A Flexible Query Graph Based Model for the Efficient Execution of Continuous Queries

Yijian Bai
UCLA
bai@cs.ucla.edu

Hetal Thakkar
UCLA
hthakkar@cs.ucla.edu

Haixun Wang
IBM T. J. Watson
haixun@us.ibm.com

Carlo Zaniolo
UCLA
zaniolo@cs.ucla.edu

Abstract

In this paper, we propose a simple and flexible execution model that (i) supports a wide spectrum of alternative optimization and execution strategies and their mixtures, (ii) provides for dynamic reconfiguration when adding/deleting queries and changing optimization goals, (iii) optimizes response time in idle-waiting prone operators, such as union, joins, and operators used in time series and temporal sequence queries. Thus, we introduce a flexible and concrete model of execution semantics for continuous DSMS queries, and demonstrate its many applications. Our tuple-oriented model dovetails and complements the abstract set-oriented semantics of current DSMS constructs and operators, which are often based on relational algebra and SQL enhanced with windows.

1 Introduction

Data Stream Management Systems (DSMSs) must support efficiently (i) continuous queries on (ii) massive and bursty data streams, with (iii) real-time, or almost real-time responses. These requirements demand execution models of great efficiency and flexibility. For instance, since active queries must run continuously and without interruptions, the addition/activation of new queries (and other changes at the meta-level) must also be supported with minimal overhead and delays. At the data level, the bursty and often unpredictable nature of arriving streams often forces the DSMS to revise scheduling and execution strategies or make a number of other QoS changes. These changes might involve switching to a new execution strategy [4, 7, 9], including those like CHAIN [7] that demand a repartitioning of the original query graph. In order to support these different execution strategies and the ability to change between them with little overhead, we need execution models that are very general, flexible, and easily adaptable.

Furthermore, inasmuch as real-time, or near real-time, queries is the raison-d'etre of DSMS, a fast response is a key goal of DSMS execution strategies and continuous query optimization. However, it was shown in [10] that the temporal nature of the union and join operators used by DSMS can significantly delay the response for queries involving these operators. In [2] it was shown that the best solution to this problem consists of generating additional timestamps on-demand to reactivate idle-waiting operators. While on-demand timestamp generation is by far the best solution, it could not be supported by the execution strategy in [10] due to complexity involved. However, timestamp generation on demand is supported by our execution model, which is used in the Stream Mill system [2, 5].

The Stream Mill System supports a range of advanced features [5], including SQL extensions to specify and search complex sequence of events in data streams [15]. In fact, the execution model and window-join operators described in this paper were extended to support efficiently the search for temporal sequence patterns.

Thus this paper discusses the following contributions:

1. An execution model, where complex queries and different execution strategies can be naturally represented. These include: Breadth-First-Search (BFS), Depth-First-Search(DFS), Round-Robin(RR), Tuple-batching, etc.
2. Our model can be easily turned into a Deterministic Finite Automata (DFA), using different macros for implementing different optimization goals. Changes in queries and execution strategies can be achieved with minimum overhead.
3. The execution model can minimize response time for union and join operators, and other idle-waiting-prone (IWP) operators. This is achieved by the generation of on-demand timestamps and their efficient propagation through an Enhanced Query Graph(EQG).
4. Temporal pattern detection can be efficiently supported in our execution model using specialized window join operators.

The rest of this paper is organized as follows: Section 2 introduces our flexible query execution model with its reconfigurable implementation, and demonstrates how to support different execution strategies. Section 4 discusses how the execution model can utilize Enhanced Query Graphs for on-demand timestamp propagation, to achieve query latency minimization. Then in Section 5 we discuss the support for temporal pattern detection operators, which are specialized window-join operators that can also benefit from on-demand timestamps. Finally in Section 6 we present several experiments that demonstrate the effectiveness of the model.

2 A Flexible Execution Model

2.1 Basic Operators

The basic operators in the Stream Mill system include selection, projection, union, window-join and aggregates. Stream Mill also supports User Defined Aggregates (UDAs) written in a procedure language or native SQL, which gives the query language strong expressive power. Most basic operators in Stream Mill follow the same semantics as those used in relational databases; the execution of them is straightforward: we compute the result(s) of the operator and add it to the output (production)—with timestamp equal to that of the input tuple—and remove the tuple from the input (consumption).

However, for some operators the above simple execution steps is not sufficient. For example, in Figure 2 we summarize the execution steps for union and window join, which are multi-input operators that require timestamp information for the completion of the execution. (These are also the Idle-Waiting Prone, or IWP operators. The IWP operators will be further discussed in Section 4).

The union operator performs a simple sort-merge operation on multiple input streams based on timestamps (e.g., the query graph in Figure 1 has a union operator). The symmetric window-join maintains internal window buffers to convert unbounded data streams to bounded sequences, for which we will use the widely accepted semantics proposed in [14]. In Figure 2 we specify this operator directly against the input streams. For simplicity of discussion we omit here the discussion of multi-way joins and asymmetric joins, whose treatment is however similar to that of binary joins.

2.2 Query Graphs in DSMS

We now discuss the query operator graph, which have been widely used in DSMSs to describe the scheduling of continuous queries that are composed of multiple basic operators. In general, the nodes of the graph denote query operators and the arcs denote the buffers connecting them. Figure 3 shows an example where the graph is a simple path. The directed arc from Q_i to Q_j *represents a buffer*, whereby Q_i adds tuples to the tail of the buffer (production) and Q_j takes tuples from the front of the buffer (consumption). In addition to the actual query operators, the graph also contains *source* nodes and *sink* nodes as shown in Figure 3. The arcs leaving the source nodes represent input buffers. In Stream Mill (and many other DSMSs) these are being filled by external wrappers. Thus, the source nodes monitor these buffers, and when these are empty they might (i) wait until some tuple arrives in the buffer, (ii) return control to the DSMS scheduler (that will then attend to other tasks), or (iii) generate timestamp information and propagate it through the operator network (as will be discussed later). Likewise, the arcs leading to the *sink* nodes denote the output buffers from which output wrappers take the tuples to be sent to users or to other processes.

