Achieving High Freshness and Optimal Throughput in CPU-limited Execution of Multi-Join Continuous Queries⋆

Abhishek Mukherji and Elke A. Rundensteiner and Matthew O. Ward

Computer Science Department, Worcester Polytechnic Institute, Worcester MA, USA.
mukherab | rundenst | matt @wpi.edu

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 data loss. We explore *join direction adaptation* (JDA) to tackle CPU-limited processing of multi-join stream queries. 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*, typically ignored by most state-of-the-art techniques, as a critical issue in CPU-limited processing. It gets further aggravated if throughput optimizing techniques are employed. We establish the novel *path-productivity* model and the *Freshness* predicate. Our proposed *JAQPOT* approach is the first integrated solution to achieve *near optimal* query throughput while also guaranteeing freshness satisfiability. JAQPOT runs in quadratic time of the number of streams irrespective of the query plan shape. Our experimental study demonstrates the superiority of JAQPOT in achieving higher throughput than the state-of-the-art JDA strategy while also fulfilling freshness predicates.

1 Introduction

Motivation. *Data Stream Management Systems* (DSMS) [2, 5, 15] are in high demand for real-time decision support as they transform 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 and bursty data volumes.

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 [8,9,11]. When processing complex join queries, either the processor may fail to keep up with the arrival rates of the streams (the CPU-limited case) or the available main 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. [8] observe that with increasing stream arrival rates and large join states, the CPU typically becomes strained before the memory does. Temporary data flushing [11] 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 *CPU*-limited cases.

⋆ This work was supported by NSF grants IS-0812027, CCF-0811510, IIS-0917017 and IIS-1018443.

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

Fig. 1. Example Query.

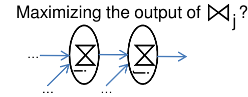


Fig. 2. A pipeline of join operators.

When resources are limited, yet another pressing issue, namely, *result staleness* arises. In Query Q1 (Fig. 1) 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 DSMS may be required 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*. A *freshness* predicate may be defined on each stream, i.e., 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 and tackles the dual problems of achieving optimal throughput while satisfying freshness of the join results.

The State-of-the-art. Two directions for tackling join queries under *computing* limitations are *load shedding* [4, 9, 16] and *join direction adaptation* (JDA) [8, 10]. The main focus of load shedding is to reduce the input rates by directly dropping tuples from the source streams [4]. This makes the plan incapable of recuperating with the production of accurate results in moments of low workloads as data is permanently lost.

Unlike load shedding, JDA preserves in-memory tuples as per the join semantics for opportunities of joining with future incoming tuples. Existing JDA techniques [8, 10] 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 Sec. 6.

Research Challenges. In general, the ability of multi-join queries to achieve *high* result throughput and to maintain result freshness under heavy workloads relies on resolving the following aspects of the problem:

- While *operator scheduling* [6, 7] tends to allocate resources at the *coarse* granularity of query operators, **adaptation at a finer granularity within the operators** is required to produce optimal throughput.
- The existing JDA technique [10] optimizes every join operator individually. In a pipeline of join operators (Fig. 2), an uncoordinated attempt to optimize operators $\bowtie_i$ and $\bowtie_j$ individually may jeopardize the real goal of optimizing the overall query throughput. The 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.
- We identify **result staleness**[2] as a critical issue for CPU-limited processing of multi-join queries. The biased resource allocation by **JDA** may potentially aggravate this problem.

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

[2] This challenge is not identified by prior work [4, 9, 16].

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 CPU resources with the goal to achieve maximal throughput. 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 of the query. We summarize our contributions as follows:

1. We demonstrate that the state-of-the-art **JDA** [10] technique fails to achieve optimal throughput for the **CPU**-limited processing of *multi-join* plans (Sec. 3).
2. We establish the **path productivity** metric as the plan-level throughput contribution of each input stream (Sec. 4.1).
3. We formulate the query throughput maximization as a **knapsack** problem and propose a **Greedy Path Productivity-based Adaptation (GrePP)** to solve it (Sec. 4.2).
4. We identify result staleness as a key challenge under CPU-limited scenarios and formulate the **freshness satisfiability** as a **weighted set-cover** problem (Sec. 4.3).
5. We integrated the above two strategies into the JAQPOT algorithm (Sec. 4.4). 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**.
6. Our experimental study (Sec. 5) demonstrates the superiority of JAQPOT over the state-of-the-art JDA solution in a large set of tested cases.

2 Preliminaries

2.1 Background

In this paper we focus on multi-join plans composed of sliding window binary join operators. We assume standard semantics as in CQL [3]. We use the **unit-time basis cost model** proposed by Kang et al. [10] that computes the join cost in terms of the three sub-tasks, namely, *probe*, *insert* and *purge* operations. For simplicity, the model assumes *count-based* windows. The key idea is that **the cost of probe dominates the total join cost while insert and purge operations are relatively inexpensive**. For details on the model and its extension to *time-based* windows refer to [10]. Fig. 3 lists the notation.

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

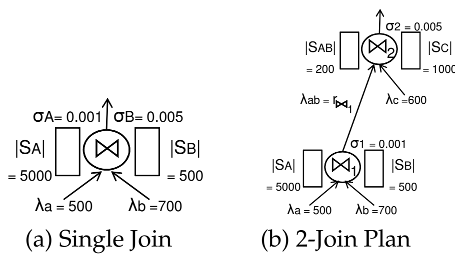
Fig. 3. List of notation.

