Performance Modeling of Stream Joins

Vincenzo Gulisano
Chalmers University of Technology
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

Alessandro V. Papadopoulos
Mälardalen University
Västerås, Sweden
alessandro.papadopoulos@mdh.se

Yiannis Nikolakopoulos
Chalmers University of Technology
Gothenburg, Sweden
ioaniko@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se

Philippas Tsigas
Chalmers University of Technology
Gothenburg, Sweden
philippas.tsigas@chalmers.se

ABSTRACT

Streaming analysis is widely used in a variety of environments, from cloud computing infrastructures up to the network's edge. In these contexts, accurate modeling of streaming operators' performance enables fine-grained prediction of applications' behavior without the need of costly monitoring. This is of utmost importance for computationally-expensive operators like stream joins, that observe throughput and latency very sensitive to rate-varying data streams, especially when deterministic processing is required.

In this paper, we present a modeling framework for estimating the throughput and the latency of stream join processing. The model is presented in an incremental step-wise manner, starting from a centralized non-deterministic stream join and expanding up to a deterministic parallel stream join. The model describes how the dynamics of throughput and latency are influenced by the number of physical input streams, as well as by the amount of parallelism in the actual processing and the requirement for determinism. We present an experimental validation of the model with respect to the actual implementation. The proposed model can provide insights that are catalytic for understanding the behavior of stream joins against different system deployments, with special emphasis on the influences of determinism and parallelization.

CCS CONCEPTS

• **Information systems → Stream management**;

KEYWORDS

Data Streaming, Stream Join, Modeling

ACM Reference format:
Vincenzo Gulisano, Alessandro V. Papadopoulos, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2017. Performance Modeling of Stream Joins. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 12 pages.
DOI: http://dx.doi.org/10.1145/3093742.3093923

DEBS '17, Barcelona, Spain
© 2017 ACM. 978-1-4503-5065-5/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3093742.3093923

1 INTRODUCTION

Data streaming is one of the main processing paradigms that can generate information useful for upcoming systems' contexts, such as production environments or financial applications, that require adaptiveness for resource-efficiency and need information that is generated continuously. Practitioners and scientists in the stream processing domain strive to improve these systems' functionality and efficiency, usually measured in terms of the *throughput* and *latency* with which streams' *tuples* are processed.

Stream joins, comparing tuples fed by two streams, are key operators in this form of data processing [5]. Due to the unbounded nature of data streams, it is often the case that stream joins are performed on subsets of recent tuples, referred to as *windows*. Due to their computation-intensive nature [1], it is all the more important to understand their behavior and expected performance, especially under a variety of deployment parameters (e.g., processing capacity and level of parallelism) and time-varying input parameters (e.g., varying sources and rates of tuples). Knowing their expected performance and processing capacity needs is very useful both for the corresponding applications' set-up and for continuous tuning for provisioning of the appropriate amount of resources they need.

In order to have such information, a rather intuitive approach is to collect it through monitoring of the system's performance and load. However, as also summarized in [4], such monitoring can (a) be intrusive and cause bottlenecks, thus deteriorating the performance of the system or (b) provide the information too late for timely reaction, thus potentially causing oscillations in the system's performance. An alternative approach is through proper *mathematical modeling* of the system behavior, taking into account important parameters of the input, the application and the system deployment. Such a model would provide statistics about the performance of stream join deployments without executing the code, which would in turn help fine-tuning the deployment parameters. It is challenging, however, to capture the actual behavior of a stream join in a comprehensive and precise way due to complex dependencies between its configuration and the properties of its deployment. Aspects of a stream join configuration that influence the latter's throughput and latency include its windows' configuration, whether the stream join should process tuples deterministically (i.e., resulting in a processing outcome that is not dependent on the incoming tuples' interleaving, which may vary significantly in *asynchronous* systems) and its parallelism degree. Deployment aspects affecting the throughput and latency of a stream join include the varying rates with which the incoming tuples are fed, the number of *distributed sources* feeding such tuples, the underlying data distribution of the tuples (which affects the probability with which two such tuples are successfully matched and result in an output tuple) and, finally, the processing capacity available to the stream join itself.

Here we present a *comprehensive and dynamic model* of stream joins' throughput and latency behavior that embraces all these aspects. We describe the model in detail in steps, identifying the impact of each of the above aspects and building up the performance models for complex settings from those for simpler settings. Further, we provide a detailed experimental study comparing the outcome of the model in connection to an actual running implementation. The results show agreement between the model and the exact system behavior, showing the reliability of the model in predicting the throughput and the latency with a fairly high accuracy. Our main contribution is a precise modeling of the throughput and latency of stream joins that allows for:

(1) The evaluation of benefits and cost of features such as parallelism and determinism, thus providing a gnomon on which to base decisions.
(2) The possibility to have statistics about the performance of stream join deployments without the need of instrumenting the code, as the latter often requires the intervention of the operating system, which induces costly overhead.
(3) Information useful in taking decisions about how to adjust parameters (load, processing capacity) in order to tune the performance of stream joins in a static or in a run-time fashion (e.g., to know how throughput and latency will be affected depending on decisions to shed load) based on the underlying dependencies of features such as determinism and parallel execution.

The paper is structured as follows. We discuss related work in Section 2 and introduce preliminary concepts in Section 3. The modeling is developed throughout Section 4 and evaluated in Section 5. We conclude in Section 6.