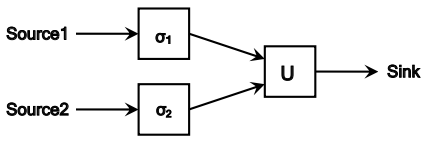


Figure 1. Simple Union

Union. When tuples are present at all inputs, select one with minimal timestamp and

- (production) add this tuple to the output, and
- (consumption) remove it from the input.

Window Join of Stream A and Stream B. When tuples are present at both inputs, and the timestamp of A is $\leq$ than that of B then perform the following operations (symmetric operations are performed if timestamp of B is $\leq$ than that of A):

- (production) compute the join of the tuple in A with the tuples in W(B) and add the resulting tuples to output buffer (these tuples take their timestamps from the tuple in A)
- (consumption) the current tuple in A is removed from the input and added to the window buffer W(A) (from which the expired tuples are also removed)

Figure 2. Basic Execution of Query Operators

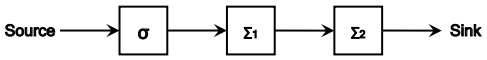


Figure 3. Simple Path Query

In general, query graphs can have several strongly connected components, where each component is a DAG. Each such DAG represents a scheduling unit that is assigned a share of the system resources by the DSMS scheduler/optimizer—not discussed in this paper, where we instead focus on the execution of each component.

The execution of each component takes place using the two-step cycle shown in Figure 4. Thus, we first execute the current operator and then select the next operator according to the execution strategy being implemented. The choice made by the execution strategy is based on the boolean values of two state variables: **yield**, and **more**. The variable **yield** is set to true if the output buffer of the current operator contains some tuples (typically, just produced by the operator); **more** is true if there are still tuples in the input buffer of the current operator. Various execution strategies can be specified using this framework, which can also support advanced optimization methods such as timestamp propagation, and special operators such variations of window-join operators for temporal pattern detection.

1. **[Execution Step]** Execute the current operator, and
2. **[Continuation Step]** Select the **next** operator for execution according to the conditions **yield** and **more** denoting the presence of output and input tuples, respectively, for the current operator

Figure 4. The *Basic Execution Cycle* forever iterates over these two steps

2.3 Different Execution Strategies

The execution model described above is very general and can be used to specify a wide range of execution strategies

with minor modifications. As examples we discuss how to use the above framework to specify the following strategies and optimization techniques: Depth-First Strategy (DFS), Breadth-First Strategy (BFS), Round-Robin (RR), and Tuple-Batching. We first present them for simple path queries, then discuss how to extend them to query graphs with arbitrary topologies.

The Depth-First Strategy (DFS) The depth-first strategy (DFS) is basically equivalent to a first-in-first-out strategy: to expedite their progress toward output, the tuples are sent to the next operator down the path as soon as they are produced.

Next Operator Selection (NOS) (Depth-First Rules)

Forward: if $\mathbf{yield}$ then $\mathbf{next} := \mathbf{succ}$

Encore: else if $\mathbf{more}$ then $\mathbf{next} := \mathbf{self}$

Backtrack: else $\mathbf{next} := \mathbf{pred}$

Repeat this NOS step on $\mathbf{next}$

Thus, after executing the query operator Q_1 in Figure 3, the algorithm checks if Q_1's output buffer is empty. If that is not the case, $\mathbf{yield}$ is true, and we execute the Q_2 operator. The Q_2 operator is the last operator in the path before the sink node: thus the tuples produced by Q_2 will actually be consumed by an output wrapper—a separate process in Stream Mill. Therefore, the algorithm continues with the consumption of all the input tuples of Q_2—i.e., the *Forward* condition is ignored when the current operator is the last before *Sink*, and instead we directly execute the *Encore* condition.

Once all the input tuples of Q_2 are processed, the $\mathbf{more}$ variable becomes false and the algorithm backtracks to its predecessor, i.e., the Q_1 operator, and executes the NOS rules on this operator. When operator Q_1's $\mathbf{more}$ condition becomes false, the algorithm should backtrack to its predecessor: however in this case, the predecessor is the *Source* node, denoting that an external wrapper is responsible for filling the input buffer of Q_1 with new tuples. Until these new tuples arrive, there is nothing left to do on this path. In this situation, control could be returned to the query-scheduler to allow the DSMS to attend to other tasks, or timestamp information could be propagated from the source as discussed later.

The Breadth-First Strategy (BFS) Breath-First Strategy (BFS) can be specified by taking the DFS rules and switching the order of the *Forward* and the *Encore* rules:

Next Operator Selection (Breadth-First Rules)

Encore: if $\mathbf{more}$ then $\mathbf{next} := \mathbf{self}$

Forward: else if $\mathbf{yield}$ then $\mathbf{next} := \mathbf{succ}$

Backtrack: else $\mathbf{next} := \mathbf{pred}$

Repeat this NOS step on $\mathbf{next}$

Round Robin (RR) Round Robin behaves very similarly to BFS—we finish every tuple on the input of one operator before visiting the next operator. However, when input becomes empty and no result tuples are produced, we simply exit this component and return control to the scheduler, instead of backtracking to look for more tuples as BFS does[1].

Next Operator Selection (Round-Robin Rules)

Encore: if $\mathbf{more}$ then $\mathbf{next} := \mathbf{self}$

Forward: else if $\mathbf{yield}$ then $\mathbf{next} := \mathbf{succ}$

Exit: else exit

Repeat this NOS step on $\mathbf{next}$

Tuple Batching All the above execution models make an execution decision after each tuple is processed. To reduce overhead, we may choose to do tuple-batching of $k-$tuples, as used in [4]—thus we only make another execution decision after $k-$tuples have been processed from each operator (or until the input is empty). Tuple-batching is primarily applied on DFS (since BFS and RR naturally performs tuple-batching already). It renders DFS to become a bit like a hybrid of DFS and BFS, thus achieve some balance on latency reduction and overhead reduction.