Throughput. The run-time throughput (Eq. 1) of a join operator A⋈B (Fig. 4.a) consists of two contributing half-way join components, namely, $r_{\ltimes} = (a \ltimes S_B)$ and $r_{\rtimes} = (b \ltimes S_A)$. Throughput is also called the *output rate* and is defined as *the number of joined tuples produced per time unit*. For tuple t^A, S_A is the *own* state whereas S_B is the *partner* state.

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

CPU Limitation in a Join. When CPU is limited, the throughput of A⋈B can be rewritten as in Eq. 2. The total *available computing resources*, denoted as μ, may be determined from the system. Terms *available resources* and *service rate* are used interchangeably.

$$r_{\bowtie}=r_{\ltimes} + r_{\rtimes}=\lambda'_a \times \sigma_B \times |S_B| + \lambda'_b \times \sigma_A \times |S_A|, \lambda'_a + \lambda'_b \leq \mu. \quad (2)$$



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

Fig. 4. Join plans with parameter settings.

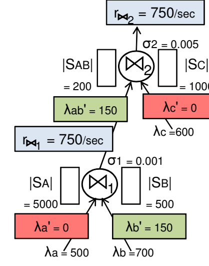


Fig. 5. JDA over a 2-join plan.

In Eq. 2, the μ resources allocated to a join is 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 μ not exceeding the input rate λ_a, i.e., $\lambda'_a = \min(\mu_a, \lambda_a)$. Similarly, $\lambda'_b = \min((\mu\text{-}\mu_a), \lambda_b)$. As the *probe* cost dominates the total join cost, the resource restriction only affects the *probe*. All input tuples undergo the *insert* and *purge* operations. In Fig. 4.a A⋈B have input rates $\lambda_a = 500$ and $\lambda_b = 700$ tuples/sec. Assume $\mu = 300$ tuples/sec as the available resources. 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. However, all 1200 arriving tuples undergo *insert* and *purge* operations every time unit. Thus, the μ resources (here 300 tuples/sec) must be divided among the *probe allowances* (here λ'_a and λ'_b) for throughput maximization.

2.2 Problem Definition

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

CPU-limited Execution of Multi-Join Plans. In the 2-join plan of Fig. 4.b[4] composed of $\bowtie_1$ and $\bowtie_2$, the *throughput optimization* problem is quite different from the single operator case. Now, the goal is to maximize the *throughput* $\mathrm{r}_{\bowtie_{root}}$ of the root operator (here $\bowtie_2$).

$$\mathbf{r}_{\bowtie_1} \mathbf{=} \lambda'_a \times \sigma_B \times |\mathrm{S}_B| + \lambda'_b \times \sigma_A \times |\mathrm{S}_A|; \mathbf{r}_{\bowtie_2} \mathbf{=} \lambda'_{ab} \times \sigma_C \times |\mathrm{S}_C| + \lambda'_c \times \sigma_{AB} \times |\mathrm{S}_{AB}|, \\ \lambda'_a + \lambda'_b + \lambda'_{ab} + \lambda'_c \leq \mu. \tag{3}$$

Equation 3 depicts the CPU-limited case in a *multi-join* plan, where μ needs to be divided among four half-way joins, namely, λ'_a, λ'_b, λ'_{ab} and λ'_c. In general, μ gets split at two levels. First, among the n join operators (say $\mu_{\bowtie_1}, \mu_{\bowtie_2}, \ldots, \mu_{\bowtie_j}, \ldots, \mu_{\bowtie_n}$). Then, for each join $\bowtie_j$, $\mu_{\bowtie_j}$ is divided among each of its respective half-way joins $\mu_{\ltimes_j}$ and $\mu_{\rtimes_j}$.

Freshness of Multi-Join Results. When the resources become limited, the produced query results may no longer be perfectly *fresh*, as in Query Q1 (Fig. 1). The result freshness is further compromised by throughput optimizing resource allocation to highly productive half-way joins. 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 become *stale*. The join results produced using such stale states are also stale, thus further deteriorating the *result freshness*. In Fig. 4.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.

[3] A fine-grained *time correlation-awareness* [9] can be used for subset selection in conjunction with JDA.

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

Definition 1. *The **freshness** predicate, namely,* $\mathbb{F}^I$ *for a stream I, requires that joined tuples produced beyond time T must not contain stream I tuples with arrival times older than* $|T - \mathbb{F}^I|$.

Under CPU-limited execution, the user can supply a *freshness* predicate $\mathbb{F}^I$ for each stream I (Def. 1). The type of the *freshness* and the window predicate must be the same, i.e., time or count-based. By default the freshness predicate $\mathbb{F}^I$ equals 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. *In this work we focus on achieving maximal throughput while satisfying the user-defined freshness specification.*

3 The JDA Technique

The state-of-the-art JDA technique uses a *half-way join productivity-based* optimization. We first define the *half-way join productivity* ρ_h metric (Def. 2) and then present the JDA policy.

