Just-In-Time Processing of Continuous Queries

Yin Yang, Dimitris Papadias

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 $S_A$ 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, $S_A$ 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.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Tuple (attribute values)</th>
<th>Partial results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$b_x(x=1), b_y(x=1)$</td>
<td>$b_x$</td>
</tr>
<tr>
<td>1</td>
<td>$a_x(x=1, y=100)$</td>
<td>$a_x, a_y, a_z$</td>
</tr>
<tr>
<td>2</td>
<td>$b_y(x=1)$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$a_x(x=1, y=100)$</td>
<td>$a_x, a_y, a_z$</td>
</tr>
</tbody>
</table>

Next, suppose that at time 2 a new record $b_y$ matching $a_y$ arrives, while there are still no join partners of $a_x$ in $S_A$. By

---

1 Some query processing algorithms (e.g., M-Join [VNB03]) do not store intermediate results. In this case the resources for producing $a_x, a_y, a_z$ are wasted regardless of whether matching C tuples of $a_x$ 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 $S_B$. 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 $S_B$ 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 ($a_2b_1$, $a_2b_2$, $a_2b_3$, $a_2b_4$ shown in Table I) are generated and 4 pointless probes against $S_B$ 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 $a1b1$ cannot generate join results due to the lack of matching tuples in $S_B$, 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 $[t.ts, t.ts+w]$. 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 $t_1$, $t_2$, ..., $t_n$ is usually assigned a timestamp of $t.ts = \max_i t_i.ts$ [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 latest 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 input 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 $S_B$, if 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 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](image)

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 processing 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 \( O_p \).

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 \( O_p \). 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 \( O_{op} \) 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](image)

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 \( O_p \) processes input \( a_1b_1 \), identifies \( a_1 \) as an MNS, and sends a feedback \( f = \langle \text{suspend}, \{a_1\}\rangle \) to producer \( O_p \). Once \( O_p \) receives such a feedback, it immediately stops generating results that are super-tuples of the specified MNSs. Continuing the example, \( O_p \) 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 \( S_t \) of super-tuples of \( s \) that have not been produced before, and returns \( S_t \) to \( O_c \). After obtaining \( S_t \) from \( O_c \), \( O_p \) joins \( t \) with \( S_t \) to generate results, and appends \( S_t \) 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 \( c_1 (c_1y = 100) \) arrives from source \( C \). \( O_p \) finds that \( c_1 \) matches MNS \( a_1 \), and sends the feedback \( f = \langle \text{resume}, \{a_1\}\rangle \) to \( O_p \). \( O_p \) joins \( a_1 \) and \( a_2 \) (whose processing is also
suspended since its y attribute is identical to \( a_1 \) with tuples in \( S_2 \), obtaining \( S_{a_1} = \{ a_1 b_2, a_1 b_3, a_1 b_4, a_1 b_5 \} \). Note that \( a_1 b_1 \) is not included in \( S_{a_1} \) because it has already been generated (before \( a_1 \) is found as an MNS). \( Op_i \) returns \( S_{a_1} \) to \( Op_i \), which joins it with \( c_i \), generating 7 results. Finally, all tuples in \( S_{a_1} \) are appended to \( S_{a_5} \).

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 NRSs 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 \( S_2 \), say \( b_1 \). 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 \( S_5 \). 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 \( Q_{C_P} \) between \( O_C \) and \( O_P \). Note that \( O_C \) cannot simply delete them from \( Q_{C_P} \) 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 \( t, t' \in S \), 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 \( f \) 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.