When tuple-batching is applied on DFS, we change the NOS rules to the following, where $processedCount$ indicates how many tuples have been taken from the input buffer.

Next Operator Selection (DFS with k-tuple-batching)

Encore: if $\mathbf{more}$ and $processedCount \leq$ k then $\mathbf{next} := \mathbf{self}$

Forward: else if $\mathbf{yield}$ then $\mathbf{next} := \mathbf{succ}$

Backtrack: else $\mathbf{next} := \mathbf{pred}$

Repeat this NOS step on $\mathbf{next}$

So far, we have studied simple path queries, we next extend our approach to query graphs containing operators with multiple inputs, such as that of Figure 5.

2.4 Unions and Joins

For operators such as unions and joins, the $\mathbf{more}$ condition evaluates to true when tuples are present in all their inputs. Thus, when some of their input buffers do not contain any tuples, **more** evaluates to false, and both DFS and BFS backtrack to a predecessor operator. Naturally, the algorithm backtracks to a predecessor feeding into a buffer that is currently empty. Therefore, if **more** is false because, say, the j^{th} input for the current operator is empty (if multiple inputs are empty, pick randomly), and $\mathbf{pred}_j$ is the operator feeding into that buffer, then the *backtrack* rule will be modified as follows:

Backtrack: $\mathbf{next} := \mathbf{pred}_j$

[1]We assume that under RR, the scheduling units are path-queries. Therefore when the query graph is not a path we need to setup virtual break nodes, as discussed later, to break the query graph into a collection of paths.

Except for these changes, the execution of operator graphs containing joins and unions is the same as that of graphs consisting of only single-input operators.

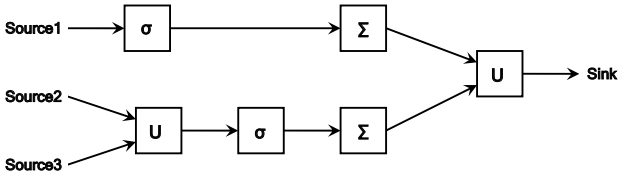


Figure 5. Query with Nested Union Nodes

3 Reconfigurable Implementation

Our simple execution model discussed above can be implemented by the following two components:

1. The maintenance of the current topology of the query graph. Topology changes may include actual query graph changes (e.g., addition or removal of operators), or *virtual* graph changes (e.g., break nodes for memory optimization, discussed later).

2. The implementation of individual *NOS* rules into a macro-library. From the library the rules are picked and matched with the operators based on the current graph topology and the execution strategy being used.

Our implementation can be viewed as a Deterministic Finite Automata (DFA). In the DFA, the states are the operators and the rules dictates how we transition from state to state. The state-transition rules change when the graph topology and optimization goal changes. Therefore, in our implementation we maintain the states of the DFA as a table, and map states to pre-defined transition rules based on both topology and the optimization goal. Table 1 shows such a mapping corresponding to Figure 3.

state	Pred	Succ	Transition
σ	-	Σ_1	DFS-Source
Σ_1	σ	Σ_2	DFS-NOS
Σ_2	Σ_1	-	DFS-Sink

Table 1. State table for single path

The rules in the mapping table are basically the Next Operator Selection (*NOS*) rules discussed in Section 2.3. For example, the *DFS-NOS* rule is what we presented for DFS in Section 2.3, and the *DFS-Source* and *DFS-Sink* rules are special cases when the input is a *Source* buffer or the output is a *Sink* buffer.

The mapping of [Σ_1,σ,Σ_2,DFS-NOS] leads to an invocation of a pre-defined macro:

$$dfs\_nos(\Sigma_1, \sigma, \Sigma_2);$$

This macro performs the NOS rule in Section 2.3, where the current operator in the rule is now Σ_1, and the *pred* is σ and the *succ* is Σ_2. Since the macro-function bodies are fixed given a graph topology and execution strategy, we maintain them as pre-defined transition-rules in a library. Thus, modification on the DFA simply becomes a re-assembly of rules and states in the table—at run time we associate each state with a function call to traverse the DFA. Such a dynamic DFA construction using a mapping table gives us tremendous flexibility and allows us to i) easily change graph topology ii) easily change optimization goals. In fact, the flexibility is so high that we can easily allow different parts of the same graph to operate under different optimization goals.

Topology Changes and Virtual Break Nodes Our execution model allows query graph to undergo constant changes, as queries/operators can be added/removed on-the-fly. Also, *virtual* graph changes, such as *break points* used by memory optimization techniques can be set/unset dynamically, as discussed next.

Memory optimization methods based on *break points* have been proposed in the literature [4, 7]. Under this optimization, a path is broken into multiple fragments by *break points*, and the fragments are directly scheduled by the scheduler. The *break points* are chosen such that the fragments of a path always achieve the largest possible memory size reduction.

Our execution model needs very little modification to support such memory optimization. In fact, all we need to do is to treat some buffers as *virtual break points*, which behave as the sink node for the previous fragment and the source node for the next fragment. For example, suppose we need to break the path of Figure 3 into two segments based on memory reduction rate, with the break point set at the output of operator Σ_1, the mapping of Table 1 changes into that of Table 2.

state	Pred	Succ	Transition
σ	-	Σ_1	DFS-Source
Σ_1	σ	-	DFS-Sink
Σ_2	-	-	DFS-Source-Sink

Table 2. State table for single path with a virtual break node

Note that, modifying the graph topology and changing the associated rules based on the new topology can be done dynamically while the system is running. Similarly, when there is an actual query graph change, when operators are added/deleted, we also perform a similar two-step process of topology update and rule-mapping change.

