

Just-In-Time Processing of Continuous Queries

Yin Yang^{#1}, Dimitris Papadias^{#2}

[#]Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong

¹yini@cse.ust.hk

²dimitris@cse.ust.hk

Abstract— In a data stream management system, a continuous query is processed by an execution plan consisting of multiple operators connected via the “consumer-producer” relationship, i.e., the output of an operator (the “producer”) feeds to another downstream operator (the “consumer”) as input. Existing techniques execute each operator separately and push all results to its consumers, without considering whether the consumers need them. Consequently, considerable CPU and memory resources are wasted on producing and storing useless intermediate results. Motivated by this, we propose just-in-time (JIT) processing, a novel methodology that enables a consumer to return feedback expressing its current demand to the producer. The latter selectively generates results based on this information. We show, through extensive experiments, that JIT achieves significant savings in terms of both CPU time and memory consumption.

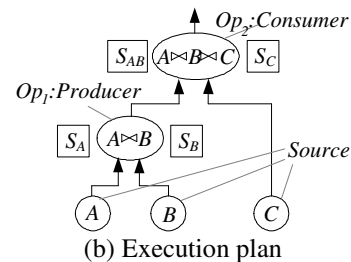
I. INTRODUCTION

In typical data stream applications, including wireless sensor networks [5], road traffic monitoring [3] and publish-subscribe services [10], data continuously flow into a DSMS. Users of the DSMS pose long-running queries, whose results are incrementally evaluated as data records arrive or expire. To answer such a query, the DSMS runs an execution plan consisting of multiple basic operators (e.g., selections, joins) connected via the *producer-consumer* relationship, where the output of the *producer* comprises the input of the *consumer*. Besides a few top-level operators whose results are directly presented to the user, most operators generate output for the sole purpose of feeding their consumers.

Figure 1a shows an example of a continuous query expressed in CQL [1]. Tuples from three streaming sources A , B and C are joined to detect a certain event. As a real-world example, an abnormal combination of readings from close-by humidity, light and temperature sensors may trigger the alarm in a factory [5]. The clause “RANGE 5 minutes” specifies that each record is *alive* for exactly 5 minutes, after which it *expires* and is *purged* from the system. Figure 1b illustrates a possible execution plan for this query consisting of two binary join operators Op_1 and Op_2 (denoted by ovals). Op_1 joins sources A and B , whereas Op_2 joins the result of Op_1 (i.e., $A \bowtie B$) with source C . Op_1 (Op_2) is the corresponding producer (consumer) of Op_2 (Op_1) respectively. The rectangles S_A , S_B (S_{AB} and S_C) denote operator states of Op_1 (Op_2), which hold tuples that came in the past, and are still valid. For example, at any time instant, S_B holds B tuples that have arrived in the last 5 minutes. A more detailed explanation of streaming join operators and their states is given in Section II.

```
SELECT * FROM
A [RANGE 5 minutes],
B [RANGE 5 minutes],
C [RANGE 5 minutes]
WHERE A.x = B.x
AND A.y = C.y
```

(a) CQL Expression



(b) Execution plan

Figure 1 Continuous query example

An important fact overlooked in most previous work is that *the producer does not have to generate a result that is not used by any of its consumers*. We illustrate it with the tuple arrival sequence of Table I. Suppose a record a_1 from source A arrives at time 1, while there are three join partners b_1, b_2, b_3 of a_1 in S_B , but no matching tuples of a_1 in S_C . Under conventional methods, operator Op_1 (i.e., the producer) uses a_1 to *probe* (i.e., to identify join partners) S_B , generating three partial results a_1b_1, a_1b_2 and a_1b_3 . Operator Op_2 (the consumer) then uses each of them to probe S_C , obtaining no results since no tuple in S_C matches a_1 . Tuple a_1 and partial results a_1b_1, a_1b_2, a_1b_3 are inserted into operator states S_A and S_{AB} respectively. Note that, at the current time instant, it is not necessary for the producer (Op_1) to generate any of the three intermediate results a_1b_1, a_1b_2, a_1b_3 since the consumer (Op_2) is unable to obtain any output with them. Yet, CPU time and memory resources are spent on computing and storing them. If no matching tuple of a_1 appears in C before the expiration of these intermediate results, the resources spent on them are wasted¹.

TABLE I
EXAMPLE TUPLE ARRIVAL SEQUENCE

Timestamp	Tuple (attribute values)	Partial results
0	$b_1(x=1), b_2(x=1), b_3(x=1)$	—
1	$a_1(x=1, y=100)$	a_1b_1, a_1b_2, a_1b_3
2	$b_4(x=1)$	a_1b_4
3	$a_2(x=1, y=100)$	$a_2b_1, a_2b_2, a_2b_3, a_2b_4$

Next, suppose that at time 2 a new record b_4 , matching a_1 , arrives while there are still no join partners of a_1 in S_C . By

¹ Some query processing algorithms (e.g., M-Join [VNB03]) do not store intermediate results. In this case the resources for producing a_1b_1, a_1b_2, a_1b_3 are wasted regardless of whether matching C tuples of a_1 arrive in the future.

probing S_A with b_4 , Op_1 generates a partial result a_1b_4 , and subsequently performs a futile probing against S_C . Similar to the previous three partial results, the computation of a_1b_4 is a waste of resources. Furthermore, the sheer presence of a_1 enlarges the size of S_A , making the probing against S_A more expensive even for incoming B tuples that do not match a_1 . Finally, assume that at time 3, a new record a_2 arrives with identical values on the join attributes x and y as a_1 . This time, 4 intermediate results (a_2b_1 - a_2b_4 shown in Table I) are generated and 4 pointless probes against S_C are performed. In general, as tuples like b_4 and a_2 keep coming, an increasing number of unwanted intermediate results are produced each time. This problem is amplified when more sources participate in the query.

Motivated by these observations, we propose Just-In-Time (JIT), a novel processing approach that dynamically adjusts the execution of producer operators based on the requirements of their consumers. Applying JIT to our example, after Op_2 finds out that a_1b_1 cannot generate join results due to the lack of matching tuples in S_C , it sends a *feedback* to Op_1 , which immediately *suspends* the processing of a_1 . Meanwhile, Op_1 stores a_1 in a *blacklist* instead of S_A to prevent future join partners of source B (e.g., b_4), or similar tuples of A (e.g., a_2), from generating unnecessary intermediate results. If a matching tuple of a_1 arrives later in C , Op_2 reports a change of demand to Op_1 , which then *resumes* the processing of a_1 and related tuples (e.g., b_4 , a_2), producing the required partial results in a just-in-time fashion.

We show experimentally that JIT achieves significant performance gains, especially for queries with comparatively high selectivity. The rest of the paper is organized as follows. Section II surveys related work. Section III outlines the general framework of JIT. Section IV provides efficient implementation of key components of JIT. Section V discusses JIT in various query plans. Section VI contains an extensive experimental evaluation. Finally, Section VII concludes with directions for future work.

II. RELATED WORK

Existing work in the data stream literature can be classified into two categories: the first aims at summarizing streaming data into synopsis structures (e.g., histograms, wavelets, sketches) and using them to provide fast, approximate answers to specific aggregate queries (e.g., [15, 12]); the second focuses on the design of general-purpose DSMSs (e.g., Aurora [2, 8], STREAM [1], TelegraphCQ [4], etc.) with formal semantics, expressive query languages and efficient query processing techniques. This work falls in the latter category, as a novel approach to continuous query processing.