### 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 5b. The join predicate checked at \( Op_4 \) is illustrated in Figure 5b. For simplicity, we assume that all tuples shown in the sequence \( (a_1, c_1) \) match each other. Initially, records \( b_1 \) and \( c_1 d_1 \) are present in operator states \( S_5 \) and \( S_{CD} \), respectively. Then, tuple \( a_1 \) from source \( A \) is joined with \( b_1 \) generating \( a_1 b_1 \), which is subsequently joined with \( c_1 d_1 \), producing \( a_1 b_1 c_1 d_1 \).
tuples containing \( a_i \) or \( c_i \) (e.g., \( a_i b_i, c_i d_i \) respectively) with their corresponding partners (inputs from \( O_p \) and \( O_p' \)). Consequently, \( O_p' \)'s demand for inputs has changed; in particular, it does not want inputs that are super-tuples of \( a_i \) or \( c_i \) any longer. Hence, it propagates the feedback \(<\text{suspend, } \{a_i\}>\) to \( O_p \) and \(<\text{suspend, } \{c_i\}>\) to \( O_p' \). Similarly, when \( c_i \) (matching \( a_i \) and \( c_i \)) arrives at time \( 2 \), \( O_p \) receives the resumption feedback \(<\text{resume, } \{a_i, c_i\}>\) from \( O_p' \). It then propagates \(<\text{resume, } \{a_i\}>\) to \( O_p \) and \(<\text{resume, } \{c_i\}>\) to \( O_p' \), obtains the required inputs from them, and computes the partial results requested by \( O_p' \).

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: \( \text{Process\_Input} \) (performed by the consumer) and \( \text{Handle\_Feedback} \) (by the producer).

<table>
<thead>
<tr>
<th>\text{Process_Input (Tuple } t, \text{ Operator } O_p )</th>
<th>// Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>// INPUT = t, an input tuple</td>
<td></td>
</tr>
<tr>
<td>// ( O_p ): the producer operator that generates ( t )</td>
<td></td>
</tr>
<tr>
<td>1. Let ( S_i(S_j) ) be the operator state corresponding to ( t ) (opposite of ( t ))</td>
<td></td>
</tr>
<tr>
<td>2. Let ( NB ) be the MNS buffer opposite of ( t )</td>
<td></td>
</tr>
<tr>
<td>3. Initialize MNS set ( \Pi ) to empty</td>
<td></td>
</tr>
<tr>
<td>4. Purge ( NB ), and probe ( t ) against ( NB )</td>
<td></td>
</tr>
<tr>
<td>5. For each MNS ( s \in NB ), matching ( t )</td>
<td></td>
</tr>
<tr>
<td>6. Remove ( s ) from ( NB ), and add ( s ) to ( \Pi )</td>
<td></td>
</tr>
<tr>
<td>7. If ( \Pi ) is not empty</td>
<td></td>
</tr>
<tr>
<td>8. Send a feedback (&lt;\text{resume, } \Pi&gt;) to ( O_p )</td>
<td></td>
</tr>
<tr>
<td>9. Assign ( O_p ) a higher priority than the current operator</td>
<td></td>
</tr>
<tr>
<td>10. Purge ( S_i ), and probe ( t ) against ( S_i ), generating results</td>
<td></td>
</tr>
<tr>
<td>11. Compute the MNS set ( \Omega = \text{Identify_MNS}(\Pi) )</td>
<td></td>
</tr>
<tr>
<td>12. If ( \Omega ) is not empty, send a feedback (&lt;\text{suspend, } \Omega&gt;) to ( O_p' )</td>
<td></td>
</tr>
<tr>
<td>13. Insert ( t ) into ( S_j )</td>
<td></td>
</tr>
<tr>
<td>14. If ( \Pi ) is not empty</td>
<td></td>
</tr>
<tr>
<td>15. Retrieve input set ( S_h ) corresponding to ( \Pi ) from ( O_p' )</td>
<td></td>
</tr>
<tr>
<td>16. Join ( t ) with ( S_h ), generating results</td>
<td></td>
</tr>
<tr>
<td>17. Append ( S_h ) to ( S_j )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Handle_Feedback (Feedback} f, \text{ Operator } O_c )</th>
<th>// Producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>// INPUT = f, a feedback of the form (&lt;\text{command, } \Pi&gt;)</td>
<td></td>
</tr>
<tr>
<td>// ( O_c ): the consumer that sends ( f )</td>
<td></td>
</tr>
<tr>
<td>1. ( \text{Suspend current operation} )</td>
<td></td>
</tr>
<tr>
<td>2. ( \text{Propagate_Feedback}(f) )</td>
<td></td>
</tr>
<tr>
<td>3. If \text{command is suspend}, call ( \text{Suspend_Production}(\Pi, O_c) )</td>
<td></td>
</tr>
<tr>
<td>4. Else, call ( \text{Resume_Production}(\Pi, O_c) )</td>
<td></td>
</tr>
<tr>
<td>5. ( \text{Resume the operation suspended at Line 1} )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6 General framework of JIT

