# On-the-fly Progress Detection in Iterative Stream Queries

Badrish Chandramouli[#] Jonathan Goldstein[#] David Maier*
[#]Microsoft Research
Redmond, WA
{badrishc, jongold}@microsoft.com

*Portland State University
Portland, OR
maier@cs.pdx.edu

## ABSTRACT

Multiple researchers have proposed cyclic query plans for evaluating iterative queries over streams or rapidly changing input. The Declarative Networking community uses cyclic plans to evaluate Datalog programs that track reachability and other graph traversals on networks. Cyclic query plans can also evaluate pattern-matching and other queries based on event sequences.

An issue with cyclic queries over dynamic inputs is knowing when the query result has progressed to a certain point in the input, since the number of iterations is data dependent. One option is a "strictly staged" computation, where the query plan quiesces between inputs. This option introduces significant latency, and may also "underload" inter-operator buffers. An alternative is to settle for soft guarantees, such as "eventual consistency". Such imprecision can make it difficult, for example, to know when to purge state from stateful operators.

We propose a third option in which cyclic queries run continuously, but detect progress "on the fly" by means of a *Flying Fixed-Point* (FFP) operator. FFP sits on the cyclic loop and circulates speculative predictions on forward progress, which it then validates. FFP is always able to track progress for a class of queries we term *strongly convergent*. A key advantage of FFP is that it works with existing algebra operators, thereby inheriting their capabilities, such as windowing and dealing with out-of-order input. Also, for stream systems that explicitly model input-event lifetimes, we know exactly which values are in the query result at each point in time.

A key implementation decision is the method for speculating. Using the high-water mark of data events minimizes the number of speculative punctuations. Probing operators on the cyclic loop to determine their external progress circulates many more speculative messages, but tracks actual output progress more closely. We show how a hybrid approach limits predictions while coming close the progress-tracking ability of Probing.

## 1. INTRODUCTION

We are seeing increased interest in iterative queries over streaming events or rapidly changing input. The Declarative Networking [19] community in particular has seen wide application of such queries for declarative routing [23], declarative overlays [17] and network monitoring and forensics [18]. Such queries are sometimes expressed as (recursive) Datalog programs. For example, the following reachability query is adapted from Condie et al. [16]. It determines which nodes in a network are reachable over links from designated source nodes (which might, for example, represent certificate servers).

```
reachable(X, [X]) :- source(X).
reachable(X, [X|P]) :- link(Y, X),
                       reachable(Y, X), notIn(X, P).
```

It derives output tuples of the form `reachable(X, P)`, meaning `X` is reachable from a source node along path `P`. The first rule says that a source `X` is reachable from itself via the trivial path `[X]`. The second rule says that node `X` is reachable by the path consisting of `P` followed by `X` (denoted `[X|P]`) if there is a direct link to `X` from a node `Y` that is reachable by path `P`. (It also includes a check to see that `X` does not lie along `P`, in which case `X` was already determined to be reachable.)

Most database techniques for evaluating such a query transform it into an algebraic expression that represents one application of the rules, which is then applied repeatedly. For example, one iteration of `reachable` can be expressed as

$$Q(r) = \pi_{M1}(\texttt{source}) \cup \pi_{M2}(\sigma_{C2}(\texttt{link} \bowtie_{C1} r))$$

where M1 adds the unit path to each source item, C1 combines a link and an input item on a common node, C2 checks for path membership and M2 augments the path from the input item. Q is initially called on the empty set, then iteratively called on results:

$$r_0 = Q(\varnothing) \qquad r_i = Q(r_{i-1})$$

until no new outputs are produced. The developers of the P2 system [17][19] and others have noted that it is not necessary to create each distinct $r_i$. Rather, a cyclic query plan can be created that simply feeds its output back in to one of its inputs. (See Figure 1.) Moreover, such a plan will function even in the presence of updates to the base data (`source` and `link` in this example).

We are interested in adding support for iterative queries in a data-stream system through similar use of cyclic query plans, to gain expressiveness. In addition to graph-traversal-type queries as seen in networking applications, we will show that cyclic plans can be used for general pattern-matching queries, such as seen in complex-event detection and temporal causality tracking [14].

One issue in such dynamic situations, however, is that the number of iterations (that is, the number of times data must circulate around the graph) is state dependent. In general, it is difficult to know when all answers have been derived up to a certain point in the input. One alternative is to "strictly stage" the query computation, taking one input (or a batch of inputs) and executing


*VLDB '09*, August 24-28, 2009, Lyon, France.

the query plan to convergence (no further outputs), temporarily buffering later input. At convergence, the answer is known to be complete up to that point in the input. However, this approach is undesirable because it introduces latency from delaying input. It can also "underload" inter-operator buffers as the last few results trickle through, leading to heavy scheduling overhead.

Another possibility is to allow "free-running" evaluation, where new external inputs (such as `source` and `link`) are added to the evaluation as they arrive. However, the best guarantee on results in such an approach is usually only *eventual consistency*: The evaluation will eventually converge to the correct result if input stops. But if input never pauses, there can be uncertainty about what results hold when. This indeterminacy can be a problem if we are trying to use such a query to monitor for a particular condition, such as "Is node *k* reachable?" In the free-running case, we can get both false negatives and false positives to such a question. It may appear at the moment that *k* is not reachable, but it actually is, it is just that the query computation has not progressed to that point yet. In the case that inputs can be retracted or expire (for example, if they are part of "soft state" [16]), it may take the query result some time to reflect such a change. Thus it may seem *k* is reachable when it is not. Also, not knowing the progress of the query interferes with purging state, for stateful operators on the loop.

This paper presents a third option to progress detection in cyclic query plans. It still allows queries to be free running, but detects the point of current progress "on the fly," using an operator we term *Flying Fixed-Point* (FFP). We rely on external streams providing punctuations that represent input progress. Punctuations have been demonstrated to effectively track progress in non-cyclic query plans, even in the presence of disorder [25]. However, punctuations will not work directly with cyclic query plans. Any cycle will have at least one binary operator (such as a Union or Join), and that operator will block on propagating punctuation until it receives corresponding punctuation on both inputs. However, since one of its inputs is based on its own output, punctuation will block forever at the operator.

FFP overcomes this problem by sitting on the cyclic loop in the query plan and issuing a *speculative punctuation*, which is essentially a guess about where computation has progressed to. FFP monitors the stream contents while the speculative punctuation circulates through the loop, in order to validate if its guess was correct. If so, FFP can issue a regular punctuation both to the query output and to the cyclic loop. (The latter is important for purging the state of stateful operators on the loop.) It performs this process without blocking its input or output, hence the "Flying" in the name.

Our initial focus for FFP is queries that are "strongly convergent" – not only do they give finite results on finite inputs, but there are finite derivations for any result. We prove that we can always detect progress for such queries. Later, we discuss useful classes of queries with this property, and also ways the strong convergence condition might be relaxed, based on recent work of others.

Cyclic query plans strictly enhance the expressive power of stream algebras. They can express queries that are not representable otherwise. Of course, for any particular query or query class, one could build a specialized algebra operator to support it. The FFP solution uses existing operators, and inherits beneficial properties they might have, such as windowing, disorder tolerance and handling retractions of events. In a stream engine such as CEDR [2] with operations that explicitly track event lifetimes, our technique is able to determine exactly what data is in the query result at any point in time. Also, the FFP approach supports making certain query parameters, such as a pattern being matched, a run-time rather than compile-time input, and hence changeable over the lifetime of a running query.

Our FFP framework admits different approaches to handling speculative punctuation. We initially devised two approaches:

- In the *High-Water-Mark* (HWM) approach, the maximum timestamp seen at the FFP operator is used as the speculation time. The speculative punctuation temporarily blocks at any loop operator that has not progressed to that time.
- In the *Probing* approach, FFP starts speculation with a high guess, but lets loop operators revise that guess downward, so as not to block speculative punctuation.

We implemented both methods in Microsoft CEP [27] to compare them. We saw that HWM does not track progress as closely as Probing, especially for disordered inputs. However, Probing can issue excess speculative punctuations in certain cases, which wastes CPU resources. Both also have issues when there are *lulls* (periods of time with punctuations but no data) in the input. Based on this experience, we developed a third approach – *Hybrid* – that attempts to get the "promptness" of Probing but with the more "stingy" behavior of HWM relative to generation of speculative punctuation. Hybrid requires adding a new non-blocking event type that communicates progress at query inputs to higher levels in the query, in particular, FFP. Further evaluation has confirmed the advantages of this approach.