A fundamental difference between a traditional DBMS and a DSMS is that the latter faces infinite inputs from the streaming sources, which cannot be handled by *blocking* operators such as joins [19]. To tackle this problem, most DSMSs adopt the *sliding-window* semantics. Specifically, for each source, the user specifies a *window* of fixed length. In the example of Figure 1, all three sources are assigned a window of 5 minutes. Hereafter, for simplicity we assume the

existence of a *global window* of length w . Each incoming tuple t is associated with a timestamp $t.ts$, and is considered *alive* during the *lifespan* of $[t.ts, t.ts+w)$. Accordingly, two input tuples t and t' with timestamps $t.ts$ and $t'.ts$ can join only if $|t.ts-t'.ts| \leq w$. A join result t with component inputs t_1, \dots, t_m is usually assigned a timestamp of $t.ts = \max_{i=1}^m t_i.ts$ [1]. For example, in Figure 1b, let ab be an output tuple of the operator $A \bowtie B$, produced by joining a (from A) and b (from B). Then, $ab.ts$ is the later timestamp between $a.ts$ and $b.ts$. Assuming that $a.ts$ and $b.ts$ represent the arrival time of the two tuples, then $ab.ts$ can be interpreted as the earliest time that ab can be created. In addition, most DSMSs require the results of a query to be reported according to their *temporal order*: for any two result tuples t and t' , t is reported before t' if and only if $t.ts \leq t'.ts$.

Query processing in a DSMS entails the construction and execution of a query plan. This work focuses on the execution part. Under this context, one of the most extensively studied problems is join processing, which is inherently more complex than single-input operators such as selections and projections. The state-of-the-art binary join algorithms (e.g., [16]) involve three steps: *purge-probe-insert*. Consider for instance, the operator $A \bowtie B$ in Figure 1b. An incoming tuple a from input stream A first purges tuples of S_B , whose timestamp is earlier than $a.ts-w$; then, it probes S_B and joins with its tuples; finally, a is inserted into S_A .

An m -way join can be computed through $m-1$ binary join steps. For example, the query plan in Figure 1b answers a 3-way join with two binary join operators. Note that such a plan (commonly referred to as *X-Join* [11]) stores intermediate join results (e.g., those of $A \bowtie B$) in the operator state (S_{AB}). In contrast, an *M-Join* [23] plan, illustrated in Figure 2a, does not store any intermediate results. Instead, tuples in each source go through a linear path of $m-1$ operators to join with tuples from other sources. This approach costs less memory than the X-Join, but more CPU time due to repeated computations of intermediate results. *Adaptive caching* [11] provides a tradeoff between memory and CPU resources, resembling a tree of M-Join operators. Finally, the *Eddy architecture* [4], shown in Figure 2b, features the novel *Eddy* operator that dynamically *routes* source tuples and intermediate results to appropriate operators to complete their processing. The proposed algorithms can be applied to all these types of join plans.

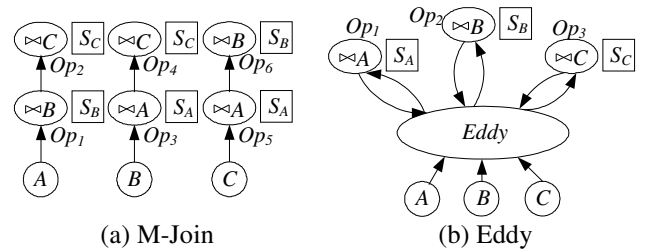


Figure 2 Example of alternative m-way join plans

A plethora of optimizations for continuous query processing have been proposed in the literature. When there are numerous operators in the system, *operator scheduling*

(e.g., [9]) finds the best execution order for minimizing memory consumption and maximizing throughput. *Adaptive query processing* techniques (e.g., dynamic plan migration [25, 24]) dynamically adjust the query to optimize performance in the presence of changing stream characteristics. When the system has insufficient CPU or memory resources to process the query completely, *load shedding* (e.g., [22]) or *operator spilling* (e.g., [20]) aims at generating a maximal (or well-represented) subset of the actual results. In case of multiple running queries, performance can be improved through shared execution (e.g., [14]) and query indexing (e.g., [18, 6]). Finally, novel hardware, such as the Cell processor [13], can be used to improve performance. Our work is orthogonal to the above methods.

Demand-driven operator execution (DOE) [21], recently proposed in the context of stream keyword search, suspends a join operator whenever (i) one of its states becomes empty, or (ii) all its consumers are suspended. As we demonstrate later, this is an extreme case that a producer generates only unwanted intermediate results; thus, DOE is subsumed by JIT. Furthermore, DOE focuses on keyword search systems, following some specific assumptions (e.g., the execution plan is always a left-deep tree), whereas the proposed solutions are generally applicable to all query plans.

III. GENERAL FRAMEWORK OF JIT

Section III-A describes JIT at an abstract level. Section III-B deals with operator scheduling issues. Section III-C discusses feedback propagation in plans where the same operator acts both as a consumer and producer.

A. Main Concepts

Let Q be a continuous query registered in the DSMS, and $EP(Q)$ be the execution plan of Q constructed by the query optimizer. For ease of presentation, hereafter we focus on the case that $EP(Q)$ is a tree of binary join operators, i.e., an X-Join plan [11], and discuss more complicated plans in Section V. JIT does not rely on any assumptions about the shape of $EP(Q)$ (which can be left-deep, right-deep or bushy), or the value distributions of the records arriving from the data sources. Let $O_C, O_P \in EP(Q)$ be two operators forming a consumer-producer relationship, i.e., the output of O_P is one of O_C 's inputs. JIT considers the case where O_C is *selective* with respect to the inputs supplied by O_P . This means that several partial results generated by O_P never contribute to the output of O_C , which we call *fruitless partial results* (FPRs). In Figure 1, assuming that during a_1 's lifespan, a matching tuple never appears in C , then a_1b_1, a_1b_2 , etc., are all FPRs with respect to consumer Op_2 .

However, given an intermediate result t from O_P , it is impossible for O_C to determine whether t is an FPR or not before its expiration, because a join partner of t may arrive at a later time. On the other hand, O_C knows those partial results that are *currently* not needed, which we call *non-demanded partial results* (NPRs). In the running example, at timestamp 1, a_1b_1, a_1b_2, a_1b_3 are NPRs with respect to Op_2 . Clearly, each NPR has two possible destinies: (i) to be matched by a future

partner, or (ii) to become a FPR after its expiration. JIT postpones the generation of NPRs of type (i) until they are demanded, i.e., when a matching partner arrives, and eliminates the production of type (ii) NPRs altogether.

JIT exploits the observation that there is a broad class of partial results that can be detected as NPRs before their generation. Their common characteristic is that they contain *minimal non-demanded sub-tuples* (MNSs), such that any output of O_P that is super-tuple of an MNS must be an NPR. In the running example, a_1 is an MNS; joining a_1 with any B tuple leads to an NPR. We require a non-empty MNS to be *minimal*, i.e., not to contain another MNS as sub-tuple. The empty tuple \emptyset is sub-tuple of any record. It is possible for \emptyset to be a valid MNS, when the opposite operator state (of O_P) at O_C is empty. In this case, *all* results computed by O_P are NPRs, and O_P can be simply suspended, achieving the same effect as DOE [21]. Figure 3 visualizes the relationship between FPR, NPR and MNS.