The Tuple-sharing *Fork* Structure When two or more operators read from the same input, we need to extend our execution model to accommodate a *fork* structure. For example in Figure 6 two different aggregates read from the same input buffer, which constitute such a *fork* structure. Memory overhead will be high if we create duplicated tuples for each consuming operator. Therefore, we need to support such tuple-sharing in our execution model.

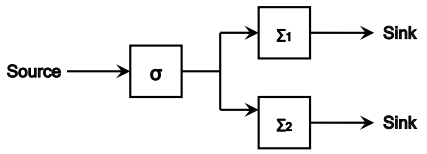


Figure 6. Tuple-sharing *Fork* structure

Instead of significantly modifying the execution rules of existing operators to handle the *fork* structure, we create *dummy* buffers in the query graph, and add a new *Fork* operator (*Fk* in Figure 7), which serve to select the next path to move forward on. The *Fork* operator has the original buffer as its single input. Then one dummy buffer is created for each branch—the buffer does not physically exist but serves as the logical input buffer for the downstream operator, where input tuples can be conceptually *deleted* independent of other operators.

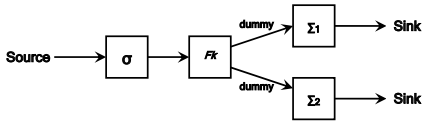


Figure 7. *Fork* Operator and Dummy buffers

Thus, the changes needed here are: i) Deletion on a dummy buffer does not physically remove the tuple, only serves to modify the tuple status stored in the dummy buffer. The dummy buffers are responsible to present the correct *current* input tuple to its own consuming operator. (Basically, each dummy buffer maintain its own head-of-queue pointer on the actual buffer.) ii) the *Fork* operator performs a round-robin execution of the downstream operators when invoked. iii) A tuple is physically removed from the buffer by the *Fork* only after all the consuming operators have deleted it from its own dummy input buffer.

Table 3 reflects the addition of a *fork* operator, which is done implicitly by the scheduler, as shown in Figure 7.

state	Pred	Succ	Transition
σ	-	$fork$	DFS-Source
$fork$	σ	$\Sigma 1$	DFS-Fork
$fork$	σ	$\Sigma 2$	DFS-Fork
Σ	$fork$	-	DFS-Sink

Table 3. State table for a *fork* structure

As a summary, the flexible DFA-based implementation enables the dynamic reconfiguration of the query graph and switching optimization goals at run time. To switch to a different strategy, we re-map transition rules for each state by modifying the rule-entry for each state inside a table. In fact, this flexible framework can allow us to run different optimization goals in different parts of the same graph, should there become such a need.

In the next two sections, we discuss how our framework can be extended to support more complex optimization mechanisms and operators. I.e., to support timestamp information propagation on Enhanced Query Graphs (EQGs), and to support temporal pattern detection using variations of the window join operator.

4 Minimizing Response Time by On-demand Timestamp Propagation

One of the critical optimization goals for continuous queries is to minimize query response time. A class of operators suffer from an *idle-waiting* problem that can significantly increase query response time. In this section we first briefly review the idle-waiting problem, which was studied in detail in [2]. Then we discuss how Enhanced Query Graphs (EQGs) can be used to efficiently propagate on-demand timestamp information in the query network to resolve the idle-waiting problem. We show how our execution model can be modified to support this timestamp-based optimization mechanism, which further improves upon the operator-by-operator propagation model used in [2].

The Idle-waiting Problem Data streams are normally ordered according to their timestamps. Usually, such timestamp ordering is a property expected and maintained by all operators in a DSMS. For instance, in the union operator a sort-merge operation is performed based on input tuple timestamp values, such that the output tuples are also ordered by their timestamps. Due to possible timestamp skews on different input streams, when any of the inputs are empty (due to lack of arrival, or data being dropped at upstream selection operators), the union operator cannot proceed and has to wait idly, since future tuples on the empty inputs could have timestamps smaller than those of the current input tuples. This *idle-waiting* problem could significantly reduce the performance of queries in a DSMS system.

The IWP Operators We term the class of operators that share the *idle-waiting* problem the *Idle-Waiting Prone* (IWP) operators. These include, for example, the union and window-join operators, which have multiple inputs and the timestamp skew on different inputs lead to idle-waiting[13, 10][2].

In [10] heartbeat tuples carrying timestamp information are sent through the query graph, in regular intervals, to release idle-waiting tuples and improve the performance of the operators. In [2], it was shown that on-demand timestamps information, which are carried by punctuation tuples and propagated through the query graph, can reduce memory consumption and improve response time much more efficiently than regular-interval heartbeats. The on-demand timestamp information for resolving idle-waiting, which we term *Enabling Time-Stamp* (ETS), basically informs the IWP operators about guarantees of future timestamp values when no new data tuples are available. Thus the IWP operators may be able to process its input tuples without waiting

[2] In fact single-input operators could also be IWP operators. For instance, the *slide* version of an aggregate on time-based windows is an IWP operator. We will omit the treatment of this operator due to space limitations.

for input data, if the timestamp guarantee is determined as of a satisfactory value.

The ETS information is generated at the *source* inputs of a query graph component (as discussed briefly later, and more detailed in [2]). Using the execution model introduced in the previous sections, the execution strategies (e.g., BFS and DFS), can be extended to support on-demand propagation of ETS information. Indeed, once the backtracking process takes us all the way back to the source node and we know that there are downstream IWP operators in an idle-waiting state, we can generate a new ETS value and send it down along the path on which backtracking just occurred, to reactivate the IWP operators.

The $\mathcal{TSM}$ Registers Timestamp information propagation and utilization in our execution model relies on the $\mathcal{TSM}$ registers, which we introduce next.

