Adaptive Optimisation For Continuous Multi-Way Joins Over RDF Streams

Danh Le-Phuoc

Technical University of Berlin

Berlin, Germany

danh.lephuoc@tu-berlin.de

ABSTRACT

The join operator is a core component of an RDF Stream Processing engine. The join operations usually dominate the processing load of a query execution plan. Due to the constantly updating nature of continuous queries, the query optimiser has to frequently change the optimal execution plan for a query. However, optimising the join executing plan for every execution step might be prohibitively expensive, hence, dynamic optimisation of continuous join operations is still a challenging problem so far. Therefore, this paper proposes the first adaptive optimisation approach towards this problem in the context of RDF Stream Processing. The approach comes with two dynamic cost-based optimisation algorithms which use a light-weight process to search for the best execution plan for every execution step. The experiments show the encouraging results towards this direction.

CCS CONCEPTS

• **Information systems** → **Join algorithms; Stream management; Resource Description Framework (RDF);**

KEYWORDS

RDF Stream Processing, Adaptive Optimisation, Continuous Multi-way joins

ACM Reference Format:

Danh Le-Phuoc. 2018. Adaptive Optimisation For Continuous Multi-Way Joins Over RDF Streams. In *WWW '18 Companion: The 2018 Web Conference Companion, April 23-27, 2018, Lyon, France*. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3184558.3191653>

1 INTRODUCTION

Most of current RDF Stream Processing (RSP) engines use the logical query operators introduced in [3] construct their logical query plans from the SPARQL-like query dialects of their choice. Subsequently, the query optimisation might be applied at the *logical level* by rewriting the plan to improve efficiency, called *algebraic optimisation*. The common rewriting rules such as reordering selection before joins and evaluating inexpensive predicates before complex ones were used in [8, 21]. Particularly for continuous queries, Arasu et al. [3] proposed rules on window-based operators such as commutative rules on time-based and count-based windows.

The logical query plan needs to be scheduled to be executed in the execution engine of a data stream management system (DSMS) with a

physical plan composed of concrete physical operators and costs. As the data arrives to the engine continuously, the DSMS scheduler can use different equivalent physical plans to execute a logical query plan during the life-time of a query. Traditional DBMSs use selectivity information and available indices to choose efficient physical plans (e.g., those which require fewer disk accesses). However, this cost metric does not apply to (possibly approximate) continuous queries, where processing cost per-unit-time is more appropriate [27]. Alternatively, if the stream arrival rates and output rates of query operators are known, then it may be possible to optimise for the highest output rate or to find a plan that takes the least time to output a given number of tuples [40, 43, 45]. Finally, quality-of-service metrics such as response time may also be used in DSMS query optimisation [1, 10, 38, 39].

Optimisation by rescheduling physical query plans are similar to those used in relational databases, e.g., re-ordering a sequence of binary joins in order to minimise a particular cost metric. There has been work in join ordering for data streams in the context of the rate-based model [45, 46]. Furthermore, adaptive re-ordering of pipelined stream filters is studied in [6], and adaptive materialisation of intermediate join results are considered in [7]. Note the prevalence of the notion of adaptivity in query rescheduling: operators may need to be re-ordered on-the-fly in response to changes in system conditions. In particular, the cost of a query plan may change for three reasons: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream [4].

Join operator is a core component of an RDF Stream Processing engine as it usually uses a SPARQL-like language fragment consisting of basic graph patterns which are commonly translated to join operations. The join operations usually dominate the processing load of a query execution plan. Due to the constantly updating nature of the continuous query, the query optimiser has to frequently change the optimal execution plan for a query. However, optimising the join executing plan for every execution step might be prohibitively expensive, hence, dynamic optimisation of continuous join operation is a still challenging problem so far even in the wider context, i.e. DSMS (cf. Section 6). Learning from our experience implementing several version of CQELS engines in [32] and [31], we propose the first adaptive optimisation approach towards this problem. This paper will presents this approach with two dynamic cost-based optimisation algorithms which use a light-weight process to search for best query plan. Our experiments in Section 5 show the encouraging results towards this direction.

The rest of the paper is outlined as following. The next section will introduce the continuous multiway join with the general cost model for dynamically finding the optimal query plan in each execution step. Section 3 will follow up with two adaptive optimisation algorithms. The experiments of such algorithms are reported in the following Section 5. Finally, we conclude the paper in Section 7.

This paper is published under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.

WWW '18 Companion, April 23-27, 2018, Lyon, France

© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC BY 4.0 License.

ACM ISBN 978-1-4503-5640-4/18/04.

<https://doi.org/10.1145/3184558.3191653>

2 CONTINUOUS MULTI-WAY JOINS OVER RDF STREAMS

2.1 Overview of Continuous Multiway Join

The basic graph patterns associated with sliding windows in RSP query languages such as C-SPARQL [9] or CQELS-QL [32] could be translated to a multiway join query. A multiway join query can be evaluated by trees of binary, partially blocking, and pipelined join operators. However, this approach is not sufficient for processing streaming inputs that need to dynamically reorganise the evaluation tree in response to the changes of stream data. Therefore, similar to MJoin [46], we introduce a single multiway join that works over more than two input buffers. This multiway join generates and propagates results in a single step without having to pass these results through a multi-stage binary execution pipeline. Because the multiway join is symmetric, without loss of generality, we extend the incremental equations of the binary join to the n-way join as shown in Equations 1 and 2 where the inserting and expiring happen in the input buffer R^1 of window W^1 . Note that, R^i is the input buffer of window W_i of each execution step for $i = 1..n$. In order to employ our ring-index for window buffers introduced in [31], we represent the incremental evaluation of these equations in Equation 3 using the select operator $\sigma_{\mu_1}(R^2)$. The operator $\sigma_{\mu_1}(R^2)$ returns all the mappings stored in the input buffer R^2 which are compatible with μ_1 . As shown in the experimental result of [31], this operator is supported by the high throughput *probe* method on the data structure that has ring-indexes on the variables to check the compatibility.

