WPI-CS-TR-11-01 February 2011

Achieving High Freshness and Optimal Throughput in Resource-Limited Execution of Multi-Join Continuous Queries

by

Abhishek Mukherji
Elke A. Rundensteiner

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Abstract. Due to high data volumes and unpredictable arrival rates, continuous query systems processing expensive queries in *real-time* may fail to keep up with the input data streams - resulting in buffer overflow and uncontrolled loss of data. In this work, we explore *join direction adaptation* (JDA) to tackle resource-limited processing of multi-join stream queries. While the existing JDA solutions allocate the scarce CPU resources to the most productive half-way join within a *single* operator, we instead leverage the operator *interdependencies* to optimize the overall query throughput. We identify *result staleness* as an impending issue in resource-limited processing, which gets further aggravated if throughput optimizing techniques are employed. For throughput optimization we propose the *path-productivity* model and further extend it for fulfilling the *Freshness* tolerance. Our proposed *JAQPOT* approach is the first integrated solution to achieve *near optimal* query throughput while also guaranteeing fulfillment of the desired *result freshness*. JAQPOT runs in quadratic time of the number of streams irrespective of the query plan shape. Our experimental study, using both synthetic and real data sets, demonstrates the superiority of JAQPOT in achieving higher throughput than the state-of-the-art strategies while, unlike the other methods, also fulfilling freshness predicates.

1 Introduction

Motivation. *Data Stream Management Systems* (DSMS) [1, 4, 23] are in high demand for real-time decision support as they specialize in transforming huge amounts of streaming data into *usable* knowledge. Due to rapid expansions in the diversity of data sources and the volume of data these sources deliver, DSMS are faced with the challenge of processing user queries demanding *real-time* responsiveness even under conditions of unpredictability, high data volumes and bursty workloads.

Windowed joins across streams, while among the most common queries in DSMS applications, are more costly compared to other operations such as selection, aggregation and projection [12, 13, 16]. When processing complex join queries, either the processor may fail to keep up with the arrival rates of the streams (the computing-limited case) or the available high-speed memory may become insufficient to hold all relevant tuples (the memory-limited case). For queries composed of joins with large states across multiple high-speed data streams, the system is even more prone to such resource deficiencies. Gedik et al. [12] observe that with increasing stream arrival rates and large join states, the computing resources typically become strained before the memory does. Temporary data flushing [16] and compressed data representations further counteract the chances of a memory-limited scenario. If under duress complete results can no longer be produced at run-time, then the DSMS must employ the available resources to ensure the production of maximal run-time throughput (output rate). Therefore, in this work, we aim at optimizing the throughput of *multi-join queries* in *computing*-limited environments.

```
Q1: SELECT B.symbol, B.price
FROM stocksNYC A, stocksTokyo B
WHERE A.symbol = "GOOG" and B.volume > A.volume
WINDOW 10 mins
```

When resources are limited, yet another pressing issue, namely, *result staleness* arises. In Query Q1 a stock trader is interested in *the companies whose stocks got traded at Tokyo in higher volumes than Google stocks traded in NYC*. He wants the comparable transactions to happen *within 10 minutes of each other*. For real-time decision making, the stock trader may require the DSMS to produce results continuously (say, once every minute). However, if the system faces high workloads and backlogs in processing, result tuples may get delayed. For example, the trader may receive results about transactions that took place 15 minutes before the current time. Such results, despite satisfying the 10-minute window predicate, would be considered *stale* and useless by the trader. Clearly, *high throughput results with no freshness guarantees are unacceptable in real-time applications as they may be producing results already deemed useless*.

In addition to the *WINDOW* predicate, the trader may want to specify a *FRESHNESS* predicate to indicate his *tolerance to staleness*. This predicate would assure that the returned join results consist of transaction tuples that took place within a threshold, say at most 12 minutes of the current time, in addition to being within 10 minutes of each other. Such delay would still be acceptable to the trader. Each stream may have an associated *FRESHNESS* predicate, i.e., it may be 12 mins for *stocksNYC* whereas 15 mins for *stocksTokyo*. To the best of our knowledge, our work is the first to identify the *result staleness* problem in the context of resource-limited execution of multi-join plans. Our work thus is the first to tackle the dual problems of achieving optimal throughput while guaranteeing adequate freshness of the join results.

The State-of-the-art. Two directions for tackling join queries under *computing* limitations are *load shedding* [3, 13, 25, 26] and *join direction adaptation* (JDA) [12, 15]. Tatbul et al. first applied load shedding to DSMS for query networks (mostly filter queries) [25] and for aggregation queries [26]. A detailed review of other load shedding solutions, including GrubJoin [13] that addresses MJoins, is given in Section 6. The main focus of load shedding is to reduce the input rates by directly dropping tuples from the source streams [3]. This makes the plan incapable of recuperating with the production of accurate results in moments of low workloads as data is permanently lost.

Rather than dropping tuples completely from the input streams as done in load shedding, JDA preserves in-memory tuples as per the join semantics for opportunities of joining with future incoming tuples. Existing JDA techniques [12, 15] exploit the asymmetry in the productivities of half-way join directions within a join operator. However, JDA techniques have so far been explored only in the context of a *single* join operator. We demonstrate in this work that new challenges arise in the multi-join case. A detailed review of the related work is provided in Section 6.

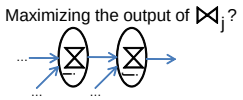


Fig. 1. A pipeline of join operators.

Research Challenges. Consider the pipeline of operators in Figure 1 where the output of $\bowtie_i$ feeds directly into $\bowtie_j$. The existing JDA technique [15] applied here will attempt to optimize operators $\bowtie_i$ and $\bowtie_j$ *individually*, and thus might fail to optimize the overall query throughput as the two operators may get optimized in an *uncoordinated* manner. In general, the ability of multi-join queries to achieve *high* result throughput (output rate) and to maintain result freshness under heavy workloads relies on resolving the following aspects of the problem:

1. While *operator scheduling* [5] tends to allocate resources at the *coarse* granularity of complete query operators, we instead propose to design algorithms to tackle the problem of join adaptation at the *finer* granularity of the *half-way join directions* within each operator to maximally leverage the asymmetry between the half-way join components.
2. The existing join direction adaptation (JDA) techniques [12, 15] optimize a single join operator *individually*. However, join operators within a multi-join plan are *interdependent*, namely, an operator depends on the output of its upstream[1] operator(s) for input. *Consideration of operator interdependency* is crucial for a successful plan-level join direction adaptation design.
3. We identify *result staleness*[2] as an impending issue for resource-limited processing of multi-join queries. The biased allocation of resources by *JDA techniques* may potentially aggravate this problem. We elaborate on this challenge in Section 3.

Proposed Approach. Unlike load shedding that discards data once the system is on the verge of crashing from overload, we propose to preemptively allocate the available computing resources with the goal to achieve maximal throughput. In short, we design, develop and evaluate a *synchronized* join adaptation strategy at the plan level that tackles the *result staleness* problem while maximizing the overall throughput (output rate) of the query.

We summarize our contributions as follows:

• We demonstrate both analytically (Section 3) and experimentally (Section 5) that for the *computing*-limited execution of *multi-join* plans the traditional *join direction adaptation* [12,15] and its extensions fail to achieve optimal throughput.

• We establish the *path productivity* metric as the plan-level throughput contribution of each input stream by leveraging operator interdependencies (Section 4.1).

• We formulate the query throughput maximization as a *knapsack* problem and propose the *Greedy Path Productivity-based Multi-Join Adaptation* (GrePP) to solve

[1] Operators closer to the stream input are *upstream* and the ones closer to the query output are *downstream*.

[2] This challenge is not identified by prior work [3, 13, 25].

Table 1. List of notation.

Symbol	Definition
t^I	Tuple of stream I
t^I.ts	Timestamp of tuple t^I
λ_i	Arrival rate of stream I
$\|S_I\|$	Window size of state I
σ_i	Selectivity of join on state S_I
λ'_i	Probe allowance of stream I, ($\leq \lambda_i$).

it.

• We identify *result staleness* as the key challenge that is further aggravated when throughput optimizing methods are employed under resource-limited execution of multi-joins. We establish the *Freshness* predicate for streams as the foundation for tackling this problem (Section 3).

• To solve the *result staleness* problem we formulate the fulfillment of *freshness* predicates as a *weighted set-cover* problem (Section 4.2).

• We then design the JAQPOT algorithm (Section 4.3) as an integration of the above two strategies. To the best of our knowledge, this is the first solution that guarantees fulfillment of result freshness predicates while achieving *near optimal* query throughput. We further note that JAQPOT achieves this effective adaptation in quadratic time in the number of input streams.

• Our experimental study (Section 5) using both synthetic and real data, demonstrates the superiority of JAQPOT over the traditional JDA solutions including multiple variants in a large set of tested cases.

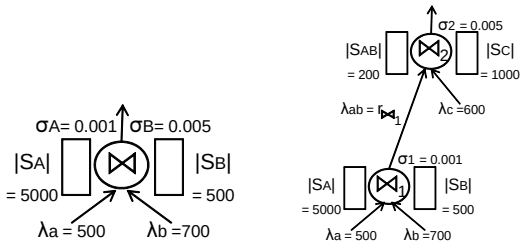
2 Background On JDA Techniques

In this paper we focus on multi-join plans composed of sliding window binary join operators. We assume standard semantics as in CQL [2]. We use the *unit-time basis cost model* proposed by Kang et al. [15] that computes the cost of a join operator in terms of the three sub-tasks, namely, *probe*, *insert* and *purge* operations. The key idea is that *the cost of probe dominates the total join cost while insert and purge operations are relatively inexpensive*. For simplicity, the model assumes *count-based* windows. The model itself is described in Appendix A together with its extension to *time-based* windows. Table 1 lists the notation.