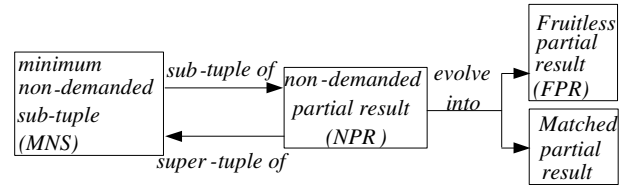


Figure 3 Concepts of FPR, NPR and MNS

According to JIT, consumer O_C detects MNSs during join processing, and reports them to O_P as a *suspension feedback*. In our example, consumer Op_2 processes input a_1b_1 , identifies a_1 as an MNS, and sends a feedback $f = \langle \text{suspend}, \{a_1\} \rangle$ to producer Op_1 . Once O_P receives such a feedback, it immediately stops generating results that are super-tuples of the specified MNSs. Continuing the example, Op_1 stops joining a_1 with B tuples, thus avoiding intermediate results a_1b_2, a_1b_3, a_1b_4 . Furthermore, if later a new tuple a_2 arrives, with identical attribute value on y (the join attribute of A and C) as a_1 , it is not joined with B , eliminating the generation of more NPRs.

Similar to an NPR, an MNS may be matched by a later partner. Therefore, O_C stores all detected MNSs in an *MNS buffer* until their expiration, and probes each incoming tuple from the opposite input against the MNS buffer. The MNS buffer may be organized as a hash table, or other index structure, to speed up the probing. Whenever O_C finds a matching tuple t of an MNS s , it removes s from the MNS buffer, and sends a *resumption feedback* containing s to producer O_P . Upon receiving this message, O_P immediately starts generating the set S_s of super-tuples of s that have not been produced before, and returns S_s to O_C . After obtaining S_s from O_P , O_C joins t with S_s to generate results, and appends S_s to the corresponding operator state. We call the producer's reactions to both kinds of feedback collectively as *dynamic production control*.

Using the running example, suppose that at timestamp 4, a new tuple c_1 ($c_1.y = 100$) arrives from source C . Op_2 finds that c_1 matches MNS a_1 , and sends the feedback $f' = \langle \text{resume}, \{a_1\} \rangle$ to Op_1 . Op_1 joins a_1 and a_2 (whose processing is also

suspended since its y attribute is identical to a_1) with tuples in S_B , obtaining $S_{a_1} = \{a_1b_2, a_1b_3, a_2b_4, a_2b_1, a_2b_2, a_2b_3, a_2b_4\}$. Note that a_1b_1 is not included in S_{a_1} because it has already been generated (before a_1 is found as an MNS). Op_1 returns S_{a_1} to Op_2 , which joins it with c_1 , generating 7 results. Finally, all tuples in S_{a_1} are appended to S_{AB} .

The general framework of JIT is flexible, in that it can be adapted to the stream (e.g., arrival rate) and query characteristics (e.g., operator selectivity). Specifically, a consumer O_C may choose not to detect *all* MNSs for a given input. Intuitively, detecting more MNSs gives better guidance to producer O_P (at the expense of higher cost at O_C), but does not affect the correctness of the output. Furthermore, O_P may decide to ignore the message and keep producing NPRs.

B. JIT and Operator Scheduling

Just-in-time processing necessitates the cooperation of O_C , O_P and the DSMS's operator scheduler to maximize its performance. In this section we present some scheduling policies starting with the case that f is a suspension feedback. At the time that O_C issues f , it is possible that O_P is currently working on producing NPRs specified by f . In the running example, when Op_2 sends $f = \langle \text{suspend}, \{a_1\} \rangle$, Op_1 may be joining a_1 with another tuple in S_B , say b_2 . Upon receiving f , JIT requires O_P to suspend its current work and immediately handle f . O_P resumes previous work only after finishing dealing with f . In the example, after handling f , Op_1 learns that a_1 is an MNS and stops joining it with S_B . Moreover, it is also desirable for the scheduler to assign O_P a higher priority than its upstream operators while processing f , because (as discussed in Section III-C) O_P may *propagate* the feedback to them.

A complication arises when the DSMS places an inter-operator queue between each pair of consumer / producer operators to store the partial results not yet processed by the consumer (in order to enable more flexible operator scheduling). After O_C identifies an MNS s , super-tuples of s may have already been produced and stored in the queue Q_{CP} between O_C and O_P . Note that O_C cannot simply delete them from Q_{CP} because they are considered "future inputs" at this moment. For instance, let t be a super-tuple of s . There may exist another tuple t' in the opposite queue (i.e., the queue of the other O_C input) such that t' matches t and $t'.ts \leq t.ts$, meaning that t and t' can generate a result. In our prototype, we process these super-tuples as normal input since the size of an inter-operator queue is usually small. When O_C detects one of them, it sends a feedback to O_P specifying s as an MNS. If O_P has already suspended generating such NPRs, it simply ignores the message.

Next we discuss resumption messages. Recall that a resumption feedback is issued by O_C during the processing of an incoming tuple t , requesting O_P to produce a set S of suppressed inputs. Because of the temporal ordering requirement, O_C must process t before generating results for subsequent inputs. This means that if O_C finishes the purge-probe-insert routine of t before S is ready, it has to *wait* for O_P to compute S . This waiting may lead to temporary silence of

O_C 's output and, in a distributed setting (where O_C and O_P are on different sites) idle CPU cycles.

JIT takes several measures to eliminate this waiting. First, for each incoming tuple t , O_C first probes t against the MNS buffer before the opposite operator states. The rationale is that if matching MNSs of t are found, O_C continues to purge / probe t against the opposite operator state, and *at the same time* O_P starts to compute the demanded partial results S . O_C waits for O_P only if the latter does not complete computing S before the former finishes probing t against the corresponding operator state. Second, after O_C sends a resumption feedback to operator O_P , the scheduler assigns O_P a higher priority than O_C . Finally, similar to the case of suspension messages, upon receiving a resumption feedback, O_P immediately suspends its current work, computes S , and then resumes the previous job. We summarize the timeline of the production resumption process in Figure 4.

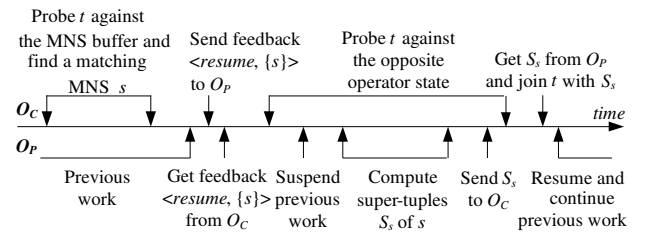


Figure 4 Timeline of production resumption

C. Feedback Propagation

In complex query plans, the producer O_P may also have upstream operators that supply its inputs. In Figure 5a, Op_3 is simultaneously a producer for Op_4 , and a consumer with regard to Op_1 and Op_2 . A subtlety in this situation is that the dynamic production control performed by an operator (as a producer) may change its demand for inputs (as a consumer). Consider the tuple arriving sequence of Figure 5c. The join predicate checked at Op_4 is illustrated in Figure 5b. For simplicity, we assume that all tuples shown in the sequence (a_1-e_1) match each other. Initially, records b_1 and c_1d_1 are present in operator states S_B and S_{CD} respectively. Then, tuple a_1 from source A is joined with b_1 , generating a_1b_1 , which is subsequently joined with c_1d_1 , producing $a_1b_1c_1d_1$.