$$(R^1 \uplus \mu_1) \bowtie (R^2 \dots \bowtie R^n) = (R^1 \bowtie R^2 \dots \bowtie R^n) \cup (\mu_1 \bowtie (R^2 \dots \bowtie R^n)) \quad (1)$$

$$(R^1 - \mu_1) \bowtie (R^2 \dots \bowtie R^n) = (R^1 \bowtie R^2 \dots \bowtie R^n) \setminus (\mu_1 \bowtie (R^2 \dots \bowtie R^n)) \quad (2)$$

$$\mu_1 \bowtie (R^2 \dots \bowtie R^n) = (\{\mu_1\} \times \sigma_{\mu_1}(R^2)) \bowtie (R^3 \dots \bowtie R^n) \quad (3)$$

The evaluation of a new mapping μ_1 inserted in to the input buffer R^1 is illustrated in Figure 1.

Figure 1: Multiway join process.

When a mapping μ_1 is inserted into the input buffer R^1 , it will be used to probe one of the other input buffers $R^2 \dots R^n$. Let us assume that R^2 is the next input buffer in the probing sequence. For each mapping μ_2^i in R^2 that is compatible with μ_1 , an intermediate joined mapping in the form $\mu_1 \circ \mu_2^i$ is generated. Subsequently, $\mu_1 \circ \mu_2^i$ is recursively used to probe the other input buffers to generate the final mappings. When a buffer

that does not return any compatible mapping is found, the probing sequence stops. For dealing with the data expiration of each buffer, we use the negative tuple approach which assigns "negative signatures" to the expired tuples to forward to upper operator to invalidate expired output of the join. We call them as negative mappings respectively in this paper. Due to the page limit, we would like to refer readers to [31] for further details.

Algorithm 1 shows our incremental evaluation algorithm for a multiway join with n input buffers. Lines 2-4 handle new mappings and line 6 is for forwarding the negative mappings to the upper operator to deal with the expiration case [31]. Line 4 calls the recursive sub-routine *probingPropagate* given in Algorithm 2 to initialise the probing sequence. The probing sequence is given in sub-routine *findNextProbWin* in line 2. Each step of the probing sequence is dynamically identified by the output generated from previous steps. Therefore, the probing sequence plays an important role in the performance of this multiway join algorithm. Hence, the sub-routine *findNextProbWin* is the place holder to introduce the adaptive optimization algorithms in Section 3.

Algorithm 1: Muti-Way Join

Input: n input buffers $W_1, \dots, W_n$

```

1 if a new mapping  $\mu$  arrives at window  $W[i]$  then
2   remove expired tuples from all windows
3    $W[i].insert(\mu)$ 
4   probingPropagate( $\mu, \{W[1], \dots, W[n]\} \setminus \{W[i]\}$ )
5 else
6   propagate negative mapping to upper operator
```

Algorithm 2: Probing propagation *probingPropagate*

Input: μ, k sliding windows $\{W[i_1], \dots, W[i_k]\}$

```

1 if  $k \neq 0$  then
2    $i_{next} \leftarrow \text{findNextProbWin}(\mu, \{W[i_1], \dots, W[i_k]\})$ 
3   for  $\mu^* \in W[i_{next}].\text{probe}(\mu)$  do
4     probingPropagate( $\mu \circ \mu^*, \{W_1, \dots, W_k\} \setminus \{W[i_{next}]\}$ )
5 else
6   dispatch  $\mu$ 
```

2.2 Cost Model for Optimizing Multiway join

Data stream sources may suffer fluctuations from data arrival, most of them from unpredictable, slow or bursty network traffic [44]. On top of that, due to the streaming nature of the input, the data distribution is unpredictable, thus, making it difficult to have a model for operator selectivity [25]. To deal with changes in the selectivity of the join predicates in multiway joins, we exploit our adaptive multiway join algorithm to find the best execution plans at run-time. As shown in section, we can use the sub-routine *findNextProbWin* to dynamically chose best sub-probing sequence for each new mapping arriving at an input window. A long this line, we will define the cost model to specify the criteria for choosing the best probing sequence for each new mapping at run-time. Base on the recursive nature of our continuous multiway join, the optimal cost function can be recursively defined in Definitions 4 and 5.

$$C(\mu \bowtie R) = C(\sigma_{\mu}(R)) + C(\{\mu\} \times \sigma_{\mu}(R)) \quad (4)$$

$$\mathbb{C}(\mu_1 \bowtie \{R^2 \dots R^n\}) = \min_{i_2 \in \{2..n\}} \left(\mathbb{C}(\mu_1 \bowtie R^{i_2}) + \sum_{\mu_{i_2}^* \in \sigma_{\mu_1}(R^{i_2})} \mathbb{C}((\mu_1 \circ \mu_{i_2}^*) \bowtie (\{R^2 \dots R^n\} \setminus \{R^{i_2}\})) \right) \quad (5)$$

Definition 4 is the optimal cost for a binary join which has only one possible probing sequence. When a new mapping μ arrives at an input buffer of the binary join, the only option to compute new results is to probe the other input buffer R for the mappings compatible with μ . The cost includes $\mathbb{C}(\sigma_\mu(R))$ as the cost of probing the input buffer and $\mathbb{C}(\{\mu\} \times \sigma_\mu(R))$ as the cost of creating new mappings from μ and the output of $\sigma_\mu(R)$. Definition 5 is used to recursively choose the next input buffer for the probing sequence $\mu_1 \bowtie \{R^2 \dots R^n\}$ that has the minimal cost, provided that the optimal costs of for all possible next probing sequences $\mathbb{C}((\mu_1 \circ \mu_{i_2}^*) \bowtie (\{R^2 \dots R^n\} \setminus \{R^{i_2}\}))$ are known. However, to compute the value of this cost function, we need to compute $\mathbb{C}$ for all possible probing sequences, which is prohibitively expensive. Therefore, we aim at finding a near optimal probing sequence by approximating this cost function through two light-weight estimation methods described next.