Definition 2. *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*** $\ltimes_i$.

$$\rho_h(\mathrm{i} \ltimes \mathrm{S}_J)=\frac{\mathrm{r}_{\ltimes_i}}{\lambda'_i}=(\sigma_J \times |\mathrm{S}_J|) \tag{4}$$

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

In a single join operator (Fig. 4.a) the μ resources (= 300 tuples/sec) must be divided among the *probe allowances* (here λ'_a and λ'_b). such that the throughput $r_{\bowtie}$ of A⋈B is maximized. By Equation 4, the ρ_h of half-way joins 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/input tuple/sec. For μ = 300 tuples/sec, JDA achieves a throughput of $r_{\bowtie} = 300 \times 5 = 1500$ by assigning all of μ to λ'_b and none to λ'_a.

Applying JDA to a Multi-Join Plan. Now, we derive a variant of the JDA policy, called JDA_M, to make it applicable to the multi-join plans. It is defined as follows:

JDA_M Policy: *Allocate* μ *in two levels:*

1. *Divide* μ *equally among all the n join operators, so for each join* $\bowtie_j$ ($\forall j = 1,2,\ldots,n$), $\mu_{\bowtie_j} = \frac{\mu}{n}$.
2. *Within each operator* $\bowtie_j$*, apply BestHJP to assign all* $\mu_{\bowtie_j}$ *towards the most productive half-way join component of* $\bowtie_j$.

In Figure 5, JDA_M first splits the μ *equally* among the two joins $\bowtie_1$ and $\bowtie_2$, i.e., 150 tuples per second are allocated to each join. In the next step, JDA is applied locally within each operator. In $\bowtie_1$, $\rho_h(\mathrm{b} \ltimes \mathrm{S}_A) > \rho_h(\mathrm{a} \ltimes \mathrm{S}_b)$, so ($\mathrm{b} \ltimes \mathrm{S}_A$) is assigned the 150 tuples per second. Similarly, in $\bowtie_2$, $\rho_h(\mathrm{ab} \ltimes \mathrm{S}_C) > \rho_h(\mathrm{c} \ltimes \mathrm{S}_{AB})$, thus ($\mathrm{ab} \ltimes \mathrm{S}_C$) is assigned the 150 tuples per second. An estimated query throughput of 750 joined tuples per second is achieved (Equation 3). We observe here that while $\bowtie_1$ produces 750 tuples per second, only 150 out of those can actually be used to probe the partner join state in $\bowtie_2$. Hence, there is an **over-utilization** of resources in $\bowtie_1$.

The local nature of the JDA technique and its failure in producing optimal throughput in a multi-join plan motivates us to explore operator interdependencies for solving the identified problems as described next in Section 4. In our experiments (Section 5) we compare the JDA_M strategy against our proposed approach.

4 The Proposed JAQPOT Approach

4.1 Optimizing Throughput in Multi-Join Queries

Throughput optimization in a *multi-join* plan requires **producer-consumer matches** between every successive join pair where all intermediate tuples produced by a *producer* join must get consumed by the downstream *consumer* join for probing the partner join state. Based on this insight, we propose a synchronized plan level resource allocation strategy.

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

In our example plan (Fig. 4.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 and propagates through n half-way joins upto the root. Similar to the half-way join productivity (ρ_h), we now define the *cardinality of intermediate joined tuples*, denoted as ϕ^I_j, produced by the jth half-way join along $Path^I$. As in Eq. 5, ϕ^I_j is computed by multiplying the productivities (ρ_h) along $Path^I$ upto j. Here, superscript p denotes the *partner* join state. ϕ^I_j forms an important component of the core formulae that we define next.

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

Fig. 6. Division of X^I resources within $Path^I$.

$$X^I = X^I_1 + \ldots + X^I_j + \ldots + X^I_n = X^I_1 \times [1 + \sum_{g=1}^{j-1} \phi^I_g] \longrightarrow X^I_j = X^I_1 \times \phi^I_{j-1} = (\frac{X^I \times \phi^I_{j-1}}{1+\phi^I_1+\ldots+\phi^I_{n-1}}). \tag{6}$$

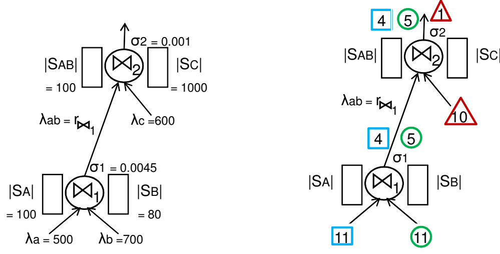
Division of Resources within an Input Path. X^I of the total μ resources are allocated to an *n-hop* path $Path^I$. Figure 6 depicts the division of X^I among the half-way joins of $Path^I$ as *probe allowances* $X^I_1, X^I_2, \ldots, X^I_j, \ldots, X^I_n$. For an effective division of X^I **producer-consumer matches** must be established between every pair of successive half-way joins, such that the output of a producer equals the *probe allowance* of the consumer join. Each such *probe allowance* for the jth half-way join, denoted as X^I_j, is expressed in terms of X^I as in Eq. 6. Here, ϕ^I_{j-1} denotes the *cardinality of the (j-1)th intermediate joined tuples*. **Path Productivity.** We now establish a *novel* metric that measures the contribution of an input path to the overall query throughput (Def. 4).