First, we start by discussing simultaneous tuples. Simultaneous tuples are tuples that have the same timestamp. To illustrate the issues arising from simultaneous tuples, let us consider a union operator with inputs A and B. If both A and B contain simultaneous tuples, these can all be processed and added to the output. But the current set of rules in Figure 2 fail to do that since they only move one tuple at a time: thus, either A or B will be emptied first and the other will be left holding one or more simultaneous tuples. To deal with simultaneous tuples, we make the following improvements to IWP operators:

- A Time-Stamp Memory ($\mathcal{TSM}$) register is introduced for each input of the IWP operator. The value of the $\mathcal{TSM}$ register is automatically updated with the timestamp value of the current input tuple and it remains in the register until the next tuple updates it.

Then the **more** condition must be modified as follows:

> **more** holds true for the query operator $\overline{Q}$ if there is at least one input tuple with timestamp value τ, where τ is the minimal value in the input $\mathcal{TSM}$ registers of $\overline{Q}$.

Figure 8. A Relaxed *more* Condition

Using this relaxed **more** condition, the execution of the union operator no longer requires that tuples must be present in all input buffers: as long as the timestamp value of a data tuple is less than or equal to the $\mathcal{TSM}$ values of the other input buffers, this tuple will be removed from the input buffer and moved to the output buffer. Thus simultaneous tuples can be all processed, even if one of the inputs becomes empty.

Besides enabling the processing of the simultaneous tuples, the $\mathcal{TSM}$ register can be used for timestamp propagation, as discussed next.

The Enhanced Query Graph (EQG) Although ETS information can be propagated by ETS carrying punctuation tuples from operator to operator, as used in [2], one observation suggests that the ETS propagation could be made more efficient. In fact, all the non-IWP operators (e.g., selections, projections, aggregates without slide) do not utilize or change ETS information. Therefore, these operators can be completely skipped during ETS propagation[3]. Furthermore, since timestamp propagation will only occur after back-tracking has been performed from the idle-waiting operator, we are sure that no data tuples will be left on intermediate non-IWP operators (any such data tuples would have been consumed first before the backtrack reaching the *Source*). Therefore, an IWP operator can deliver ETS information directly to other IWP operators down the graph, without going through intermediate operators.

Thus, we enhance our query graphs with special ETS arcs, displayed as dotted arrows in Figures 9 and 10. Observe that in Figure 9, the Q_1 operator is not an IWP operator and thus is bypassed completely by the IWP arc. Therefore, to support ETS propagation in EQG, only the execution of IWP operators need to be modified and the others are left unchanged. (A significant advantage compared to the approach used in [2], where every operator has to be modified to handle the timestamp propagation.) A second important advantage is that the propagation of the ETSs is faster since non-IWP operators are bypassed.

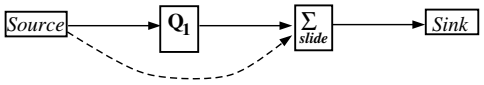


Figure 9. EQG for Simple Path Query

Given the standard query graph $QG(P)$ for a query program P, the *enhanced query graph*, $EQG(P)$ is obtained by adding the following ETS arcs: there is a ETS arc from Q_i to Q_j provided that (i) Q_j is an IWP operator and Q_i is either an IWP operator or a source node, and (ii) there is a directed path in $QG(P)$ that leads from Q_i to Q_j without going through any intermediate IWP operators. Observe that *Source* nodes are treated as IWP operators, while the *Sink* nodes are not.

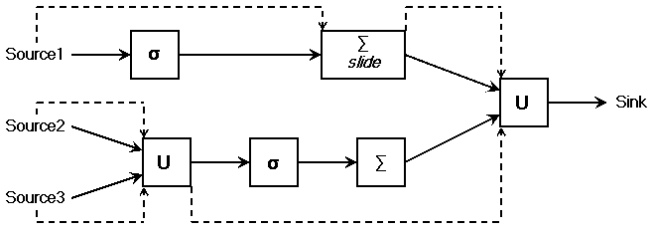


Figure 10. The Enhanced Query Graph

The solid arcs in our $EQG(P)$ denote buffers containing an arbitrary number of tuples generated by one operator and consumed by the next operator. However, dashed arcs represent *$\mathcal{TSM}$ registers* that always contain a single timestamp value. These $\mathcal{TSM}$ registers are updated in two ways:

[3] An IWP operator could potentially modify the ETS information before passing it on. For example, an union operator has multiple inputs each with its own ETS value, and it will only propagate the minimum value among them.

1. automatically, they are set to the timestamp value of the current input tuple, i.e., the first tuple in the input buffer, or
2. explicitly by an IWP operator feeding into this register.

An explicit update only occurs when the regular input buffer is empty. When the input buffer contains a tuple, the $\mathcal{TSM}$ register is updated with the timestamp value of this tuple, as in the case of punctuation tuples.

Using the EQG, the **more** condition also remains the same as that of Figure 8. However, the NOS rules must be modified to take into account that there are two different kinds of arcs and thus two different kinds of yields and successors for IWP operators, as follows: the **tuple-yield** is true when actual tuples are produced and the $\mathcal{TSM}$-**yield** is true when no tuple is produced but the $\mathcal{TSM}$ register is updated. Thus if **tuple-succ** and $\mathcal{TSM}$-**succ** respectively denote the successors along the data arc and $\mathcal{TSM}$ arc, then the forward condition becomes:

Forward: if **tuple-yield** then **next** := **tuple-succ**, else if $\mathcal{TSM}$-**yield** then **next** :=$\mathcal{TSM}$-**succ**

Of course, for non-IWP operators, we can always set $\mathcal{TSM}$ -**yield** to false since there is no $\mathcal{TSM}$-**succ**. This is also true for the last IWP operators in the $EQG(P)$: these are the IWP operators from which no $\mathcal{TSM}$ arc emanates.