3 ADAPTIVE OPTIMISATION ALGORITHMS

3.1 One-step Adaptation

As shown by the experimental results of the throughput of the probing operation in [30, 31], we can assume that the cost of probing an input buffer can be estimated by a constant (γ_σ), since it does not change dramatically with respect to the sizes of the window buffers. Moreover, we can also assume that the cost of generating a mapping from two mappings is constant ($\gamma_\times$). Therefore, we have the following definitions for γ_σ and $\gamma_\times$.

$$\gamma_\sigma \approx \mathbb{C}(\sigma_\mu(R)) \text{ for every } \mu \text{ and } R \quad (6)$$

$$\gamma_\times \approx \frac{\mathbb{C}(\{\mu\} \times \sigma_\mu(R))}{|\sigma_\mu(R)|} \text{ for every } \mu \text{ and } R \quad (7)$$

To adapt to the changes in the input buffers, our one-step adaption method assumes that the costs of the next steps in the possible probing sequences are approximatively the same for each output generated from the current probing step. Definition 8 approximates the cost of each next probing sequence $(\{R^2 \dots R^n\} \setminus \{R^{i_2}\})$ as a constant $\gamma_{\bowtie}$:

$$\gamma_{\bowtie} \approx \mathbb{C}((\mu_1 \circ \mu_{i_2}^*) \bowtie (\{R^2 \dots R^n\} \setminus \{R^{i_2}\})) \quad (8)$$

Hence, we have the approximate cost function for a probing sequence as shown below.

$$\mathbb{C}(\mu_1 \bowtie \{R^2 \dots R^n\}) \approx \gamma_\sigma + (\gamma_\times + \gamma_{\bowtie}) \min_{i_2 \in \{2..n\}} |\sigma_\mu(R^{i_2})| \quad (9)$$

This approximation enables us to build a simple algorithm for finding the potentially best probing sequence by checking the cardinalities of all possible probing operations of a mapping μ on k sliding windows $\{W[i_1], \dots, W[i_k]\}$. The algorithm is described in Algorithm 3. Note that this algorithm can use the counters stored in the ring index entries of the keys on the join predicates of the window buffer. Calling the probe function on a window at line 5 will return the index entries for identifying the cardinality of the probing output.

Algorithm 3: findNextProbWin_OneStep

Input: μ : mapping to probe, k sliding windows $\{W[i_1], \dots, W[i_k]\}$

```

1 minCard  $\leftarrow$  MAXINT
2 minProbeIdx  $\leftarrow$  0
3 for  $j \in [1..k]$  do
4   if  $\mu$  shares joining variable with window  $W[i_j]$  then
5     iterator  $\leftarrow$   $W[i_j].\text{probe}(\mu)$ 
6     if  $\text{iterator.size}() < \text{minCard}$  then
7       minCard  $\leftarrow$   $\text{iterator.size}()$ 
8       minProbeIdx  $\leftarrow$   $i_j$ 
9 return minProbeIdx;
```

3.2 Two-step⁺ Adaptation

The more probing sequences are possible for $\mu_1 \bowtie \{R^2 \dots R^n\}$ the harder it gets to compute the cost function given in Definition 5. We analyse the number of possible probing sequences of a join query based on its join graph. A join graph consists a set of windows as its vertices. Two vertices are connected if two windows share a join predicate (same mapping variable). Figure 2 depicts four join graphs for two 4-way joins and two 6-way joins. Note that each vertex is labeled with name of the shared mapping variable. Join graphs connected as a line as in Figure 2a and Figure 2c are called linear joins. If a mapping μ_1 comes from both ends of the linear join, there is only one possible probing sequence. For instance, the only possible probing sequence for $\mu_1 \bowtie \{R^2, R^3, R^4\}$ is $\mu_1 \bowtie R^2 \bowtie R^3 \bowtie R^4$. Therefore, it is easy to estimate $\mathbb{C}(\mu_1 \bowtie \{R^2 \dots R^n\})$ in this case. For instance, by monitoring the probing operation with the given probing sequence, its cost can be stored in an extra field of the ring index entry of the index used for the join predicate [31].

Figure 2: Join graphs.

Another interesting property of linear joins is that the inner nodes have only two connections to two separate linear subjoins. This property can be used to efficiently estimate the probing sequences in a linear join by breaking the graph into smaller linear sub-joins. The join graphs that have the star and bushy topologies like Figure 2b and Figure 2d are called star join and bushy join, respectively. The higher the connectivity of the graph, the more complicated it becomes to compute the $\min$ part in Definition 5. However, we have found a useful property for the high connectivity vertices as shown in the following equation.

$$\mu \bowtie (\{R^{i_1} \dots R^{i_m}\}) = \sigma_\mu(R^{i_1}) \times \dots \times \sigma_\mu(R^{i_m})$$

if $\text{dom}(R^{i_j}) \cap \text{dom}(R^{i_k}) \subseteq \text{dom}(\mu) \quad \forall j, k \in [1..m] \wedge j \neq k$ (10)

This property says that the Cartesian product of the all outputs of probing operations $\sigma_\mu(R^{i_j})$, using the mapping μ on the input buffers