Throughput. The run-time throughput of a join operator, also called the *output rate*, is defined as *the number of joined tuples produced per time unit*. For a join operator A⋈B (Figure 2.a), it consists of two contributing half-way join components, namely, $r_{\ltimes}$ for (a $\ltimes$ S_B) and $r_{\rtimes}$ for (b $\ltimes$ S_A), as in Formula 1. For tuple t^A, S_A is the *own* state whereas S_B is the *partner* state.

$$r_{\bowtie} = r_{\ltimes} + r_{\rtimes} = \lambda_a \times \sigma_B \times |S_B| + \lambda_b \times \sigma_A \times |S_A| \quad (1)$$

Computing Limitation in a Join Operator. During run-time the computing resources become insufficient to keep up with the *arrival rates* of the input streams due to high workloads. In such situations, like [15] we assume that the total available computing resources may be determined from the system. We use the terms *available resources* and *service rate* interchangeably. In the context



(a) Single Join (b) 2-Join Plan

Fig. 2. Join plans with example parameter settings.

of resource limited processing, the throughput of a join operator (Equation 1) can be re-written as in Equation 2.

$$\mathrm{r}_{\bowtie} = \mathrm{r}_{\ltimes} + \mathrm{r}_{\rtimes} = \lambda'_a \times \sigma_B \times |\mathrm{S}_B| + \lambda'_b \times \sigma_A \times |\mathrm{S}_A|, \quad \lambda'_a + \lambda'_b \leq \mu. \tag{2}$$

By Equation 2, we may allocate a total of μ computing resources to the processing of a join that gets divided between the two half-way joins. Stream A is assigned a *probe allowance*, denoted by λ'_a, which is a portion μ_a of the *join* service rate μ. λ'_a cannot exceed the input rate λ_a, i.e., $\lambda'_a = \min(\mu_a, \lambda_a)$. Similarly, $\lambda'_b = \min((\mu\text{-}\mu_a), \lambda_b)$. In the unit-time basis model, the *probe* cost dominates the total join cost. Therefore, the limitation on resources is restricted to *probe*. *Insert* and *purge* are relatively inexpensive and thus are assumed to be accomplished as would have been done under regular conditions. We illustrate this using A⋈B (Figure 2.a) with input rates λ_a = 500 and λ_b = 700 tuples/sec. Assume μ = 300 tuples per second as the total resources, i.e., the service rate. Therefore, a subset[3] of 300 tuples out of the 1200 (= 500 + 700) tuples from either of the input streams is used for the *probe* operation. Moreover, all 1200 arriving tuples undergo *insert* and *purge* operations every time unit, as they are inexpensive operations. Thus, the goal is to distribute the μ resources (here 300 tuples/sec) among the *probe allowances* (here λ'_a and λ'_b) such that the throughput $\mathrm{r}_{\bowtie}$ of A⋈B is maximized.

Review of Half-way Join Productivity. In Figure 2.a, the two half-way joins have different productivities (Definition 1). Using Formula 3, their productivities are: $\rho_h(\mathrm{a} \ltimes \mathrm{S}_B) = \sigma_B \times |\mathrm{S}_B| = 0.005 \times 500 = 2.5$ and $\rho_h(\mathrm{b} \ltimes \mathrm{S}_A) = \sigma_A \times |\mathrm{S}_A| = 0.001 \times 5000 = 5$ joined tuples per input tuple per second, respectively.

Definition 1. *The productivity of the half-way join $\ltimes_i \equiv (i \ltimes S_J)$, denoted by $\rho_h(i \ltimes S_J)$, is the throughput contribution ($r_{\ltimes_i}$) of $\ltimes_i$ per* ***input tuple processed by*** *$\ltimes_i$.*

$$\rho_h(\mathrm{i} \ltimes \mathrm{S}_J) = \frac{\mathrm{r}_{\ltimes_i}}{\lambda'_i} = (\sigma_J \times |\mathrm{S}_J|) \tag{3}$$

BestHJP Policy: *Allocate available resources μ starting with the most productive half-way join until μ gets exhausted.*

The *half-way join productivity* policy (henceforth called BestHJP) is used in the literature [15] for a *single* join operator. BestHJP [15] biases the join direction towards the most *productive* half-way join to maximize the join throughput $\mathrm{r}_{\bowtie}$.

[3] The fine-grained *time correlation-awareness* [13, 24] can be used for subset selection in collaboration with join direction adaptation.

As $\rho_h(a \ltimes S_B) < \rho_h(b \ltimes S_A)$, BestHJP assigns all of μ to λ'_b as the *probe allowance* and λ'_a gets none. For $\mu = 300$ tuples/sec, the throughput $r_{\bowtie}$ thus produced will be $300 \times 5 = 1500$ tuples/sec.

3 Problem Definition

Now, we define the two problems we target, namely, achieving *optimal throughput* and tackling *result staleness* in computing-limited execution of *multi-join*.

Computing-limited Execution of Multi-Join Plans. Given the 2-join plan in Figure 2.b[4] composed of $\bowtie_1$ and $\bowtie_2$, the problem of *throughput optimization* becomes quite different from the single operator scenario, namely, the goal is to achieve overall high *query throughput*. In other words, the throughput $r_{\bowtie_{root}}$ of the root operator (here $\bowtie_2$) must be maximized.

$$\begin{aligned} r_{\bowtie_1} &= \lambda'_a \times \sigma_B \times |S_B| + \lambda'_b \times \sigma_A \times |S_A|, \\ r_{\bowtie_2} &= \lambda'_{ab} \times \sigma_C \times |S_C| + \lambda'_c \times \sigma_{AB} \times |S_{AB}|, \\ & \lambda'_a + \lambda'_b + \lambda'_{ab} + \lambda'_c \leq \mu. \end{aligned} \tag{4}$$

Equation 4 depicts the problem of join direction adaptation in a *multi-join* plan. For this 2-join plan, μ needs to be divided among four half-way joins, namely, λ'_a, λ'_b, λ'_{ab} and λ'_c. In other words, μ needs to be split into two levels, namely, first dividing μ among the n join operators (say $\mu_{\bowtie_1}, \mu_{\bowtie_2}, \ldots, \mu_{\bowtie_j}, \ldots, \mu_{\bowtie_n}$) and then, for each join $\bowtie_j$, further dividing $\mu_{\bowtie_j}$ between each of its respective half-way joins $\mu_{\ltimes_j}$ and $\mu_{\rtimes_j}$.

For a single join operator, BestHJP [15] guarantees optimal *operator* throughput. However, for the 2-Join plan in Figure 2.b, BestHJP would individually optimize $\bowtie_1$ and $\bowtie_2$ for the throughput and thus may not achieve an overall high *query throughput* $r_{\bowtie_{root}}$. Beyond the immediate interpretation of applying BestHJP on a multi-join plan, we explore other variants of BestHJP. We further set out to design four ρ_h-based heuristics whose description is put later in Appendix B to now focus on our proposed approach. The failure of these policies then motivates us to explore operator *interdependencies* for solving the identified problems as described next in Section 4. In our experiments (Section 5) we do compare these heuristics against our proposed approach.

Freshness of Multi-Join Results. As described for Query Q1 (Section 1), when the resources become limited, the produced query results may no longer be perfectly *fresh*. The result freshness gets further compromised when JDA techniques bias resource allocation to highly productive half-way joins for achieving optimal throughput. Consequently, little or no resources are left for the less productive components of the plan. Therefore, under a *throughput optimizing* scheme, insufficient scheduling of certain operators may lead to their *starvation*. Moreover, when a *starved* upstream operator does not produce sufficient intermediate results, the dependent join state in the downstream operator tends to

[4] For simplicity σ_i denotes overall selectivity of $\bowtie_i$; in reality each half-way join has an associated selectivity as in Figure 2.a.

become *stale*. The join results produced using such stale states are also stale, thus further deteriorating the *result freshness*. In Figure 2.b, if ($c \ltimes S_{AB}$) is most productive, the assignment of complete μ to λ'_c would *starve* $\bowtie_1$, leading to the *staleness* of the state S_{AB} and eventually also to that of the final query results.

Definition 2. *The **Freshness** predicate, namely, $\oint^I$ for a stream I, requires that all the joined output produced at any time T must not contain the tuples t^I with arrival time older than (T- $\oint^I$).*

Under resource-limited execution, when 100% *freshness* is not achievable, the user can supply a *Freshness* predicate $\oint^I$ with respect to each stream I (Definition 2). The type of the *Freshness* predicate will be the same as that of the window predicate, i.e., time or count-based and the value would be equal or greater than the window value. Say for WINDOW = 10 minutes, the user may desire a maximum $\oint^I$ of, say, 12 mins indicating that a 2 minute delay is tolerable. By default the Freshness predicate $\oint^I$ may be equal to the WINDOW predicate when the users require the results to be 100 % fresh. Query results not fulfilling the *Freshness* predicate are considered stale and thus useless. *The problems of optimizing throughput and maintaining freshness are orthogonal. In this work we focus on achieving maximal throughput while fulfilling the user-defined Freshness specification*.

4 The Proposed JAQPOT Approach

Below, we first present our solution for the *query throughput optimization* problem (Section 4.1), while our approach for satisfying the *result freshness* is described in Section 4.2. Lastly, the integrated JAQPOT algorithm combining the two solutions is given in Section 4.3.

4.1 Optimizing Throughput in Multi-Join Queries

Our analysis of throughput optimization in *multi-join* plans indicates the need for a *producer-consumer match* between successive joins. Scarce resources are best utilized if any intermediate tuples produced by a *producer* join also get fully consumed by the downstream *consumer* join for probing the partner join state. Clearly, this match must be achieved between every pair of join operators in the plan for the effective utilization of the available μ resources. Thus, we propose a synchronized resource allocation strategy at the query plan level that establishes *producer-consumer matches* within the plan.