# 2. GRAPH REACHABILITY

In this section, we explain how streaming query results are computed recursively through an example query. More specifically, we consider the following graph reachability query:

Given a directed graph G = (N, L) with nodes $N = \{n_i \mid i = 1..k\}$, and links $L = \{(n1_i, n2_i) \mid i = 1..j\}$, plus a set of source nodes $S \subseteq N$, compute all pairs $(n_1, n_2)$, $n_1 \in S$, $n_2 \in N$, such that $n_2$ is reachable from $n_1$ through one or more links in L. We assume that neither L nor S is known at compile time and that both can change over time. This aspect is representative of streaming queries over networks and roads, where both link properties (e.g., traffic conditions) and graph structure (e.g., links failing and recovering in a network) are volatile.

In this discussion, we give the reader an intuition for how results get calculated, and lay the foundation for thinking about cyclic streaming queries. We therefore assume that once a data item arrives it is valid forever, that there are no retractions (for example, revisions to erroneous items) in the input, and that there are no punctuations to deal with. These assumptions will be removed later.

The plan for this query is shown in Figure 1. Note that the leaves of the graph provide the input streams, and that we have one input stream for new links, and another for new source nodes. Also note that the plan is a directed graph of streaming versions of relational operators, where each arrow in the diagram is a stream, and is labeled with the format of the events traveling along the stream. We assume that every stream event is tagged with the application time $V_s$ at which the event becomes valid, which will be shown in its first field in the discussion.

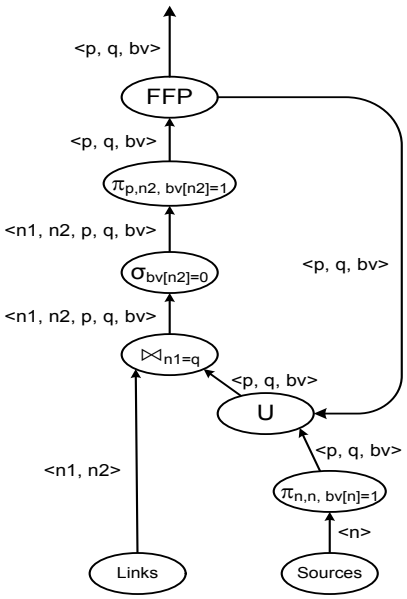



**Figure 1: Reachability Query**

We interpret the stream as describing a changing relation. The contents of the relation at any time t are all the events with $V_s \leq t$. Operators then output event streams that describe the changing view computed over the changing input according to the relational semantics of that operator. This interpretation corresponds loosely to the semantics used elsewhere [1][2][9][10][11][12].

Of note is a new operator called FFP (for Flying Fixed-Point). This operator is the means by which recursion occurs, and passes events along its input both to a conventional, non-recursive output, as well as to one of its descendants in the operator graph. The result is a form of recursion, that terminates when a fixed-point is reached (see Ramakrishnan et al. [4]). Another feature of the query plan is the schema elements labeled "bv". These are, in fact, bit vectors, each of which is k bits long, and serve the same purpose as the path field in the `reachable` example in the introduction. We use this bit vector to track visited nodes in G and avoid infinite looping through cycles.

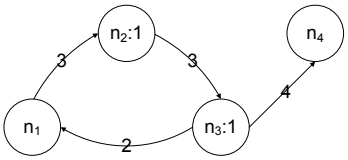



**Figure 2: Query Input**

In this example, we will feed our query the graph shown in Figure 2 through the Links input. The nodes are labeled with both the node name as well as the valid time for the Sources insertion event. Similarly, the links are also labeled with the valid times of their insertion events.

For the sake of concreteness and clarity, we will follow the execution of the query plan to completion for each distinct moment in time. We will also assume that each operator processes input events in batches such that all input events with the same valid time are processed at once. We will therefore describe the behavior of our plan at the four distinct points in time from time 1 to time 4. Since we have four distinct nodes, bv is 4 bits long.

**Time 1**: We receive two input events on the Sources stream, which correspond to nodes $n_2$ and $n_3$. The projection above the Sources stream produces the two events: (1, $n_2$, $n_2$, 0100) and (1, $n_3$, $n_3$, 0010). These events then travel through the Union and lodge in the right join synopsis. Since there is no input on the left side of the Join, we have reached a fixed-point.

**Time 2**: We receive event (2, $n_3$, $n_1$) on the Links stream, meaning that starting at time two, our input relation on the left side of the Join contains a link from $n_3$ to $n_1$. This link travels up to the Join, and lodges in its left synopsis. Given the join condition, this link joins to one row on the right side: (1, $n_3$, $n_3$, 0010). The Join then outputs (2, $n_3$, $n_1$, $n_3$, $n_3$, 0010). The Select operator then checks if there is a cycle by seeing if the path so far includes the destination in the new, derived path (by checking the 1st bit, since the path goes to $n_1$). Since this bit is not set, the event reaches the Project, which removes unneeded columns and sets the appropriate bit in bv. The result is (2, $n_3$, $n_1$, 1010), meaning that there exists a path from $n_3$ to $n_1$, starting at valid time 2. We now reach the FFP operator, which both outputs the result and inserts it into the Union below the Join. The Join then lodges the event in the right synopsis, but is unable to join it to anything in its left synopsis. We have now reached a fixed-point.

**Time 3**: We receive the events (3, $n_1$, $n_2$) and (3, $n_2$, $n_3$) in the Links stream. They travel up to the left synopsis of the Join, which already contains event (2, $n_3$, $n_1$). By joining the new events to the right synopsis, the Join produces (3, $n_1$, $n_2$, $n_3$. $n_1$, 1010) and (3, $n_2$, $n_3$, $n_2$, $n_2$, 0100). Both events get past the Select since the checked bits are 0, and so there is no cycle yet. After projection, these two events become (3, $n_3$, $n_2$, 1110) and (3, $n_2$, $n_3$, 0110). These entries are now output and loop around again to the right join synopsis. This time, however, we have not yet reached a fixed-point. By joining the new events to the left join synopsis, we produce (3, $n_2$, $n_3$, $n_3$, $n_2$, 1110) and (3, $n_3$, $n_1$, $n_2$, $n_3$, 0110).

Continuing our query, we check for cycles using our Select operator. Unlike previous times, this time we find a cycle. The first event has already visited $n_3$. We do not pass this event through to the next round of recursion and only continue with the second. After projection, it becomes (3, $n_2$, $n_1$, 1110), which is output and passed back to the Union for another round of recursion. It lodges in the right join synopsis, and produces a new event that cannot get past the select since the first three bits are set. We have again reached a fixed-point. The following output has been produced so far: (2, $n_3$, $n_1$, 1010), (3, $n_3$, $n_2$, 1110), (3, $n_2$, $n_3$, 0110), (3, $n_2$, $n_1$, 1110).

**Time 4**: We receive event (4, $n_3$, $n_4$) in the Links stream that lodges in the left join synopsis. The join then produces (4, $n_3$, $n_4$, $n_3$, $n_3$, 0010) and (4, $n_3$, $n_4$, $n_2$, $n_3$, 0110). Both events get through the Select since neither has its 4th bit set, and become (4, $n_3$, $n_4$, 0011) and (4, $n_2$, $n_4$, 0111). They are then output, and loop around to lodge in the right join synopsis without joining to anything. We have again reached a fixed-point.

A few interesting observations one can make from this example:

- For clarity, we presented the example in a way that quiesced the query between time increments. The same result, although

possibly with a different output order, would be achieved if new inputs were allowed into the recursive loop before a fixed-point had been reached. This outcome is possible because of the order insensitivity of the operators used in this recursive query plan.

- We assumed each event lives forever, once inserted. We can relax this assumption in two different ways. In the first, an event can arrive with an expiration time $V_e$, in addition to its start time. In this case, we can calculate the lifetimes of output reachability events based on the intersection of the lifetimes of all contributing input events. For example, suppose node $n_3$ was a source for the interval [1, 6], link ($n_3$, $n_1$) was present in the interval [2, 8], and link ($n_1$, $n_2$) for interval [3, 10]. Then the output event ($n_3$, $n_2$, 1110) will be valid for the interval [3, 6]. The appropriate event spans can be computed whenever two events join.

- The second relaxation is to allow an event to be explicitly retracted, as long as operators can handle retractions, such as in CEDR [2]. Such a retraction ripples through the query plan in the forward direction, removing all events derived from it as it goes. For example, suppose at time 5, the Links event ($n_3$, $n_1$) is retracted. This retraction will result in the retraction of the output event ($n_3$, $n_1$, 1010). The retraction of this event will travel around the loop, resulting in the event ($n_3$, $n_2$, 1110) also being retracted.

- The operators have bag semantics. The query can generate multiple copies of an answer if there are distinct paths using the same nodes, but the bit vector prevents infinitely many copies of the same output. This point is covered further in Section 3.