A final complication with the EQG approach is due to the fact that we need to ensure that each $\mathcal{TSM}$ value generated by the source nodes is only passed to lower IWP nodes once. For that, we add a flag to each $\mathcal{TSM}$ register, whereby the register is set to active when it is set by its predecessor IWP operator, and reset to inactive when this value is passed to the successor $\mathcal{TSM}$ register[4]. This assures that $\mathcal{TSM}$ registers operate like $\mathcal{TSM}$ punctuation tuples (which can only be consumed once by each operator) and avoid possible complications.

The IWP operators in the EQG approach will operate as shown in Figure 11: if **more** fails and we are not able to deliver a data tuple to the **tuple-succ**, then we check if the $\mathcal{TSM}$ register is active, and, if so, we deliver its value to the $\mathcal{TSM}$ register of next IWP operator. The cycle of i) backtracking, and ii) ETS propagation, repeats as long as more tuples are consumed in each cycle and the next ETS value keep improving. If ETS propagation fails to help consuming any tuple, or the next ETS value no longer improves, then the current cycle stops and the control is given back to the scheduler, which will then execute other query components.

Timestamp Generation Flexible and robust mechanisms for timestamps and heartbeats are needed to achieve power and versatility in DSMSs [1, 13, 10]. Thus, the Stream Mill system supports three kinds of timestamps: *external*, *internal*, and *latent*. Here we will only use the *internal* timestamps as an example to illustrate how on-demand ETS is generated at the source nodes[5].

[4]The *source* input initiates this $\mathcal{TSM}$ register activation, when an improved ETS value can be generated at the end of backtracking.

[5]The three different types of timestamps are discussed in [2] in detail.

Union:
If **more** is true, select a tuple with timestamp τ and deliver it to the output (production); then remove it from the input (consumption).

If **more** is false and the $\mathcal{TSM}$ register is active, then update the output $\mathcal{TSM}$ register to its value. Also activate the output $\mathcal{TSM}$ register and deactivate the input $\mathcal{TSM}$ register.

Window Join of Stream A with Stream B:
If **more** is true, and input A contains a tuple with timestamp value τ, then perform the following operations (and the symmetric operations are performed if and B contain a tuple with timestamp value τ):

- (production) join of the tuple in A with the tuples in W(B) and send the resulting tuples to the output, and
- (consumption) the current input tuple in A is removed.

If **more** is false and the $\mathcal{TSM}$ register is active, set the output $\mathcal{TSM}$ register to its value. Also activate the output $\mathcal{TSM}$ register and deactivate the input $\mathcal{TSM}$ register.

Figure 11. Execution of IWP Operators in EQG

For *Internal Timestamps*, tuples are timestamped upon entering the DSMS, using system time. Therefore, when execution backtracks to a source node, whose input buffer is empty, for internally timestamped tuples the ETS value generated is that of the current system clock. With the EQG, the source nodes write the ETS value into the register at the end of the ETS arc, and the operator at the end of that dashed arc becomes next operator to be executed.

5 Temporal Pattern Detection

Besides joins and unions, other operators will also benefit from on-demand timestamp information. One example is the window-aggregate operator with slides [5]. The other example is temporal pattern detection operators, which can be efficiently supported by specialized version of window-joins, as we discuss next.

Temporal sequence pattern detection is very important in many applications. For example, RFID applications frequently need to detect temporal sequences in order to convert raw RFID tag readings into business data [8].

Consider the following example for an RFID-enabled warehouse management application:

Example 1 *A packing case arrives and is detected by an RFID reader. Then multiple product boxes arrive subsequently and are detected by another RFID reader. Each arriving product is packed into the waiting packing case. When a new packing case arrives, the current case is moved forward and the new case starts waiting for more arrivals. We want to detect the event of case-product containment—which product is packed into which case. Therefore, an RFID tag reading for each product paired with the corresponding case represents the containment relationship.*

Suppose we have two data streams containing the RFID-tag readings: **case** and **product**. Such a containment rela-

tionship can be handled using a window-join operator, as the following:

1. In the join, we require a tuple from one data stream (i.e., **product**) to join with only one tuple with a later timestamp from the other data stream (i.e., **case**). This timestamp order enforces that an incoming **product** tuple only join with the already waiting **case** tuple, but not with a future arriving one. This can be done using the standard *asymmetric* window-join, where a sliding window is applied on the **case** stream only, not on the **product** stream. In our example, for every **product** tuple we create a one-tuple sliding window on the **case** stream, to hold the most recent **case** tuple relative to the **product** tuple. Thus every arriving **product** tuple is paired with the most recent **case** tuple then removed from the data stream, while arriving **case** tuples simply enter the window without performing any pairing.

2. We need to enforce in the *asymmetric* window-join, that the join results produced by the current *case* tuple in the window will not be outputted until the timestamp information of the next *case* tuple is known. Furthermore, the timestamp of the **product** tuple in the current join can not be later than the next incoming *case* tuple.

   The issue here is, again, due to possible timestamp skew among different streams—the arrival order of **product** tuple and **case** tuple may be disturbed. As every product is packed into the currently waiting case, if a **product** tuple with a later timestamp arrives BEFORE its corresponding **case** tuple (which has an earlier timestamp), we might mistakenly think that this product is packed into the *previous* waiting case. Therefore if we do not have timestamp information of the next incoming **case** tuple, the current join candidates will have to idle-wait.

Based on the above, we use the execution in Figure 12 to detect this sequence of *B following the most recent A*. This operator is a special case of *asymmetric* window-join, where the A-stream (corresponding to the **case** stream in our example) has a one-tuple sliding window defined.

In fact, this example corresponds to a common semantics for a *sequence* operator (i.e., the *Recent* semantics as defined in [8, 3]). Thus, our join-operator implementation can be seen as one potential way to implement the *Recent* semantics of [8].