Input Paths. We introduce the notion of *input paths* in Definition 3.

Definition 3. *Given a multi-join plan Q with k input streams[5] (I = 1, . . . , k); each pipeline of half-way joins from the leaf to the root operator forms an **input path** denoted by $Path^I$. A path having **n** join operators between input stream I and the output of query is called an **n-hop path**.*

[5] If stream I is used multiple times as input to the plan (self-joins), then separate copies of I will used as separate inputs.

In our example plan (Figure 2.b), we identify three input paths, namely, Path^A, Path^B and Path^C. Path^A, a 2-hop path, is composed of two sequential half-way joins, namely, (a $\ltimes$ S_B) followed by (ab $\ltimes$ S_C), also written as (a $\ltimes$ S_B $\ltimes$ S_C).

Along an *n-hop* Path^I, every input tuple joins at the leaf half-way join and propagates the intermediate results to the second half-way join and so on upto the root. The half-way join productivity (ρ_h) of the *leaf* half-way join represents the number of intermediate results produced per input tuple processed by it. Similarly, the *cardinality of intermediate joined tuples* produced by the jth half-way join along Path^I may be computed by multiplying the productivities (ρ_h) of half-way joins along Path^I starting with the leaf up to the jth half-way join. In Equation 5, we give the formula for this *cardinality of intermediate joined tuples*, denoted as ϕ^I_j. The superscript p denotes the *partner* join state at each level. ϕ^I_j forms an important component of the core formulae that we are about to define next.

$$\phi^I_j = \prod^{j}_{g->1}(\sigma^p_g \times |S^p_g|). \tag{5}$$

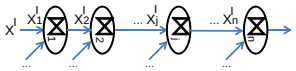


Fig. 3. Division of X^I resources within Path^I.

$$\begin{aligned} X^I &= X^I_1 + X^I_2 + \ldots + X^I_j + \ldots + X^I_n \\ &= X^I_1 \times [1 + \phi^I_1 + \ldots + \phi^I_{j-1} + \ldots + \phi^I_{n-1}] \end{aligned} \tag{6}$$

Division of Resources within an Input Path. Assume that X^I of the total μ resources are allocated to an *n-hop* path Path^I (Figure 3). X^I is further divided among the half-way joins of Path^I as *probe allowances* $X^I_1, X^I_2, \ldots, X^I_j, \ldots, X^I_n$, such that they all add up to X^I. The key for achieving the most effective division of X^I is *to establish a* ***producer-consumer match*** *between every pair of these successive half-way joins*. This ensures that no resources are wasted on the production of intermediate join tuples that ultimately do not contribute to the path's benefit. Thus, for a 2-join plan (Figure 2.b), the output $r_{\bowtie_1}$ (Formula 4) of $\bowtie_1$ must be equal to the *probe allowance* λ'_{ab} of $\bowtie_2$. In Formula 6, we show that each such *probe allowance* assignment to the input of the jth half-way join along Path^I, denoted as X^I_j, may be computed by multiplying the leaf component X^I_1 with ϕ^I_{j-1}. Here, ϕ^I_{j-1} denotes the *cardinality of the (j-1)th intermediate joined tuples*. Further, in Formula 7, we substitute the value of X^I_1 from Equation 6 to derive the value of X^I_j as a fraction of the total X^I resources. As X^I_j is the input of the jth half-way join, thus the cardinality value ϕ^I_{j-1} of the (j-1)th half-way join is used.

$$\begin{aligned} X^I_j &= X^I_1 \times \phi^I_{j-1} \\ &= \left(\frac{X^I \times \phi^I_{j-1}}{1 + \phi^I_1 + \ldots + \phi^I_{n-1}}\right). \end{aligned} \tag{7}$$

Table 2. Path productivity table.

$\textbf{Path}^I$	$\textbf{Path}^A$	$\textbf{Path}^B$	$\textbf{Path}^C$
Resources used (X^I)	15	16	10
Output rate $r^I_{\bowtie_{root}}$	4	5	1
Path Productivity $\rho_p(\text{Path}^I)$	0.266	0.312	0.1

Path Productivity. Similar to the half-way join productivity metric that estimates the contribution of a half-way join to the operator output, we now establish a *novel* metric that measures the contribution of an input path to the overall query throughput (Definition 4).

Definition 4. *The* ***path productivity*** *of* $Path^I$*, denoted by* ρ_p***(Path**^{I}**)***, is its contribution to the query throughput* ***per tuple t processed***[6] ***within Path**^{I}*.

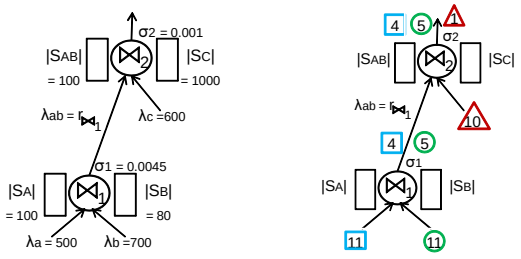
$$\begin{aligned}\rho_p(\text{Path}^I) &= (\sigma_n^p \times |S_n^p|) \times \left(\frac{\phi_{n-1}^I}{1+\phi_1^I+\ldots+\phi_{n-1}^I}\right) \\ &= \left(\frac{\phi_n^I}{1+\phi_1^I+\ldots+\phi_{n-1}^I}\right).\end{aligned} \tag{8}$$

If total X^I resources are assigned to Path^I, the component X_n^I assigned to the root half-way join may be computed using Equation 6. The throughput contribution by processing the X^I resources along Path^I, denoted as $(r^I_{\bowtie_{root}}) = (\sigma_n^p \times |S_n^p|) \times X_n^I$. Therefore, by Definition 4, the productivity of Path^I can be computed as $\rho_p(\text{Path}^I) = \frac{r^I_{\bowtie_{root}}}{X^I}$. Equation 8 gives the expression for the productivity of Path^I. The formula is further simplified to an expression entirely composed of the *cardinalities of intermediate joined tuples*.

Applying Equation 8 over the paths in our example plan (Figure 2.b), the productivity of Path^A, denoted as $\rho_p(\text{Path}^A)$ can be computed as $(\sigma_C \times |S_C|) \times (\frac{(\sigma_B \times |S_B|)}{1+(\sigma_B \times |S_B|)})$. Similarly, if X^B resources are allocated to Path^B, X^B will be divided among λ'_b and λ'_{ab}. Applying Formula 7 to this plan, an effective division of X^B over Path^B will be $\lambda'_b = (X^B/6)$ and $\lambda'_{ab} = (5 \times X^B/6)$. The total throughput contribution ($r^B_{\bowtie_{root}}$) achieved by this assignment on Path^B can be estimated as $(5 \times X^B/6) \times (\sigma_C \times |S_C|)$. Thus, for $X^B = 600$ tuples/sec, 100 tuples/sec are assigned to λ'_b, then producing $r_{\bowtie_1} = 500$ tuples/sec as intermediate output. The remaining 500 tuples/sec (= X^B - λ'_b) are assigned to λ'_{ab}, such that $\lambda'_{ab} = r_{\bowtie_1}$ and *producer-consumer match* is achieved between $\bowtie_1$ and $\bowtie_2$.

Discussion. The path productivity metric defaults to the notion of half-way join productivity ρ_h (Section 2) when applied to a single operator. ρ_h is a *local* operator level metric whereas our proposed ρ_p metric establishes the contribution of a complete input path to the *query* throughput $r_{\bowtie_{root}}$. The former takes only the tuples directly input into the half-way join into consideration, whereas the ρ_p metric instead considers all the tuples processed anywhere along the path, be it at the leaf, the intermediate and the root operators.

[6] Tuple t here refers to a leaf input tuple or an intermediate tuple along Path^I.



(a) Parameter assignment (b) Path-wise throughput

Fig. 4. Example query plan.

Path Productivity-based Join Adaptation. Given plan Q with k input paths, namely, $Path^A$, $Path^B$, . . . , and $Path^k$, their path productivities can be computed using Formula 8. The 2-join plan in Figure 4 may be translated into a *path productivity table* (Table 2). This translation is based on the *input rates*, the *selectivities* and the *state sizes* along each path within the plan. For each input path $Path^I$, the *path productivity table* lists (a.) *the resources used* (X^I), (b.) *the query output rate* ($r^I_{\bowtie_{root}}$) achieved using X^I resources, and (c.) *the path productivity* ($\rho_p(Path^I)$).

In Figure 4.b) for $Path^A$, the resources X^A = 15 tuples/sec may be divided across the two half-way joins such that λ'_a gets 11 and λ'_{ab} gets 4. The throughput contribution of $Path^A$, denoted by $r^A_{\bowtie_{root}}$, is 4 tuples/sec as ($ab \ltimes S_C$) produces 1 joined tuple t^{ABC} per t^{AB} input tuple. Thus, for every 15 tuples consumed by $Path^A$ in a second, it will produce 4 t^{ABC} joined tuples. The values in Table 2 may be *fractional*. Once a multi-join plan has been translated into a productivity table, our join adaptation problem can be formulated as a variant of the knapsack problem [21], as described below.

