Optimization of Data Stream Processing

Janusz R. Getta
School of Information Technology and Computer Science
University of Wollongong
Wollongong, NSW, Australia
jrg@uow.edu.au

Ehsan Vossough
Department of Computing and Information Technology
University of Western Sydney, Campbelltown
NSW, Australia
e.vossough@uws.edu.au

ABSTRACT
Efficient processing of unlimited and continuously expanding sequences of data items is one of the key factors in the implementations of Data Stream Management Systems (DSMS). Analysis of stream processing at the dataflow level reveals execution plans which are not visible at a logical level. This work introduces a new model of data stream processing and discusses a number of optimization techniques applicable to this model and its implementation. The optimization techniques include applications of containers with intermediate results, analysis of data processing rates, and efficient synchronization of elementary operations on data streams. The paper also describes the translation of logical level expressions on data streams into the sets of dataflow level expressions, syntax based optimization of dataflow expression, and scheduling of concurrent computations of the dataflow expressions.

Keywords
data stream, dataflow, scheduling, optimization

1. INTRODUCTION
In the last few years Data Stream Management Systems (DSMS) emerged as a new important research area [7, 1, 2]. A data stream is a theoretically unlimited sequence of data items that may originate from the sensor devices [10], financial organizations, network monitoring systems, satellite systems, etc. DSMS is a specialized software capable of storing and processing data streams. Experiments show that data processing techniques developed for the conventional database management systems cannot be directly reused for implementation of DSMS [15]. Data stream processing techniques should be *reactive*, *continuous*, *adaptable*, and *efficient*. Reactivity means that processing of a data item starts as soon as an item is appended to a stream. Continuity requires the periodic recomputations of applications in order to keep up with the changing contents of the streams. Adaptability allows for the dynamic modifications of data processing plans in a response to an external event such as, rapidly changing frequency of a stream, or the congestion of an internal queue, etc. Efficiency means that data stream processing rates should be higher than data transmission rates. This work proposes a new data stream processing model that satisfies the principles listed above and allows for a more comprehensive optimization of data stream processing. We show that computations on data streams can be implemented as the flows of data items between the elementary operations and that logical level expressions on data streams can be translated into networks of elementary operations. The optimization techniques include the optimal organizations of data flows among the operations and an efficient implementation and scheduling of the operations.

The origins of data stream processing techniques can be traced to the pipelined [17],adaptive [9], continuous [13], online, and interactive query processing [12, 8]. A comprehensive review of many works that contributed to the present state of data stream processing can be found in [5].

Majority of the data stream processing systems use pipes or queues to build the networks of elementary operations on data streams. Typical examples include Fjords [11] and Aurora [6, 1], and STREAM project [2, 4]. Synchronization of the operations in these systems is usually controlled by a central scheduler that is responsible for the order of execution, amount of time available for each operation, and memory management. CACQ system [11] treats data streams as infinite relational tables. The system uses the "eddy" operator [3] to dynamically process the items appended to data streams. Adaptive query optimization in DSMS through tuple routing between the distributed "eddies" has been recently proposed in [14]. Implementation of "distributed eddy" as a network of simple operators seems to be conceptually close to dataflow expressions proposed in this work. Optimization of stream processing described in [15] applies a model of computations where the relational algebra operations are linked via queues and data items flow from one operation to another.

The systems listed above define the computations on data streams as the expressions built over the symbols of logical operations and names of data streams. We call such level of computations as a *logical level* of data stream processing. The optimizations at the logical level do not reveal all execution plans that are possible when a new data element is appended to one of the input streams. More execution plans are visible at a level where each operation acts on a single data item from a stream with static collection of windows on the remaining streams. We call such level of computations as a *dataflow level* of stream processing. For example a rate-based optimization [15] of an expression $(r(ab) \bowtie_b s(bc)) \bowtie_{ac} t(ac)$ over the streams r, s, and t reveals 3 execution plans. The first plan is given above and two others can be obtained from the associativity of join operation. At the data flow level, a data item d_r appended to d can be processed in two different ways. In one way, d_r

is joined with a window on s and the results are joined with a window on t. In the other, the joins are performed in the opposite order. Hence, for 3 input data streams we get 6 execution plans. Analysis of the execution plans at the logical level does not reveal the plans visible at the data flow level. For example, it is impossible to have an execution plan where an item d_r is joined with a window on s, and item d_s is joined with a window on t. It suggests that comprehensive analysis of execution plans should be performed at the dataflow level.