R^{ij} , is equivalent to the join operation $\mu \bowtie (\{R^{i_1} \dots R^{i_m}\})$. The condition, called *short-cut condition*, is met when μ contains all shared variables of pairs of distinct input buffers. Based on these properties, we can provide an approximation function γ_{approx} for $\mathbb{C}((\mu_1 \circ \mu_{i_2}^*) \bowtie (\{R^2 \dots R^n\} \setminus \{R^{i_2}\}))$. An improved cost approximation function is shown in Definition 11.

$$\mathbb{C}(\mu_1 \bowtie \{R^2 \dots R^n\}) \approx \gamma_{\sigma} + \min_{i_2 \in \{2..n\}} \left(\left| \sigma_{\mu_1}(R^{i_2}) \right| \sum_{\mu_{i_2}^* \in \sigma_{\mu_1}(R^{i_2})} (\gamma_{\times} + \gamma_{approx}((\mu_1 \circ \mu_{i_2}^*) \bowtie (\{R^2 \dots R^n\} \setminus \{R^{i_2}\}))) \right) \quad (11)$$

Based on this definition, we propose the two-step⁺ algorithm to enable short-cut joins shown in Algorithm 4. Lines 2-11 check the short-cut condition. When the short-cut-condition is not met, the algorithm calls the sub-routine *findNextProbWin_MinCost* to find the next buffer to probe based on the approximate costs.

Algorithm 4: *probingPropagation_TwoStep*

Input: μ : propagating mapping, k sliding windows $\{W[i_1], \dots, W[i_k]\}$

```

1 if  $k \neq 0$  then
2   if  $\mu \bowtie \{W[i_1] \dots W[i_k]\}$  satisfies the short-cut condition then
3     for  $j \in [1..k]$  do
4        $I_j \leftarrow W[i_j].probe(\mu)$ 
5       if  $I_j.size() = 0$  then
6         return;
7     cartesianProduct( $I_1 \dots I_k$ )
8   else
9      $i_{next} \leftarrow findNextProbWin\_MinCost(\mu, \{W[i_1], \dots, W[i_k]\})$ 
10    for  $\mu^* \in W[i_{next}].probe(\mu)$  do
11      probingPropagate( $\mu \circ \mu^*, \{W_1, \dots, W_k\} \setminus \{W[i_{next}]\}$ )
12 else
13   dispatch  $\mu$ 

```

This sub-routine is described in Algorithm 5. It finds the next input buffer that potentially leads to the minimal cost value. Lines 5-19 check if all estimations γ_{approx} are available. Then, the best probing sequence will be determined in lines 17-20. If one of the estimations needed is not available, the algorithm will switch to the one-step algorithm to find the probing sequence as shown in lines 14-16. In the process, line 15 will turn on the operator monitor to measure the time spent for this probing sequence, and then assign this value to γ_{approx} for the next executions. To direct the probing sequence to the short-cut condition, we assign a zero cost for $\gamma_{approx}(\mu \bowtie (\{R^{i_1} \dots R^{i_m}\}))$ if $\mu \bowtie (\{R^{i_1} \dots R^{i_m}\})$ that meets the short-cut condition.

4 ADAPTIVE OPTIMIZATION FOR MULTIPLE MULTIWAY JOINS

4.1 Multiple-Join Operator

To share the computations and the memory when processing multiple joins that have the same set of input buffers, we modify the multiway join algorithm in Section 2.1 associated with the adaptive optimization algorithms in Section 3 to build a shared join operator, i.e., the multiple join operator. Let us assume m multiple window joins $W_j^1 \dots W_j^m$ where $j=1..k$ and W_j^i is a window buffer extracted from the RDF stream

Algorithm 5: *findNextProbWin_MinCost*

Input: μ : mapping to probe, k sliding windows $\{W[i_1], \dots, W[i_k]\}$

```

1 if  $\mu$  only shares variable with one window  $W[i_j]$  then
2   return  $i_j$ 
3 else
4   minCost  $\leftarrow$  MAXINT
5   for  $j \in [1..k]$  do
6     if  $\mu$  shares joining variable with window  $W[i_j]$  then
7        $I \leftarrow W[i_j].probe(\mu)$ 
8       if  $I.size() = 0$  then
9         return  $i_j$ 
10       $C \leftarrow 0$ 
11      for  $\mu_{i_j}^* \in I$  do
12        if
13           $\gamma_{approx}((\mu \circ \mu_{i_j}^*) \bowtie (\{W[i_1] \dots W[i_k]\} \setminus \{W[i_j]\}))$ 
14          is already estimated then
15             $C \leftarrow C + I.size() * \gamma_{approx}((\mu \circ \mu_{i_j}^*) \bowtie (\{W[i_1] \dots W[i_k]\} \setminus \{W[i_j]\}))$ 
16          else
17            turn on the operator monitor to estimate
18             $\gamma_{approx}((\mu \circ \mu_{i_j}^*) \bowtie (\{W[i_1] \dots W[i_k]\} \setminus \{W[i_j]\}))$ 
19            return
20            findNextProbWin_OneStep( $\mu, \{W[i_1], \dots, W[i_k]\}$ );
17      if minCost >  $C$  then
18        minCost  $\leftarrow C$ 
19         $i_{next} \leftarrow i_j$ 
20 return  $i_{next}$ 

```

$S^i, i=1..n$. Let W_{max}^i be the window buffer that has a window size equal to the maximum window size over all $W_j^i, j=1..k$. Then, the containment property [24] as shown in Definition 12 holds.