- Traditional notions of punctuations [1][3][5][6] would fail if used in the context of this query, since operators in the recursive loop wait on themselves for a punctuation. The punctuations would therefore become blocked at the Union and Join, which would receive punctuations from their non-loop inputs, but never the ones on the loop. This issue is addressed fully in Section 3.

# 3. FORMALISM

In this section, we formally define concepts related to streams, punctuations and queries; describe what is required for an operator implementation to be speculation friendly; and prove that FFP functions correctly with appropriate inputs, streams and operators.

## 3.1 Streams and Progress

We adopt a formal model of streams that we believe encompasses most previous stream models. A *stream* R is a potentially unbounded sequence $e_1$, $e_2$, … of events. An *event e* consists of one or more control parameters $c_1$, $c_2$, …, $c_n$, plus an optional *payload p*, which we write as $e = < c_1, c_2, \ldots, c_n; p>$. A payload will typically be a relational tuple, but might be something else, such as a punctuation pattern. All we require is a notion of *conformance* of a payload $p$ to a schema **R**. We say a stream R *conforms to* schema **R** if the payload of every event in R conforms to **R**.

The exact nature of control parameters varies from system to system. Some of the alternatives we have seen are:

A1. A single control parameter that contains a sequence number assigned at the inputs to a query.

A2. One control parameter that indicates what the event represents (regular tuple, punctuation, end of stream), and a second control parameter giving a timestamp supplied by the stream source [6].

A3. A control parameter indicating whether the event represents a positive tuple (insertion) or negative tuple (deletion) [10].

A4. A pair of control parameters defining a time interval over which the payload is valid [1].

We do not constrain the details of the control parameters. What we require is that for stream R(**R**), any prefix P of R can be *reconstituted* [11] into a linear sequence $r_1, r_2, \ldots, r_m$ of *snapshots* over **R**. Each snapshot is just a finite relation over **R**. It is useful to consider how each additional event modifies the reconstitution. For example, with Alternative A1 above, we can treat an event $<sn, p>$ as adding a new snapshot to the list that adds $p$ to the previous snapshot. That is, it extends $r_1, r_2, \ldots, r_{sn-1}$ to $r_1, r_2, \ldots, r_{sn-1}, r_{sn}$, where $r_{sn} = r_{sn-1} \cup \{p\}$. For Alternative A4, we can view snapshots as being indexed by timestamps, and an event $<s, e; p>$ as inserting $p$ into any snapshot $r_{tk}$ in $r_{t1}, r_{t2}, \ldots, r_{tm}$ where $s \le tk < e$, plus possibly adding a snapshot $r_e$ to the end of the list if $e > tm$.

We would like to treat a stream R as representing a potentially infinite list $r_1, r_2, \ldots$ that is the limit for the reconstitution as we take longer and longer prefixes of R. We term this sequence the *canonical history* of R [1], and consider the intent of applying a function $f$ to R to be a stream S whose canonical history is $f(r_1), f(r_2), \ldots$ . However, there is no guarantee that R converges to a well defined canonical history. New events might continue to update a particular snapshot indefinitely. Thus, we require that a stream *make progress*, meaning that for each snapshot $r_i$, there comes a point in the stream where $r_i$ no longer changes.

For an event $e$ in stream R, let P be the prefix of R up to $e$, and P:$e$ be P with the addition of $e$. Let the reconstitution of P be $r_1, r_2, \ldots, r_m$, and the reconstitution of P:$e$ be $s_1, s_2, \ldots, s_n$. We define the *stabilization point* of $e$ relative to R, stable($e$), as the maximum $i$ such that

$$r_1 = s_1, r_2 = s_2, \ldots, r_i = s_i.$$

That is, $e$ does not modify any of $r_1, r_2, \ldots, r_i$. We say that stream R *progresses* if for any index $j$, there is a point after which for any event $e$, stable($e$) $\ge j$. At that point, snapshot $r_j$ is stabilized – it will no longer change. If R progresses, then every snapshot eventually stabilizes, and the canonical history is well defined. In this case, we can use R@$i$ to denote snapshot $r_i$ in the canonical history of R. Note that snapshots in a reconstitution or canonical history need not be indexed by sequential integers. Any strictly increasing sequence works; we will sometimes use timestamps in the sequel.

We consider only progressing streams, so that the canonical history is always defined. However, we must detect progress to make use of it. For some streams, this task is easy – for example, in Alternative A1, if events are assumed to be in order of increasing sequence number. Our approach accommodates disordered streams (at least in the recursive part of the query), so we will need a form of punctuation to explicitly mark progress. An event $e$ in stream R constitutes a *punctuation at i* if every event $d$ after $e$ in R has stable($d$) $> i$. We say that stream R *explicitly progresses* if for any index $j$, there is some event $e$ in R that is punctuation at $i$, where $i > j$. In some cases, such as ordered streams, "normal" events can serve as punctuations. However, to handle disordered streams, we need specific punctuation events (flagged as such with a control parameter). We assume that all

stream operators produce explicitly progressing output given explicitly progressing inputs. Thus, they must propagate punctuation appropriately.

In our definition of FFP we will also have *speculative* punctuation, which is similar to regular punctuation, but does not actually guarantee stream progress. We will refer to non-speculative punctuation as *definite* punctuation when we need to distinguish the two. In our discussion, we use dp($i$) to denote a definite punctuation event at index $i$, and sp($i$) to denote a speculative punctuation event at index $i$.

## 3.2 Queries and Fixed Points

To accommodate the algebraic representation of queries with FFP, we view a relational query Q over which we want to compute a fixed-point as having two relational parameters, r and s, designated as Q(r, s). Parameter r names an external input (and can be generalized to a set of relations). Parameter s is the recursion parameter, which represents data headed around the recursive loop. We require that schema(Q) = schema(s), and that Q is monotone on its second argument. That is, we have $Q(r, s) \subseteq Q(r, s \cup s_1)$ for any $s_1$.

We now define the fixed point of Q on r. Let

$Q^0(r) = Q(r, \emptyset)$

$Q^i(r) = Q(r, Q^{i-1}(r))$ for $i > 0$

We say tuple $t$ has level $i$ if it appears in $Q^i(r)$. The *fixed point of Q on r* is

$Q^*(r) = \bigcup_{0 \le i} Q^i(r)$.

Our goal for recursive queries over a stream R is to compute the fixed point of each snapshot in the canonical history of R. That is, given progressing stream R and Query Q, we want to produce a progressing stream S such that, for every index $i$,

$S@i = Q^*(R@i)$.

We call such an S a *fixed-point stream* for R under Q, and write $S \in Q^*(R)$. (We use membership because there could be many streams with this property.)

As we noted in the introduction, we need to avoid certain kinds of divergent behavior in computing fixed-points. The need for finite answers and finite derivations are captured in the following two definitions.

**Definition 2.1:** Query Q(r, s) is *convergent* if for each value of r, there exists a $k$ such that $Q^k(r) = Q^{k+1}(r)$.

If Q(r, s) converges at $k$, then

$Q^*(r) = \bigcup_{0 \le i \le k} Q^i(r)$.

and so must be finite.

**Definition 2.2:** Query Q(r, s) is *strongly convergent* if for each value of r, there exists a $k$ such that $Q^k(r) = \emptyset$.

Note that strongly convergent implies convergent, and that for a strongly convergent query Q, there is a maximum level ($k$) that any tuple $t$ in Q*(r) has, hence the number of derivations is finite.

## 3.3 Operations

To use FFP with a target query Q(r, s), we will need to express Q with algebraic operators that behave appropriately, particularly with regard to speculative punctuation. We say a streaming operator G is *speculation-friendly* if the following three conditions hold.

S1. G speculates correctly.

S2. G does not block on definite punctuation.

S3. G is forward moving.

We explain each of these conditions below.

S1. G *speculates correctly* if given a speculative punctuation sp($i$) in one input stream, and that every other input stream is explicitly progressing, G will eventually emit speculative punctuation sp($j$) where $j \le i$. Moreover, if it turns out that sp($i$) actually holds (that is, G receives no later event $e$ with stable($e$) $\le i$), then sp($j$) actually holds (G will emit no event $d$ with stable($d$) $\le j$). Also, if G has previously emitted a definite punctuation dp($k$), then $j \ge \min(i, k)$. This last condition says that G doesn't "back up" from previously emitted definite punctuation. In practice, it will always turn out that $i > k$, so $j > k$. To speculate correctly, G will typically need to track definite punctuation on its other inputs.

S2. (G does not block on definite punctuation.) We already assume that G will produce explicitly progressing output on explicitly progressing input. Our method further requires operators to emit output in the absence of any particular definite punctuation. Such a G must output the same collection of non-punctuation events on any two input streams with the same non-punctuation events. Any monotonic operator has a non-blocking implementation. (Section 6 discusses handling non-monotonic operators by being able to revise previous outputs.)