There are more possible semantics for forming a temporal sequence pattern. Another example is the *Chronicle* semantics[8, 3]. Consider the scenario of one-product-per-case packaging. We still enforce that the packing case arrives before the product, however a case only can hold one product, and there might be more than one packing case waiting when an incoming product arrives—therefore we always use the earliest packing case for the incoming product. For this scenario, the **case** tuple is still followed by the **product** tuple, however the **product** tuple is now paired with the *earliest available* **case** tuple (instead of the most-recent **case** tuple in the previous example), and both the **product** and the **case** tuples can only be used *once* for successful pairing.

Asymmetric Window Join of stream A (with One-Tuple window) and stream B.
If $\mathbf{more}$ is true, and input A contains a tuple with timestamp value τ, then replace the A-tuple in W(A) with the current input A tuple, and remove the current input A-tuple from the stream.
If $\mathbf{more}$ is true, and input B contains a tuple with timestamp value τ, then perform the following operations:

- (production) join of the tuple in B with the tuples in W(A), and send the result of the join to the output (since the *TSM* register on A has confirmed that the next A-tuple has a later timestamp).
- (consumption) the current input tuple in B is removed.

If $\mathbf{more}$ is false and the $\mathcal{TSM}$ register is active, set the output $\mathcal{TSM}$ register to its value. Also activate the output $\mathcal{TSM}$ register and deactivate the input $\mathcal{TSM}$ register.

Figure 12. Detecting B-following-most-recent-A in EQG.

We can still use a window-join operator to implement the *Chronicle* semantics. However, instead of joining on a fixed window (whose size is difficult to determine), we use the full history (i.e. the **UNLIMITED PRECEDING** window) and modify the window-join operator with the *key-join* constraints discussed next.

The *key-join* constraints are defined on the streams that participate in the join, and specifies that every tuple (identified by its key), can only participate in one successful join operation before removed out of the window from further consideration. Besides enforcing one-to-one pairing, this constraint also enables us to effectively purge tuples in the window once they are used for pairing, therefore we can freely use a large window size without paying a big performance penalty.

We use an *asymmetric* window-join operator on streams **case** and **product**, with slight modification on its execution. An **UNLIMITED PRECEDING** window and a *key-join* constraint is applied on the **cases** stream. The execution of this operator for our model is shown in Figure 13.

6 Experiments and Results

In the following experiments, we demonstrate: i) Our execution model can support different execution strategies. We use DFS and BFS as examples, both combined with ETS propagation. We study the behavior of the two strategies under different burstiness of tuple arriving pattern. ii) We study timestamp propagation using the original query graph vs. using EQG. We show that EQG reduces overhead of ETS propagation, which could amount to significant savings in a complex query graph. iii) We show that timestamp propagation on EQG can reduce output burstiness caused by idle-

Asymmetric Window Join of stream A (with unlimited preceding window and key-join constraint) and stream B.
If **more** is true, and input A contains a tuple with timestamp value τ, then put the A-tuple into W(A), and remove the current input A-tuple from the stream.
If **more** is true, and input B contains a tuple with timestamp value τ, then perform the following operations:

- (production) join of the tuple in B with the earliest tuple in W(A), and send the result of the join to the output.
- (consumption) the current input tuple in B is removed, also the earliest tuple in W(A) is removed.

If **more** is false and the $\mathcal{TSM}$ register is active, set the output $\mathcal{TSM}$ register to its value. Also activate the output $\mathcal{TSM}$ register and deactivate the input $\mathcal{TSM}$ register.

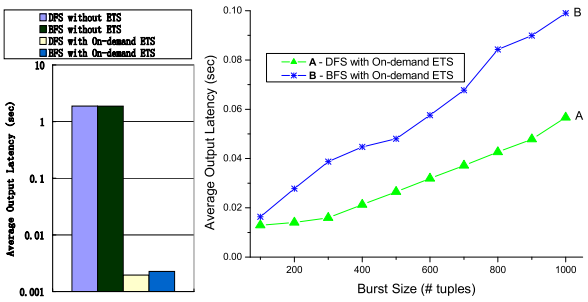
Figure 13. Detecting B-following-A with Chronicle semantics.

waiting, thus not only reduce query response time but also smooth out traffic peaks in the results.

Experiment Setup For our experiments, the Stream Mill DSMS server was hosted on a Linux machine with P4 2.8GHz processor and 1 GB of main memory. The input data tuples were randomly generated under a Poisson arrival process with the desired average arrival rates.

Depth-First versus Breadth-First In our first experiment we demonstrate how our execution framework supports the BFS and DFS strategies. We compare how BFS and DFS work with on-demand ETS propagation, and demonstrate the performance differences between BFS and DFS strategies when we consider very bursty data streams. We use the simple query graph of Figure 1, where each of the input data streams is filtered by a selection operator with low selectivity (95% tuples pass through), before the streams are unioned together. The data rates average at 1000 tuples per second on the first stream and 0.5 tuples per second for the second stream; this rate diversity can cause significant idle-waiting for tuples on the faster stream.

The graph of Figure 14 quantifies this effect. Figure 14(a) shows that BFS and DFS have similar latencies without ETS propagation, and the latencies reduce dramatically with on-demand ETS (the graph is log-scale). Figure 14(b) compares BFS and DFS performance for various burstiness factors. We keep the average rate of 1000 tuple/sec, while introducing bursts of nearly simultaneous tuples and the size of these bursts is shown on the horizontal axis. Both DFS and BFS show increasing average latency with increasing input burstiness but BFS is affected by the input burstiness more severely than DFS. This steeper increase of latency with burst size is intrinsic in the BFS execution and has little do with ETS propagation policies—there are similar delay difference even under latent timestamps. In fact, with latent timestamps, if we assume that the cost of

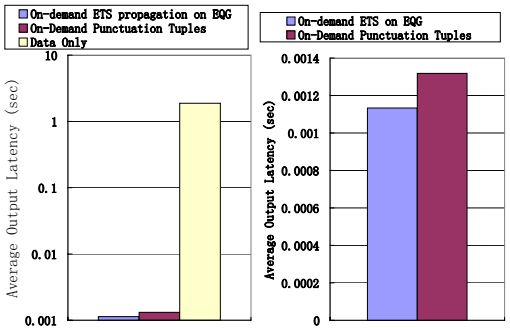


(a) (b)

Figure 14. Latency for BFS and DFS

processing one tuple is c for both the σ and the union in Figure 1, then it is easy to derive analytically that the average output latency for DFS is $(n+1)c$, while for BFS it is $(1.5n+0.5)c$, where n is the size of bursts in the Source1 input.