Lines 10 and 13 in \( \text{Process\_Input} \) 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 \( S_i \), 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 \( \text{Identify\_MNS} \) and (ii) on the producer's side, functions \( \text{Propagate\_Feedback} \), \( \text{Suspend\_Production} \) and \( \text{Resume\_Production} \). 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 \(<\text{suspend, } \{\text{MNS}(t)\}>\) 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., \( O_c = O_p \) and \( t = ab \) (received from \( O_b = O_p \)). Given the join predicate \( A \times B \times C \times D \), the CNSs are \( a \) and \( \emptyset \). In the more complex scenario of Figure 5, for an input \( t = abcd \) of \( O_p \), 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 \( O_p : (A = x) \times (B = y) \times (C = z) \times (D = w) \). 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](image)

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, \( ab \) 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 \( S_s \) 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 \( t \in S_s \), whether \( s \) matches \( t \), and (ii) given a CNS \( s \) that has no matching tuples in \( S_s \), whether \( s \) is minimal. The basic idea of the algorithm is to match all nodes with each tuple \( t \in S_s \), and subsequently report minimal CNSs that do not have a matching partner.

Figure 8 illustrates the pseudo-code. As a special case, if \( S_s \) is empty, Identify_MNS reports \( O \) 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 \( t \in S_s \), 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 \( S_s \). When all tuples in \( S_s \) 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.

Identifying MNS (Record t) // Consumer
If INPUT = t; an input record from producer;
1. Let \( S_r(S_L) \) be the operator state corresponding to \( t \) (opposite of \( t \));
2. If \( S_r \) is empty, report \( O \) 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 \( t \in S_s \);
6. For each Level 1 node \( s \in L \), mark \( s \) as matched if it matches \( t \), and unmatched otherwise;
7. For \( i = 2 \) to top level of \( L \);
8. For each \( s \) on level \( i \) 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 each \( s \) on level \( i \) of \( L \);
13. 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 \( t = abed \) of \( Op_3 \) in Figure 5 and a tuple \( e_2 \) in \( S_2 = S \), such that \( a.x = e_2.x \). The processing of \( e_2 \) will set node \( a \) to matched (Line 6) and then dead (Line 10). Now assume a subsequent tuple \( e_3 \) in \( S_3 \) such that \( c.z = e_2.z \). During the processing of \( e_3 \), 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 \( S_s \), since both probe \( t \) against all \( S_s \) records. Furthermore, when the join condition at \( O_C \) contains equi-join predicates, its performance can be accelerated using Bloom filters [7] on \( S_s \). Specifically, a Bloom filter comprises of (i) \( BF[1.k] \), a k-bit string of binary values, and (ii) a set \( I \) of hash functions \( h_1, h_2, ..., h_k \), 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 \( I \in j \) 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 \( I \in j \) 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 \( E_2 \) (\( E_2 \) and \( E_3 \) are capable of detecting (some) \( b, c, d \) sub-tuples that do not match any tuple in \( S_s \), 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 Suspending, 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 \( O_2 \) and \( O_3 \) supply the left and right inputs of \( O_5 \), respectively. Depending on the schema of results produced by \( O_2 \) and \( O_3 \), an MNS is classified into two types: a Type I MNS is a sub-tuple of results generated exclusively by \( O_2 \) or \( O_3 \), whereas a Type II MNS is a sub-tuple of the combination of results produced by \( O_2 \) and \( O_3 \). In the example of Figure 5, \( O_3 = Op_3, O_2 = Op_1 \) and \( O_3 = Op_2 \). Sub-tuples \( a, ab \) and \( c, d \) 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 \( O_2 = Op_1 \) (for \( a \)) and \( O_3 = Op_3 \) (for \( c \)). \( Op_1 \) passes \( <mark, \{ a \} > \) to \( Op_2 \) and \( <mark, \{ c \} > \) to \( Op_2 \). \( Op_1 \) \( (Op_2) \) then marks every output that is a super-tuple of \( a \) (\( c \)), respectively. At \( Op_2 \), 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 \( a*e \) output (although permitting results such as \( a*e \) and \( a*e \)).