Problem 1. ***Join adaptation as a knapsack problem:*** *Given a path productivity table representation of a plan Q having k paths $Path^1$, $Path^2$,. . . , $Path^k$, when $Path^I$ is assigned resources in multiple M_I of its resources X^I, then its throughput can be computed as $r^I_{\ltimes_{root}} = M_I \times \rho_p(Path^I)$. By defining a_I to be 1 if $Path^I$ is chosen in a solution and 0 otherwise, we can formulate this* ***JDA-Knapsack*** *problem as:*

$$\text{Maximize } \Sigma_{I=1}^{k}\ a_I \times r^I_{\bowtie_{root}} \tag{9}$$

subject to:

$$\Sigma_{I=1}^{k}\ a_I \times M_I \times X^I \leq \mu. \tag{10}$$

JAQPOT Policy: *Allocate μ iteratively to the most productive path remaining until μ is completely consumed.*

Assume μ = 30 tuples/sec. Using the JAQPOT Policy for the productivity table listed in Table 2, the most productive path $Path^B$ will be assigned 16 tuples (out of 30). The remaining 14 resources fall short of X^A=15, the minimum resources required by the second most productive path $Path^A$. Thus, $Path^C$ will be chosen and assigned 10 resources, wasting the remaining 4 tuples. The total output rate achieved using this greedy assignment is 6 tuples/cycle (assume each cycle runs for 1 second). A more effective assignment would be to instead give the complete 30 (=15 × 2) tuples/cycle to $Path^A$ and achieve 8 (=4 × 2)

tuples/cycle as throughput. *This illustrates that a **greedy** application of the JAQPOT Policy fails to achieve optimal throughput*.

Above, we find ourselves working under rigid constraints. First, each execution cycle runs *independently* of its predecessor and successor execution cycles. Second, we assume a *discrete execution model* where the X^I, $r^I_{\bowtie_{root}}$ and μ values are assumed to be *whole numbers*. Under this model the *throughput optimization* problem does not exhibit the *greedy choice property* (Refer to [21] for details). Thus, a *dynamic programming* knapsack solver must be employed to achieve an assignment yielding optimal throughput. Furthermore, the DP knapsack solver runs in $\bigcirc(k \times \mu)$, for k input streams and μ available computing resources. For higher values of μ, this solver would be extremely compute-intensive. Therefore, we now explore alternate greedy strategies for solving this problem.

The Greedy Knapsack Solver. Let us now relax the above restrictions. First, instead of *independent execution cycles*, each being assigned *distinct* μ available resources, we now consider the *coordinated execution across successive cycles*. For example, two successive execution cycles producing 3 and 2 join tuples respectively will result in the overall path productivity $\rho_p(\text{Path}^I)$ to be 2.5 tuples/cycle. As we will see shortly, this achieves even higher output rates than produced under the *discrete* execution model. Once such a group of successive execution cycles is identified, we can view their combination as a *mega* execution cycle. Secondly, X^I, $r^I_{\bowtie_{root}}$ and μ values can now be *fractional*. Thus, for Path^B (Figure 4), $X^B = 16$ tuples/sec and $r^B_{\bowtie_{root}} = 5$ tuples/sec may be re-phrased as Path^B using $X^B = 8$ tuples/sec to produce $r^B_{\bowtie_{root}} = 2.5$ tuples/sec. While *fractional* tuples cannot be consumed (or produced) in an individual execution cycle, over the span of multiple successive execution cycles a virtual consumption (or production) of *fractional* tuples per cycle may arise.

These relaxations are *mutually complementary* and their benefit is twofold. First, multiple cycles may be scheduled together. Second, as fractional resource assignment is allowed, high productivity paths consuming resources X^I greater than μ, that would otherwise be eliminated in the *discrete model*, may now be assigned resources. In a real-world resource-limited scenario, the resources are more likely to be an estimated μ value available over a duration spanning multiple cycles rather than being a *distinct* μ value available to each *independent* cycle. Under this *continuous execution model*, our *JDA-Knapsack* problem (Formulation 1) now exhibits both the *greedy choice property* and the *optimal substructure property* [21]. In other words, an optimal solution is guaranteed to contain the most productive path. This implies that we can now use a *greedy knapsack solver*, henceforth referred to as ***Gre**edy **P**ath **P**roductivity-based* Multi-Join Adaptation (**GrePP**), to tackle our problem.

For our running example 4, GrePP selects the most productive path, Path^B (Table 2), and allocates all of μ (=30 tuples/sec) such that λ'_b gets 20.62 tuples/sec and λ'_{ab} gets 9.38 tuples/sec. The estimated query throughput $r^{GrePP}_{\bowtie_{root}}$ is 9.38 tuples/sec that exceeds the throughput of the *discrete model* and is guaranteed to be *optimal* [21]. **GrePP runs in $\bigcirc(k \log(k))$ time [21] for a plan joining k streams and thus is independent of μ.**

4.2 Satisfying Freshness in Multi-Join Queries

Although GrePP is guaranteed to produce *optimal* throughput, such an adaptation may suffer from the problem of *result staleness*. Recall that *Freshness* predicates (Definition 2) are supplied by the user with respect to each input stream. Here we now show how our path-productivity based model is extended to incorporate this notion of *Freshness*.

The key idea here is that the *Freshness* predicates defined over streams are fulfilled by translating them into *refresh rates* for the join states in the query plan. To enforce the *Freshness* predicate on stream I (Definition 2), every tuple t^I from stream I and all its intermediate joined tuples must be purged from the plan within $\oint^I$ time (or tuple for count-based freshness) from its arrival time t^I.ts. This would ensure that no results having t^I will be produced beyond $\oint^I$ from its arrival. In Figure 5.a, stream C contributes the singleton t^C, the intermediate t^{CD} and t^{CDE} tuples, get stored in *own* states S_C, S_{CD} and S_{CDE}, respectively. State S_C gets refreshed at λ_c tuples/sec, which tends to be high at high arrival rates. However, the rates at which tuples t^{CD} and t^{CDE} get inserted into intermediate states S_{CD} and S_{CDE} depend on the portion of μ allocated to λ'_{cd} and λ'_{cde}, respectively. Intermediate states, such as S_{CD} and S_{CDE}, that depend on upstream operators for input are called *staleness susceptible states* (highlighted in Figure 5.a). $\frac{S_C}{\lambda_c}$, $\frac{S_{CD}}{\lambda'_{cd}}$ and $\frac{S_{CDE}}{\lambda'_{cde}}$ denote the time duration for which a singleton t^C and its corresponding t^{CD} and t^{CDE} tuples will remain in their respective *own* states S_C, S_{CD}, S_{CDE}, respectively. The Freshness predicate $\oint^C$ for stream C gets fulfilled only if $\oint^C \geq (\frac{S_C}{\lambda_c} + \frac{S_{CD}}{\lambda'_{cd}} + \frac{S_{CDE}}{\lambda'_{cde}})$.

Lemma 1. *To fulfill the Freshness predicate $\oint^I$ for stream I, $\oint^I \geq \Sigma_{j=1}^{n} \frac{S_j^o}{\lambda'_j}$, where λ'_j and S_j^o denote the* ***probe allowance*** *and the* ***own*** *join state, respectively, at j^{th} operator along $Path^I$, storing intermediate tuples having t^I tuples.*

In Lemma 1 each $\frac{S_j^o}{\lambda'_j}$ value corresponds to the average time each singleton t^I tuple and its corresponding intermediate joined tuples reside in their respective *own* state S_j^o. Therefore, the *Freshness* predicate $\oint^I$ on any stream I can only be fulfilled by allocating sufficient resources λ'_j to each operator j along $Path^I$ so that its *own* join states get refreshed at sufficient rates. Using our model of input paths within a multi-join plan, we gain further insights into the *result staleness* problem. First, we observe that each input path contains one or more of these *staleness susceptible states*. Moreover, besides ($c \ltimes S_D$) the state S_{CD} is populated by ($d \ltimes S_C$) too. Similarly, in addition to ($cd \ltimes S_E$), the state S_{CDE} is also populated by ($e \ltimes S_{CD}$). Thus, *staleness susceptible states* may be refreshed *synchronously* by allocating resources to the paths covering those states. The problem of satisfying the *Freshness* predicates can be tackled by translating them to corresponding *refresh rates* (Definition 5) for each *staleness susceptible*

state. The user-defined *Freshness* predicate is fulfilled only if the input rate λ'_j (λ_j for leaf) at each half-way join exceeds the desired R_{S_j}.

Definition 5. *For a state S_j, the **refresh rate** R_{S_j} denotes the minimum number of new tuples required to be inserted per time unit into state S_j to prevent it from becoming stale. A **staleness susceptible state** S_j is said to be covered if its refresh rate R_{S_j} is fulfilled.*

Given the foundation, we are now able to translate the problem of coverage of the *staleness susceptible states* into a *weighted multiple set cover problem* (WMSCP) [29], as described in Formulation 2. Given the set of all *staleness susceptible states* and the input paths that include those states, the goal is to identify the set of paths, called the *minimal coverage paths* that cover all the *staleness susceptible states* utilizing the *minimum* computing resources $\Delta\mu$ out of the total μ resources. The remaining (μ - $\Delta\mu$) resources can then be allocated to GrePP for throughput optimization. In Figure 5.a, Path^A and Path^C are such *minimal coverage paths* covering all staleness susceptible states in joins $\bowtie_3$ and $\bowtie_4$.

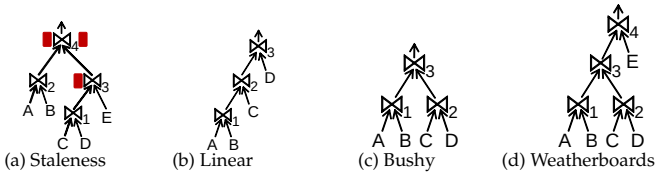


Fig. 5. Multi-join plans used.