ETS Propagation on EQG In the next experiment we compare the effectiveness of punctuation versus that of auxiliary arcs in propagating on-demand ETS information. Without EQG, punctuation tuples are generated on-demand and propagated through the operator network as special tuples, without utilizing the special ETS arcs. The performance differences between the two approaches, which are minimal on simple query graphs, start showing when we use more complex query graphs. Figure 15 shows the results obtained for the query graph of Figure 10.



(a) (b)

Figure 15. With EQG vs. Without EQG

Conclusion for Figure 15(a): Both ETS propagation through EEG or as special tuples through the network effectively resolves idle-waiting.

Conclusion for Figure 15(b): A closer look at the two methods reveal that on-demand punctuation tuple propagation has a higher overhead than the EQG graph, which produces a higher average latency (a 14% higher latency without the special arcs). This is because punctuation tuples experience a small delay as they have to go through non-IWP operators, which are bypassed by ETS arcs. This delay be-

comes more significant when query graph becomes larger and more complex, and when the fraction of non-IWP operators becomes larger.

Furthermore, since a window-aggregate with slide [5] is used in our query, the burstiness of the output also increases significantly with the size of the slide, as shown in Figure 16, curve A. (The size of the slide dictates that a result is returned once some number of tuples are processed [6, 5]). To demonstrate this, we counted the number of tuples produced during 0.02 seconds and defined burstiness as the difference between the extremes (max and min) divided by the average of those counts. Line B in Figure 16 clearly shows that, the burstiness caused by the slide is eliminated by on-demand ETS propagation. Thus the benefits of IWP operators with on-demand ETS go beyond the ability of minimizing latency and memory in their executions in as much as these operators can equalize traffic and smooth out traffic peaks caused by bursty arrivals or bursty behavior of certain query operators. Reduction of burstiness and its undesirable side effects (as shown in Figure 14) is highly desirable in DSMS.

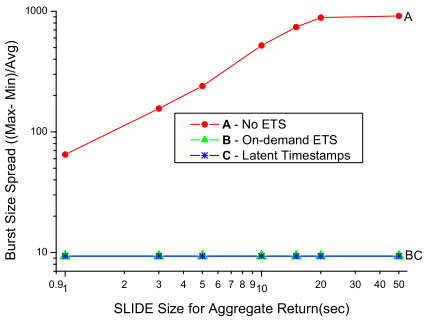


Figure 16. Burstiness vs. SLIDE Size

7 Related Work

The optimization of continuous queries has provided a major topic of DSMS research, which has primarily focused on operator scheduling [4, 9] and/or restructuring the query graph [7] to minimize memory or latency. In this paper, we do not propose new optimization strategies, rather we devise a flexible execution model on the query components that can be used to support different optimization strategies with ease.

The idle-waiting problem was discussed in [13, 10], in particular in [10] it was proposed to use regular timestamp-carrying heartbeats to solve idle-waiting problems. In [2] on-demand punctuation tuple propagation was shown to be a better approach in terms of giving rise to faster query response time with less memory overhead. In this paper, we introduces extra arcs on the query graph to further reduce the overhead and improve the response time.

Temporal event detection has been the topic of much existing research[16, 3, 11, 12], which were mostly done within the context of active databases. Here we instead use relational operators in a SQL-based data stream query language to detect such temporal patterns. The window-join variations described in this paper can be seen as a potential implementation mechanism for the temporal operators proposed in [8].

8 Conclusions

In this paper we present a flexible query graph execution model that is conducive to a dynamically reconfigurable implementation. Our model can be used to support various query optimization strategies, enabling the DSMS to change strategies at run time with ease and minimum overhead. Furthermore, our model can support advanced mechanisms for timestamp propagation and response time minimization, which is hard to achieve with other models [10]. Finally, sophisticated temporal-pattern detection operators are naturally and efficiently supported under this execution model.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[2] Yijian Bai, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. Optimizing timestamp management in data stream management systems. In *ICDE*, 2007.

[3] S. Chakravarthy and D. Mishra. Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[4] Don Carney et. al. Operator scheduling in a data stream manager. In *VLDB*, 2003.

[5] Yijian Bai, Hetal Thakkar, Chang Luo, Haixun Wang, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, 2006.

[6] Jin Li et. al. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, 2005.

[7] B. Babcock et. al. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.

[8] Yijian Bai, Fusheng Wang, Peiya Liu, Carlo Zaniolo, and Shaorong Liu. RFID data processing with a data stream query language. In *ICDE*, 2007.

[9] Mohamed A. Sharaf et. al. Preemptive rate-based operator scheduling in a data stream management system. In *AICCSA*, 2005.

[10] Theodore Johnson et. al. A heartbeat mechanism and its application in gigascope. In *VLDB*, pages 1079–1088, 2005.

[11] S. Gatziu and et al. Detecting Composite Events in Active Databases Using Petri Nets. In *Workshop on Research Issues in Data Engineering: Active Database Systems*, 1994.

[12] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB*, 1992.

[13] Utkarsh Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.

[14] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[15] Yijian Bai, Chang Luo, Hetal Thakkar, and Carlo Zaniolo. Efficient support for time series queries in data stream management systems. In *Stream Data Management–Chapter 6*. N. Chaudhry, K. Shaw and M. Abdelguerfi (EDs.), Kluwer, Vol. 30, 2004.

[16] Iakovos Motakis and Carlo Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3/4):291–325, 1997.