S3. (G is forward moving.) We require that if an input event $e$ for G contributes to output event $d$, then stable($e$) $\le$ stable($d$). In practice, it is unlikely that an operator G could arbitrarily shift events backward in time without violating condition S1.

## 3.4 The FFP Operator

To use the FFP operator to compute fixed points relative to a query Q(r, s), we need an algebraic query tree $T[O, I_r, I_s]$ for Q. O, $I_r$ and $I_s$ are essentially "ports" of this query tree, where O connects to an output stream, $I_r$ connects to an external input stream R, and $I_s$ will be for recursive input. We also view the FFP operator as having ports: $FFP[I, O_E, O_R]$. Here I connects to an input stream, $O_E$ connects to the external output stream, and $O_R$ connects to the recursive output stream. When we apply FFP to T and R, we make the following connections:

$R \rightarrow I_r$ $\quad$ $O \rightarrow I$ $\quad$ $O_R \rightarrow I_s$

$O_E$ will connect either directly to a client, or to the input of a downstream operator. We denote this arrangement of operators by FFP[R, T]. When FFP, T and R are connected in this manner, a *recursive loop* is created that passes from $O_R$ to $I_s$ to O to I. Figure 3 shows the recursive loop in our reachability query as a dashed line. Note that for this example, Q, and hence T, has two external input streams, one for sources and one for links. A useful concept in the sequel is *external progress*. The external progress of any binary operator on the loop is the maximum definite punctuation it has received on its non-loop input. The external progress of the loop is the minimum over the external progress of its binary operators. Note that stream progress in the loop may often be less far along than external progress, because events from an earlier time are still iterating through the loop.

In defining the FFP operator, we view it as operating in phases, iterating over segments of its input separated by speculative punctuations. (These phases in general will be different from the levels of recursion defined earlier.) We will assume that at startup,

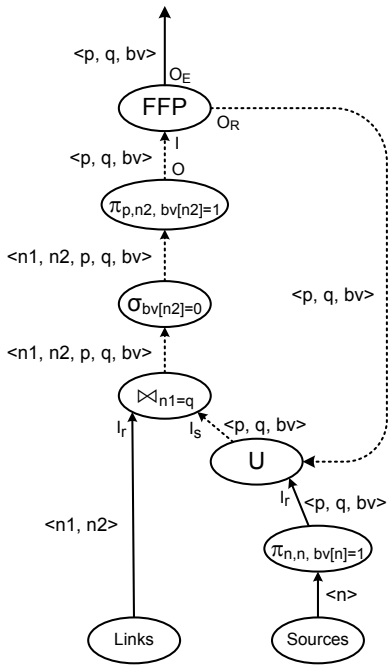



**Figure 3: Dashed Recursive Loop**

FFP emits a speculative punctuation sp($t_{min}$) on $O_R$, where $t_{min}$ is known to be before the stable points of all events on all external input streams.

A *segment* of input for FFP is a maximal sequence of events

$e_1, e_2, \ldots, e_m$, sp($t$)

received on I, where none of the $e_i$'s is a speculative punctuation. By maximality, $e_1$ must either be the first event on I, or be preceded immediately by a speculative punctuation. We allow that a segment can have $e_1, e_2, \ldots, e_m$ be the empty list

For each segment $e_1, e_2, \ldots, e_m$, sp($t$) that FFP receives on I, it performs the following steps.

F1. Emit $e_1, e_2, \ldots, e_m$ on output $O_E$.

F2. Emit those events in $e_1, e_2, \ldots, e_m$ that are not definite punctuations on output $O_R$.

F3.a. If stable($e_i$) > $t$ for $1 \le i \le k$, then emit dp($t$) on output $O_R$, followed by sp($u$) for some $u \ge t + c$ (for a fixed constant $c$).

F3.b. Otherwise, emit sp($t$) on output $O_R$.

The constant $c$ can be chosen as the minimal possible time interval, sometimes called a *chronon* [1]. Note that FFP will only ever have one speculative punctuation circulating on the recursive loop at a time. Its strategy is to keep circulating a speculative punctuation sp($t$) until it determines that the punctuation is valid, then it converts it to a definite punctuation and starts speculating at a later point. The next section will present conditions under which such speculation must always eventually succeed.

This definition of FFP might seem to indicate that it needs to buffer a whole segment in order to examine it. In fact, we will describe two implementation options that can process segments in a pipelined, continuous fashion (hence the "Flying" in "Flying Fixed-Point").

## 3.5 Specific FFP Algorithms

This general description of FFP admits several specific implementations. We describe two basic ones here, and a third, more sophisticated version in Section 5.3.

**High-Water Mark** In the High-Water Mark (HWM) approach, we monitor regular events that pass through FFP, and maintain a high-water mark `hwm` equal to the maximum value of stable($e$) over all events seen. We then use `hwm` as the time value whenever FFP issues a new speculative punctuation (as long as `hwm` is greater than the latest definite punctuation issued already). FFP remembers this value of `hwm` as `cst` (current speculation time). For every regular event $e$ subsequently received, FFP checks if stable($e$) ≤ `cst`. If so, it sets a flag `notConverged`. As the speculative punctuation sp(`cst`) travels around the loop, it will "lodge" temporarily at any binary operator whose external progress is before `cst`. When the speculative punctuation sp(`cst`) returns to FFP, and `notConverged` is set, FFP clears the flag and recirculates sp(`cst`). (Step F3.b) If `notConverged` is false, then FFP "promotes" sp(`cst`) to definite punctuation dp(`cst`), and issues it according to Step F3.a. It then issues a new speculative punctuation if `hwm` has advanced in the mean time.

For illustration, consider the example in Section 2. Suppose FFP decides to speculate immediately after getting the event (3, $n_3$, $n_2$, 1110). At this point, `hwm` = 3, so FFP sends out sp(3) and sets `cst` = 3. FFP then immediately receives the event (3, $n_2$, $n_3$, 0110), so it sets `notConverged` to true. Suppose that sp(3) travels around the loop to the Join operator and lodges, because the latest punctuation received on the `link` input is dp(2). Regular events continue to be propagated through the Join. Assume immediately after Join receives event (4, $n_3$, $n_4$), it gets dp(4). At that point, Join releases sp(3), which travels on to FFP. Since `notConverged` is set, FFP clears it and sp(3) is recirculated. This time sp(3) will not lodge at the join, but immediately propagate back to FFP. The `notConverged` flag will still be false, because only events with time = 4 are currently circulating. Hence sp(3) will be promoted to dp(3) and sent around the recursive loop. FFP then starts the speculation process over again with `hwm` = `cst` = 4.

**Probing** The HWM approach, while conceptually simple, has some limitations. Its estimate for progress (latest event seen) could be far off from the external progress of the loop as indicated by definite punctuation. Second, it will not advance progress in the absence of regular events, even if punctuations are present. To overcome those issues, we designed an alternative approach, called *Probing*. With Probing, a speculative punctuation never lodges. Rather, an operator is allowed to revise the time value downward so the punctuation can be forwarded immediately. FFP starts a new speculative punctuation with a time value of +∞, which will always be revised downward on the first circulation. Probing keeps track of the earliest time of subsequent events it receives as an input (rather than just the `notConverged` flag). Upon receiving speculative punctuation sp($s$), it compares $s$ to the earliest event time `eet`. It outputs a definite punctuation at time $d$ = min($s$, `eet`), as long as $d$ is later than the previous definite punctuation produced.

Returning to Example 1, suppose the Probing version of FFP decides to issue a speculative punctuation at the same point (just after receiving (3, $n_3$, $n_2$, 1110)). FFP emits sp(+∞) on its loop output, initializing `eet` to +∞ as well. FFP then receives (3, $n_2$,

$n_3$, 0110), which resets `eet` to 3. Assume when sp(+∞) arrives at the Union, the latest punctuation on its `nodes` input was dp(5). Thus Union will emit sp(5), which travels to the Join. If the Join's latest punctuation on its `links` input is dp(2), then Join will emit sp(2), which will travel unchanged through the Select and Project to arrive back at FFP. At this point, FFP can emit dp(2) on the loop output, since 2 is smaller than `eet` (assuming any previous definite punctuation was at a time earlier than 2). At this point, FFP can speculate again with sp(+∞). Should this speculative punctuation again return to FFP as sp(2), there will be no definite punctuation generated, since dp(2) was produced previously.

## 3.6 Correctness of FFP

We can now state our main result.

**Theorem 2.3:** Let T[O, $I_r$, $I_s$] be a query tree for a strongly convergent query Q(r, s). If T uses speculation-friendly operators and R is an explicitly progressing stream, then FFP[R, T] outputs an explicitly progressing stream S ∈ Q*(R).