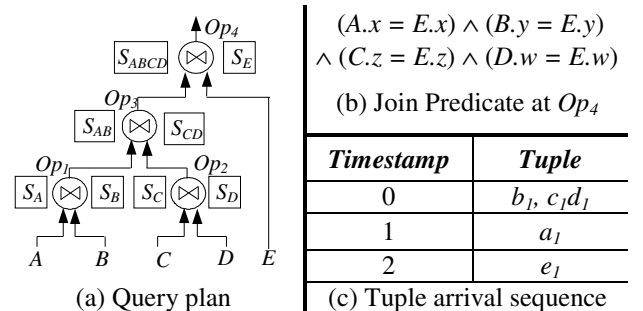


Figure 5 A 5-way join example

Now at Op_4 , suppose S_E has matching records of b_1 and d_1 , but not a_1 and c_1 . Op_4 thus sends a feedback $\langle \text{suspend}, \{a_1, c_1\} \rangle$ to Op_3 . Responding to the feedback, Op_3 stops joining

tuples containing a_1 or c_1 (e.g., a_1b_1 , c_1d_1 respectively) with their corresponding partners (inputs from Op_2 and Op_1). Consequently, Op_3 's demand for inputs has changed; in particular, it does not want inputs that are super-tuples of a_1 or c_1 any longer. Hence, it propagates the feedback $\langle suspend, \{a_1\} \rangle$ to Op_1 and $\langle suspend, \{c_1\} \rangle$ to Op_2 . Similarly, when e_1 (matching a_1 and c_1) arrives at time 2, Op_3 receives the resumption feedback $\langle resume, \{a_1, c_1\} \rangle$ from Op_4 . It then propagates $\langle resume, \{a_1\} \rangle$ to Op_1 and $\langle resume, \{c_1\} \rangle$ to Op_2 , obtains the required inputs from them, and computes the partial results requested by Op_4 .

Scheduling policies are more complex in the presence of feedback propagation, but follow the general idea described in Section III-B: (i) an operator always propagates a feedback before handling it, (ii) upon receiving a feedback, an operator suspends its current job and handles the feedback, (iii) an operator handling a suspension feedback has higher priority over its upstream ones, and (iv) an operator handling a resumption message computes the tuples requested by its consumer, while at the same time expecting inputs from its producers (to which it has propagated the feedback) and has a lower priority over these producers. Figure 6 summarizes the general framework of JIT, which consists of two procedures: *Process_Input* (performed by the consumer) and *Handle_Feedback* (by the producer).

```

Process_Input (Tuple  $t$ , Operator  $Op$ )           // Consumer
// INPUT =  $t$ : an input tuple
//  $Op$ : the producer operator that generates  $t$ 
1. Let  $S_t(S_o)$  be the operator state corresponding to  $t$  (opposite of  $t$ )
2. Let  $NB_o$  be the MNS buffer opposite of  $t$ 
3. Initialize MNS set  $\Pi$  to empty
4. Purge  $NB$ , and probe  $t$  against  $NB$ 
5. For each MNS  $s \in NB_o$  matching  $t$ 
6.   Remove  $s$  from  $NB_o$ , and add  $s$  to  $\Pi$ 
7. If  $\Pi$  is not empty
8.   Send a feedback  $\langle resume, \Pi \rangle$  to  $Op$ 
9.   Assign  $Op$  a higher priority than the current operator
10. Purge  $S_o$  and probe  $t$  against  $S_o$ , generating results
11. Compute the MNS set  $\Omega = Identify\_MNS(t)$ 
12. If  $\Omega$  is not empty, send a feedback  $\langle suspend, \Omega \rangle$  to  $Op$ 
13. Insert  $t$  into  $S_t$ 
14. If  $\Pi$  is not empty
15.   Retrieve input set  $S_\Pi$  corresponding to  $\Pi$  from  $Op$ 
16.   Join  $t$  with  $S_\Pi$ , generating results
17.   Append  $S_\Pi$  to  $S_o$ 

Handle_Feedback (Feedback  $f$ , Operator  $Op$ )     // Producer
// INPUT =  $f$ : a feedback of the form  $\langle command, \Pi \rangle$ 
//  $Op$ : the consumer that sends  $f$ 
1. Suspend current operation
2. Propagate_Feedback( $f$ )
3. If  $command$  is suspend, call Suspend_Production( $\Pi$ ,  $Op$ )
4. Else, call Resume_Production( $\Pi$ ,  $Op$ )
5. Resume the operation suspended at Line 1

```

Figure 6 General framework of JIT

Lines 10 and 13 in *Process_Input* materialize the *purge-probe-insert* processing routine for a given input t . Before that, the consumer probes t against the MNS buffer NB and sends the resumption feedback (Lines 1-9). The response of this

feedback is retrieved later (Lines 14-17), according to the asynchronous messaging protocol described above. After probing the opposite state S_o , the consumer detects MNSs of t , and sends a suspension feedback, if any MNS is found. Regarding the producer, the only change is that it now handles the pre-emptive job of responding to feedback. Specifically, it first propagates the feedback to upstream operators (Line 2), and performs appropriate operations depending on the type of the feedback (Lines 3-4). Two important aspects of JIT are left open in the above framework: (i) on the consumer's side, function *Identify_MNS* and (ii) on the producer's side, functions *Propagate_Feedback*, *Suspend_Production* and *Resume_Production*. We call them collectively as the *feedback mechanism* and discuss it in detail in the next section.

IV. IMPLEMENTATION OF THE FEEDBACK MECHANISM

Section IV-A describes MNS detection by the consumer operator. Section IV-B presents the dynamic production control, i.e., the producer's reactions to feedback.

A. MNS Detection

A suspension feedback is initiated when a consumer O_C identifies that some input tuple t does not have join partners, in which case O_C sends a message $\langle suspend, \{MNS(t)\} \rangle$ to the corresponding producer O_P of t . $MNS(t)$ is the set of minimal non-demanded sub-tuples contained in t . Any sub-tuple of t that has the potential to belong to $MNS(t)$ is called *candidate non-demanded sub-tuple (CNS)*. A CNS can only contain components that appear in the join predicate of O_C . Consider, for instance, that the consumer is the top join of Figure 1, i.e., $O_C = Op_2$ and $t = ab$ (received from $Op = Op_1$). Given the join predicate $A.y = C.y$ at Op_2 , the CNSs are a and \emptyset^2 . In the more complex scenario of Figure 5, for an input $t = abcd$ of Op_4 , there are 16 CNSs, e.g., \emptyset , a , ab , abc , $abcd$, etc., which are all combinations of components a , b , c , and d involved in the conditions of Op_4 : $(A.x = E.x) \wedge (B.y = E.y) \wedge (C.z = E.z) \wedge (D.w = E.w)$. CNSs can be organized in a lattice, where each node corresponds to a CNS and nodes are connected by the "sub-tuple" relationship. Figure 7 illustrates the lattice for input $t = abcd$ in the example of Figure 5.

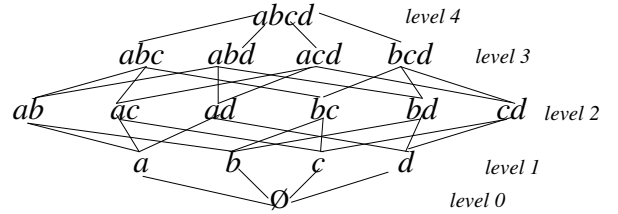


Figure 7 Example CNS lattice

Two important properties of the lattice are (i) if a CNS/node s is determined to be an MNS, then none of its ancestors can be an MNS because they are not minimal (although they are all NPRs), and (ii) given a node s above

² Recall from Section III.A that the empty tuple \emptyset is a valid MNS, when the opposite operator state of O_C is empty.