$$W_j^1 \dots W_j^n \subseteq W_{max}^1 \dots W_{max}^n \quad (12)$$

Due to this property, the processing of the query $W_{max}^1 \dots W_{max}^n$ produces an output that contains the outputs of all queries $W_j^1 \dots W_j^n, j=1..k$. Therefore, the join component only has to execute a single multiway query for $W_{max}^1 \dots W_{max}^n$. In the routing component, each resulting mapping has to be routed to the query that takes it as an input. We call the routing component of the multiple join operator *router*. The router maintains a sorted list of the windows relevant to each join. The windows are ordered by window sizes in increasing order. Each output mapping is traversed to its constituent leaf-mappings to find the mappings which are valid within valid time intervals of the windows. When a mapping satisfies the time condition of a query, it is routed to the query's output buffer. Figure 3 illustrates a multiple join operator for 3 queries over 2 streams S^1 and S^2 where $Q_1 = W_1^1 \bowtie W_1^2$, $Q_2 = W_2^1 \bowtie W_2^2$ and $Q_3 = W_3^1 \bowtie W_3^2$. This multiple join operator connects the 2-way join operator $W_{max}^1 \bowtie W_{max}^2$ to its router where $W_{max}^1 = W_3^1$ and $W_{max}^2 = W_3^2$. The left-hand side of the figure shows how the router delivers the output mappings from the 2-way join to each query. For instance, when the new mapping $\langle a_1, b_6 \rangle$ arrives at the stream S^1 , the 2-way join probes the input buffer W_{max}^2 to generate two output mappings $\langle a_1, b_6, c_1 \rangle$ and $\langle a_1, b_6, c_5 \rangle$. Based on the window

conditions of each query, the router routes $\langle a_1, b_6, c_5 \rangle$ to Q_1 and Q_2 and both $\langle a_1, b_6, c_1 \rangle$ and $\langle a_1, b_6, c_5 \rangle$ to Q_3 .

Figure 3: Shared Windows Example.

4.2 Networks of Multiple Join Operators for Shared Window Joins

In the general case, the concurrent queries registered to the system only share subsets of the streams involved in their queries. Therefore, we create a network of multiple join operators to enable sharing the execution of sub-queries for a group of queries that share the same set of streams. Figure 4 illustrates a network for 4 queries over 4 streams S^1, S^2, S^3 and S^4 , where $Q_1 = W_1^1 \bowtie W_1^2 \bowtie W_1^3$, $Q_2 = W_2^2 \bowtie W_2^3$, $Q_3 = W_3^3 \bowtie W_3^4$ and $Q_4 = W_4^4 \bowtie W_4^3 \bowtie W_4^1$. This network is composed of three multiple join operators $\bowtie_1^M, \bowtie_2^M$ and $\bowtie_3^M$ where $\bowtie_1^M$ is for Q_1 , $\bowtie_2^M$ is for Q_2 and $\bowtie_3^M$ for Q_3 and Q_4 .

Figure 4: A Network of Multiple Join Operators.

To reduce the amount of memory allocated for the window buffers, the network only uses one buffer for each stream. This buffer contains all window buffers on that stream. For example, the input buffer W_{max}^2 on the stream S^2 contains the windows W_1^2, W_2^2, W_3^2 and W_4^2 . Therefore, multiple join operators in the network share subsets of all input buffers created for the network. This enables an incremental evaluation for the network in the same way as for multiway joins. Algorithm 6 presents the probing propagation process in the network.

The algorithm recursively propagates the probing operations to find possible outputs in the join component, before forwarding join outputs to the routing component. When a new mapping arrives at an input window, it will trigger a probing graph that contains all probing sequences of the multiple joins that consumes that new mapping. For instance, a new mapping μ_2 in the stream S^2 will trigger the following probing sequences: $seq_1 = \mu_2 \bowtie \{W_{max}^1, W_{max}^3\}$ for $\bowtie_1^M$, $seq_2 = \mu_2 \bowtie W_{max}^3$ for $\bowtie_2^M$ and $seq_3 = \mu_2 \bowtie \{W_{max}^3, W_{max}^4\}$ for $\bowtie_3^M$. They form the probing graph depicted in Figure 5, in which they share a probing sub-sequence seq_2 . Note that the solid lines represent for probing sequences inside the multiway joins and the broken lines represent for routing paths from the multiway joins to the corresponding routers. The output mappings of this shared probing sequence can either be delivered to the router of $\bowtie_2^M$ or be used for further processing in seq_1 or seq_3 .

Algorithm 6: probingPropagationMultipleJoins

Input: μ : propagating mapping, k sliding windows

$\{W[i_1], \dots, W[i_k]\}$, m multiple join operators $\bowtie_1^M \dots \bowtie_m^M$

```

1 for  $\bowtie_i^M$  accepts  $\mu$  as output do
2   dispatch  $\mu$  to the router of  $\bowtie_i^M$ 
3 if there is a multiple join operator among  $\bowtie_1^M \dots \bowtie_m^M$  that
   consumes  $\mu$  for further processing then
4    $\bowtie_{next}^M \leftarrow$  the smallest multiple join operator among
      $\bowtie_1^M \dots \bowtie_m^M$  that consumes  $\mu$ 
5    $\mathcal{JS} \leftarrow$  set of input windows of  $\bowtie_{next}^M$  and but haven't
     involved in generating  $\mu$ 
6    $i_{next} \leftarrow \text{findNextProbeWin\_MinCost}(\mu, \mathcal{JS})$ 
7   for  $\mu^* \in W[i_{next}].\text{probe}(\mu)$  do
8     probingPropagationMultipleJoins( $\mu \circ \mu^*$ ,
        $\{W_1, \dots, W_k\} \setminus \{W[i_{next}]\}, \bowtie_1^M \dots \bowtie_m^M$ )

```

$\bowtie_2^M$ and $seq_3 = \mu_2 \bowtie \{W_{max}^3, W_{max}^4\}$ for $\bowtie_3^M$. They form the probing graph depicted in Figure 5, in which they share a probing sub-sequence seq_2 . Note that the solid lines represent for probing sequences inside the multiway joins and the broken lines represent for routing paths from the multiway joins to the corresponding routers. The output mappings of this shared probing sequence can either be delivered to the router of $\bowtie_2^M$ or be used for further processing in seq_1 or seq_3 .