The rest of this paper is organized as follows. Section 2 defines a data stream processing model, the systems of operations at a logical and dataflow level, and scheduling of concurrently running dataflow operations. The optimization of data stream processing at a dataflow level is presented in Section 3.1. The optimal scheduling of dataflow expressions is discussed in Section 3.2. Section 4 concludes the paper.

2. DATA STREAM PROCESSING MODEL

This section introduces a simple model of data streams and defines the systems of operations on data streams at the logical and dataflow levels.

2.1 Data streams

A *data stream* is a theoretically unlimited and continuously expanding sequence of homogeneous data items. A *window* over a data stream is a time varying collection of subsequences in the stream. A new data item appended to a stream triggers a relocation of the window accordingly to a predefined strategy. Each data item obtains a unique timestamp when it is recorded in a window on an input stream. A data item has a *status* indicator whose value is either *positive* or *negative*. The items appended to the streams have a *positive* status. The negative data items originate from the relocations of windows and specific operations that remove data items from the windows, e.g. set difference operation. All other operations generate the positive data items. The status indicator allows for a uniform treatment of the insertions and deletions from the windows.

2.2 Elementary operations

The system of logical level operations on data streams considered in this work is very similar to a slightly generalized and extended relational algebra. It includes `join`($\bowtie$), `generalized difference`($-$), `union`($\cup$), `aggregation` (γ), `split`($\prec$), `selection`(σ), and `transformation`(ϱ) operations. The semantics of `join` and `selection` are the same as the semantics of the same relational operations. An operation of `generalized difference` $r -_x s$ removes from a window w_r data items which have the same attribute values in x as at least one data item in a window w_s. An `union` merges two windows on data streams. An `aggregation` partitions an input stream according to the values of selected attributes and applies aggregation functions to each partition. A `split` operation $r \prec_{\phi_1,\phi_2} (s,t)$ sends all data items from a stream r that evaluate a formula ϕ_1 to *true* to a stream s and all data items that evaluate a formula ϕ_2 to *true* to a stream t. A `transformation` $\varrho_T(r)$ applies a function T that transforms the data items in a stream r, e.g. projects the data items on a given set of attributes. The logical level expressions are constructed in the same way as the expressions of relational algebra. The evaluation of a logical level expression

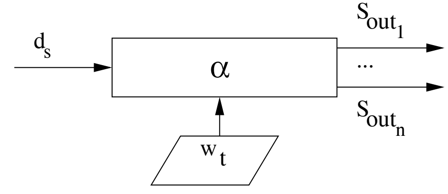

Figure 1: The inputs and outputs of dataflow operation α

at time t_0 is equal to the evaluation of the expression over the contents of windows on the input data streams at t_0.

The system of logical operations on data streams has the respective system of elementary operations at the dataflow level, later on referred to as *dataflow operations*. A dataflow operation α takes on input a data item d_s from a stream s and all data items with the timestamps lower than timestamp of d_s and included in a window w_t over a data stream t. A dataflow operation sends its results to a number of output streams $s_{out_1}, \ldots, s_{out_n}$, see Figure 1. As a simple example consider a pseudo-code specification of generalized difference $-_{left}$. The operation computes $\{d_r\} -_X w_s$.

`if` $\forall s \in w_s\ d_r[X] \neq s[X]$ `then`
  `if` $positive(d_r)$ `then` $d_r^+ \rightarrow s_{out}$ `else` $d_r^- \rightarrow s_{out}$ `endif`
`endif`

A term $d_r^+ \rightarrow w_{out}$ ($d_r^- \rightarrow w_{out}$) represents an operation that appends d_r to a window on the output stream w_{out} as a positive (negative) data item.