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

$$\rho_p(\mathrm{Path}^I) = (\sigma^p_n \times |S^p_n|) \times (\frac{\phi^I_{n-1}}{1 + \phi^I_1 + \ldots + \phi^I_{n-1}}) = (\frac{\phi^I_n}{1 + \phi^I_1 + \ldots + \phi^I_{n-1}}). \tag{7}$$

[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.

[6] Tuple t refers to either a leaf or an intermediate tuple in $Path^I$.



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

Fig. 7. Example 2-join plan.

Path^I	Path^A	Path^B	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

Fig. 8. Path productivity table.

If X^I resources are assigned to Path^I, the component X^I_n assigned to the root half-way join may be computed using Eq. 6. The throughput contribution by processing the X^I resources along Path^I, denoted as $(r^I_{\bowtie_{root}}) = (\sigma^p_n \times |S^p_n|) \times X^I_n$. Therefore, by Def. 4, the productivity of Path^I can be computed as $\rho_p(\text{Path}^I) = \frac{r^I_{\bowtie_{root}}}{X^I}$, as given in Eq. 7.

In our example plan (Fig. 4.b), X^B resources allocated to Path^B will be divided among λ'_b and λ'_{ab}. By Eq. 6, an effective division of X^B over Path^B will be $\lambda'_b = (X^B/6)$ and $\lambda'_{ab} = (5 \times X^B/6)$. Using Eq. 7, the total throughput contribution $(r^B_{\bowtie_{root}})$ achieved is estimated as $(5 \times X^B/6) \times (\sigma_C \times |S_C|)$. Thus, if X^B = 600 tuples/sec, λ'_b gets 100, then producing $r_{\bowtie_1}$ = 500 tuples/sec as intermediate output. λ'_{ab} gets 500 tuples/sec (= X^B - λ'_b) and a *producer-consumer match* is achieved between $\bowtie_1$ and $\bowtie_2$.

Discussion. The path productivity ρ_p metric defaults to the notion of half-way join productivity ρ_h (Sec. 3) when applied to a single operator. ρ_h is a *local* operator level metric whereas 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.

4.2 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 7. The 2-join plan in Fig. 7 may be translated into a *path productivity table* (Fig. 8). 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* ($\rho_p(\text{Path}^I)$.

In Fig. 7.b) for Path^A, the resources X^A = 15 tuples/sec may be divided across the two half-way joins such that $\lambda'_a = 11$ and $\lambda'_{ab} = 4$. The throughput contribution of Path^A, denoted as $r^A_{\bowtie_{root}} = 4$ as $\sigma_C = 1$. Thus, for every 15 tuples consumed by Path^A in a second, it will produce 4 t^{ABC} joined tuples. The values in Fig. 8 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 [14], as given below.

Problem 1. ***Join adaptation knapsack problem:*** *Given a plan Q with k paths $Path^1$, $Path^2$,. . . , $Path^k$, when $Path^I$ is assigned resources in multiple M^I of X^I, then its throughput $r^I_{\ltimes_{root}} = M^I \times \rho_p(Path^I)$. 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:*

$$\text{Maximize } \Sigma^k_{I=1}\, a_I \times r^I_{\bowtie_{root}} \tag{8}$$

subject to:

$$\Sigma^k_{I=1}\, a_I \times M^I \times X^I \leq \mu. \tag{9}$$

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

Assume μ = 30 tuples/sec. Using the JAQPOT Policy for the productivity table listed in Fig. 8, 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 resources. The total throughput thus achieved 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 X^I, $r^I_{\bowtie_{root}}$ and μ must be *whole numbers*. Under this model the *throughput optimization* problem does not exhibit the *greedy choice property* ([14]). Thus, a *dynamic programming* knapsack solver must be employed to achieve an assignment yielding optimal throughput which runs in $\bigcirc$(k × μ), 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. We now relax the above restrictions. First, instead of *independent execution cycles*, each being assigned *distinct* μ resources, we now consider the *coordinated execution across successive cycles*. For example, two successive cycles producing 3 and 2 join tuples respectively will result in the overall path productivity ρ_p(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 cycles is identified, we can view their combination as a *mega* cycle. Secondly, X^I, $r^I_{\bowtie_{root}}$ and μ values can now be *fractional*. Thus, for Path^B (Fig. 7), 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 a single cycle, over the span of multiple successive 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. Also in a real-world CPU-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 cycle. Under this *continuous execution model*, our *JDA-Knapsack* Problem 1 now exhibits both the *greedy choice property* and the *optimal substructure property* [14]. 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 in Fig. 7, GrePP selects the most productive path, Path^B (Fig. 8), 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 and is greater than that in the *discrete model*. It is guaranteed to be *optimal* [14]. **GrePP runs in $\bigcirc$(k** log**(k)) time [14] for a plan joining k streams and thus is independent of** μ.

4.3 Satisfying Freshness in Multi-Join Queries

The throughput optimizing allocation by GrePP may still suffer from *result staleness*. We allow users to supply the *freshness* predicates (Def. 1) for each input stream. Now, we extend our **path-productivity based** model to incorporate this notion of *freshness*.