**Proof:** We sketch a proof in two main parts. The first part establishes that S is a fixed-point stream for R under Q. The second part shows that S is explicitly progressing.

That S is a fixed-point stream for R under Q does not rely on the handling of speculative punctuations at all. Rather, it follows from the fact that FFP sends all input back around the recursive loop, that operators on that loop do not block on definite punctuations, and that R is progressing. The proof of this part is an induction on the level of recursion. Consider a specific snapshot r = R@*t* in the canonical history of R. The general statement is that FFP eventually receives (hence outputs to $O_E$) all events needed for $Q^m$(r) for every *m*.

***Basis case.*** The basis case is that FFP receives $Q^0$(r) = Q(r, ∅) on I. This case holds since R will eventually progress past *t* and stabilize *r*. Since T will have received all of ∅ at this point, it will output all of Q(r, ∅) to I. (There is no problem if T receives more data, because Q is assumed monotone on its second input.)

***Inductive step.*** This case follows from the observation that if FFP has received all of $Q^{k-1}$(r) on its input I, it will emit it on recursive output $O_R$. Thus T will eventually produce all tuples in Q(r, $Q^{k-1}$(r)) = $Q^k$(r).

Since Q is strongly convergent, there is some *j* such that $Q^j$(r) = ∅. Thus once FFP has received all input up through $Q^j$(r), there will be no more output events for Q*(r), and the output of FFP will progress past time *t*.

Demonstrating the explicit progress of S requires two things. (1) Any dp(*t*) that FFP emits on $O_E$ must be correctly placed. That is, no later event *e* will be emitted with stable(*e*) < *t*. (2) For any index *u*, FFP will eventually emit a definite punctuation tp(*t*) for some $t \geq u$.

For (1), we note that FFP will always see the end of a segment (that is, the next speculative punctuation). After FFP emits any events on $O_R$ in step F2, it will necessarily emit a speculative punctuation on $O_R$ in step F3.a or F3.b. Because every operator on the recursive loop is speculation-friendly, each must eventually pass on the speculative punctuation until it gets back to I. Now consider segment $e_1$, $e_2$, …, $e_m$, sp(*t*) that satisfies the If-statement in step F3.a. When $e_1$, $e_2$, …, $e_m$ are sent out again on $O_R$, any event *d* they will produce in the next segment will have stable(*d*) > *t*, since all operators on the recursive loop are forward moving. This situation will be true for all subsequent segments, by similar reasoning. Thus the speculative punctuation sp(*t*) was actually valid, and FFP can convert it safely to dp(*t*). Since R is explicitly progressing, T will eventually produce a definite punctuation dp(*u*) where $u \geq t$. That punctuation will be correctly placed in the output of T by the properties of its operators, and hence will be correctly placed in the output of FFP.

For (2), we note that a speculative punctuation sp(*t*) can only be recirculated a finite number of times by step F3.b before step F3.a applies. Since the input of FFP progresses, as shown in the first part of the proof, there must eventually be a segment where $e_1$, $e_2$, …, $e_m$ all have stable points after *t*. Further, each time we use step F3.a, we increase the index for the speculative punctuation by at least *c*. Thus we must eventually speculate at some index $v \geq u$.

**End of Proof.**

The hypotheses in Theorem 2.3 are actually stronger than they need be. Any operators in T that are not on the recursive loop do not need to be speculation-friendly. They only need to satisfy the condition that they emit explicitly progressing output on explicitly progressing input.

# 4. PATTERN MATCHING WITH NFAs

This section explains how to use FFP to implement arbitrary NFAs, a common paradigm for pattern matching. Pattern matching can be framed as an iterative stream query, where, given a transition table for a finite automata, and given an input sequence, we wish to find all reachable automata states [7][8]. This relationship can be formulated as a simple Datalog query:

```
Reach(B, B-1, start).
Reach(B, T, Q) :- Reach(B, T-1, R),
          Input(T, A), Transition(R,Q,A).
```

`Reach` contains all reachable automata states (3rd field), where the subsequence that matches the pattern starts at the first field and ends at the second. The first line seeds the automata with a zero-length pattern at every sequence position. The second line then combines existing found patterns with sequence elements that move the pattern to a new state through a transition. This query is strongly convergent, because we can only follow transitions along increasing sequence numbers. We are therefore limited in the number of iterative steps at any given moment by the number of received symbols, which is finite.

Figure 4 shows the resulting cyclic plan, with sample input and output. Note that the state machine is given as a streaming input, and may, in theory, change over time. Thus the plan is actually a streaming program for executing arbitrary, evolving automata. For clarity, we again assume that event lifetimes are unbounded, and explain the role of the various operators with the given input. The particular automata that we execute here searches for the pattern AB*A. The query outputs all discovered event sequences that constitute partial and complete patterns, and their associated states in the automata. The starting state is S, and the final state is called F. (Note that we could filter the output for final states if desired.)

The state machine is described using a set of transitions such that each transition absorbs an accompanying input. The Symbols input is a description of the sequence in which we attempt to find patterns. Each event has a sequence number, and a symbol, which may match a symbol in the automata transition table.

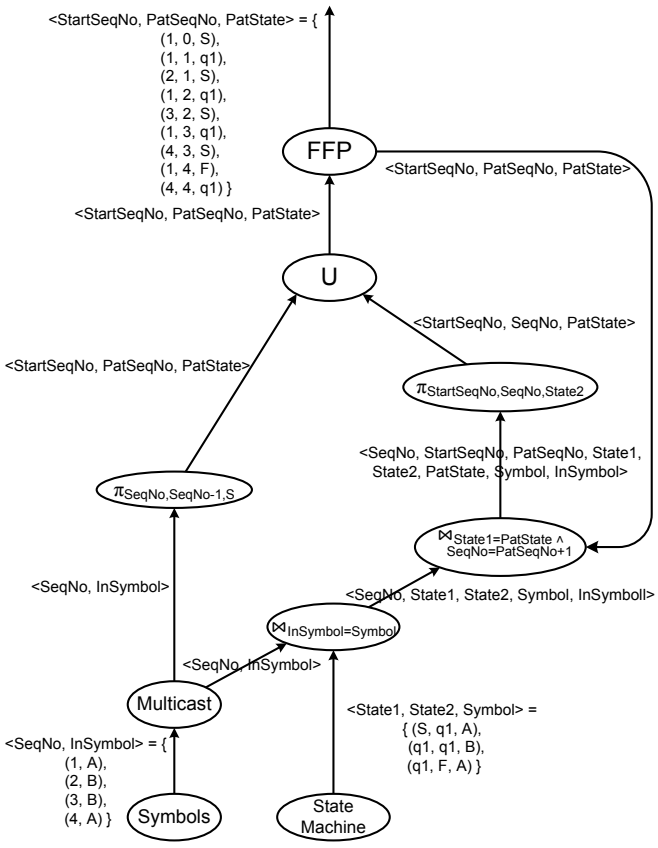



**Figure 4: Query Plan for NFAs**

While we will not describe the execution of this query in the same level of detail as the query in Section 2, we sketch its behavior. The state machine is loaded into the right join synopsis of the lower Join. When input comes along the Symbols stream, this Join finds all transitions that can be made using this symbol, and passes these transitions to the Join above, which looks for partial patterns that end at the starting state of one of the activated transitions, and which sequentially precede the new symbol. For all such matches, we have found a new (partial or complete) pattern, which we output and recursively insert back into the right side of the upper Join.

Along the left branch of the Multicast above the Symbols input, we create a seed start state on each input symbol and recursively insert it into the right join synopsis of the upper Join. In the example above, the input *sequence* is 'ABBA'. Since our query returns partial and complete discovered patterns, we should output the following patterns and their associated end sequence IDs: 'A':1, 'AB':2, 'ABB':3, 'ABBA':4, 'A':4. There are four extra outputs in Figure 4. These outputs correspond to the four seed patterns introduced by the left side of the multicast, and are regarded as patterns of length 0.

We highlight several additional aspects of the FFP approach.

**Dynamic Patterns:** Note that with support for retraction of events, we can change all or part of the NFA while the query is running. Such a capability could be useful in a fraud- or intrusion-detection scenario where data mining techniques are uncovering new suspicious patterns, and we want to incorporate them into the NFA.

**Windows:** Patterns can be restricted to occur in within a window of a particular duration *d* by giving each event an explicit expiration $V_e = V_s + d$, and intersecting lifetimes as described in the discussion of Example 1.