2.3 Expressions

We considers a class of logical level expressions where each operation has no more than two input data streams and produces only one output stream. Let $\alpha(w)$ denote a dataflow operation α that processes a data item against the static contents of window w. A dataflow expression consists of a stream identifier δ_s followed by a sequence of dataflow operations $\alpha_i(w_k)\ldots\alpha_j(w_m)$. We assume that the first dataflow operation in the sequence processes the input data items of stream s and that each next dataflow operation processes the outputs produces by the previous operation in the sequence. A *recorder* operation $\rightarrow(w)$ writes an input data item d to a window w, relocates the position of the window when necessary, and outputs the removed data items from the window as negative data items followed by the input data item d. For example, a dataflow expression δ_r: $\rightarrow(w_r)$, $\bowtie(w_s)$, $\bowtie(w_t)$, $\rightarrow(w_{out})$ records a data item appended to a stream r in a window w_r, and then joins the results from the latter operation with the contents of window w_s, and then joins the results with the contents of window w_t, and records the final results in a window w_{out}.

A logical level expression $E(s_1, \ldots, s_k)$ over the streams $s_1, \ldots, s_k$ is implemented at the dataflow level as a set of dataflow expressions $\{\delta_{s_1}, \ldots, \delta_{s_n}\}$. For example, a set of dataflow expressions $\{\delta_r$: $\rightarrow(w_r)$, $-_{left}(w_s)$, $\rightarrow(w_{out})$, δ_s: $\rightarrow(w_s)$, $-_{right}(w_r)$, $\rightarrow(w_{out})\}$ (see Figure 2) implements a logical operation of generalized difference $r -_X s$. A pseudo-code specification of an operation $-_{right}$ on data item d_s

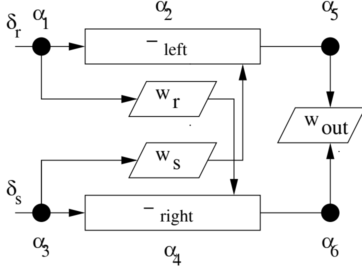

Figure 2: A dataflow level implementation of difference operation

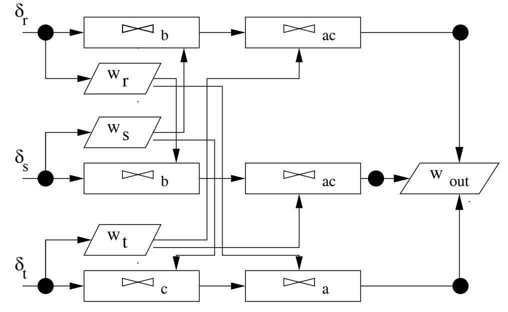

Figure 3: A dataflow implementation of $(r(ab) \bowtie_b s(bc)) \bowtie_{ac} t(ac)$

and window w_r is as follows.

if $\exists r \in w_r\ r[X] = d_s[X]$ **then**
  if $positive(d_r)$ **then** $d_r^- \rightarrow s_{out}$ **else** $d_r^+ \rightarrow s_{out}$ **endif**
endif