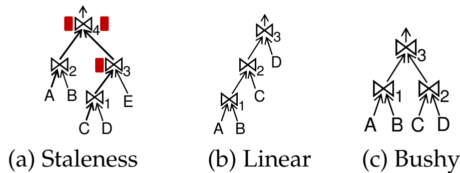
The key idea here is that the **freshness predicates** defined over streams are fulfilled by translating them into **refresh rates** for the join states inside the plan. By Def. 1, tuple t^I from stream I must not be part of the joined results beyond time (t^I.ts + $\mathbb{F}^I$), where t^I.ts denotes the arrival time of t^I. To enforce this constraint, every tuple t^I from stream I and all its intermediate joined tuples must be purged from the plan by (t^I.ts + $\mathbb{F}^I$) time (or tuple for count-based freshness). In Fig. 9.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 by incoming tuples at λ_c tuples/sec, whereas the intermediate states intermediate states S_{CD} and S_{CDE} get refreshed with tuples t^{CD} and t^{CDE} at a rate dependent on the portion of μ allocated to λ'_{cd} and λ'_{cde}, respectively. Such intermediate states, such as S_{CD} and S_{CDE}, are called *staleness susceptible states* (highlighted in Fig. 9.a). In a steady stream, $\frac{S_C}{\lambda_c}$, $\frac{S_{CD}}{\lambda'_{cd}}$ 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}. To satisfy the predicate $\mathbb{F}^C$ for stream C, $\mathbb{F}^C \geq (\frac{S_C}{\lambda_c} + \frac{S_{CD}}{\lambda'_{cd}} + \frac{S_{CDE}}{\lambda'_{cde}})$.

Lemma 1. *To fulfill the freshness predicate $\mathbb{F}^I$ for stream I, $\mathbb{F}^I \geq \Sigma_{j=1}^{n} \frac{S_j^o}{\lambda'_j}$, 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.*

Using Lemma 1, the *freshness* predicate $\mathbb{F}^I$ 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. Our model of input paths enables us to gain further insights into the *result staleness* problem. We observe that each input path contains one or more of these *staleness susceptible states*. Moreover, each susceptible state, say S_{CD}, may receive input from multiple paths. For example, S_{CD} gets input from (c $\ltimes$ S_D) and (d $\ltimes$ S_C). Also S_{CDE} gets input from (cd $\ltimes$ S_E) and (e $\ltimes$ S_{CD}). Thus, *staleness susceptible states* may be refreshed *synchronously* by allocating resources to the paths covering those states. The *freshness* predicates can be satisfied by fulfilling the corresponding *refresh rates* of a half-way join (Def. 5) of each *staleness susceptible state*, i.e, if the input rate λ'_j (λ_j for leaf) at each half-way join exceeds the desired R_{S_j}.

Definition 5. *The* ***refresh rate*** *R_{S_j} of a state 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, in Problem 2 we translate the *freshness* satisfiability problem into a **weighted multiple set cover problem (WMSCP)** [18] over the *staleness susceptible*



(a) Staleness (b) Linear (c) Bushy

Fig. 9. Multi-join plans.

Algorithm 1: The JAQPOT Algorithm

Input: Path productivity table τ_p[ ] [1,. . . ,k] for plan Q, refresh rates $R_{S_{1,...,j}}$ $\forall$ j states, available resources μ

Output: Assignment of μ to plan Q *PathAssign*[ ][ ]

1: *PathAssign*[ ][ ] ← GH-WMSCP(τ_p[ ][1,. . . ,k], $R_{S_{1,...,J}}$)
2: $\Delta\mu \leftarrow \sum_{i->1}^{k}$ *PathAssign*[*i*][ ]
3: *PathAssign*[ ][ ] ← GrePP($\mu - \Delta\mu$, τ_p[ ][1,. . . ,k])
4: return *PathAssign*[ ][ ]

states. 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 are used by GrePP for throughput optimization. In Fig. 9.a, Path^A and Path^C are such *minimal coverage paths* covering all staleness susceptible states in $\bowtie_3$ and $\bowtie_4$.

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. The k input paths* $P = Path^1, Path^2, \ldots, Path^k$ *cover all the staleness susceptible states 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*. Total weight of the* ***multiple cover*** *is defined as* $\Sigma_{I=1}^{k} X^I \times M_I$*. WMSCP seeks the minimum weight multiple cover for U such that every state* S_j *is covered for at least its refresh rate* R_{S_j}*. By defining* b^I_j *to be 1 if* $S_j \in Path^I$ *and 0 otherwise, we can now write our WMSCP problem as:*

$$\Delta\mu = \text{Minimize } \Sigma_{I=1}^{k} \text{X}^I \times \text{M}_I \quad (10)$$

subject to:

$$\Sigma_{I=1}^{k} \text{b}^I_j \times \text{X}^I_j \times \text{M}_I \geq \text{R}_{S_j} \quad \forall \text{ j} = 1,2,\ldots,\text{m}. \quad (11)$$

Complexity and Optimality Analysis. WMSCP is strongly NP-Hard and the cost of an optimal solution may be too high for our dynamically scenario. Thus, we use a greedy algorithm called *GH-WMSCP* [18]. We use GH-WMSCP to satisfy the refresh rates and in turn to fulfil *freshness* predicates. The time complexity TC(GH-WMSCP) = $\bigcirc(\text{m} \times \text{k} + \text{m}^2)$ for m *staleness susceptible states* and k input paths. The cover found by GH-WMSCP will atmost differ from the optimal cover for WMSCP, denoted as OPT(WMSCP), by a factor of $\ln(m)$ [18].