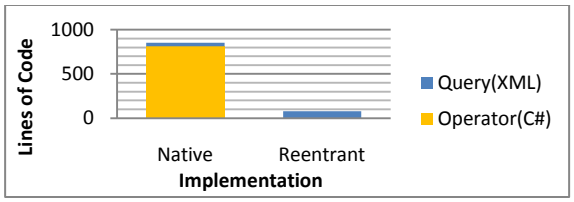
**Auxiliary State:** This example uses an unadorned NFA, but it is possible to maintain state associated with each path in the query that is used regulate transitions and test conditions. Such state is carried as additional fields in the events that circulate in the cyclic loop. As an example, consider the "W" query that looks for patterns of the form $UD^iU^jD^kU^l$, where $i + k = j + l$. Here we are looking for two segments of downticks followed by upticks that bring us back to the starting point. (The initial U is to ensure we have found the largest such pattern.) We use a basic NFA for UD*U*D*U* and keep an auxiliary field *M* in the loop events to track the number of downticks minus the number of upticks. In this case, we need a transition restriction going from the second group of downticks to the final group of upticks that $M > 1$. (Otherwise the value of M can never reach zero again.)

# 5. Evaluation and Improvement

In this section, we evaluate and improve upon the proposed techniques by examining their efficacy in the context of Microsoft CEP [27], a streaming product based on the CEDR research project [2]. This system uses the valid-time-interval approach mentioned in earlier examples and described in Goldstein et al. [2], and fully implements speculation with out-of-order input over an algebra more expressive than standard SQL (SPJ with GB, aggregation, and union). We have implemented speculation-friendly versions of Select, Project, Join, Union, Multicast, Aggregation, and Group-and-Apply (similar to SQL Group-By). All operators immediately propagate changes to the minimum punctuation seen on all inputs, and are either speculation friendly, or conform to the more relaxed notion described in Section 6. As a result, all operators may be used in the recursive loop except one. (AlterLifetime, which is used to window data, is not always forward moving. As a result, windows are applied to data before entry in the recursive loop.) Handlers were added to operators for speculative punctuations according to the different approaches, and a new FFP operator, based on the existing Multicast operator was added to the system. Overall, the modifications and additions needed to support FFP were surprisingly few.

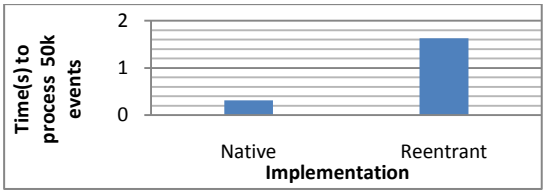
## 5.1 Comparison to Native Pattern Operator

In order to understand the tradeoffs associated with using iterative queries as opposed to writing specialized operators to implement the same functionality, we implemented a non-reentrant *ATN* (*augmented transition network* [26]) operator with similar power to the recursive plan shown in Figure 4. The only difference in expressiveness is that the cyclic query plan allows modifications to the automata over time, while the ATN operator must be halted and restarted if the automata changes. Great care was taken in implementing this operator, as its use will likely last beyond the lifetime of this evaluation. It produces exactly the same output as the cyclic plan, and state was kept to the minimum needed to provide the right in-memory indices for very high performance. The amount of time required to build this operator was vastly more than the time to write the reentrant automata query. As a proxy to represent the time required to build the operator, we report lines of code for the combined query (verbose XML) and operator implementation (C#) for the ATN operator versus lines of code for the cyclic query in the following figure:




On the other hand, one would expect the ATN operator to perform significantly better than the solution easily built using a cyclic query. We implemented a version of the W query described in Section 4, using a single integer register in our ATN to count the number of upticks and downticks, ensuring that the second trough is not above the first trough, and that we end with a zero count. The resulting ATN has 7 states, 11 transitions, and one final state.

We ran this ATN over an ordered stream of evenly weighted coin tosses with a window size of 30 events. All data was first read from disk and parsed into events before timing began. The events were then processed through the system as quickly as possible with the standard level of batching. Output was dropped to avoid including the output cost in the result. The results are shown in the figure below. Note that we achieve a respectable 30K events/s with our reentrant implementation, comparable to solutions proposed by others. In comparison, a carefully tuned and indexed native implementation in our system achieves approximately 150K events/s, a factor of 5 difference.




To sum up, the iterative query was vastly easier to write (a tenth the code), did not require source-level access to our system or knowledge of system internals, is capable of modifying the automata on the fly (possible for the native operator at increased development cost), and has respectable performance. In comparison, however, to our highly optimized pattern-matching operator, it is 5x slower for this particular query.

## 5.2 High-Water Mark versus Probing

In this section, we experimentally study the comparative strengths and weaknesses of the HWM and Probing versions of progress detection. We begin with a discussion and motivation for our performance metrics and the various parameters that we vary.

### *5.2.1 Evaluation of FFP Speculation Alternatives*

Our basic FFP framework allows some latitude in selection and processing of speculative punctuations. Before proceeding to results, we discuss our main evaluation metrics and the experimental parameters we vary in performance experiments.

**Lag:** We want to characterize how closely the punctuations output by FFP track actual stream progress. Figure 5 illustrates a relevant metric, called *lag*. The x-axis represents system time and the y-axis application time, in arbitrary units. The dots represent output events from FFP, plotted by the time each is output versus the timestamp it carries. The solid "Real Progress" line represents the low-water mark for application time: the minimum application time of all future events. The crosses are punctuations, and the dashed "Explicit Progress" line represents the bound on low-water-mark time provided by the punctuation. The lag at any point on the x-axis is the distance between the two lines. For example, at x =18, the lag is 11.1 – 10.0 = 1.1 units. We report lag averaged across system time. In the figure, the average lag is about 0.64 units.

There are at least two sources of lag in the FFP setting. The first is high or low estimates of progress in speculative punctuations. For example, with HWM, if the HWM time of regular events is far ahead of actual progress, the speculative punctuation will lodge in some loop operator for an extended period. Conversely, if the progress estimate is too low, it might result immediately in a definite punctuation, but will not be as tight of a bound on actual progress as it could be. The second source of lag is the batching of events. In most stream engines, an operator tries to process a batch of events when invoked, to amortize scheduling overheads. FFP will read at most one speculative punctuation from any batch, hence will output definite punctuation no more than once per batch. It is also worth noting that under the assumption that external punctuations are *d* time units apart, then average lag can be no better than *d*/2, even for non-cyclic stream processing. As a result, this quantity serves as a useful lower bound on lag.

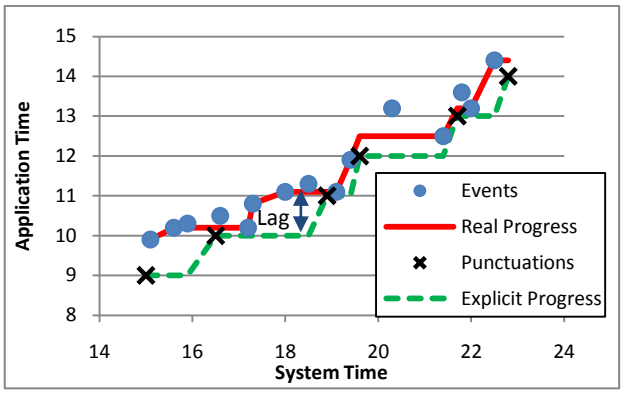



**Figure 5: Lag**

**Number of Speculative Punctuations:** We measure the number of speculative punctuations emitted by FFP and compare that to the total number of definite punctuations that it outputs. When there are no regular events to process, the Probing approach continues to circulate speculative punctuations, which may waste CPU resources.

**Join-Synopsis Size:** We also report the average size (in number of events) of the synopses maintained by joins on the cyclic loop. Joins are the major contributor to operator state in our queries, and are the main beneficiary, memory-wise, when FFP produces a definite punctuation.

**Liveliness:** We wish to capture the responsiveness of the system, so we also measure the maximum system time between consecutive outputs. If the system becomes unresponsive for long periods despite a steady incoming stream of events, this situation indicates a problem with liveliness.

**Test Parameters:** We examine three main variables when comparing speculation approaches to FFP. (1) We believe that HWM will be more sensitive to the *amount of disorder* in the input stream than Probing, particularly to events that arrive "early" compared to the rest of the stream. (2) Periods of inactivity may also expose differences between HWM and Probing, since Probing is always trying to discover the passage of

external time. We therefore vary the *input rate* of the query. (3) The frequency and duration of *lulls* in the input affects both approaches, but in different ways. The HWM approach will not establish new punctuations in the presence of lulls, while the Probing approach may over-speculate.

### *5.2.2 Experimental Results*

In all experiments, the query is the same one described at the end of Section 4 (the "W" query), and the data is generated and query executed identically to Section 5.1, except where varied as described in the individual experiments.