A logical level expression $E(s_1, \ldots, s_k)$ is translated into a set of dataflow expressions in the following way. Let T_E be a syntax tree of the expression. For each leaf node s_i, $i = 1, \ldots, n$, perform the following steps.
(1) Create an empty dataflow expression δ_{s_i}.
(2) Let $A_1, \ldots, A_n$ be a sequence of logical operations along a path from a leaf node s_i to the root node of T_E. Replace $A_1(s_i, s_j)$ with a dataflow operation $\alpha_1(w_j)$ where α_1 implements $A_1(\{d_i\}, w_j)$ and w_j is a window over a stream s_j. For all $k = 2, \ldots, n$ replace A_k with $\alpha_k(w_{T'})$ where α_k implements $(A_k(A_{k-1}(\ldots), T')$ where T' is a root of subexpression being the second argument of A_k.
(3) Append the path obtained in step(2) to δ_{s_i} and insert into the path the recorder operations $\rightarrow(w_{T'})$ whenever $w_{T'}$ is used in any dataflow expression.
(4) Append a recorder operation $\rightarrow(w_{out})$ at the end of each dataflow expression.
For example, the translation of a logical level expression $(r(ab) \bowtie_b s(bc)) \bowtie_{ac} t(ac)$ produces a set of dataflow expressions:
$\{\delta_r$: $\rightarrow(w_r)$, $\bowtie_b(w_s)$, $\bowtie_{ac}(w_t)$, $\rightarrow(w_{out})$,
δ_s: $\rightarrow(w_s)$, $\bowtie_b(w_r)$, $\bowtie_{ac}(w_t)$, $\rightarrow(w_{out})$,
δ_t: $\rightarrow(w_t)$, $\bowtie_b(w_s)$, $\bowtie_{ac}(w_r)$, $\rightarrow(w_{out})\}$.
A graphical representation of the expressions is given in Figure 3. An expression δ_t comes from a transformation of the logical level expression into $r(ab) \bowtie_b (s(bc)) \bowtie_{ac} t(ac))$

Correctness of the transformation above depends on the properties of operations involved. Consider a data element d_i appended to a stream s_i. The transformation produces an execution path δ_i that represents the computations of $\Delta E(w_{s_1}, \ldots, \{d_i\}, \ldots, w_{s_k})$ where $w_{s_1}, \ldots, w_{s_k}$ are the windows over the input streams. The transformation is correct if it is possible to use ΔE to update the result of $E(w_{s_1}, \ldots, w_{s_i}, \ldots, w_{s_k})$ to the result of $E(w_{s_1}, \ldots, w_{s_i} \cup \{d_i\}, \ldots, w_{s_k})$. The above is true if for each operation $\alpha(w_r, w_s))$ used in E it is possible to compute $\alpha(w_r \cup \{d\}, w_s)$ using the values of $\alpha(w_r, w_s))$ and $\Delta\alpha(\{d\}, w_s)$ where d denote an element appended or removed from a window w_r. Such incremental computations are always possible for the system of operations defined in 2.2. For example, $(w_r \cup \{d\}) \bowtie w_s = (w_r \bowtie w_r) \cup (\{d\} \bowtie w_s)$.

2.4 Scheduling

Order preservation of stream processing means that serial processing of a sequence of data items $<d_{r_1}, d_{r_2}, \ldots, d_{r_n}>$ should output the results in an order consistent with the input sequence, i.e. $<out(d_{r_1}), out(d_{r_2}), \ldots, out(d_{r_n})>$. If an input sequence is processed concurrently then the scheduling of dataflows is conceptually similar to the scheduling of database transactions. A hypothetical transaction starts when a data item is written to a window and ends when the final results are recorded in all output windows or when a dataflow operation returns no values. To enforce the correctness and order preservation the executions of transactions representing the dataflows must be *order-preserving serializable* [16]. However, scheduling of data flow expressions in the same way as scheduling of database transactions is not feasible because of the following reasons. All operations of dataflow expressions are known in advance and their execution is always sequential and deterministic. It is possible to anticipate all conflicts between the dataflow operations. The computations of dataflow expressions cannot be aborted and restarted. Order preservation and serializability do not need to be strictly enforced.

