactively-scheduled pipelined join operator. In: IEEE Data Enginerring Bullentin, pp. 27–33. IEEE Computer Society Press, Los Alamitos (2000)
15. Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R.: Processing complex aggregate queries over data streams. In: Proceedings of ACM SIGMOD, Madison, Wisconsin, pp. 61–72. ACM Press, New York (2002)
16. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In: Proceedings of VLDB, pp. 79–88. Morgan Kaufmann, San Francisco (2001)
17. Guha, S., Koudas, N.: Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In: Proceedings of ICDE, pp. 567–579 (2002)
18. Vitter, J.S., Wang, M.: Approximate computation of multidimensional aggregates of sparse data using wavelets. In: Proceedings of ACM SIGMOD, pp. 193–204. ACM Press, New York (1999)
19. Jiang, Z., Luo, C., Hou, W.-C., Yan, F., Zhu, Q.: Estimating aggregate join queries over data streams using discrete cosine transform. In: Bressan, S., Küng, J., Wagner, R. (eds.) DEXA 2006. LNCS, vol. 4080, pp. 182–192. Springer, Heidelberg (2006)

20. Chaudhuri, S., Shim, K.: Including group-by in query optimization. In: Proceedings of VLDB, pp. 354–366. Morgan Kaufmann, San Francisco (1994)
21. Yan, W.P., Larson, P.-Å.: Eager aggregation and lazy aggregation. In: Proceedings of VLDB, pp. 345–357. Morgan Kaufmann, San Francisco (1995)
22. Tran, T.M., Lee, B.S.: Transformation of continuous aggregation join queries over data streams. Technical Report CS-07-02, Department of Computer Science, University of Vermont (2007)
23. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal 12(2), 120–139 (2003)
24. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query processing, approximation, and resource management in a data stream management system. In: Proceedings of CIDR, pp. 22–34 (2003)
25. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: TelegraphCQ: continuous dataflow processing. In: Proceedings of ACM SIGMOD, San Diego, California, pp. 668–668. ACM Press, New York (2003)
26. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for internet databases. In: Proceedings of ACM SIGMOD, Dallas, Texas, United States, pp. 379–390. ACM Press, New York (2000)
27. Bai, Y., Thakkar, H., Wang, H., Luo, C., Zaniolo, C.: A data stream language and system designed for power and extensibility. In: Proceedings of CIKM, pp. 337–346 (2006)
28. Hammad, M.A., Mokbel, M.F., Ali, M.H., Aref, W.G., Catlin, A.C., Elmagarmid, A.K., Eltabakh, M., Elfeky, M.G., Ghanem, T.M., Gwadera, R., Ilyas, I.F., Marzouk, M.S., Xiong, X.: Nile: A query processing engine for data streams. In: Proceedings of ICDE, pp. 851–863. IEEE Computer Society Press, Los Alamitos (2004)
29. Sullivan, M.: Tribeca: A stream database manager for network traffic analysis. In: Proceedings of VLDB, pp. 594–606. Morgan Kaufmann, San Francisco (1996)
30. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of ACM SIGMOD, San Diego, California, pp. 647–651. ACM Press, New York (2003)
31. Srivastava, U., Widom, J.: Memory-limited execution of windowed stream joins. In: Proceedings of VLDB, pp. 324–335. Morgan Kaufmann, San Francisco (2004)
32. Hammad, M.A., Aref, W.G., Elmagarmid, A.K.: Stream window join: Tracking moving objects in sensor-network databases. In: Proceedings of SSDBM, pp. 75–84 (2003)
33. Ojewole, A., Zhu, Q., Hou, W.-C.: Window join approximation over data streams with importance semantics. In: Proceedings of CIKM, pp. 112–121 (2006)
34. Zhang, R., Koudas, N., Ooi, B.C., Srivastava, D.: Multiple aggregations over data streams. In: Proceedings of ACM SIGMOD, pp. 299–310. ACM Press, New York (2005)
35. Tatbul, N., Zdonik, S.B.: Window-aware load shedding for aggregation queries over data streams. In: Proceedings of VLDB, pp. 799–810 (2006)
36. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In: Proceedings of ICDE, p. 350. IEEE Computer Society Press, Los Alamitos (2004)
37. Considine, J., Li, F., Kollios, G., Byers, J.W.: Approximate aggregation techniques for sensor databases. In: Proceedings of ICDE, pp. 449–460. IEEE Computer Society Press, Los Alamitos (2004)
38. Yan, W.P., Larson, P.-Å.: Performing group-by before join. In: Proceedings of ICDE, pp. 89–100. IEEE Computer Society Press, Los Alamitos (1994)