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.*

Lemma 2 relates the counts of the input paths (k) and the susceptible states (m) in a query. For example, in both the plans, namely, *linear* (Fig. 9.b) and *bushy* (Fig. 9.c), k=4 and m=2. Therefore, substituting m with (k-2) in the expression for the time complexity of GH-WMSCP, TC(GH-WMSCP) = $\bigcirc(\text{k}^2)$.

4.4 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

[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 (Sec. 5) that in practice realistic *freshness* predicates are indeed fulfillable using only a small share of the resources.

resources assigned to the jth half-way join of $Path^I$. The overall time complexity of our solution TC(JAQPOT) = TC(GH-WMSCP) + TC(GrePP) = $\bigcirc(k^2 + k \times \log(k)) \simeq \bigcirc(k^2)$. Thus, **JAQPOT runs in quadratic time of k** irrespective of the plan shape.

Run-time Query Adaptation. An initially optimal resource allocation by JAQPOT may become sub-optimal due to the dynamic nature of the streams. Thus, we adopt a simple yet effective strategy for runtime adaptation (details in [13]). We measured the runtime overheads of JAQPOT and found that GH-WMSCP incurs the highest cost. Performance of GH-WMSCP for different parameters, such as sizes of input and sets, has been thoroughly studied in [18]. Thus, we instead focus our experimental study on throughput and freshness.

5 Experimental Evaluation

Parameter	Value
Arrival rates (λ_i)	300 ~ 1200 tuples/sec
Window sizes of states ($\lvert S_I \rvert$)	200 ~ 5000 tuples
Join selectivities (σ_i)	0.01 ~ 0.1
Available Resources (μ)	0 ~ 100 % of saturation
Freshness predicates ($\mathbb{F}^I$)	1.5× ~ 5× of $\lvert S_I \rvert$

Fig. 10. Experimental parameters with values.

We now examine the effectiveness of JAQPOT compared against the state-of-the-art JDA technique (described in Section 3). We implemented JAQPOT and JDA_M within the CAPE stream engine [15].

Objectives. The goal of our experimental study is to substantiate the observations of our analytical study (Sec. 4) that JAQPOT is capable of producing *near optimal throughput* together with maintaining *result freshness*. We evaluate JAQPOT and its competitor JDA_M policy by measuring their performance in (a) *producing query throughput*, and (b) *fulfilling the freshness predicates*. We examine the following critical questions with focus on the two performance measures:

• *What is the impact of stream and query parameters, such as λ_i, σ_i, and S_I as well as the query shape on the throughput produced by the JDA techniques?*

• *How does the throughput produced by JAQPOT and JDA_M techniques compare with the saturation throughput*[8] *with change in μ?*

• *In the absence of the set-coverage solver (GH-WMSCP), how badly do the throughput optimization techniques perform in terms of the result freshness?*

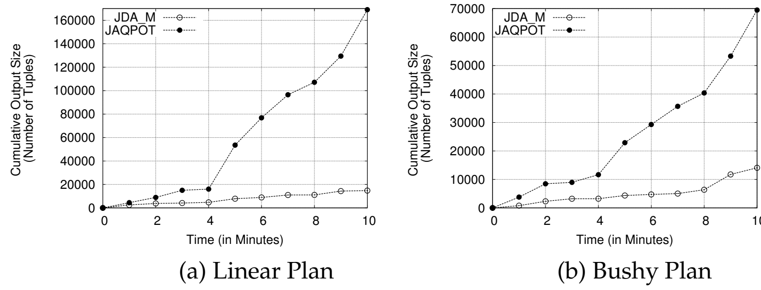
• *In JAQPOT, what fraction of resources get assigned for fulfilling freshness as opposed to achieving high throughput?*

Experimental Setup. All experiments are run on a machine with Java 1.6 runtime, Windows 7 with Intel(R) Core(TM)2 Duo CPU@2.13 GHz processor and 4 GB RAM. All techniques are tested rigorously using synthetic streams and distinct query shapes with arbitrary parameter settings (Table 10). Further, the applicability to a real-world application is also verified using the Weatherboards data set [1]. The results for the experiments with the real data set are not included in this paper. Please refer to technical report [12] for details.

5.1 Throughput Production in Synthetic Data

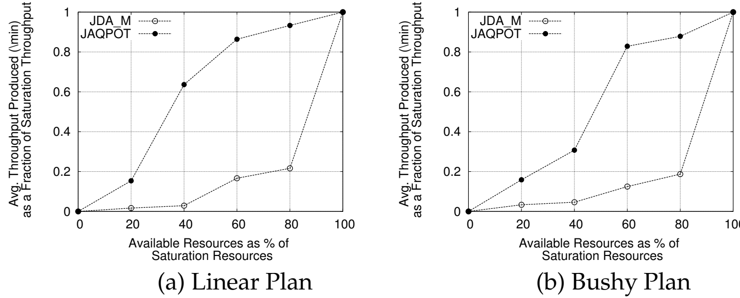
The goal of our experiments is to compare the throughput produced by both the JDA techniques under (a) fluctuating streams, and (b) changing resource availabilities. We

[8] 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) Linear Plan (b) Bushy Plan

Fig. 11. Impact of fluctuations in streams.