A scheduler described below should be considered as a formal model for concurrent processing of data streams and not as a hypothetical implementation. We say that dataflow operations α_i and α_j conflict if both of them operate on the same window and least one of them modifies the window. The conflicts between the operations of dataflow expressions $\delta_1, \ldots, \delta_n$ are recorded in a binary matrix $\mathcal{C}_{m \times m}$. All elements $c_{ii} \in \mathcal{C}$, $i = 1, \ldots, m$ are set to 1. The elements c_{ij}, $i \neq j$ are set to 1 if α_i conflicts with α_j. All other elements c_{ij} are set to 0. A state of running computations is represented by a scheduling graph $G = <N, E>$ where N is a set of nodes labelled with the names of elementary operations and E is a set of edges that consists of solid edges (E_s) and dashed edges (E_d). A scheduling graph is dynamically created and maintained during the computations of dataflow expressions. Whenever a new data element triggers the execution of dataflow expression δ_i: $\alpha_{i_1}, \ldots, \alpha_{i_k}$, a solid linked list of the nodes $\alpha_{i_1}, \ldots, \alpha_{i_k}$ is appended to the graph. Then, $\mathcal{C}$ is used to find what new dashed edges should be added the graph. An operation α_{i_j} is allowed to start its computations when the computations of $\alpha_{i_{j-1}}$ are completed and there is no dashed edge $<\alpha, \alpha_{i_j}>$ in the scheduling graph. When the computations of α_{i_j} are completed then a node α_{i_j} and all dashed and solid edges $<\alpha_{i_j}, \alpha>$

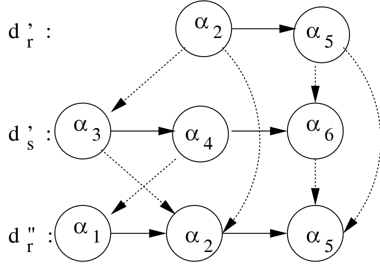

Figure 4: A sample instance of scheduling graph.

are removed from the scheduling graph. Figure 4 shows an instance of a scheduling graph G for the concurrent computations of dataflow expressions given in Figure 2 after the data items d'_r,d'_s,d''_r have been appended to the streams r, s and after the computations triggered by d'_r have reached operation α_2. The formal model of scheduling described above allows for the evaluation of different implementation methods. The quality of implementation depends on a number of dashed edges in a scheduling graph. In the perfect case, no conflicts between dataflow operations means no dashed arrows in the graph and no operations are blocked.

3. OPTIMIZATION TECHNIQUES

Optimization of data streams processing is performed at the logical, dataflow, and scheduling levels. Optimization at the logical level is very similar to the traditional syntax based optimization of relational algebra expressions that pushes the most selective operations down a syntax tree and merges unary operations with binary operations. The rate based optimization [15] extends this technique with the transformations of logical level expressions that maximize the stream processing rates. This section concentrates on the optimizations performed at the dataflow and scheduling levels. The optimization techniques include reduction of conflicts over the accesses to the intermediate results, transformations of dataflow expressions to maximize the processing rates, and efficient implementation of scheduling.

3.1 Optimization of dataflow expressions

The objective of optimization is to find a set of dataflow expressions that maximizes the processing rates. One of the optimization questions addresses the possibilities and benefits from elimination of containers with the intermediate results. As a simple example consider the following dataflow implementation of a logical level expression $(r(ab) \bowtie_b s(bc)) \bowtie_{ac} t(ac)$:
$\{\delta_r$: $\rightarrow(w_r)$, $\bowtie_b(w_s)$, $\rightarrow(w_{rs})$, $\bowtie_{ac}(w_t)$, $\rightarrow(w_{out})$,
δ_s: $\rightarrow(w_s)$, $\bowtie_b(w_r)$, $\rightarrow(w_{rs})$, $\bowtie_{ac}(w_t)$, $\rightarrow(w_{out})$,
δ_t: $\rightarrow(w_t)$, $\bowtie_{ac}(w_{rs})$, $\rightarrow(w_{out})\}$
The expressions δ_r and δ_s perform two extra write operations to a window w_{rs} when compared with the earlier translation given in Figure 3. As a consequence a subexpression $\bowtie_b(w_s)$, $\bowtie_{ac}(w_r)$ is replaced in δ_t with one operation $\bowtie_{ac}(w_{rs})$. The analysis of stream frequencies is a key to the selection of the best translation. To simplify the problem, we assume that all elementary operations are computed in a block mode, i.e. such that an operation is initiated only when its predecessor in the same dataflow expression completes the computations. The average time T_t spent on processing of item d_t appended to a stream t includes the time spent on processing the dataflow operations and the time spent on waiting for access to w_{rs}, i.e. $T_t = \tau_{\rightarrow w_t} + \tau_{\bowtie w_{rs}} + \tau_{wait}$. A data item d_t must wait when it arrives at the operation $\bowtie w_{rs}$ before the earlier processed data items from the streams r and s are written to w_{rs}. Alternatively, if we do not use a window w_{rs} (see Figure 3) then the average time spent on the processing of d_t is equal to $T'_t = \tau_{\rightarrow w_t} + \tau_{\bowtie w_s} + f_s * \tau_{\bowtie w_r}$ where f_s is an average number of data items produced by $\bowtie w_s$ from one input item. We benefit from the existence of w_{rs} if $T_t < T'_t$, i.e.