Level 1 and a tuple t' , s matches t' if and only if all its children match t' . Regarding property (i), if one of a , b , ab or ac is an MNS, abc cannot be an MNS since it contains an MNS as a sub-tuple. Similarly, for property (ii), if abc matches a tuple t' both ab and ac must match t' . Let S_o be the opposite state of t in O_C . *Identify_MNS* uses the CNS lattice to efficiently determine (i) given a CNS s and a tuple $t' \in S_o$, whether s matches t' , and (ii) given a CNS s that has no matching tuples in S_o , whether s is minimal. The basic idea of the algorithm is to match all nodes with each tuple $t' \in S_o$, and subsequently report minimal CNSs that do not have a matching partner.

Figure 8 illustrate the pseudo-code. As a special case, if S_o is empty, *Identify_MNS* reports \emptyset as the only MNS and returns immediately. Otherwise, it initializes each node of the CNS lattice L to *alive*, meaning that it has the potential to become an MNS. Then, for each tuple $t' \in S_o$, the algorithm tests every Level 1 node s against t' . If s matches t' , s is marked as *matched*; otherwise, it is set to *unmatched*. Next, it proceeds to examining nodes in increasing order of their level. Each node is marked as *matched*, if all its children are matched. After completing the traversal of L for t' , all *matched* nodes are set to *dead*, and *Identify_MNS* proceeds to the next tuple in S_o . When all tuples in S_o have been processed, the algorithm picks nodes that are both *alive* and minimal. Specifically, it starts from Level 1 and reports all *alive* nodes as MNSs. Then, it checks higher nodes level by level. For each *alive* node s , if any of its children is an MNS or non-minimal, s is marked as non-minimal; otherwise, s is reported as an MNS.

```

Identify_MNS (Record  $t$ )                                     // Consumer
// INPUT =  $t$ : an input record from producer;
1. Let  $S_i$  ( $S_o$ ) be the operator state corresponding to  $t$  (opposite of  $t$ )
2. If  $S_o$  is empty, report  $\emptyset$  as the only MNS and return
3. Let  $L$  be the CNS lattice of  $t$ 
4. Initialize all nodes in  $L$  to be alive
5. For each record  $t' \in S_o$ 
6.   For each Level 1 node  $s \in L$ , mark  $s$  as matched if it matches  $t'$ ,
   and unmatched otherwise
7.   For  $l = 2$  to top level of  $L$ 
8.     For each  $s$  on level  $l$  of  $L$ 
9.       Mark  $s$  as matched if all its children are matched, and
       unmatched otherwise
10.  Set a node to dead if marked as matched during Lines 6-9
11. Report each Level 1 node that is alive as MNS
12. For  $l = 2$  to top level of  $L$ 
13.   For each  $s$  on level  $l$  of  $L$ 
14.     Mark  $s$  as non-minimal if any of its children is MNS or non-
        minimal; otherwise, report  $s$  as MNS

```

Figure 8 Algorithm *Identify_MNS*

Note that a *matched* node may become *unmatched* during the processing of a subsequent tuple. On the other hand, once a node *dies*, it stays so for the entire execution. Consider again input $t = abcd$ of Op_4 in Figure 5 and a tuple e_1 in $S_E = S_o$ such that $a.x = e_1.x$. The processing of e_1 will set node a to *matched* (Line 6) and then *dead* (Line 10). Now assume a subsequent tuple e_2 in S_E such that $c.z = e_2.z$. During the processing of e_2 , node c becomes *matched* and *dies*. However, node ac remains

unmatched (and alive) because the status of a has switched to *unmatched* (but still dead) for e_2 .

Identify_MNS can be combined with a nested loop join of t and S_o , since both probe t against all S_o records. Furthermore, when the join condition at O_C contains equi-join predicates, its performance can be accelerated using *Bloom filters* [7] on S_o . Specifically, a Bloom filter comprises of (i) $BF[1..k]$, a k -bit string of binary values, and (ii) a set of l hash functions h_1, h_2, \dots, h_l , each of which maps all values in the domain to integers in $[1, k]$. Given a set of values V , $BF[i]$ is 1 if there exists $v \in V$ and $1 \leq j \leq l$ such that $h_j(v) = i$, and 0 otherwise. Clearly $BF[i]$ can be built with a single scan of V . Given a value v , if there exists $1 \leq j \leq l$ such that $BF[h_j(v)]$ is 0, one can be sure that v does not exist in the value set V . Continuing the example, a Bloom filter maintained on $E.y$ ($E.z$, $E.w$) is capable of detecting (some) b (c , d) sub-tuples that do not match any tuple in S_E , respectively, and thus are MNSs. This method has lower computation cost than *Identify_MNS*, but may not detect all MNSs.

B. Dynamic Production Control

In this section we focus on the handling of feedback messages by *Suspend_Production*, *Resume_Production* and *Propagate_Feedback*. In all these procedures, each MNS in the feedback is handled independently. Thus, without loss of generality, we assume that the feedback contains only one MNS. Let operators O_L and O_R supply the left and right inputs of O_P , respectively. Depending on the schema of results produced by O_L and O_R , an MNS is classified into two types: a Type I MNS is a sub-tuple of results generated exclusively by O_L or O_R , whereas a Type II MNS is a sub-tuple of the combination of results produced by O_L and O_R . In the example of Figure 5, $O_P = Op_3$, $O_L = Op_1$ and $O_R = Op_2$. Sub-tuples a , ab and c belong to Type I, while ac belongs to Type II.

We also distinguish two cases for suspension of production. The first refers to conventional *suspension* feedback, i.e., an operator completely stops producing NPRs containing an MNS. On the other hand, a *mark-result* feedback requires the producer to *mark*, rather than to suspend production of, super-tuples of the specified NPRs. A *mark-result* message is generated for type II MNSs. Consider, for instance, that Op_3 in Figure 5 wants to stop MNS $\{ac\}$, generated from inputs $O_L = Op_1$ (for a) and $O_R = Op_2$ (for c). Op_3 passes $\langle mark, \{a\} \rangle$ to Op_1 and $\langle mark, \{c\} \rangle$ to Op_2 . Op_1 (Op_2) then marks every output that is a super-tuple of a (c), respectively. At Op_3 , marked AB tuples from Op_1 containing a as a sub-tuple are not joined with marked CD tuples from Op_2 containing c , thus eliminating a^*c^* output (although permitting results such as $a^*c_1^*$ and $a_1^*c^*$).

Let s be an MNS of Type I from O_L . *Suspend_Production* scans the operator state S_L of O_L , extracting all super-tuples of s , and moves them to a *blacklist* B_L associated with S_L . If right before handling the feedback, O_P was joining a super-tuple t of s , t is also inserted to B_L . After finishing feedback handling, O_P continues to process the next input tuple t' succeeding t . In the example of Figure 1 and Table I, if Op_1 receives $\langle suspend,$

$\{a_i\}$ while joining a_i with $b_2 \in S_B$, it moves a_i to blacklist B_A , and then continues with the next incoming tuple b_4 .

The blacklist B_L is organized as follows. Each entry of B_L consists of an MNS s and a list of s 's super-tuples, each associated with a timestamp specifying when they are inserted to B_L . For a *suspension* feedback, incoming tuples from O_R are not joined with B_L , so as to prevent the generation of NPRs. For a *mark-result* feedback, however, new tuples from O_R have to join B_L , generating marked outputs. Hence, when there is hash table or index structure maintained on S_L , it is desirable to extend the structure to cover the "marked" tuples in B_L for efficient probing.