Figure 5: A probing graph.

At each stage of the probing graph, the algorithm checks if the mapping used for the probing is the output of a multiple join operator. In this case, the mapping is sent to the router of this multiple join operator (lines 1-2). Line 3 is used to check if the propagated mapping can be used to generate joined results by a multiple join operator. If so, line 4 chooses the one that has the smallest set of inputs. This multiple join operator applies the two-step⁺ algorithm for choosing the best probing sequence on a single multiway join (line 6). The algorithm is only applied to the subset of the input buffers that have been processed on the sequence created by propagated mappings.

5 EXPERIMENTAL RESULTS

In this section, we present our experimental results via two types of experiments. The first type aims to show how two optimisation algorithms in Section 3 behave in a single multiway join. The second one will focus on algorithms presented in Section 4. The implementations of these algorithms are based on the latest version of CQELS engine [31] which will be open sourced at <http://cqels.org/>. The experiments are executed on a standard workstation with Debian Squeeze AMD64 2x E5606 Intel Quad- Core Xeon 2.13GHz with 16GB RAM, running the Java SE Runtime Environment (build 1.7.0_51-b13) Java HotSpot 64-Bit Server VM.

Figure 6: Join throughputs of optimisation algorithms.

5.1 Single Multiway join

In the first type of the experiments on single multiway join queries we compare them against the fixed probing sequences used in in [30, 31]. We also compare them against the ESPER engine¹ which serves as the baseline. Similar to previous experiments, we have created an ad-hoc data structure specifically for each window buffer. We have measured the throughputs for the four types of queries given in Figure 2 with different window sizes. As shown in [30, 31], the ratio of the number of distinct values in the join predicates to the window size has a significant impact in the throughput of the join operator. Hence, we fix this ratio in all test runs. To check the adaptability of the algorithms, in each test run, we stream solution mappings to the window buffers with three stages. In the first stage, we fill the window buffers with mappings to keep the selectivity on the join predicate at 100%. Then, in the next stage, we stream the same number of mappings but with a selectivity of 10% to test if the engines could adapt to the changes of the new selectivity. Finally, in the last stage, we stream the same number of mappings with the selectivity of 100%.

¹<https://github.com/espertechinc/esper>

Figure 6 reports on the experimental results. The results show that the two-step+ algorithm outperforms the other algorithms by orders of magnitude. For the 6-way join with the bushy graph, ESPER reported an out-of-heap-size error for window sizes greater than 100 mappings. In summary, the one-step algorithm performs 175%-400% better than the fixed probing sequence algorithm. The two-step+ algorithm provides in average a throughput which is twice as high as the one given by the one-step algorithm. The higher the likelihood of short-cut joins, the higher the throughput of the two-step+. In particular, the 4-way join with star graph and 6-way join with bushy graph produces more than 7 times and 10 times higher throughputs than the one-step-based version, respectively.

5.2 Multiple Multiway Joins

In the next type of experiments, we evaluate the performance of the multiple query optimisation algorithm by testing its scalability when the number of concurrent queries increases. The evaluation includes three cases: The first two cases are 2-way joins and 3-way joins that share the same 2 and 3 streams respectively. In the third case, there is

Figure 7: Join throughputs of multiple query optimisation.

a mixture of three query types, namely 2-way joins, 3-ways join, and 4-way joins whose input streams are randomly chosen from 4 streams. The window size of each input stream is chosen randomly between 100,000 and 200,000 mappings. In each case we executed a series of runs for a number of queries ranging from 2 to 1,000. For each test run, we first register its concurrent queries, then we filled up all windows. After all windows were filled, we measured the processing throughput of the engine used. The engine consumes incoming data distributed evenly among 4 streams. In these experiments, we also used ESPER with ad-hoc data structures for each stream as the baseline engine and we compared against the CQELS engine equipped with the multiple query optimiser.

Figure 7 shows the experimental results. We can see that the throughput of ESPER drops rapidly when the number of queries increases. The sudden stops in the plot line are due to out-of-heap error during the processing. In all tests, the throughput of CQELS only decreases marginally. This stems from the fact that adding more queries only adds small additional loads on the routers, while the number of streams are unchanged.

Since the processing load only depends on the maximum window sizes of all query windows on each input stream, this guarantees scalability.

6 BACKGROUND & RELATED WORK

Initial efforts on adaptive query plans include mid-query re-optimisation [26] and query scrambling, where the objective was to pre-empt operators that become blocked and to schedule other operators instead [2, 44]. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddies approach [4] performs scheduling of each tuple separately by routing it through the operators that make up the query plan. Thereby, the operators of the query plan are dynamically re-ordered to adapt to the current system conditions. This is driven by tuple routing policies that attempt to find which operators are fast and selective, such that those operators are executed first. This approach was applied to continuous queries in [13, 36] and was evaluated in [17]. The extended version for multi-way joins can be found in [37, 42]. On top of that, it was also extended to consider semantic information such as attribute correlations during routing [11].

For distributed settings, the queue length is considered as a third factor for tuple routing strategies [41].

To achieve adaptivity, the processing engine has to deal with some overheads. The first overhead is having to re-route each tuple separately. The next overhead is migrating internal states stored in some operators from the current query plan to the new query plan that has a new arrangement of operators. The issue of state migration across query plans was studied in [18, 50]. More details on adaptive query processing can be found in [5, 19, 22].