$$\tau_{wait} < (\tau_{\bowtie w_s} + f * \tau_{\bowtie w_r}) - \tau_{\bowtie w_{rs}} \quad (1)$$

This means that faster processing of $\bowtie w_{rs}$ compensates for the blocking time τ_{wait} because operation $\bowtie w_{rs}$ does not need to recompute the join of w_r and w_s. The only component of (1) that depends on the frequencies of input streams is τ_{wait}. Assume that the processing of d_t is blocked by the processing of item d_s from a stream s. Then

$$\tau_{wait} = \tau_{\rightarrow w_s} + \tau_{\bowtie w_r} + \tau_{\rightarrow w_{rs}} - (\Delta t + \tau_{\rightarrow w_t}) \quad (2)$$

where Δt represents a time slot between the arrivals of d_s and d_t. If the time slots $\tau_{\rightarrow w_s}$, $\tau_{\bowtie w_r}$, $\tau_{\rightarrow w_{rs}}$, and $\tau_{\rightarrow w_t}$ do not depend on the frequencies of input streams then the value of τ_{wait} would depend only on Δt. If the frequencies of r and s are lower than the frequency of t then Δ_t will be longer, the blocking time will be shorter, (1) will be true, and as a consequence we will benefit from the existence of w_{rs}. In the extreme case, when the frequencies of r and s are equal to zero i.e. r and s are the static data containers, joining of the latter containers performed before the processing of any item from t will be the optimal solution. Elimination of the intermediate results requires the transformations of the syntax tree of logical level expression into left- or right-deep syntax trees such that after each transformation at least one of the arguments is at left- or right-deep position in a syntax tree.

The next group of optimizations addresses the processing rates. Consider a dataflow expression δ:α_1, ..., α_n and assume that each operation α_i needs the average time τ_i to process a single data item and that α_i generates on average f_i data items from the processing of one input item. The total time T_δ spent on computations of δ is equal to

$$T_\delta = \tau_1 + \beta_1 * \tau_2 + \ldots + \beta_{n-1} * \tau_n \quad (3)$$

where $\beta_i = f_1 * f_2 * \ldots * f_i$ when the operations run in a block mode and $\beta_i = (1 + f_1 * f_2 * \ldots * f_i)/2$ when the operations run in an item mode, i.e. an operation starts processing a data item as soon as the item is appended to operator's input. The execution of operation α_i in a block mode is delayed relatively to the start of α_{i-1} by a time slot needed for processing of α_{i-1} and equals to $\beta_{i-2} * \tau_{i-1}$. The execution of α_i in an item mode is delayed relative to the start of α_{i-1} by a time slot used for processing of either one, or two or $\ldots f_1 * f_2 * \ldots * f_{i-2}$ data items. An average delay is $((1 + f_1 * f_2 * \ldots * f_{i-2})/2) * \tau_{i-1}$. The objective of optimization is to find the order of operations $\alpha_1 \ldots \alpha_n$ that minimizes a value of (3). A simple solution is to permute the commutative dataflow operations in expression δ and to pick a sequence of operations that minimizes T_δ. As an example, consider the implementation of logical level expression $(r(ab) \bowtie_b s(bc)) -_c t(cd)$ as the following set