Recall from the example of Figure 1 that O_P can detect new MNSs (e.g., a_2) if they have the same join attribute values as an existing MNS s . This is realized by two additional operations. First, during the scan of the operator state (S_L) to identify super-tuples of s , O_P also finds those records in S_L that contain a sub-tuple s' with identical join attributes as s , and moves these records to B_L as well, under the entry for s . Second, when a new record t (e.g., a_2) arrives from O_L , O_P compares it with MNSs in B_L . If t contains such a sub-tuple s' , t is inserted to B_L . For a *suspension* feedback, no further processing of t is necessary, whereas for a *mark-results* feedback, t is joined with S_R , marking the results.

For a Type II MNS s , *Suspend_Production* first decomposes it into s_L and s_R . Then, O_P (i) scans both states S_L and S_R , moving super-tuples of s_L and s_R to black lists B_L and B_R , respectively, and (ii) sends *mark-results* messages to O_L and O_R . If the process is initiated by a *suspension* feedback, each marked incoming tuple only probes against S_L (or S_R), while an unmarked tuple joins both S_L/S_R and B_L/B_R . Otherwise, (the process is initiated by *mark-result*), a marked input is also probed against S_L/S_R , and the outputs are marked.

Next, we discuss *Resume_Production*, assuming that s is a Type I sub-tuple of O_L 's results. First, O_P finds the entry with MNS s in B_L . For an *unmark-results* feedback, O_P simply moves all super-tuples of s to S_L . For a *resumption* feedback, O_P joins each super-tuple t of s with tuples in S_R whose arrival timestamps are larger than the suspension time of t , and inserts t to S_L . In the example of Figure 1, suppose B_A contains a_1 and a_2 , when Op_1 receives $\langle resume, a_1 \rangle$ from Op_2 . Op_1 moves both tuples back to S_A , and joins a_1 with b_2 - b_4 and a_2 with b_1 - b_4 . Note that a_1 is not joined with b_1 because the suspension time of the former (1) is not earlier than the arrival time of the latter (1), suggesting that when a_1 is inserted into B_A , it has already been joined with b_1 . Type II MNSs are handled in a similar manner.

During feedback propagation, O_P simply relays a Type I MNS to O_L and/or O_R in its original form, e.g., if it receives $\langle suspend, \{a\} \rangle$, Op_3 passes $\langle suspend, \{a\} \rangle$ to Op_1 . For an MNS s of Type II, O_P first decomposes s into two sub-tuples s_L and s_R based on the schema of O_L and O_R 's results, e.g., sub-tuple ac is decomposed into a and c . Then, it sends s_L to O_L and s_R to O_R , using a *mark-result* feedback. Similarly for a *resumption* feedback containing a Type II MNS s , O_P passes s_L and s_R to O_L and O_R respectively, enclosed in *unmark-result* feedback, which stops the marking process. We end this

section with a note that practical implementations of the above functions have a high degree of flexibility since JIT serves as an optimization, not a core requirement, for query processing. For example, an implementation may choose not to handle Type II MNSs, or not to detect new MNSs based on known ones.

V. EXTENSIONS TO OTHER OPERATORS AND PLANS

So far we have focused on binary tree plans and the case that both the consumer and producer operators are joins. However, the applicability of JIT is not restricted to this context. We first extend JIT to operators beyond joins. When O_P is not a join operator, it may be unable to perform dynamic production control; on the other hand, if an upstream operator O' of O_P is a join, O_P can simply pass feedback from a downstream consumer O_C to O' , and the latter then adjusts its production accordingly.

A consumer O_C can be an arbitrary operator as long as it is able to detect MNSs using an algorithm similar to *Identify_MNS* (see Section IV-A). For instance, consider the plan of Figure 9a, in which $O_C = Op_2$ is a selection. For the sequence of inputs in Table I, $Op_2 = \sigma_{A.x > 200}$ detects a_1 as an MNS once it receives $a_1 b_1$ from Op_1 . It thus sends $\langle suspend, \{a_1\} \rangle$ to Op_1 , which stops joining a_1 with records in S_B . Instead of maintaining a black list, Op_1 can simply delete a_1 , as Op_2 will never issue a resumption message. In Figure 9b, consumer Op_2 joins its inputs from Op_1 with a static relation R_C , rather than another streaming source. JIT applies to this plan in a similar way to the case of Figure 9a, i.e., Op_2 may send suspension, but not resumption, feedback.

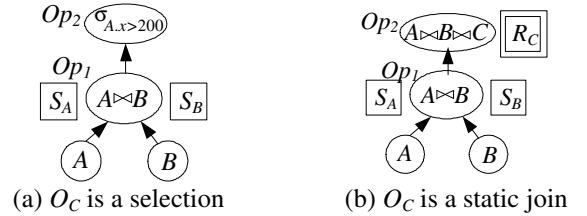


Figure 9 Alternative plans

Next we discuss JIT in plans with complex operators, specifically, *M-Join* and *Eddy*. As shown in Figure 2a, an M-Join involves multiple "half join" operators, each of which has only a single operator state (e.g., operator $\bowtie B$ has only one state S_B). Such operators are similar to the join in Figure 9b, except that the operator states (e.g., S_B) get updated as tuples arrive and expire in the corresponding stream (e.g., B). Therefore, resumption feedback is necessary as new matching partners of an MNS may appear later. The processing of this feedback, however, is simpler than the case of binary stream joins, in that the consumer does not request inputs from the producer from such messages. For example, in the leftmost operator path in Figure 2a, suppose Op_2 has issued a suspension feedback to its Op_1 specifying a_1 as an MNS. Later a matching partner c_1 of a_1 arrives and is inserted to S_C . Because c_1 is processed along a different path (Op_5 and Op_6),

Op_2 is not activated and thus does not need super-tuples of a_1 from Op_2 .

The Eddy architecture (Figure 2b) is similar to M-Joins in the sense that tuples are processed in the “half join” units (called STeMs in [4]), but now there are no fixed consumer-producer connections between them. We view every STeM as both a consumer and a producer. Once an MNS s is detected, it is sent back to the Eddy that propagates it to STeMs, whose operator state may contain s as a sub-tuple. For instance, MNS a_1 is propagated to STeM Op_1 , which then puts a_1 in a blacklist. Resumption feedback can be handled in a similar manner. Finally, the JIT logic can also be programmed into *user defined aggregates* (UDAs), which let the user organize simple operations with control flows, achieving Turing-complete expressive power [19].

VI. EXPERIMENTAL EVALUATION

We have implemented JIT in C++ following the PIPES [17] stream management framework, and performed all experiments on a Pentium 4 3.0G CPU with 2GB of RAM. For each set of experiments, we generate synthetic data for N streaming sources, and process a continuous join query over the N sources with a clique-join predicate. Specifically, there is an equi-join condition between each pair of sources; every tuple from any of the N sources contains $N-1$ columns corresponding to the other $N-1$ sources. For example, if there are 4 sources A, B, C and D , the join predicate is $(A.x_1=B.x_1) \wedge (A.x_2=C.x_2) \wedge (A.x_3=D.x_3) \wedge (B.x_4=C.x_4) \wedge (B.x_5=D.x_5) \wedge (C.x_6=D.x_6)$, where x_1-x_6 are distinct columns. Unless otherwise specified, each source has an average tuple arrival rate of λ tuples per second, and each column value is a random integer uniformly distributed in the range $[1..d_{max}]$. Intuitively, a larger d_{max} leads to a more selective join because the probability of two values to be equal is smaller. A sliding window of size w is applied to all sources.