When there are multiple continuous queries registered, memory and computing resources can be shared to optimise the overall processing. For selection queries, a possible multi-query optimisation is to index the query predicates and store auxiliary information in each tuple that identify which queries it satisfies [12, 13, 29, 35, 48]. When a new tuple arrives for processing, its attribute values are extracted and matched against the query index to see which queries are satisfied by this tuple. Data and queries may be thought of as duals, in some cases reducing query processing to a multi-way join of the query predicate index and the data tables [13, 35]. Indexing range predicates is discussed in [35, 48], whereas a predicate index on multiple attributes is presented in [34, 35].

In addition, memory usage may be reduced by sharing internal data structures that store operators' states [16, 20, 49]. Moreover, in the context of complex queries containing stateful operators such as joins, computation may be shared by building a common query plan [15]. For example, queries belonging to the same group may share a plan, which produces the union of the results needed by the individual queries. A final selection is then applied to the shared result set and new answers are routed to the appropriate queries. An interesting tradeoff appears between doing similar work multiple times and doing too much unnecessary work. Techniques that balance this tradeoff are presented in [14, 28, 47]. For example, suppose that the workload includes several queries referencing a join of the same windows, but having a different selection predicate. If a shared query plan performs the join first and then routes the output to the appropriate queries, it is possible that unnecessary tuples are generated, because, some of the joined tuples may not satisfy any selection predicate. On the other hand, if each query performs its selection first and then joins the surviving tuples, then the join operator cannot be shared and the same tuples will be probed many times. Finally, sharing a single join operator among queries referencing different window sizes is discussed in [23].

Even there are several lines of work on adaptive query optimisation and optimisation of continuous query execution in general in the last 20 years, most of the proposed approaches are implemented and tested in a single system or a limited sets of systems within small scopes of certain application domains. Therefore, to the best of our knowledge, there have not been any insightful investigation of dynamic optimisation at physical execution level in the context of RDF Stream Processing, especially in multiway join query. Therefore, we made the first effort modelling the problem to design the adaptive optimisation algorithms in Section 3 test in our engine CQELS [31]. Note that, there is a derived work [33] from our continuous multiway join concept that was firstly introduced in [30]. The algorithms and cost models of this work were inspired from ours.

7 CONCLUSIONS

The paper presents our effort on addressing a very challenging problem of dynamic optimisation of join operation in continuous query over RDF Streams which we believe is a very central performance problem of RSP engines. We modelled the join operations on RDF Streams as a

continuous multiway join which help us to introduces a general recursive cost model. Via this cost model, we propose two approximation versions which help us to introduce two light-weight adaptive optimisation algorithms. Our experimental results on simulated data show very encourage performance gain in controlled settings. This opens up interesting and potential options for implementing dynamic query optimisers for RSP engines.

ACKNOWLEDGEMENT

This research has been supported by the Marie Skłodowska-Curie Programme H2020-MSCA-IF-2014 under Grant No. 661180.

REFERENCES

- [1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (Aug. 2003), 120–139.
- [2] Laurent Amsaleg, Michael J. Franklin, Anthony Tomic, and Tolga Urhan. 1996. Scrambling query plans to cope with unexpected delays. In *Proceedings of the fourth international conference on Parallel and distributed information systems (DIS '96)*. IEEE Computer Society, Washington, DC, USA, 208–219.
- [3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142.
- [4] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: continuously adaptive query processing. *SIGMOD Rec.* 29, 2 (2000), 261–272.
- [5] Shivnath Babu. 2005. Adaptive query processing in the looking glass. In *Second Biennial Conference on Innovative Data Systems Research (CIDR '05)*. 238–249.
- [6] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04)*. ACM, New York, NY, USA, 407–418.
- [7] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. 2005. Adaptive Caching for Continuous Queries. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, Washington, DC, USA, 118–129.
- [8] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. 2004. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.* 29, 3 (Sept. 2004), 545–580.
- [9] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. 2010. An execution environment for C-SPARQL queries. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*. ACM, New York, NY, USA, 441–452. <https://doi.org/10.1145/1739041.1739095>
- [10] Henrike Berthold, Sven Schmidt, Wolfgang Lehner, and Claude-Joachim Hamann. 2005. Integrated resource management for data stream systems. In *Proceedings of the 2005 ACM symposium on Applied computing (SAC '05)*. ACM, New York, NY, USA, 555–562.
- [11] Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. 2005. Content-based routing: different plans for different data. In *Proceedings of the 31st international conference on Very large data bases (VLDB '05)*. VLDB Endowment, 757–768.
- [12] Chris Barnes, J. B. Park, and Albert Vernon. 1999. Scalable Trigger Processing. In *Proceedings of the 15th International Conference on Data Engineering (ICDE '99)*. IEEE Computer Society, Washington, DC, USA, 266–.
- [13] Sirish Chandrasekaran and Michael J. Franklin. 2003. PSoup: a system for streaming queries over streaming data. *The VLDB Journal* 12, 2 (Aug. 2003), 140–156.
- [14] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. 2002. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*. Washington, DC, USA.
- [15] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: a scalable continuous query system for Internet databases. *SIGMOD Rec.* 29, 2 (2000), 379–390.
- [16] Matthew Denny and Michael J. Franklin. 2005. Predicate result range caching for continuous queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)*. ACM, New York, NY, USA, 646–657.
- [17] Amol Deshpande. 2004. An initial study of overheads of eddies. *SIGMOD Rec.* 33, 1 (March 2004), 44–49.
- [18] Amol Deshpande and Joseph M. Hellerstein. 2004. Lifting the burden of history from adaptive query processing. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30 (VLDB '04)*. VLDB Endowment, 948–959.
- [19] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. *Found. Trends databases* 1 (January 2007), 140. Issue 1.
- [20] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2004. Sketch-Based Multi-query Processing over Data Streams. In *9th International Conference on Extending Database Technology, EDBT 2004*. 551–568.
- [21] Lukasz Golab, Theodore Johnson, and Oliver Spatscheck. 2008. Prefilter: predicate pushdown at streaming speeds. In *Proceedings of the 2nd international workshop on Scalable stream processing system (SSPS '08)*. ACM, New York, NY, USA, 29–37.
- [22] Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. 2002. Adaptive Query Processing: A Survey. In *Proceedings of the 19th British National*