**Effect of Disorder:** The first experiment introduces disorder in the input stream by shifting every 100th event forward in the input stream by a varying number of punctuations. The results are shown in Figure 6, Figure 8, and Figure 10. (The Hybrid results are described in the next section.)

As expected, the increasing tendency of HWM to choose time stamps further and further into the future leads to fewer output punctuations, which both prevents state cleanup and increases lag. Note the effect seems to be linear with the amount of disorder. Probing, on the other hand, is constantly looking for opportunities to issue definite punctuations, which results in no observable dependency on the type of disorder introduced here. We also collected results for the number of speculative punctuations circulated. The graph looks very similar to the graph for the number of output punctuations. While the number of speculative punctuations was insensitive to disorder with Probing, the total varied between 20,000 and 30,000, whereas for HWM, the numbers were almost the same as the number of output punctuations.

**Varying Input Rate** In the next experiment, we vary the rate at which data enters the query to generate periods of query inactivity. Specifically, we vary the interval at which we introduce each 100 events. The results are shown in Figure 7. As expected, the number of speculative punctuations circulated by Probing increases very quickly as inactivity increases, and quickly becomes orders of magnitude higher than HWM. While this behavior might seem benign, if another query is running in the system, the speculation can degrade the other query's performance. There were no appreciable differences concerning join size, so that graph is omitted.

**Effect of Lulls** In the next experiment, we introduce lulls by randomly removing all events between successive punctuations if we lose a coin toss. We vary the weight of the coin to generate lulls of varying duration in the input. In addition, we introduce events into the query in such a way the passage of time between punctuations entering the query closely matches the passage time reflected by the punctuation timestamps. The results are shown in Figure 9 and Figure 11.

There are two noteworthy phenomena here. First, because the system experiences increasing periods of inactivity as we increase the removal rate, the number of speculative punctuations for Probing increases dramatically. In addition, Probing, due to its aggressive polling of external time, produces many more output punctuations than HWM. But most interesting is the liveliness graph. During periods where punctuations are received but no data, HWM is unable to establish a water mark that can move output time forward. Thus increasingly long periods of time go by when HWM produces no output punctuations, even though input punctuations are received. This problem is particularly bothersome when there are stateful operators downstream of FFP that are unable to unblock during these periods. In contrast, Probing does not suffer from this problem at all. While the other measures are relatively uninteresting, it is of some interest that Probing maintains a consistent factor of 2 lag advantage over HWM, and is always close to optimal.

## 5.3 The Hybrid Approach

The evaluation of our initial implementations of FFP showed that on the whole, the Probing approach gives lower lag values (and

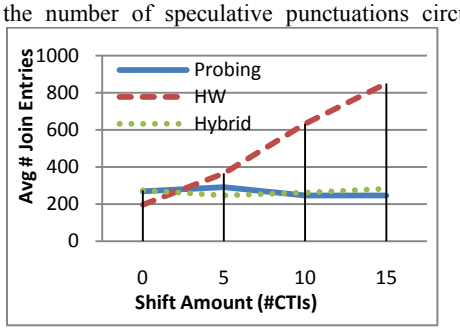



**Figure 6: Disorder vs. State**

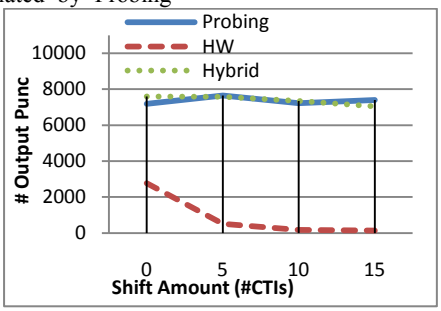



**Figure 8: Disorder vs. Output Puncts**

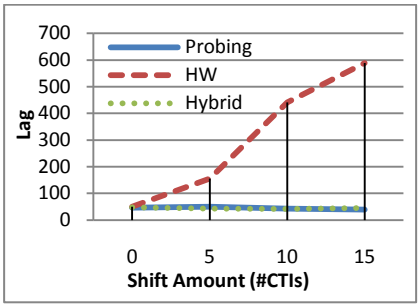



**Figure 10: Disorder vs. Lag**

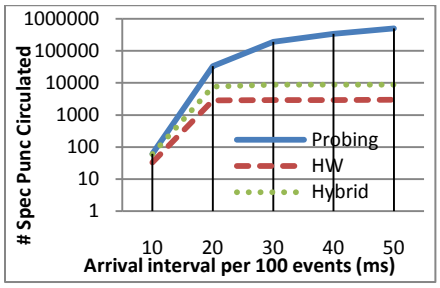



**Figure 7: Inactivity vs. Spec Punc**

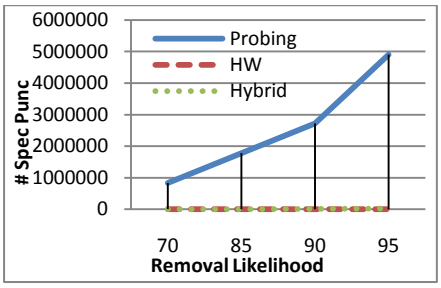



**Figure 9: Lull likelihood vs. Spec Punc**

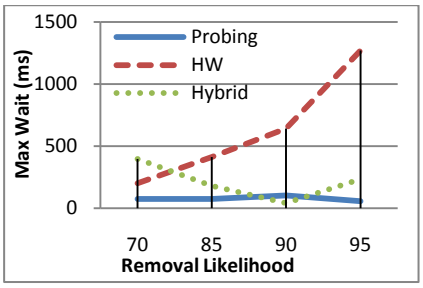



**Figure 11: Lull likelihood vs. Unresponiveness**

consequentially less memory use) than the HWM approach, especially with disorder. However, as we have seen, Probing can "overgenerate" speculative punctuations and waste CPU if input rates drop. Also, both approaches perform poorly in the presence of lulls in the input data. This experience led us to a third alternative – the *Hybrid* approach – that combines aspects of both, while adding explicit notification of input progress. Hybrid uses *external progress* (EP) events that let binary operators on the loop communicate progress on their external inputs. FFP can use EP events to help decide when to speculate on progress. For each definite punctuation dp($t$) on a query input, there is an external progress event ep($t$) with the same time value. An operator receiving an EP event immediately passes it on to its output.

In the Hybrid approach, speculation is *enabled* only if the latest EP event ep($t$) is later than latest definite punctuation dp($u$) produced so far. There are two cases where Hybrid initiates a new speculative punctuation. The first is if it receives an EP event, speculation is enabled, and there is not a speculative punctuation already in circulation. The second is on receipt of a speculative punctuation when enabled. As with Probing, Hybrid initially speculates at $+\infty$, and lets loop operators adjust this time value downward, and keeps track of the earliest time `eet` of subsequent events it receives as input. Upon receiving sp($s$), it outputs a definite punctuation at time $d$ = min($s$, `eet`), if $d$ is later than the latest definite punctuation produced so far.

### *5.3.1 Hybrid Experiments*

Figure 6 through Figure 11 also report results for the Hybrid approach. Of key importance is that in every aspect, Hybrid assumes the best qualities of both HWM and Probing. Hybrid has very low lag, and is responsive, similar to Probing, and does not generate unbounded speculative punctuations during inactivity, similar to HWM. Also worth mentioning is that Hybrid is actually easier to implement than HWM, but slightly more difficult than Probing. Hybrid is therefore an easy choice over either of the other two approaches.

## 6. DISCUSSION

We discuss further here two requirements, strong convergence of queries and the speculation-friendly requirement on operators.

**Strong Convergence:** Does the requirement for strong convergence limit the FFP method too much? Not necessarily. First, there are non-trivial classes of queries that are strongly convergent. Second, we believe we can extend the method to work with convergent queries. Considering classes of queries, we note that many queries from Declarative Networking, such as for overlays [17] and routing [23] are strongly convergent by virtue of maintaining the path being explored and avoiding revisiting nodes. Also, bounded recursive queries fall into the strongly convergent class. In bounded recursion, the number of iterations is limited by a function of the input size. Pattern-matching queries tend to be linearly bounded, because each iteration consumes at least one input event. Basic finite-automata-based pattern matching, as in Example 2, is in this class, as are, we believe, the NFA$^b$ automata of Agrawal et al. [14] and non-reentrant Augmented Transition Networks [26]. We also believe the "traversal recursions" of Rosenthal, et al. [24] fall in this class.

For relaxing strong convergence, when there are no retractions in the input stream, placing a duplicate-elimination operator between a convergent query and FFP suffices. If retractions are allowed, we have observed a problem we term "historical divergence" in which an event and its retraction may circulate around the loop endlessly. In this case it seems necessary for duplicate elimination to track the derivation provenance of tuples it receives, and the compact representation of provenance for incremental recursive view maintenance of Liu et al. [22] may be applicable. For convergent queries that involve a monotone aggregate, such as MIN for shortest path, the technique of *aggregate selection* [19][20] may help. Aggregate selection is a generalized form of duplicate elimination in which dominated tuples (for example, paths longer than the minimum so far) can be suppressed.