We investigate the effect of JIT on two different categories of binary join trees: bushy and left-deep, both of which are commonly used in practice. Table II summarizes all query plans used in the experiments. A plan is executed twice, each time for 5 hours application time, with and without JIT. We refer to the execution without JIT as REF (for reference solution). The two solutions are compared in terms of total CPU time and peak memory consumption. All joins are implemented using the nested loop algorithm [16]. Table III summarizes the ranges of all parameters, with default values in bold.

TABLE II
EXECUTION PLANS

N	<i>Bushy Plan</i>	<i>Left-Deep Plan</i>
3	—	$(A \bowtie B) \bowtie C$
4	$(A \bowtie B) \bowtie (C \bowtie D)$	$((A \bowtie B) \bowtie C) \bowtie D$
5	$((A \bowtie B) \bowtie (C \bowtie D)) \bowtie E$	$((A \bowtie B) \bowtie C) \bowtie D \bowtie E$
6	$((A \bowtie B) \bowtie (C \bowtie D)) \bowtie (E \bowtie F)$	$((A \bowtie B) \bowtie C) \bowtie D \bowtie E \bowtie F$
7	$((A \bowtie B) \bowtie (C \bowtie D)) \bowtie ((E \bowtie F) \bowtie G)$	—
8	$((A \bowtie B) \bowtie (C \bowtie D)) \bowtie ((E \bowtie F) \bowtie (G \bowtie H))$	—

TABLE III
PARAMETERS UNDER INVESTIGATION

<i>Parameter</i>	<i>Range & Default</i>	
	<i>Bushy</i>	<i>Left Deep</i>
window size w (min)	10, 15, 20 , 25, 30	5, 7.5, 10 , 12.5, 15
stream rate λ (/sec)	0.4, 0.7, 1 , 1.3, 1.6	0.4, 0.7, 1 , 1.3, 1.6
#sources N	4, 5, 6 , 7, 8	3, 4 , 5, 6
max data value d_{max}	100, 150, 200 , 250, 300	30, 40, 50 , 60, 70

We first present the results for the bushy plans. Figure 10 shows the CPU time and memory consumption as a function of the window size w . In terms of CPU time, JIT outperforms REF by more than an order of magnitude (Figure 10a), while saving up to 62% of memory (Figure 10b). Note that the performance gains are amplified with increasing w . The advantage of JIT mainly comes from the reduction of FPRs (i.e., unnecessary partial results). In general, a longer window has two effects on the number of FPRs: (i) it leads to a larger number of total intermediate results, causing more FPRs; (ii) it increases the chance that an intermediate result has matching partners, reducing FPRs. The former effect prevails because it is magnified through multiple join operators, while the latter is always linear to w .

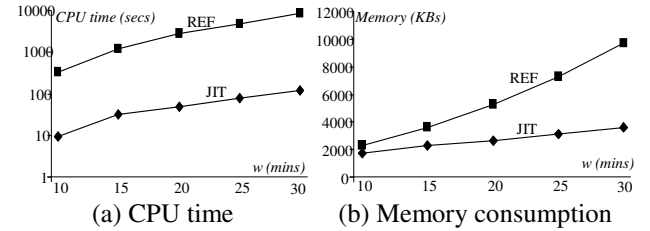


Figure 10 Overhead vs. window size w (bushy plan)

Figure 11 plots the CPU time and memory consumption against the stream rate λ . The effect of λ is similar to that of the window size w , and as λ increases, JIT yields larger savings compared to REF. Intuitively, a rapid stream rate leads to more intermediate results, many of which are not demanded by their corresponding consumers and are, therefore, eliminated by JIT.

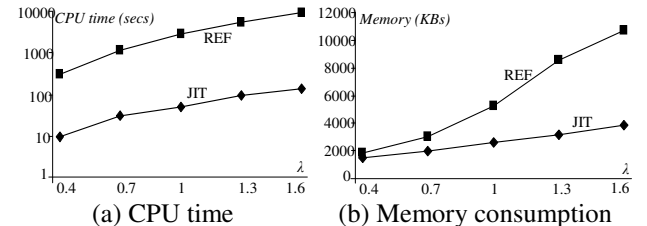


Figure 11 Overhead vs. stream rate λ (bushy plan)

Figure 12 studies the effect of the number N of streaming sources. Again, JIT outperforms REF by large margins on both metrics and its overhead increases slower with N . An interesting observation is that in terms of memory consumption, REF exhibits a step-wise pattern, i.e., the plans with 4 and 5 (also 6 and 7) inputs consume similar amounts of

memory, which reflects the nature of the bushy plans. In JIT, however, this pattern does not exist because many intermediate results are eliminated.

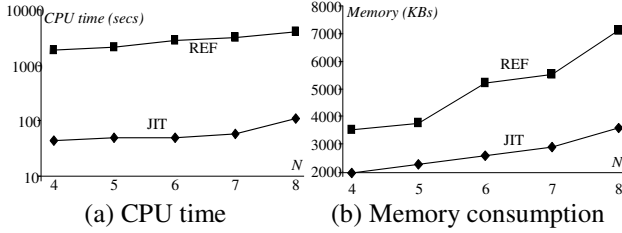


Figure 12 Overhead vs. number of sources N (bushy plan)

Figure 13 demonstrates the impact of d_{max} , i.e., the maximum column value for each tuple. Recall that a large d_{max} leads to a small output size. Consequently, fewer intermediate results are generated and the overhead of both JIT and REF decreases. Note that in JIT, when d_{max} exceeds 200, both the CPU cost and the memory consumption remain relatively stable because, after this point, very few intermediate results are produced.

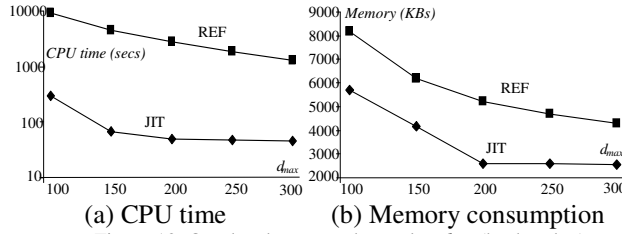


Figure 13 Overhead vs. max data value d_{max} (bushy plan)

Having established the superiority of JIT in high selectivity settings and bushy plans, we next investigate its performance for low selectivity and left-deep plans. As shown in Table III, the default d_{max} used in this set of experiments is as low as 50, compared with 200 in the previous settings. In order not to overload the system, we apply smaller w (window size) and N (number of sources), and feed stream D (C when $N = 3$) with values from $[1..10^2 d_{max}]$. Figures 14 and 15 illustrate the results for varying window size w and stream rate λ , respectively. Due to the relatively low selectivity, many intermediate results have matching partners and, thus, the effect of JIT is less pronounced. Nevertheless, JIT still has a clear advantage over REF, especially for higher values of w and λ .