Problem 2. ***Coverage of staleness susceptible states as a weighted multiple set cover problem (WMSCP):*** *The Universe U consists of m staleness susceptible states = $S_1, S_2, \ldots, S_m$ with required refresh rates = $R_{S_1}, R_{S_2}, \ldots, R_{S_m}$, respectively. There are k input paths covering all the staleness susceptible states P = $Path^1, Path^2, \ldots, Path^k$ where $\cup_{I=1}^{k} Path^I = U$ such that each path $Path^I$ has a positive real cost (resources used) X^I. If a n-hop $Path^I$ contains state S_j, then the resources used for S_j in $Path^I$ are denoted as X^I_j, such that for the n states of $Path^I$ $\Sigma_{j=1}^{n} X^I_j = X^I$.*
*A k-tuple M = $M_1, M_2, \ldots, M_k$ constitutes a multiple cover for U in which the number of times state S_j is covered is defined to be the sum of M_I's for those $Path^I$'s which contain S_j and the total weight of the **multiple cover** is defined to be $\Sigma_{I=1}^{k} X^I \times M_I$. WMSCP seeks the minimum weight multiple cover for U such that every state S_j is covered for at least R_{S_j} refresh rate. By defining b^I_j to be 1 if $S_j \epsilon Path^I$ and 0 otherwise, we can now write our WMSCP problem as:*

$$\Delta\mu = \text{Minimize } \Sigma_{I=1}^{k} X^I \times M_I \tag{11}$$

subject to:

$$\Sigma_{I=1}^{k} b^I_j \times X^I_j \times M_I \geq R_{S_j} \quad \forall\, j = 1,2,\ldots,m. \tag{12}$$

Complexity and Optimality Analysis. As WMSCP is shown to be strongly NP-Hard, an optimal solution for it may be exttremely time consuming to find. Thus, we utilize a greedy algorithm called *GH-WMSCP* proposed in [29]. We use

this set cover algorithm to satisfy the refresh rates and in turn to fulfil *Freshness* predicates. First, the time complexity of the algorithm TC(GH-WMSCP) = $\bigcirc$(m $\times$ k + m^2) for m *staleness susceptible states* and k input paths. Second, the cover found by GH-WMSCP will atmost differ from the optimal cover for WMSCP, denoted as OPT(WMSCP), by a factor of ln(m). For details refer to [29].

Lemma 2. *For **k input streams** in a join query Q, there are exactly **(k-2) staleness susceptible states** irrespective of the query shape, be it linear, semi-bushy or bushy.*

By analyzing the binary join trees we observe the relationship between the number of input streams (k) and the number of staleness susceptible states (m) as given in Lemma 2. Using Lemma 2, the *linear* plan (Figure 5.b) and the *bushy* plan (Figure 5.c) both having four input streams, have exactly two *staleness susceptible states*. Therefore, substituting m with (k-2) in the expression for the time complexity of GH-WMSCP, TC(GH-WMSCP) = $\bigcirc(k^2)$.

4.3 The Integrated JAQPOT Algorithm

We now present our algorithm called ***J**oin **A**daptation at **Q**uery plan-level using **P**ath-productivity for **O**ptimizing **T**hroughput*, in short, JAQPOT (Algorithm 1). JAQPOT first assigns a fraction $\Delta\mu$[7] out of μ available resources towards fulfilling the *Freshness* requirements using the GH-WMSCP. Further, the greedy knapsack solver GrePP achieves an *optimal* query throughput using the remaining resources (μ - $\Delta\mu$). JAQPOT returns the join adaptation assignment in *PathAssign*[ ][ ], where *PathAssign*[I][j] denotes the resources assigned to the j^{th} half-way join of $Path^I$. The overall time complexity of our solution TC(JAQPOT) = TC(GH-WMSCP) + TC(GrePP) = $\bigcirc(k^2 + k \times \log(k)) \simeq \bigcirc(k^2)$, for k *input paths*. Thus, **JAQPOT runs in quadratic time of k** irrespective of the plan shape.

Algorithm 1 ***J**oin **A**daptation at **Q**uery plan-level using **P**ath-productivity for **O**ptimizing **T**hroughput* (JAQPOT)

Input: Path productivity table τ_p[ ] [1,. . . ,k] for plan Q, refresh rates for all j states $R_{S_{1,\ldots,j}}$, available resources μ tuples/sec
Output: Assignment of μ to plan Q *PathAssign*[ ][ ]
1: *PathAssign*[ ][ ] $\leftarrow$ GH-WMSCP(τ_p[ ][1,. . . ,k], $R_{S_{1,\ldots,j}}$)
2: $\Delta\mu \leftarrow \sum_{i->1}^{k}$ *PathAssign*[i][ ]
3: *PathAssign*[ ][ ] $\leftarrow$ GrePP($\mu - \Delta\mu$, τ_p[ ][1,. . . ,k])
4: return *PathAssign*[ ][ ]

[7] We chose to satisfy the *Freshness* predicates while *optimizing* throughput as this adaptation is sufficient for real world applications. We found in our experimental study (Section 5) that in practice realistic *Freshness* predicates are indeed fulfillable using only a small share of the resources.

Table 3. Experimental parameters for synthetic data set.

Parameter	Value
Arrival rates (λ_i)	300 ~ 1200 tuples/sec
Window sizes of states ($\|S_i\|$)	200 ~ 5000 tuples
Join selectivities (σ_i)	0.01 ~ 0.1
Available Resources (μ)	0 ~ 100 % of saturation
Freshness predicates (ϕ^I)	1.5× ~ 5× of window predicate

4.4 Run-time Query Adaptation

Due to fluctuations in the arrival patterns, an initially optimal resource allocation by JAQPOT may become sub-optimal after some time. In that case, the plan would need to be adapted at run-time. We now describe the framework we use for adapting the resource allocation across the join plan at run-time.

Our adaptation strategy, adopted from [20], works in three steps: *monitoring*, *analysis* and *actuation*. *Monitoring* entails collecting a running estimate of the statistics such as the available resources and the output rate of each join in order to derive the selectivities using windowed averages. When the query performance varies by more than a certain error threshold ϵ, the *analysis* step is triggered. Effectively, where we re-run the JAQPOT optimizer in a separate thread. Only if the predicted performance of the adaptation recommended by JAQPOT exceeds that of the existing plan by performance threshold p, then we *actuate* the adaptation. The *actuation* step is extremely efficient as it entails to simply changing the resource allocations along input path within the plan.

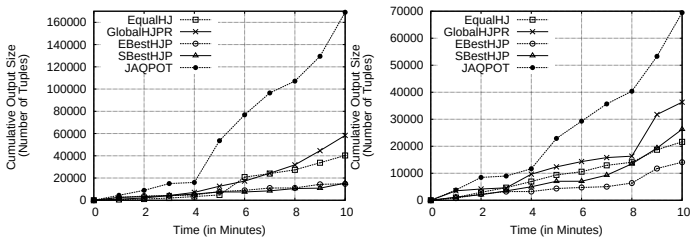
The processing is split into two execution threads. The *monitoring* and the *actuation* steps are interleaved with query execution within one thread. However, the *analysis* (i.e., optimizer calls and generation of recommendations) is run in a separate thread as it requires employment of the weighted set-cover and the knapsack solvers. This prevents blocking of the query executor while the system is being periodically analyzed for adaptation in parallel. The effectiveness of our run-time adaptation strategy is demonstrated in our experimental study (Section 5).

5 Experimental Evaluation

We now examine the effectiveness of our proposed JAQPOT approach. We compare JAQPOT against the four alternate heuristic policies based on the state-of-the-art half-way join productivity (ρ_h) metric, as described in Appendix B. For our experimental study we implemented JAQPOT and the four ρ_h-based heuristic policies within the XXX[8] stream processing engine [23].

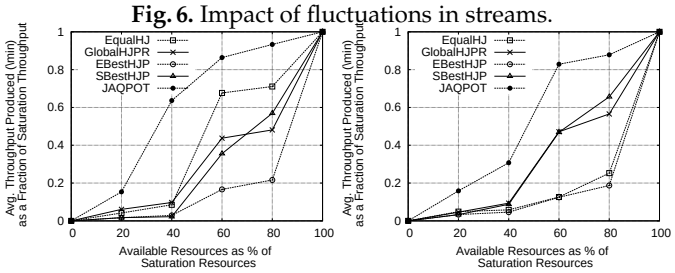
Objectives. Our analytical study (Section 4) establishes that JAQPOT is capable of producing *near optimal throughput* together with maintaining *result freshness*. The goal of this experimental study is to further substantiate whether the capabilities of JAQPOT hold true for real applications and in resource-limited environments. We evaluate JAQPOT and its competitor policies by measuring their performance in (a.) producing *query throughput*, and (b.) fulfilling the *freshness* predicates. We examine the following impending questions with focus on

[8] The name XXX is used for anonymity.



(a) Linear Plan (b) Bushy Plan

Fig. 6. Impact of fluctuations in streams.



(a) Linear Plan (b) Bushy Plan

Fig. 7. Impact of resource availability.

the two performance measures:

• *How do the throughput produced by the each of the JDA techniques compare when stream and query parameters, such as λ_i, σ_i, and S_I, fluctuate?*

• *How does the throughput produced by the JDA techniques compare with the saturation throughput*[9] *when the availability of resources is changed?*

• *In the absence of the freshness set-coverage solver (GH-WMSCP), how badly do the JDA techniques perform with respect to result freshness?*

• *In JAQPOT, what fraction of resources get assigned for fulfilling freshness as opposed to achieving high throughput?*

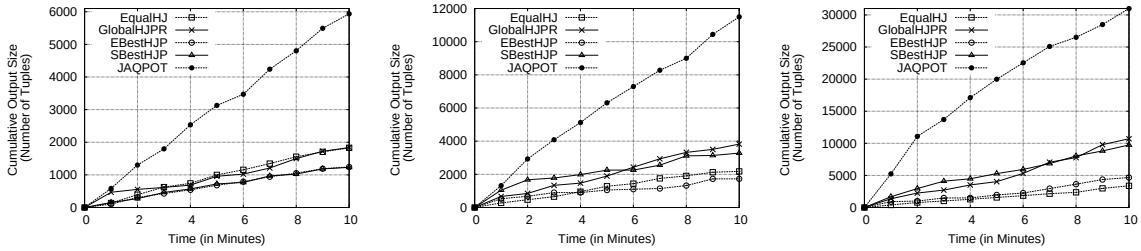
• *How costly are the adaptation steps? Are the benefits of JAQPOT larger than the incurred costs?*

Experimental Setup. All our experiments are run on a machine with Java 1.6.0.0 runtime, Windows 7 with Intel(R) Core(TM)2 Duo CPU@2.13 GHz processor and 4 GB of RAM. All techniques are tested rigorously using synthetic streams and distinct query shapes with arbitrary parameter settings (Table 3). Further, the applicability to a real-world application is also verified using the weatherboards data set [7].