**Speculation-Friendly:** Can the definition of "speculation-friendly" be relaxed to accommodate an operator that blocks on definite punctuation? Yes, but the operator must have the ability to issue events for "speculative" tuples—result tuples that may need to be revised later. However, in most punctuation-based systems we are aware of (including NiagaraST [6] and Gigascope [13]), aggregate operators block waiting for punctuation, which makes them unsuitable for the recursive loop of FFP. Such an operator will not output tuples in a snapshot from which other tuples might be derived. CEDR, in contrast, supports speculative output. Thus, on receiving a speculative punctuation with time $t$, it can emit events with speculative values for the MIN, in snapshots before $t$. Those events can travel around the recursive loop, possibly lowering the MIN values for other pairs. However, eventually each pair will reach its minimum value for a particular snapshot, and the speculative punctuation will cause no revisions of previous events. FFP will detect convergence at $t$, and can issue a definite punctuation for that time.

## 7. RELATED WORK

Related work in this area falls into five categories. One is event streams being defined in terms of changing relational snapshots, and the associated relational semantics for the operators [1][2][9][10][11][12]. The work in this paper is based on these notions of streams and operators, and may therefore be easily adapted to most implementations based on these designs.

The Declarative Networking community [19], as we have discussed earlier, use cyclic plans to evaluate Datalog programs that track reachability and other graph traversals on networks. They also concentrate on partitioning of data and computation across nodes, which has not been a focus for us.

Another category of related work is on punctuations [1][3][5][6]. While our approach makes extensive use of punctuations, as observed in Section 2, traditional punctuation semantics are insufficient to fully support recursion. We therefore introduced the idea of speculative punctuation in Section 3.

Previous work on using windowed automata to perform regular expression matching in a streaming system [7] involves creating a special-purpose pattern-matching operator, but does not expose the internal recursion for more general use. Our approach, in contrast, is a minor addition to the system and is mostly comprised of pre-existing operators and can automatically make use of existing features such as incremental evaluation, the ability to speculate, and the robustness to out-of-order input found in our previous work [1]. While the native-operator approach can provide efficiency, FFP may be used for other problems requiring recursion, such as graph traversal. Also, our approach has the unusual property that the automata itself is described using an input stream, and may change over time.

The fourth area of related work is the Cayuga project [8], which provides an Iteration operator. While that operator is expressive enough for regular-expression matching, it is not as expressive as the form of iteration used here. In fact, one can write the Iteration operator using FFP, Select, Project, Join, and Union. The Iteration operator adds recursion to an existing, non-recursive engine via a new, complex operator. In contrast, FFP is just a special case of Multicast that handles punctuation differently. Finally, the Cayuga project, while significantly improving the expressiveness of streaming systems, did not consider out-of-order event arrival.

The last area of related work is the vast literature on recursive query processing. We leverage the semantics of these approaches by describing our semantics in terms of recursive queries over snapshots. We have chosen Ramakrishnan et al. [4] as a representative survey paper.

## ACKNOWLEDGMENTS

We thank Balan Sethu Raman, Beysim Sezgin, and the entire Microsoft CEP team for suggestions and comments on this work. This work was supported in part by NSF grant IIS 0612311.

## 8. CONCLUSIONS AND FUTURE WORK

Through this work, we have come to the surprising conclusion that cyclic query plans are a simple extension to a stream query engine, highly expressive, and practical. They benefit from all the capabilities of existing operators such as incremental window evaluation and disorder tolerance. They are sufficiently expressive to attack both graph-walking queries and regular-expression pattern matching. We believe even further expressiveness is available in CEDR by speculating when necessary to ensure disorder tolerance. This ability allows operators such as aggregation and difference to be used in recursive loops, which are useful for expressing branch and bound execution strategies. Our progress-detection mechanisms may also be of use in other settings with cyclic event processing, such as continuous workflow systems [28].

Detecting forward time progress is relatively straightforward with the addition of speculative punctuations, which function similarly to regular punctuation. Through implementation and experimentation, we have developed the Hybrid approach to FFP that tracks progress closely, yet does not speculate unduly.

## 9. REFERENCES

[1] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, Mingsheng Hong: *Consistent Streaming Through Time: A Vision for Event Stream Processing*. CIDR 2007: 363-374

[2] Jonathan Goldstein, Mingsheng Hong: *Consistency Sensitive Operators in CEDR*. Microsoft Research Technical Report MSR-TR-2007-158, 2007

[3] Utkarsh Srivastava, Jennifer Widom: *Flexible Time Management in Data Stream Systems*. PODS 2004: 263-274

[4] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan: *Efficient Bottom-up Evaluation of Logic Programs. The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic Publishers, 1992

[5] Peter Tucker, David Maier, Tim Sheard, Leonidas Fegaras: *Exploiting Punctuation Semantics in Continuous Data Streams*. IEEE TKDE 15(3): 555-568 (2003)

[6] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter Tucker: *Semantics and Evaluation Techniques for Window Aggregates in Data Streams*. SIGMOD 2005: 311-322

[7] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, Neil Immerman: *On Supporting Kleene Closure over Event Streams*. ICDE 2008

[8] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M. White: *Cayuga: A General Purpose Event Monitoring System*. CIDR 2007.

[9] Arvind Arasu, Shivnath Babu, Jennifer Widom: *CQL: A Language for Continuous Queries over Streams and Relations*. DBPL 2003: 1-19

[10] Moustafa A. Hammad et al.: *Nile: A Query Processing Engine for Data Streams*. ICDE 2004: 851

[11] David Maier, Jin Li, Peter Tucker, Kristin Tufte, Vassilis Papadimos: *Semantics of Data Streams and Operators*. ICDT 2005: 37-52

[12] Sankar Subramanian, Srikanth Bellamkonda, Hua-Gang Li, Vince Liang, Lei Sheng, Wayne Smith, James Terry, Tsae-Feng Yu, Andrew Witkowski: *Continuous Queries in Oracle*. VLDB 2007: 1173-1184

[13] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, Oliver Spatscheck: *A Heartbeat Mechanism and Its Application in Gigascope*. VLDB 2005: 1079-1088

[14] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, Neil Immerman: *Efficient Pattern Matching over Event Streams*. SIGMOD Conference 2008: 147-160

[15] Eugene Wu, Yanlei Diao, Shariq Rizvi: *High-performance Complex Event Processing over Streams*. SIGMOD 2006:407-418

[16] Tyson Condie, David Chu, Joseph M. Hellerstein, Petros Maniatis: *Evita Raced: Metacompilation for Declarative Networks*. PVLDB 1(1):1153-1165 (2008)

[17] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, Ion Stoica: *Implementing Declarative Overlays*. SOSP 2005:75-90

[18] Atul Singh, Petros Maniatis, Timothy Roscoe, Peter Druschel: *Using Queries for Distributed Monitoring and Forensics*. EuroSys 2006:389-402

[19] Boon Thau Loo et al.: *Declarative Networking: Language, Execution and Optimization*. SIGMOD 2006:97-108

[20] S. Sudarshan and Raghu Ramakrishnan: *Aggregation and Relevance in Deductive Databases.* VLDB 1991:501-511

[21] Atul Singh, Petros Maniatis, Timothy Roscoe, Peter Druschel: *Using Queries for Distributed Monitoring and Forensics*. EuroSys 2006:389-402

[22] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, Boon Thau Loo: *Recursive Computation of Regions and Connectivity in Networks*. ICDE 2009

[23] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, Raghu Ramakrishnan: *Declarative Routing: Extensible Routing with Declarative Queries*. SIGCOMM 2005: 289-300

[24] A. Rosenthal, S. Heiler, U. Dayal, F. Manola: *Traversal Recursion: A Practical Approach to Supporting Recursive Applications*. SIGMOD 1986: 166-176

[25] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, David Maier: *Out-of-order Processing: A New Architecture for High-Performance Stream Systems*. PVLDB 1(1):274-288 (2008)

[26] W. A. Woods: *Transition Network Grammars for Natural Language Analysis*. Communications of the ACM 13(10): 591-606 (1970)

[27] M. Ali et al.: *Microsoft CEP Server and Online Behavioral Targeting.* VLDB 2009 (demonstration, to appear)

[28] Panayiotis Neophytou, Panos K. Chrysanthis, and Alexandros Labrinidis: *Towards Continuous Workflow Enactment Systems*. International Conference on Collaborative Computing (CollaborateCom'08), 2008