- Conference on Databases: Advances in Databases (BNCOD 19)*. Springer-Verlag, London, UK, UK, 11–25.
- [23] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. 2005. Optimizing in-order execution of continuous queries over streamed sensor data. In *Proceedings of the 17th international conference on Scientific and statistical database management (SSDBM'2005)*. Lawrence Berkeley Laboratory, Berkeley, CA, US, 143–146.
 - [24] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Scheduling for shared window joins over data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29 (VLDB '03)*. VLDB Endowment, 297–308.
 - [25] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. 1999. An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD '99)*. ACM, New York, NY, USA, 299–310. <https://doi.org/10.1145/304182.304209>
 - [26] Navin Kabra and David J. DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD Rec.* 27, 2 (June 1998), 106–117. <https://doi.org/10.1145/276305.276315>
 - [27] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. 2003. Evaluating Window Joins over Unbounded Streams. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*. 341–352.
 - [28] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. 2004. The case for precision sharing. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30 (VLDB '04)*. VLDB Endowment, 972–984.
 - [29] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06)*. ACM, New York, NY, USA, 623–634.
 - [30] Danh Le-Phuoc. 2013. *A Native and Adaptive Approach for Linked Stream Data Processing*. Ph.D. Dissertation. Digital Enterprise Research Institute, National University of Ireland, Galway.
 - [31] Danh Le-Phuoc. 2017. Operator-aware approach for boosting performance in RDF stream processing. *Web Semantics: Science, Services and Agents on the World Wide Web* 42 (2017), 38 – 54. <https://doi.org/10.1016/j.websem.2016.04.001>
 - [32] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. 2011. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of 10th International Semantic Web Conference*. 370–388.
 - [33] Chan Le Van, Feng Gao, and Muhammad Intizar Ali. 2017. Optimizing the Performance of Concurrent RDF Stream Processing Queries. In *The Semantic Web*, Eva Blomqvist, Diana Maynard, Aldo Gangemi, Rinke Hoekstra, Pascal Hitzler, and Olaf Hartig (Eds.). Springer International Publishing, Cham, 238–253.
 - [34] Ken C. K. Lee, Hong Va Leong, and Antonio Si. 2004. QUAY: A Data Stream Processing System Using Chunking. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS '04)*. IEEE Computer Society, Washington, DC, USA, 17–26.
 - [35] Hyo-Sang Lim, Jae-Gil Lee, Min-Jae Lee, Kyu-Young Whang, and Il-Yeol Song. 2006. Continuous query processing in data streams using duality of data and queries. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06)*. New York, NY, USA, 313–324.
 - [36] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *2002 ACM SIGMOD International Conference on Management of Data*. 49–60.
 - [37] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. 2003. Using State Modules for Adaptive Query Processing. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*. 353–364.
 - [38] Sven Schmidt, Henrike Berthold, and Wolfgang Lehner. 2004. QStream: deterministic querying of data streams. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30 (VLDB '04)*. VLDB Endowment, 1365–1368.
 - [39] Sven Schmidt, Thomas Legler, Sebastian Schär, and Wolfgang Lehner. 2005. Robust real-time query processing with QStream. In *Proceedings of the 31st international conference on Very large data bases (VLDB '05)*. VLDB Endowment, 1299–1301.
 - [40] Yufei Tao, Man Lung Yiu, Dimitris Papadias, Marios Hadjieleftheriou, and Nikos Mamoulis. 2005. RPJ: producing fast join results on streams through rate-based optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)*. ACM, New York, NY, USA, 371–382.
 - [41] Feng Tian and David J. DeWitt. 2003. Tuple routing strategies for distributed eddies. In *Proceedings of the 29th international conference on Very large data bases - Volume 29 (VLDB '03)*. VLDB Endowment, 333–344.
 - [42] Wee Hyong Tok and Stéphane Bressan. 2002. Efficient and Adaptive Processing of Multiple Continuous Queries. In *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '02)*. Springer-Verlag, London, UK, UK, 215–232.
 - [43] Tolga Urhan and Michael J. Franklin. 2001. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 501–510.
 - [44] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. 1998. Cost-based query scrambling for initial delays. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data (SIGMOD '98)*. ACM, New York, NY, USA, 130–141.
 - [45] Stratis D. Viglas and Jeffrey F. Naughton. 2002. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data (SIGMOD '02)*. ACM, New York, NY, USA, 37–48.
 - [46] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. 2003. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th international conference on Very large data bases - Volume 29 (VLDB '03)*. VLDB Endowment, 285–296.
 - [47] Song Wang, Elke Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. 2006. State-slice: new paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd international conference on Very large data bases (VLDB '06)*. VLDB Endowment, 619–630.
 - [48] Kun-Lung Wu, Shyh-Kwei Chen, and Philip S. Yu. 2004. Interval query indexing for efficient stream processing. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management (CIKM '04)*. ACM, New York, NY, USA, 88–97.
 - [49] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. 2005. Multiple aggregations over data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)*. ACM, New York, NY, USA, 299–310.
 - [50] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. 2004. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04)*. ACM, New York, NY, USA, 431–442.