(a) Linear Plan (b) Bushy Plan

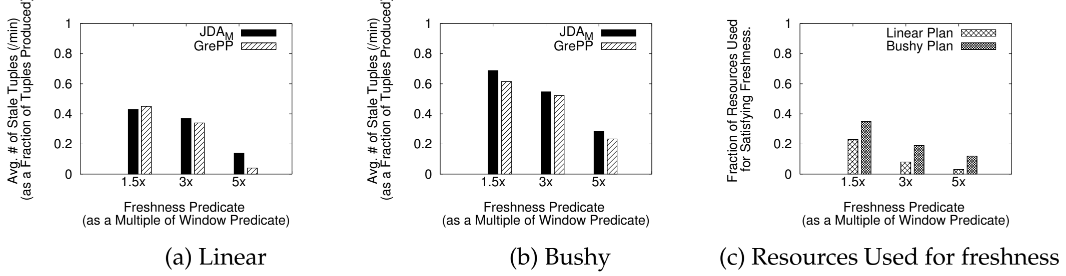
Fig. 12. Impact of resource availability.

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 with two different query shapes, namely, *linear* (Fig. 9.b) and *bushy* (Fig. 9.c). While the join order of the *linear* plan is $(((A\bowtie B)\bowtie C)\bowtie D)$, that of the *bushy* plan is $((A\bowtie B)\bowtie(C\bowtie D))$. 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 (Fig. 10).

Impact of Fluctuating Stream Parameters. The fluctuating input streams are simulated by changing *operator selectivities*. The *window sizes* and *arrival rates* were observed to have *similar* effects 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 $\mathbb{F}^I$ is set to 1.5× WINDOW predicates on each stream I.

In Fig. 11, 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 (Fig. 11.a), the selectivities first change at *3 mins.* from SEL1 ($\bowtie_1 = 0.01 \mid \bowtie_2 = 0.01 \mid \bowtie_3 = 0.05$) to SEL2 ($\bowtie_1 = 0.03 \mid \bowtie_2 = 0.03 \mid \bowtie_3 = 0.05$) and further *at 7 minutes* from SEL2 to SEL3 ($\bowtie_1 = 0.03 \mid \bowtie_2 = 0.03 \mid \bowtie_3 = 0.1$). 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 performance of the JDA_M strategy is significantly lower than JAQPOT. Figure 11.b illustrates the results for the *bushy* plan with changes in the selectivities at 3 and 7 mins, just like the linear plan. Here JAQPOT again produces high throughput while JDA_M continues to produce output at a very low rate.

Impact of Changing Available Resources. These charts (Fig. 12) summarize the performance of the techniques over the complete range of μ values from 0% to 100% of



(a) Linear (b) Bushy (c) Resources Used for freshness

Fig. 13. Evaluation of freshness predicates.

saturation. A variety of parameter settings are used, as in Table 10. The charts depict μ as a percentage of the *saturation resources*. On the y-axis, the throughput produced by JAQPOT and JDA$_M$ strategies, averaged over several runs, is shown as a percentage of the saturation throughput.

For the linear plan (Fig. 12.a) JAQPOT outperforms the JDA$_M$ strategy producing more than 80% of the saturation throughput while using only 60% of the resources. Averaged over the different μ values, JAQPOT consistently beats the JDA$_M$ strategy by producing 3× as many tuples/min on average, with a maximum of 6.5× at 40% of saturation resources.

For the bushy plan (Fig. 12.b), the trends are similar. Overall, JAQPOT performs much better for the linear plan than the bushy plan. In the linear plan, all the *staleness susceptible states* can be synchronously refreshed as they are covered by fewer (possibly single) path, whereas, for the bushy plan, atleast two paths require resources for *freshness* fulfillment, thus leaving fewer resources for throughput optimization.

5.2 Evaluating Result Freshness

The purpose of these experiments is twofold, namely, (a) to establish that result staleness is aggravated by the JDA approaches, and (b) to analyse how much of the resources are spent in satisfying *freshness*. The staleness of results is measured by counting the number of tuples produced that violate a given *freshness* predicate $\mathbb{F}^I$ as per Def. 1.

Result Staleness in Join Adaptation. Next, we substantiate our hypothesis that the *throughput optimizing schemes aggravate the result staleness problem*. We compare the JDA$_M$ and the GrePP knapsack solver omitting the GH-WMSCP component such that GrePP produces stale tuples in the absence of GH-WMSCP. We perform these experiments on the linear and bushy plans (Figs. 9.b and c). For each plan, we create many scenarios by varying the parameter settings (Table 10). Here, μ is fixed at 300. We evaluate three distinct settings of the *freshness* predicate, namely, 1.5×, 3×, and 5× of the window size. 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 (Fig. 13.a), as the freshness predicate is relaxed from 1.5× to 5×, there is a marked drop in number of stale tuples. The JDA$_M$ strategy produces high amounts of stale tuples. In absence of GH-WMSCP, even GrePP produces a substantial amount of stale tuples. The trend is similar in the bushy plan (Fig. 13.b). However, an even larger number of stale results are produced in the bushy plan as compared to the linear plan.