of dataflow expressions:
$\{\delta_r$: $\rightarrow(w_r)$, $\ltimes_b(w_s)$, $-_c(w_t)$, $\rightarrow(w_{out})$,
δ_s: $\rightarrow(w_s)$, $\ltimes_b(w_r)$, $-_c(w_t)$, $\rightarrow(w_{out})$,
δ_t: $\rightarrow(w_t)$, $-_c(w_s)$, $\ltimes_b(w_r)$, $\rightarrow(w_{out})\}$.
If we assume that the frequencies of input data streams are more or less the same then, a dataflow expression δ_s can be transformed to an equivalent form δ'_s: $\rightarrow(w_s)$, $-_c(w_t)$, $\ltimes_b(w_r)$, $\rightarrow w_{out}$. The new expression is better than δ_s because operation $-_c(w_t)$ returns on output no more than one data item in reply to one input data item.

3.2 Optimization of the scheduling

Optimization of scheduling is achieved through elimination of conflicts between the simultaneously processed dataflow operations. Scheduling quality is measured in a number of dashed edges in the scheduling graph. No dashed edges means no conflicts and no blocking of data-flow operations. We propose to use timestamps to eliminate some of the conflicts over access to the windows and to abolish order-preservation and serializability. Assume, that unique timestamps are attached to data items when recorded in the windows on input data streams. Then, read/write conflicts disappear when each data flow operation processing a data item d_i reads from a window only the data items with timestamps lower than timestamp of d_i. For example, the comparison of timestamps eliminates all dashed edges between the nodes α_1, α_2, α_3, and α_4 in a graph given in Figure 4.

The write/write conflicts over access to the windows on output streams are eliminated or reduced by the total or partial relaxation of the order-preservation principle. The principle can be relaxed when the frequencies of input data streams are too high to observe all modifications of the outputs. Let a dataflow expressions $E = \{\delta_1, \ldots, \delta_n\}$ processes the input streams $s_1, \ldots, s_n$. Let $D = d_1, d_2, d_3, \ldots$ be a sequence of data items appended to the streams. We say that computation of E over the data items in D is serial if the processing of each data item d_i, $i = 1, 2, 3, \ldots$ starts as soon as it is recorded in a window on its input stream and immediately after the results of d_{i-1} are recorded in w_{out}. We say that computation of E over the data items in D is concurrent if the processing of each data item d_i, $i = 1, 2, 3, \ldots$ D starts as soon as the item is recorded in a window on its input stream. Let $w_1, w_2, w_3, \ldots$ be a sequence of states of the window w_{out} obtained from the serial computations of dataflow expressions E over D. A concurrent computation of E over D is *order-preserving* if, for each $i \in \{1, 2, 3, \ldots\} w'_i = w_i$ where $w'_1, w'_2, w'_3, \ldots$ is a sequence of states of window w_{out} recorded during the concurrent computation. A concurrent computation of E over D is *weak order preserving* if there exists $i \in \{1, 2, 3, \ldots\}$ such that $w'_i = w_i$. It means that only some states of the output window are correct.

A weak order preservation adopted as a correctness criterion has an important consequence on the implementation of an effective scheduling technique. Let D_f be a finite subsequence of D. We call a dataflow expression E as *order invariant* if for any order of data elements in D_f the computations of E over D_f provide exactly the same result. For example, a dataflow expression given in Figure 3 is order invariant because for any permutation of a given finite sequence of input data items the results produced by the expression are always the same. It means that implementation of weak order preserving computations of dataflow expressions that satisfy *order invariant* property and do not access the windows on intermediate streams does not need run-time maintenance of scheduling graph. This is because the read/write conflicts over access to the windows on input streams, are sorted out by the timestamp comparisons, write/write conflicts over access to the output window are immaterial due to weak order preservation, and write/read conflicts never happen because no intermediate results are stored.