Let \( s \) be an MNS of Type I from \( O_5 \). Suspending_Production scans the operator state \( S_r \) of \( O_5 \), extracting all super-tuples of \( s \), and moves them to a blacklist \( B_{i} \) associated with \( S_r \). If right before handling the feedback, \( O_5 \) was joining a super-tuple \( t \) of \( s \), \( t \) is also inserted to \( B_{i} \). After finishing feedback handling, \( O_5 \) continues to process the next input tuple \( t \) succeeding \( t \). In the example of Figure 1 and Table I, if \( Op_1 \) receives \( <suspend,
BA, BA, SL s MNS to MNS handled time of the latter that contains new tuples from O₂ are not joined with B₂ as so as to prevent the generation of NPRs. For a mark-result feedback, however, new tuples from O₂ have to join B₂, generating marked outputs. Hence, when there is hash table or index structure maintained on S₂, it is desirable to extend the structure to cover the "marked" tuples in B₂ for efficient probing.

Recall from the example of Figure 1 that O₂ can detect new MNSs (e.g., a₂) 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 (S₂) to identify super-tuples of s, O₂ also finds those records in S₂ that contain a sub-tuple s' with identical join attributes as s, and moves these records to B₂ as well, under the entry for s. Second, when a new record t (e.g., a₃) arrives from O₁, O₂ compares it with MNSs in B₂. If t contains such a sub-tuple s', t is inserted to B₂. For a suspension feedback, no further processing of t is necessary, whereas for a mark-results feedback, t is joined with S₂, marking the results.

For a Type II MNS s, Suspend_Production first decomposes it into s₂ and s₈. Then, O₂ (i) scans both states S₂ and S₈, moving super-tuples of s₂ and s₈ to black lists B₂ and B₈ respectively, and (ii) sends mark-results messages to O₃ and O₄. If the process is initiated by a suspension feedback, each marked incoming tuple only probes against S₂ (or S₈), while an unmarked tuple joins both S₂/S₈ and B₂/B₈. Otherwise, (the process is initiated by mark-result), a marked input is also probed against S₂/S₈, and the outputs are marked.

Next, we discuss Resume_Production, assuming that s is a Type I sub-tuple of O₂'s results. First, O₂ finds the entry with MNS s in B₂. For an unmark-results feedback, O₂ simply moves all super-tuples of s to S₂. For a resumption feedback, O₂ joins each super-tuple t of s with tuples in S₂ whose arrival timestamps are larger than the suspension time of t, and inserts t to S₂. In the example of Figure 1, suppose B₂ contains a₁ and a₃, when O₂ receives <resume, a₁> from O₃, O₂ moves both tuples back to S₂, and joins a₁ with b₂, b₂, and a₃ with b₁, b₄. Note that a₁ is not joined with b₁ because the suspension time of the former (1) is not earlier than the arrival time of the latter (1), suggesting that when a₁ is inserted into B₂, it has already been joined with b₁. Type II MNSs are handled in a similar manner.

During feedback propagation, O₂ simply relays a Type I MNS to O₃, and/or O₄ in its original form, e.g., if it receives <suspend, {a₁}>, O₂ passes <suspend, {a₁}> to O₄. For an MNS s of Type II, O₂ first decomposes s into two sub-tuples s₁ and s₈ based on the schema of O₁ and O₄'s results, e.g., sub-tuple ac is decomposed into a and c. Then, it sends s₁ to O₃ and s₈ to O₄, using a mark-result feedback. Similarly for a resumption feedback containing a Type II MNS s, O₂ passes s₁ and s₈ to O₃ and O₄ 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 O₃ is not a join operator, it may be unable to perform dynamic production control; on the other hand, if an upstream operator O' of O₃ is a join, O₃ can simply pass feedback from a downstream consumer O₂ to O', and the latter then adjusts its production accordingly.