Resource Utilization for Satisfying Freshness. We aim to evaluate the fraction of the available resources allocated by JAQPOT towards freshness fulfillment. For these experiments, we run JAQPOT (GH-WMSCP + GrePP) for several settings of the linear

and the bushy plans by changing the query parameters, including different μ values. We again evaluate three distinct *freshness* settings, namely, 1.5×, 3×, and 5× of the window predicates.

In Fig. 13.c, 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 requirement also reduces drastically. For freshness tolerance of 5×, the linear plan utilizes 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×). Thus, most reasonable freshness predicates may be fulfilled by allocating less than 10% of μ on average. Moreover, our proposed solution is guaranteed to find the best solution, if one exists. The proof can be easily implied from optimality of the set-cover [18] and the knapsack [14] solvers.

Infeasible plans. 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 the bushy plans.

Experimental Conclusions. The findings of our experimental study are:

1. JAQPOT continuously produces near-optimal throughput even in bursty streams.
2. JAQPOT consistently outperforms the state-of-the-art ρ_h-based JDA_M policy by producing 2~6 times higher throughput for all tested cases.
3. JAQPOT performs better in *linear* plans compared to *bushy* plans, as bushy plans utilize more resources for *freshness* fulfillment.
4. In CPU-limited processing, *result staleness* problem is further aggravated if throughput optimizing techniques are employed.
5. For all satisfiable cases, JAQPOT guarantees a throughput optimizing allocation.

6 Related Work

Load shedding [4, 16] is popular in CPU-limited scenarios. Shedding directly drops the tuples from the streams and the data is permanently lost. Shedding solutions, with an exception of [4,16] focus on optimizing a *single* join operator or a single MJoin operator [9]. Tatbul et al. [16] first applied load shedding to streaming databases. As indicated in [16], they *do not* address the additional issues related to processing windowed joins over streams. Ayad et al. [4] 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. GrubJoin [9] targets the MJoin operator by leveraging *time correlation-awareness*. It focuses on a *single* MJoin operator, whereas our work tackles an orthogonal problem of operator interdependencies within a plan. Although MJoins utilize less memory, they are typically computationally expensive [17] and are less likely to be selectd by the query optimizer in a CPU-limited scenario.

Closest to our work, *join direction adaptation* (JDA) [8, 10] explores the *half-way 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 becomes ineffective for multi-join queries. Further, all these approaches typically address a single optimizing function. None of these approaches focus on leveraging the inter-operator dependency to adapt to run-time fluctuations nor do they consider *result staleness*. Whenever a query with interconnected join operators is used, our solution leveraging operator *interdependency* can be applied in conjunction with the existing approaches [9, 17].

While **operator scheduling** [6,7] tends to allocate resources at the **coarse** granularity of a query operators, we focus on adaptation at a **finer** granularity of half-way joins within an operator for optimizing throughput. Our work utilizes an adaptive query processing [13] framework for adjusting the join direction of the query plan at run-time.

7 Conclusion

This paper addresses the CPU-limited execution of multi-join queries using join direction adaptation. We propose the *path productivity* metric that leverages the *operator interdependencies* instead of localized operator-centric optimization. We identify *result staleness* as a pressing issue under CPU limitations, and throughput optimizing techniques further aggravate it. Our key 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.

8 Acknowledgements

We are grateful to Song Wang, Luping Ding and other DSRG members for their efforts in building the CAPE system. We thank Prof. Murali Mani and the anonymous reviewers for their insightful comments.

References

1. Weatherboards dataset from intel berkeley research lab. http://db.csail.mit.edu/labdata/labdata.html.
2. D. J. Abadi, D. Carney, and et al. Aurora: a new model and architecture for data stream management. *VLDB*, 12(2), 2003.
3. A. Arasu and et al. The cql continuous query language: semantic foundations and query execution. *VLDB*, 15(2), 2006.
4. A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, 2004.
5. B. Babcock, S. Babu, and et al. Models and issues in data stream systems. In *PODS*, 2002.
6. B. Babcock, S. Babu, and et al. Chain: operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.
7. D. Carney and et al. Operator scheduling in a data stream manager. In *VLDB*, 2003.
8. B. Gedik and et al. Adaptive load shedding for windowed stream joins. In *CIKM*, 2005.
9. 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, 2007.
10. J. Kang, J. Naughton, and et al. Evaluating window joins over unbounded streams. In *ICDE*, 2003.
11. B. Liu, Y. Zhu, and E. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD*, 2006.
12. A. Mukherji and E. A. Rundensteiner. Achieving high freshness and optimal throughput in resource-limited execution of multi-join continuous queries: The jaqpot approach. Technical report, WPI, 2011.
13. R. V. Nehme, E. A. Rundensteiner, and E. Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *EDBT*, 2009.
14. D. Pisinger. Where are the hard knapsack problems? *Comput. Oper. Res.*, 32(9), 2005.
15. E. A. Rundensteiner and et al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, 2004.
16. N. Tatbul and et al. Load shedding in a data stream manager. In *VLDB*, 2003.
17. S. Viglas, J. Naughton, and et al. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.
18. J. Yang and J. Y.-T. Leung. A generalization of the weighted set covering problem. *Naval Research Logistics*, 2005.