It is possible to extend timestamping on a class of dataflow expressions that satisfy an invariant property and perform write and read operations on the intermediate results of computations. Such extension requires the dynamic modification of timestamps when the data items are recorded in the windows on intermediate streams. We associate with each window on intermediate data stream the highest timestamp of a data item whose timestamp was compared against all items in the window. For instance, an operation $\ltimes_{ac}(w_{rs})$ in the first example in Section 3.1 produces new items with a higher timestamp than existing item in w_{rs}. Then, all data items written to window (w_{rs}) obtain a higher timestamp associated with the items in w_{rs}.

If in the same example the operations $\rightarrow(w_{rs})$ change a timestamp of an item written to a window w_{rs} if the timestamp is lower than a timestamp associated with the window. This technique virtually reorders the intermediate results and eliminates all conflicts from the respective scheduling graph.

Weak order preserving computations do not determine precisely whether the results are correct at any moment in time. *K-level order preserving* computations ensure that, after at least, k data items appended to the input streams the outputs are consistent with a hypothetical serial execution. The implementation of k-level order preserving computations requires synchronization of the system every k write operations to a window on the output data stream. The processing of input data items is delayed until the results from processing of the last data item in a sequence of k items is received on output.

4. SUMMARY AND CONTRIBUTIONS

This work investigates the optimization of data stream processing at the dataflow and scheduling levels. We extend a standard append-only model of data streams with a concept of negative data items to uniformly represent the results of non-monotonic operations, e.g. stream difference and relocations of windows over data streams. Our approach identifies the logical, dataflow, and scheduling levels of data stream processing. The optimizations at a dataflow level include the optimal translation of logical level expressions into the sets of dataflow expressions, the optimization of individual dataflow expressions, the elimination and reduction of blocking, and analysis of benefits and costs of intermediate data streams. The optimizations at a scheduling level include efficient implementation of concurrent computations with dataflow expressions, possible relaxation of order-preservation principle, and the scheduling of stream processing in the presence of the intermediate data streams.

The major contribution of this work is a clear separation of the logical and dataflow levels of stream processing. Stream processing at the dataflow level captures the execution plans which are not visible at a logical level or which are buried in the implementations of logical level operations. A sample system of dataflow operations and expressions provides a formal tool for detailed analysis of logical level computations on data streams. The model in which all elementary dataflow operations have only one input queue, and the operations read from a set of windows on remaining data streams captures the processing of single data items more precisely than the systems of operations with many input queues. The other important contributions of this work include transformation of logical level expressions into sets of dataflow expressions, analysis of the costs and benefits from intermediate data streams, and optimization of dataflow expressions. This paper also contributes to the development of a formal model of scheduling and to an efficient implementation of a model for a given set of sample operations.

5. REFERENCES

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. In *Proceedings of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 663–663, San Diego, California, June 9-12 2003.

[2] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosensstein, and J. Widom. Stream: The stanford stream data manager (demonstration descrioption). In *Proceedings of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 662–662, San Diego, California, June 9-12 2003.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, 2000.

[4] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 253–264, San Diego, California, June 9-12 2003.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, 2002.

[6] D. Carney, U. Cetintemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams a new class of data management applications. In *Proceedings of Intl. Conf. on Very Large Databases(VLDB)*, Homg Kong, China, August 2002.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 665–665, San Diego, California, June 9-12 2003.

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD 1997, Proceedings ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182, 1997.

[9] A. Levy. Special issue on adaptive query processing. *Bulletin of the Technical Committee on Data Engineering*, 23(2), June 2001.

[10] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *18th Intl. Conf. on Data Engineering*, 2002.

[11] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, 2002.

[12] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB'99, Proceedings of 25th Intl. Conf. on Very Large Data Bases*, pages 709–720, 1999.

[13] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, 1992.

[14] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB'2003, Proceedings of 29th Intl. Conf. on Very Large Data Bases*, 2003.

[15] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 37–48, 2002.

[16] G. Weikum and G. Vossen. *Transactional Information Systems:theory, algorithms and the practice of concurrency control and recovery.* Morgan Kaufmann, 2002.

[17] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First Intl. Conf. on Parallel and Distributed Information Systems (PDIS 1991)*, pages 68–77, 1991.