5.1 Throughput Production in Synthetic Data

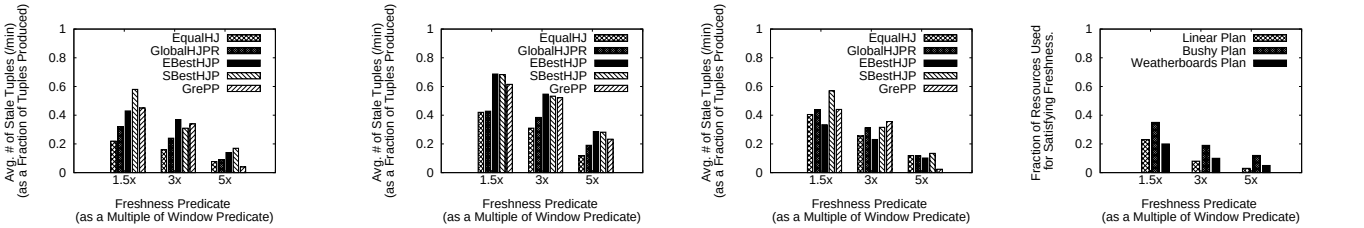
The goal of these experiments is to compare the throughput produced by each JDA technique under (a.) fluctuating streams, and (b.) changing resource avail-

[9] The minimum total resources required to process the full query workload with no CPU limitation are called the *saturation resources*. The corresponding throughput produced is called the *saturation throughput*.



(a) μ = 300 tuples per second (b) μ = 600 tuples per second (c) μ = 900 tuples per second

Fig. 8. Experiments using the intel berkeley weatherboard data set for three different available resources (μ).



(a) Linear (b) Bushy (c) Weatherboards (d) Resources Used for Freshness

Fig. 9. Evaluation of freshness predicates.

abilities. We measure throughput as the cumulative join output tuples produced over time. We use an equi-join of 4 streams, namely, A, B, C and D. We use two different query shapes, namely, *linear* (Figure 5.b) and *bushy* (Figure 5.c) plans. While the join order of the *linear* plan is (((A⋈B)⋈C)⋈D), that of the *bushy* plan is ((A⋈B)⋈(C⋈D)). The data streams are generated according to the *Poisson* distribution that models the arrival pattern of several real-world stream applications. Overall, a variety of scenarios are evaluated by changing the λ_i, σ_i and S_I parameters for each query shape (Table 3).

Impact of Fluctuating Stream Parameters. We change the *operator selectivities* to simulate fluctuations in the input streams. The other parameters, namely, *window sizes* and *arrival rates*, were observed to have a similar effect on the workload as that of the selectivities, thus we omit them here. Query workloads can be adjusted by generating streams such that the join selectivities become high (or low) as desired. Here, we fixed the μ to 30% of saturation whereas $\oint^I$ is set to 1.5× WINDOW predicates on each stream I.

In Figure 6, we measure the cumulative throughput (y-axis) as time progresses (x-axis) for a total of 10 mins of steady state query execution. In the *linear* plan (Figure 6.a), the selectivities first change at *3 mins.* from SEL1 ($\bowtie_1$ = 0.01 | $\bowtie_2$ = 0.01 | $\bowtie_3$ = 0.05) to SEL2 ($\bowtie_1$ = 0.03 | $\bowtie_2$ = 0.03 | $\bowtie_3$ = 0.05) and further *at 7 minutes* from SEL2 to SEL3 ($\bowtie_1$ = 0.03 | $\bowtie_2$ = 0.03 | $\bowtie_3$ = 0.1). From SEL1 to SEL2, the selectivities of $\bowtie_1$ and $\bowtie_2$ *triple* while keeping $\bowtie_3$ constant. From SEL2 to SEL3, the selectivity of the root $\bowtie_3$ *doubles* while the selectivities of $\bowtie_1$ and $\bowtie_2$ remain unchanged. This change in the root operator $\bowtie_3$ improves the throughput production by JAQPOT even more significantly than the change in non-root operators. The adaptation by the heuristic policies is marginal. Glob-

alHJPR outperforms the rest of the ρ_h-based policies, as its allocation criteria is based on join selectivities itself. Figure 6.b illustrates the results for the *bushy* plan with changes in the selectivities at 3 and 7 mins, just like in the case of the linear plan. Here JAQPOT again produces high throughput while GlobalHJPR performs only reasonably.

Impact of Changing Available Resources. These experimental results (Figure 7) summarize the performance of the techniques over the whole range of resource availability from 0% to 100%. A variety of parameter settings are used, as in Table 3. The charts depict the available resources as a percentage of the *saturation resources*. On the y-axis, the throughput produced by each policy, averaged over several runs, is shown as a percentage of the saturation throughput.

For the linear plan (Figure 7.a) JAQPOT utilizes the resources effectively. Specifically JAQPOT is able to produce more than 80% of the saturation throughput while using only 60% of the resources. JAQPOT consistently outperforms all the ρ_h-based policies. Averaged over the different cases of the available resources, JAQPOT produces about 2.5× as many tuples/min as those produced by the leading ρ_h-based policy, with a maximum of 6.5× at 40% of saturation resources. Among the heuristic policies, GlobalHJPR is found to be most effective at low resource availability, whereas the simplistic EqualHJ policy performs well when resources are above 60% of saturation.

For the bushy plan (Figure 7.b), JAQPOT consistently outperforms the ρ_h-based policies and on average produces about *twice* as many tuples/min. as produced by the most productive ρ_h-based policy. GlobalHJPR and SBestHJP are the two most effective ρ_h-based policies. Clearly, JAQPOT performs much better for the linear plan as compared with the bushy plan. This happens because all the *staleness susceptible states* in the linear plan can be synchronously refreshed as they belong to a *single* path, Path^A. On the other hand, for the bushy plan, atleast two paths require resources for fulfilling the *freshness* predicates, thus leaving less resources for throughput optimization.

5.2 Throughput Production in Real Data

We now test the applicability of the techniques for real data streams, namely, the weatherboard data set from Intel Berkeley Lab [7]. The data set contains readings collected from 54 sensors deployed in the Intel Berkeley Research lab between Feb. 28th and Apr. 5th of 2004. Each row of data is an 8-tuple [date, time, epoch, moteid, temperature, humidity, light, voltage]. We cleanse the data by removing all tuples with missing or incorrect (negative) values. The cleansed version contains 2.2 million tuples. The test query for the data set is an equijoin of the 5 streams in order (((A⋈B)⋈(C⋈D))⋈E) (Figure 5.d), where streams A. . .E represent different sensor groups (in proximity to each other). A constant window size of 1000 tuples is used for each join state. We evaluate three value settings of μ, namely, 300, 600 and 900, all measured in tuples/sec.

In the experimental results for $\mu = 300$ (Figure 8.a), we observe the cummulative throughput produced by each technique (on y-axis) as time progresses (on x-axis). During the complete run, JAQPOT produces 2.8× as many tuples/minute

as those produced by the best heuristic policy on average, with a maximum of 3.4× at 4 mins. Among the heuristic policies, EqualHJ and GlobalHJPR produce high throughput. In experiments with $\mu = 600$ (Figure 8.b), JAQPOT on average outperforms the best heuristic policy by 2.5×. GlobalHJPR and SBestHJP are the best heuristic policies. Finally, for $\mu = 900$, Figure 8.c), JAQPOT averages 3.4× better than GlobalHJPR and SBestHJP, which continue to top the heuristic policies. Comparing the output rate achieved by JAQPOT across the charts, doubling the μ from 300 to 600 also doubles the average throughput produced per minute. Moreover, a change of 1.5× from 600 to 900 *triples* the average throughput.

5.3 Evaluating Result Freshness

The purpose of these experiments is twofold, namely, (a.) to establish that result staleness is indeed imminent in resource-limited processing, and (b.) to verify that satisfying *freshness* predicates often does not require a substantial portion of the resources. The staleness of results is measured by counting the number of tuples produced that violate a given *Freshness* predicate. For example, if $\oint^{I} =$ 12 minutes, we evaluate every join tuple produced per time unit. A joined tuple is *stale* with respect to stream I, if it contains a stream I tuple whose timestamp is earlier than 12 mins before the current time.

Result Staleness in Join Adaptation. Next, we substantiate our hypothesis that the *throughput optimizing schemes aggravate the result staleness problem*. We compare the four heuristic policies with only the GrePP knapsack solver. The GH-WMSCP component for tackling result staleness is not used. We perform these experiments on the same three plans used in the throughput experiments, namely, the linear, the bushy and the weatherboards (Figures 5.b, c and d). For each plan shape we create many scenarios by varying the parameter settings (Table 3). Here, μ is fixed at 300. We evaluate three distinct settings of the *Freshness* predicate, namely, 1.5×, 3×, and 5× the window predicate. The higher the freshness predicate, the more tolerant the user query is to staleness. For each freshness value, we count the number of stale tuples produced by each technique. The three *freshness* predicate values (x-axis) are plotted against the average fraction of stale tuples /min (y-axis).