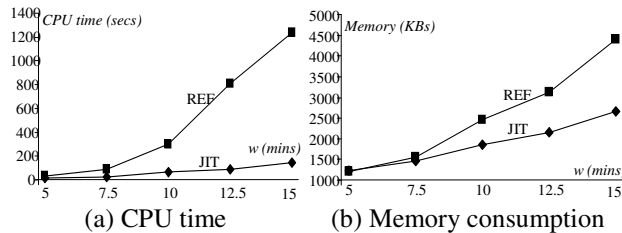


Figure 14 Overhead vs. window size w (left-deep plan)

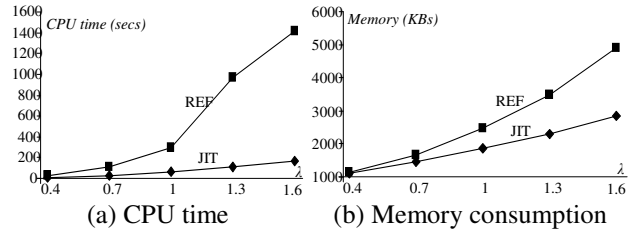


Figure 15 Overhead vs. stream rate λ (left-deep)

Figure 16 compares JIT and REF with respect to the number of streaming sources (N). JIT outperforms REF and scales more gracefully as N grows, especially in terms of CPU time. Finally, Figure 17 illustrates the effect of d_{max} . REF incurs high cost for low selectivity ($d_{max} < 50$), whereas JIT, successfully handles even very low selectivity.

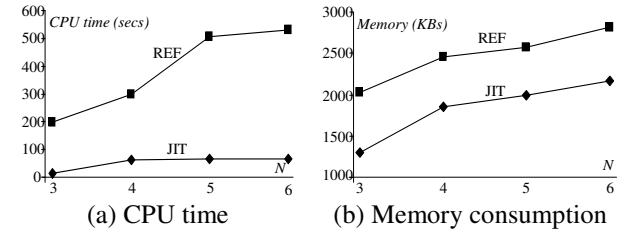


Figure 16 Overhead vs. number of sources N (left-deep)

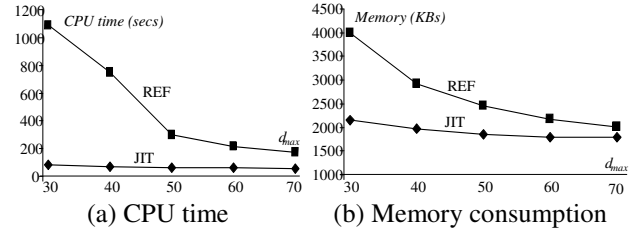


Figure 17 Overhead vs. max data value d_{max} (left-deep)

VII. CONCLUSION

This paper proposes JIT, a novel framework for continuous query execution. JIT eliminates unnecessary intermediate results through the collaboration between the consumer and producer operators, thus achieving significant savings in terms of both CPU time and memory consumption. We first present a general framework of JIT, including (i) a feedback mechanism, (ii) scheduling policies that maximize parallelism, and (iii) a message propagation scheme to amplify the effects of JIT. Then, we describe efficient implementations of key components of JIT, specifically, the feedback generation algorithm at the consumer and the dynamic production control routines performed by the producer. A comprehensive set of experiments confirm that JIT improves performance, often by orders of magnitude.

This work opens several directions for future work. So far, we have focused on the case that the exact results are required. The first interesting problem is to integrate JIT with approximate query processing methods, such as load shedding [22]. Furthermore, we intend to investigate the application of JIT in wireless sensor networks [5], where the elimination of

unnecessary partial results is critical for minimizing network transmissions and prolonging the battery life of sensors.

ACKNOWLEDGEMENTS

This work was supported by the grant HKUST 6184/05E from Hong Kong RGC.

REFERENCES

- [1] Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *Very Large Data Bases Journal (VLDBJ)*, 15(2), 2006.
- [2] Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S. B. Aurora: a New Model and Architecture for Data Stream Management. *Very Large Data Bases Journal (VLDBJ)* 12(2), 2003.
- [3] Arasu, A., Cherniack, M., Galvez, E. F., Maier, D., Maskey, A., Ryzkina, E., Stonebraker, M., Tibbetts, R. Linear Road: A Stream Data Management Benchmark. *Very Large Data Bases Conference (VLDB)*, 2004.
- [4] Avnur, R., Hellerstein, J. M. Eddies: Continuously Adaptive Query Processing. *ACM Conference on the Management of Data (SIGMOD)*, 2000.
- [5] Abadi, D., Madden, S., Lindner, W. REED: Robust, Efficient Filtering and Event Detection in Sensor Networks. *Very Large Data Bases Conference (VLDB)*, 2005.
- [6] Agarwal, P., Xie, J., Yang, J., Yu, H. Scalable Continuous Query Processing by Tracking Hotspots. *Very Large Data Bases Conference (VLDB)*, 2006.
- [7] Bloom, B. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422-426, 1970.
- [8] Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, C., Galvez, E., Salz, M., Stonebraker, M., Tatbul, N., Tibbetts, R., Zdonik, S. Retrospective on Aurora. *Very Large Data Bases Journal (VLDBJ)*, 13(4), 2004.
- [9] Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D. Operator Scheduling in Data Stream Systems. *Very Large Data Bases Journal (VLDBJ)*, 13, 2004.
- [10] Bizarro, P., Babu, S., DeWitt, D., Widom, J. Content-Based Routing: Different Plans for Different Data. *Very Large Data Bases Conference (VLDB)*, 2005.
- [11] Babu, S., Munagala, K., Widom, J., Motwani, R. Adaptive Caching for Continuous Queries. *IEEE International Conference on Data Engineering (ICDE)*, 2005.
- [12] Cormode, G., Garofalakis, M. Sketching Probabilistic Data Streams. *ACM Conference on the Management of Data (SIGMOD)*, 2007.
- [13] Gedik, B., Yu, P., Bordawekar, R. Executing Stream Joins on the Cell Processor. *Very Large Data Bases Conference (VLDB)*, 2007.
- [14] Krishnamurthy, S., Franklin, M. J., Hellerstein, J. M., Jacobson, G. The Case for Precision Sharing. *Very Large Data Bases Conference (VLDB)*, 2004.
- [15] Korn, F., Muthukrishnan, S., Wu, Y. Modeling Skew in Data Streams. *ACM Conference on the Management of Data (SIGMOD)*, 2006.
- [16] Kang, J., Naughton, J. F., Viglas, S. Evaluating Window Joins over Unbounded Streams. *IEEE International Conference on Data Engineering (ICDE)*, 2003.
- [17] Krämer, J., Seeger, S. PIPES – a Public Infrastructure for Processing and Exploring Streams. *ACM Conference on the Management of Data (SIGMOD)*, 2004.
- [18] Lim, H., Lee, J., Lee, M., Whang, K., Song, I. Continuous Query Processing in Data Streams Using Duality of Data and Queries. *ACM Conference on the Management of Data (SIGMOD)*, 2006.
- [19] Law, Y.-N., Wang, H., Zaniolo, C. Query Languages and Data Models for Database Sequences and Data Streams. *Very Large Data Bases Conference (VLDB)*, 2004.
- [20] Liu, B., Zhu, Y., Rundensteiner, E. Run-time Operator State Spilling for Memory Intensive Long-Running Queries. *ACM Conference on the Management of Data (SIGMOD)*, 2006.
- [21] Markowetz, A., Yang, Y., Papadias, D. Keyword Search on Relational Data Streams. *ACM Conference on the Management of Data (SIGMOD)*, 2007.
- [22] Tatbul, N., Zdonik, S. Window-aware Load Shedding for Aggregation Queries over Data Streams. *Very Large Data Bases Conference (VLDB)*, 2006.
- [23] Viglas, S., Naughton, J. F., Burger, J. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. *Very Large Data Bases Conference (VLDB)*, 2003.
- [24] Yang, Y., Krämer, J., Papadias, D., Seeger, B. HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 19(3), 2007.
- [25] Zhu, Y., Rundensteiner, E., Heineman, G. T. Dynamic Plan Migration for Continuous Queries Over Data Streams. *ACM Conference on the Management of Data (SIGMOD)*, 2004.