A consumer O₃ 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 O₃ = O₄ is a selection. For the sequence of inputs in Table I, O₄ = O₃ A>200 detects a₁ as an MNS once it receives a₁b₁ from O₃. It then sends <suspend, {a₁}> to O₁, which stops joining a₁ with records in S₂. Instead of maintaining a black list, O₁ can simply delete a₁, as O₃ will never issue a resumption message. In Figure 9b, consumer O₃ joins its inputs from O₅ with a static relation Rₒ rather than another streaming source. JIT applies to this plan in a similar way to the case of Figure 9a, i.e., O₅ may send suspension, but not resumption, feedback.

![Figure 9 Alternative plans](image)

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 e-B has only one state S₂). Such operators are similar to the join in Figure 9b, except that the consumer states (e.g., S₂) 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 O₅ has issued a suspension feedback to its O₃ specifying a₁ as an MNS. Later a matching partner c₁ of a₁ arrives and is inserted to S₂. Because c₁ is processed along a different path (O₃ and O₆),
$O_p_2$ is not activated and thus does not need super-tuples of $a_1$ from $O_p_2$.

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 $O_p_1$, which then puts $a_1$ in a blacklist. Resolution 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 $(A.x_1=B.x_1) \land (A.x_2=C.x_2) \land (A.x_3=D.x_3) \land (B.x_2=C.x_2) \land (B.x_3=D.x_3)$, where $x_1...x_6$ are distinct columns. Unless otherwise specified, each source has an average tuple arrival rate of $\lambda$ 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

<table>
<thead>
<tr>
<th>$N$</th>
<th>Bushy Plan</th>
<th>Left-Deep Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>$(A=B) \land C$</td>
</tr>
<tr>
<td>4</td>
<td>$(A=B) \land (C=D)$</td>
<td>$(A=B) \land (C=D) \land E$</td>
</tr>
<tr>
<td>5</td>
<td>$(A=B) \land (C=D) \land E$</td>
<td>$(A=B) \land (C=D) \land E$</td>
</tr>
<tr>
<td>6</td>
<td>$(A=B) \land (C=D) \land (E=F)$</td>
<td>$(A=B) \land (C=D) \land (E=F)$</td>
</tr>
<tr>
<td>7</td>
<td>$(A=B) \land (C=D) \land (E=F) \land (G=H)$</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>$(A=B) \land (C=D) \land (E=F) \land (G=H)$</td>
<td>—</td>
</tr>
</tbody>
</table>

### TABLE III

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Bushy &amp; Default</th>
<th>Left Deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>window size $w$ (min)</td>
<td>10, 15, 20, 25, 30</td>
<td>5, 7.5, 10, 12.5, 15</td>
</tr>
<tr>
<td>stream rate $\lambda$ (sec)</td>
<td>0.4, 0.7, 1, 1.3, 1.6</td>
<td>0.4, 0.7, 1, 1.3, 1.6</td>
</tr>
<tr>
<td>#sources $N$</td>
<td>4, 5, 6, 7, 8</td>
<td>3, 4, 5, 6</td>
</tr>
<tr>
<td>max data value $d_{max}$</td>
<td>100, 150, 200, 250, 300</td>
<td>30, 40, 50, 60, 70</td>
</tr>
</tbody>
</table>

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](image)

(a) CPU time

(b) Memory consumption

Figure 11 plots the CPU time and memory consumption against the stream rate $\lambda$. The effect of $\lambda$ is similar to that of the window size $w$, and as $\lambda$ 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](image)

(a) CPU time

(b) Memory consumption

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.

![CPU time vs. number of sources N (bushy plan)](image)

**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.

![CPU time vs. max data value $d_{max}$ (bushy plan)](image)

**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 $[1, 10^4 d_{max}]$. Figures 14 and 15 illustrate the results for varying window size $w$ and stream rate $\lambda$, 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 $\lambda$.

![CPU time vs. window size $w$ (left-deep plan)](image)

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

![CPU time vs. stream rate $\lambda$ (left-deep)](image)

**Figure 15** Overhead vs. stream rate $\lambda$ (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.

![CPU time vs. max data value $d_{max}$ (left-deep)](image)

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

![CPU time vs. stream rate $\lambda$ (left-deep)](image)

**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, 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