For the linear plan (Figure 9.a), as the freshness predicate is relaxed from 1.5× to 5×, there is a marked drop in number of stale tuples. Among the heuristics, EBestHJP and SBestHJP produce high amounts of stale tuples. Even GrePP produces a substantial amount of stale tuples in the absence of GH-WMSCP. The trend is similar for the bushy plan (Figure 9.b). However, the bushy plan produces an even larger number of stale results compared to the linear plan. For 1.5×, about 70% of the tuples produced by SBestHJP are stale. The staleness trends are similar in the weatherboards data (Figure 9.c). It suffers less from result staleness because the μ value was sufficient such that the throughput optimizing allocation inadvertently covered the staleness susceptible states as well.

Resource Utilization for Satisfying Freshness. The goal of these experiments is to evaluate what fraction of the available resources are allocated by JAQPOT for fulfilling the freshness requirements. For these experiments, we again evaluate three distinct *Freshness* settings, namely, 1.5×, 3×, and 5× of the window predicates. We run the JAQPOT algorithm, including both GrePP and GH-WMSCP, for several settings of the linear and the bushy plans by changing the query parameters, including different μ values. Further, JAQPOT is also run for the weatherboards data using each of the three *Freshness* predicates and $\mu = 300$. As the remaining resources would be used for *throughput optimization*, the less resources JAQPOT uses for freshness fulfillment, the better the performance.

In Figure 9.d, the Freshness predicate (x-axis) is plotted against the fraction of resources used for satisfying freshness. We find that as the freshness predicate is relaxed, the demand for resources for satisfying freshness is drastically reduced. For Freshness tolerance of 5×, the linear and the weatherboards plans utilize only 3% and 5% resources for freshness, respectively. Further, as the bushy plan faces higher risk for staleness, the bushy plan uses significantly larger portions of resources for Freshness (35% for 1.5×).

Among the plans we evaluated, we periodically found some *infeasible* plans, i.e., whose freshness predicates were not achievable under existing conditions. In particular, about 8% of the evaluated plans were infeasible, 85% of which were for the rigid Freshness predicate 1.5× and 65% were for bushy plans. However, for the weatherboards query, JAQPOT always found a feasible assignment and utilized minimal resources. Thus, for most reasonable Freshness predicates, an allocation utilizing not more than 10% of μ on average is possible. Moreover, our proposed solution is guaranteed to find the best solution, if one exists. This fact may be proven using the implicit nature of the set-cover and the knapsack solvers, i.e., the set-cover [29] finds the minimal resources utilized for fulfilling the freshness requirements whereas the knapsack solver [21] maximizes the throughput with the remaining resources.

5.4 Run-time Overhead of JAQPOT

As described in Section 4.4, JAQPOT has three overheads: *monitoring*, *analysis* and *actuation*. The monitoring and the actuation steps are inexpensive as they do not require much computation. The dominant cost among the overheads is the *analysis* step. The analysis overhead is measured as the time taken to detect the violation of the error threshold, run the optimizer, and recommend the alternate resource allocation.

Among the analysis tasks, running GH-WMSCP for set coverage was found to be the costliest computation. While the performance of GH-WMSCP for different parameters, such as sizes of input and sets, has been thoroughly studied in [29], our observations are similar. For all tested cases the execution time of GH-WMSCP was found in 2 millisec to 300 millisec range. The plan recommended by GH-WMSCP deviates from OPT(WMSCP) by 11% on average and at most by

19%. Moreover, we run the optimizer on a separate thread such that acceptable relative to the benefits gained.

Yet another aspect of the adaptation framework is when and how often to adapt. The monitoring step performs sampling and makes the decision whether to discard or keep the sample. When any statistics vary by more than the error threshold ϵ, the *analysis* step is triggered. Similarly, the performance threshold p governs when the actuation step is triggered. We acknowledge that the performance p and error ϵ thresholds are critical settings that must be determined empirically within a given context. We conduct empirical runs to find reasonable settings for these thresholds for a given environment. Subsequently those are held stable for the reported experiments. For the different settings of the synthetic data, p was between 3% and 7% and ϵ was between 5% and 8%. For Weatherboards both were fixed at 5%. A self-tuning technique where these thresholds get adjusted on-the-fly could be applied. However, such advanced tuning strategies [10,14,18,20] are orthogonal to our core optimization solution, and are left for future work.

Experimental Conclusions. The overall findings using both the synthetic and real datasets are:
• JAQPOT continuously produces near optimal throughput even under fluctuating streams.
• JAQPOT consistently outperforms the ρ_h-based policies and produces 2~6 times the throughput produced by them for all tested cases.
• Among the ρ_h-based policies, GlobalHJPR outperforms the other policies in most cases.
• JAQPOT performs better in *linear* plans compared to *bushy* plans, as bushy plans utilize more resources for *freshness* fulfillment.
• Under resource-limited processing *result staleness* is aggravated by throughput optimizing techniques.
• If the freshness predicates are *satisfiable*, JAQPOT is guaranteed to find a resource allocation that utilizes the minimum resources.
• Performance trends observed in experiments over synthetic data consistently also hold true for the real data as well.

6 Related Work

Existing research on resource-limited execution of join queries may be classified into two categories, namely, memory-limited [11, 16, 17, 24] and computing-limited [3, 12, 13, 15, 25]. All of these approaches typically address a single optimizing function. Solutions addressing the memory-limited scenarios typically either focus on a single join operator [11, 17, 24] or optimize multi-join query plans using flushing [16] or memory management [6, 24]. Alternatively, load shedding [3, 25] is popular in computing-limited scenarios. Shedding directly drops the tuples from the streams and the data is permanently lost. Shedding solutions, with an exception of [3, 25] as further discussed below, focus on optimizing a *single* join operator or a single MJoin operator [13].

Tatbul et al. [25] are among the first to apply load shedding to streaming databases. They propose two shedding algorithms, namely, *drop-based* and *filter-based* shedding for query networks (mostly filter queries). In subsequent research [26], they focus on shedding for aggregation queries. As indicated in [25], they *do not* address the additional issues related to processing windowed joins over streams. Ayad et al. [3] explore the inter-relationship between *query optimization* and *load shedding*. They propose static optimization and in the absence of a feasible plan they pick a plan augmented with shedding operators placed on the input streams to make it feasible. However, there is no focus on leveraging the inter-operator dependency to adapt to run-time fluctuations nor do they consider *result staleness*. Tu et al. [27] propose a control-based approach of load shedding to tackle processing delays. However, they do not focus on the additional challenges of multi-join plans neither do they address throughput optimization.

GrubJoin [13] targets the MJoin operator by leveraging *time correlation-awareness*. Despite a common focus on multi-join queries, GrubJoin is different from our work. First, it focuses on a *single* MJoin operator, whereas our work tackles an orthogonal problem of operator interdependencies within a plan. Moreover, the use of MJoin under *compute*-intensive workloads has been questioned in the literature [22, 28] as it repeatedly recomputes results compared against result reuse in a pipelined plan of binary operators. Under *compute*-intensive workloads, a query optimizer is thus highly unlikely to select a single MJoin over a binary join plan. The experiments in Viglas et al. [28] indeed exclude worst case scenarios for MJoin. For such cases, materialization of intermediate results [22,28] or breaking the MJoin into smaller MJoins or binary joins is proposed. Whenever such a plan of interconnected join operators is used, our solution tackling operator *interdependency* issues can be applied in conjunction with the existing approaches [13, 28].

Closest to our work, *join direction adaptation* (JDA) [12, 15] explores the *halfway join productivity* to *selectively allocate* computing resources to maximize the output rate. They focus on a *single* join operator only. In this work, we establish that such traditional JDA technique is ineffective for multi-join queries.

Operator *scheduling* [8, 9, 19] achieves efficient processing of continuous queries by determining two execution decisions, namely, (a.) what order should the operators be scheduled in?, and (b.) how many tuples should an operator process at each execution step? Operator ordering has been studied well in the literature such as round robin scheduling [19] and *chain* scheduling [5]. Little work has been done on the second aspect. In some sense our work now relates to this second aspect - though in the context of resource-limited environments.

Another area of related research is adaptive query processing [10, 14, 18] that aims to identify at run-time when sub-optimal performance arises. This is typically accomplished by comparing the estimated and the measured factors in the query. When such an anomaly is detected, the query optimizer alters the plan at run-time to enhance the overall performance. Our work utilizes such

an adaptive framework for adjusting the join direction of the query plan at run-time using our proposed metric and algorithms.

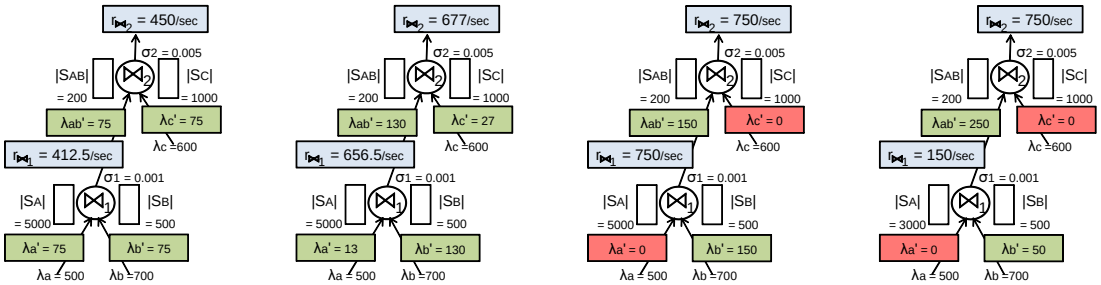
7 Conclusion

This paper addresses the computing-limited execution of multi-join queries using join direction adaptation. By leveraging the *operator interdependencies* instead of localized operator-centric optimization, we propose the *path productivity* metric. We identify *result staleness* as a pressing issue under resource limitations, and throughput optimizing techniques further aggravate it. Our most important contribution is the integrated JAQPOT algorithm that tackles the *result staleness* problem while producing *optimal* query throughput. We validate our analytical findings using experimental studies with both synthetic and real data.

References

1. D. J. Abadi, D. Carney, and et al. Aurora: a new model and architecture for data stream management. *VLDB*, 12(2):120–139, 2003.
2. A. Arasu and et al. The cql continuous query language: semantic foundations and query execution. *VLDB*, 15(2):121–142, 2006.
3. A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, pages 419–430, 2004.
4. B. Babcock, S. Babu, and et al. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
5. B. Babcock, S. Babu, and et al. Chain: operator scheduling for memory minimization in data stream systems. In *SIGMOD*, pages 253–264, 2003.
6. S. Babu, K. Munagala, and et al. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.
7. P. Bodik, S. Madden, and et al. Intel berkeley research lab: Weatherboards dataset. http://db.csail.mit.edu/labdata/labdata.html.
8. D. Carney, U. C, and et al. Monitoring streams: A new class of data management applications. In *VLDB*, pages 215–226, 2002.
9. D. Carney and et al. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.
10. A. Deshpande, Z. G. Ives, and et al. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
11. J.-P. Dittrich and et al. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB*, pages 299–310, 2002.
12. B. Gedik and et al. Adaptive load shedding for windowed stream joins. In *CIKM*, pages 171–178, 2005.
13. B. Gedik, K.-L. Wu, and et al. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding. *IEEE TKDE*, 19:1363–1380, 2007.
14. Z. G. Ives, A. Deshpande, and et al. Adaptive query processing: Why, how, when, and what next? In *VLDB*, pages 1426–1427, 2007.
15. J. Kang, J. Naughton, and et al. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.
16. B. Liu, Y. Zhu, and E. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD*, pages 347–358, 2006.

17. G. Luo, C. Ellmann, and et al. A scalable hash ripple join algorithm. In *SIGMOD*, pages 252–262, 2002.
18. S. Madden, M. Shah, and et al. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, 2002.
19. R. Motwani, J. Widom, and et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
20. R. V. Nehme, E. A. Rundensteiner, and E. Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *EDBT*, pages 803–814, 2009.
21. D. Pisinger. Where are the hard knapsack problems? *Comput. Oper. Res.*, 32(9):2271–2284, 2005.
22. V. Raghavan, Y. Zhu, and et al. Multi-join continuous query optimization: Covering the spectrum of linear, acyclic, and cyclic queries. In *BNCOD*, pages 91–106, 2009.
23. E. A. Rundensteiner and et al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.
24. U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.
25. N. Tatbul and et al. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
26. N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.
27. Y.-C. Tu, S. Liu, S. Prabhakar, and et al. Load shedding in stream databases: a control-based approach. In *VLDB*, pages 787–798, 2006.
28. S. Viglas, J. Naughton, and et al. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.
29. J. Yang and J. Y.-T. Leung. A generalization of the weighted set covering problem. *Naval Research Logistics*, pages 142–149, 2005.



(a) EqualHJ (b) GlobalHJPR (c) EBestHJP (d) SBestHJP

Fig. 10. The ρ_h-based Heuristic Strategies for Allocation of μ to the Multi-join Query Scenarios.

A Additional Background

Unit-time Basis Cost Model. A cardinality-based cost model is inapplicable for continuous queries because the streams are infinite whereas it computes the

Table 4. The ρ_h-based heuristics for multi-join query scenarios.

Policy	Application to 2-join query (Figure 10)
Equally among Half-way Joins (EqualHJ): Allocate μ equally among all m half-way joins such that for each half-way join $\ltimes_i$ ($\forall$ i = 1,2,...,m), $\mu_{\ltimes_i} = \frac{\mu}{m}$.	$\lambda'_a = 75$ |$\lambda'_b = 75$ | $\lambda'_{ab} = 75$ | $\lambda'_c = 75$ (tuples per second), $r_{\bowtie_1} = 412.5$ ab join tuples per second, $\mathbf{r_{\bowtie_{root}} = 450}$ **abc join tuples per second.** **Producer-consumer mismatch:** out of 412.5 ab tuples produced by $\bowtie_1$ only 75 will be used as probe allowance in $\bowtie_2$.
Global Half-way Join Productivity Ratio (GlobalHJPR): Allocate μ to all m half-way joins in the ratio of their respective half-way join productivities (ρ_h) such that for each half-way join $\ltimes_i$ ($\forall$ i = 1,2,...,m), $\mu_{\ltimes_i} = \frac{\mu \times \rho_h(\ltimes_i)}{\sum_{j=1}^{m} \rho_h(\ltimes_j)}$.	$\lambda'_a = 13$ | $\lambda'_b = 130$ | $\lambda'_{ab} = 130$ | $\lambda'_c = 27$ (tuples per second), $r_{\bowtie_1} = 656.5$ ab join tuples per second, $\mathbf{r_{\bowtie_{root}} = 677}$ **abc join tuples per second.** **Producer-consumer mismatch:** out of 656.5 ab tuples produced by $\bowtie_1$ only 130 will be used as probe allowance in $\bowtie_2$.
Equally among $\bowtie$s, then Local BestHJP (EBestHJP): Allocate μ in two levels: Divide μ *equally* among all n join operators such that for each join $\bowtie_j$ ($\forall$ j = 1,2,...,n), $\mu_{\bowtie_j} = \frac{\mu}{n}$. Within each operator $\bowtie_j$, apply BestHJP to assign all $\mu_{\bowtie_j}$ towards the most productive half-way join component of $\bowtie_j$.	$\mu_{\bowtie_1} = \mu_{\bowtie_2} = \frac{300}{2} = 150$ tuples per second, $\lambda'_a = 0$ | $\lambda'_b = 150$ | $\lambda'_{ab} = 150$ | $\lambda'_c = 0$ (tuples per second), $r_{\bowtie_1} = 750$ ab join tuples per second, $\mathbf{r_{\bowtie_{root}} = 750}$ **abc join tuples per second.** **Producer-consumer mismatch:** out of 750 ab tuples produced by $\bowtie_1$ only 150 will be used as probe allowance in $\bowtie_2$.
σ-ratio among $\bowtie$s, then Local BestHJP (SBestHJP): Allocate μ in two levels: Divide μ among all n join operators in the ratio of their respective *join selectivities* such that for each join $\bowtie_j$ ($\forall$ j = 1,2,...,n), $\mu_{\bowtie_j} = \frac{\mu \times \sigma_{\bowtie_j}}{\sum_{i=1}^{n} \sigma_{\bowtie_i}}$. Within each operator $\bowtie_j$, apply BestHJP to assign all $\mu_{\bowtie_j}$ towards the most productive half-way join component of $\bowtie_j$.	$\mu_{\bowtie_1}:\mu_{\bowtie_2} = 0.001:0.005 = 50:250$ tuples per second, $\lambda'_a = 0$ | $\lambda'_b = 50$ | $\lambda'_{ab} = 250$ | $\lambda'_c = 0$ (tuples per second), $r_{\bowtie_1} = 150$ ab join tuples per second, $\mathbf{r_{\bowtie_{root}} = 750}$ **abc join tuples per second.** **Producer-consumer mismatch:** only 150 tuples are produced by $\bowtie_1$ whereas 250 probe allowance is, assigned to λ'_b of $\bowtie_2$ thus wasting 100 probe allowance.

time needed by the query to run to completion. Thus, we adopt the unit-time basis cost model proposed by Kang et al. [15].

$$\text{Cost}(A \bowtie B) = \text{Cost}(a \ltimes S_B) + \text{Cost}(b \ltimes S_A) \tag{13}$$

$$\text{Cost}(a \ltimes S_B) = \lambda_a \times (\text{probe}(S_B) + \text{insert}(S_A) + \text{purge}(S_A)) \tag{14}$$

$$\text{Cost}(b \ltimes S_A) = \lambda_b \times (\text{probe}(S_A) + \text{insert}(S_B) + \text{purge}(S_B)) \tag{15}$$

In a unit-time basis cost model, the cost of the single join operator (Figure 2.a) is divided into two independent *half-way* join components, also called *join directions*(Equation 13). In Equation 14, λ_a tuples arriving from stream A *probe* state S_B per time unit. Those λ_a tuples are then *inserted* into state S_A and *purge* tuples previously present in state S_A. The *probe* translates to a *search* on the *partner* join state (here, S_B), whereas *insert* and *purge* translate to *updates* on *own* state (S_A).

Analysis of Join Cost Factors. The search and update costs depend on the data structure employed in the implementation of the state. While *search* is costlier in a *nested loop join*, using an efficient circular list the update costs (insert + purge) are constant time. In a *symmetric hash join* using hash buckets, the probe cost is less than that in nested loop join, but now depends on the number of distinct buckets and the population of each bucket. In addition, an efficient circular list of tuples maintaining the arrival order keeps the update (insert + purge) costs low. Overall, while probe dominates the join cost, the update (insert + purge) costs are fairly minor [15].

Extension to Time-based Windows. The unit-time basis cost model is developed for *count*-based windows only. However, it may be extended to *time*-based windows by applying logic similar to that used by Ayad et al. [3] as follows. As

we are concerned with steady state conditions and are using average rate, it is easy to adapt the model for *time*-based windows using the following argument. On average, the number of active tuples in a time-based window state S_I of time T units is ($\lambda_i \times$ T). So, by replacing the size $|S_I|$ of a *count*-based window with ($\lambda_i \times$ T), all our equations will be applicable to *time*-based windows as well.

B Heuristic HJP-based Policies for Multi-Join Plan

We further design four ρ_h-based heuristic policies. In Table 4, we describe each of them including their principle and their application to a 2-join plan (Figure 10). These policies do not guarantee optimal throughput as they suffer from *producer-consumer mismatch*. Moreover, the *result staleness* may arise from them, as the *Freshness* predicates are also not guaranteed to be covered.