2 RELATED WORK

Throughput and latency have been studied in many related papers [1, 6, 8, 9], but mainly from an empirical point of view. In this work we provide a detailed model for a stream join that shows what contributes to the throughput and latency and studies the dependencies between aspects such as window semantics, possible exceeding of the processing quota, non-deterministic versus deterministic processing, existence of multiple distributed data sources and centralized versus parallel processing.

Analytical study of stream joins' throughput can be found in [7, 9, 17]. Differently from us, [7] focuses on a centralized execution only (with the goal of tuning a load shedding scheme); [17] models throughput for a concrete ad-hoc parallelization approach (time-slicing); finally, [9] provides equations for the throughput of stream joins but does not account for its time-varying state size and its evolution over time (the equations have the sole purpose of estimating the scalability of the proposed parallelization approach).

Regarding latency, the closest in spirit work is in [15]. Nevertheless, the modeling focuses on a specific stream join (the original handshake join of [16]) and is intended to show the considerable latency the original method incurs and thus motivate the improvement discussed in [15]. Furthermore, the model does not consider distributed data sources, the latency overhead introduced when enforcing deterministic processing nor the possibility for a stream join to exceed its processing quota. A model for the latency cost is also presented in [14] for the SplitJoin stream join, as well as for the average latency for 2 consecutive tuples to be compared. However, the cost for determinism (or respectively a strong order with dense punctuation) is not taken into account.

Further modeling, which does not cover the throughput and latency of stream joins under the configuration and deployment characteristics we consider can be found in [2, 3, 6, 11, 13]. In [3] the memory cost, as in the size of the join state, is considered for a join model combining window-based and punctuation-based semantics. A model focusing on the memory cost can be also found in [13] for the join-biclique method. The modeling in [6] is leveraged to find the optimal size of the window but is very specific to the coupled hardware architecture (Cell processor) and parallelization approach. In [2] the SECRET model analyzes the execution semantics of stream processing systems. The paper presents technology-independent definitions of stream, batch, window, time-based window, and window size, relating these quantities to the semantics of the data streaming process. In [11], Kang et al. proposed a unit-time-basis cost model for estimating the time needed for a query to be run to completion for different join algorithms. The cost model focuses on the handling of single individual tuples from each input stream separately.

3 PRELIMINARIES

We introduce here stream joins' semantics along with the formal notation used in the remainder, we discuss determinism and parallelism aspects of their implementations and state our modeling goal.

Stream join. Data streaming operators (e.g. filters, aggregates, joins) continuously process streams of data. Each stream is an unbounded sequence of tuples $t_0, t_1, \ldots$ sharing schema $\langle ts, A_1, \ldots, A_n \rangle$. Given a tuple t, attribute $t.ts$ represents its creation timestamp while the rest of the attributes are application-related. Stream processing *continuous queries* are modeled as directed acyclic graphs of operators that continuously consume incoming tuples and produce output tuples. Streams can be distinguished between *physical* and *logical* [8, 9]. A physical stream consists of tuples from a single source of data (e.g., from another operator), while a logical stream is a collection of physical streams delivering the same type of information from multiple sources (e.g., from multiple operators).

In a stream join (or simply join in the remainder) tuples are received from two input streams, R and S, and are compared by evaluating a given predicate. An output tuple joining the schema of R and S tuples is produced for each pair of tuples from R and S for which the predicate holds. Since the input streams are unbounded, tuples from each stream are compared only with a finite portion (window W_R from stream R and window W_S for stream S) of the opposite stream. *Time-based* windows are defined to cover a fixed period of time with respect to the timestamp attribute of the tuples (e.g., the last 5 minutes). Since R's and S' rates can vary over time, time-based windows do not specify the actual number of tuples to be contained in the window. If this is preferred over a window that spans a fixed time period, *tuple-based windows* can be employed

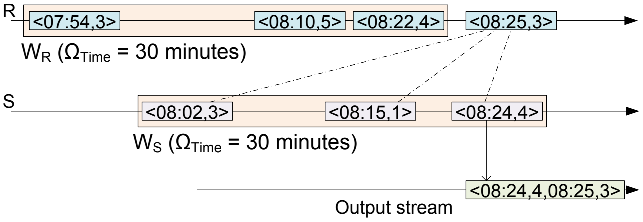


Figure 1: Example of a time-based join with Ω_{Time} set to 30 minutes. Tuples from R and S are composed by attributes $\langle ts, value\rangle$. A tuple t_R from R and a tuple t_S from S are successfully matched if $t_R.value < t_S.value$. In the example, an incoming tuple from R is matched with the tuples in W_S and one output tuple (joining the schema of R's and S' tuples) is produced.

(e.g., to contain the last 100 tuples). In the remainder we refer to a join relying on time-based and tuple-based windows as *time-based join* and *tuple-based join*, respectively.

The semantics of joins initially evolved in a tight connection with the underlying implementation. In the seminal work [11] Kang et al describe the problem of joins over unbounded streams as a nested-loop join (aka in the literature as the *3-step procedure*). Based on that, Teubner and Mueller [16] define their semantics as follows[1]:

Window-Based Stream Join Semantics. For $t_R \in R$ and $t_S \in S$, the tuple $\langle t_R,t_S\rangle$ appears in the join result $R \bowtie S$ iff

a) t_R arrives after $t_S(t_R.ts > t_S.ts)$ and $s \in W_S$ the moment that t_S arrives, or

b) t_R arrives earlier than $t_S(t_R.ts < t_S.ts)$ and $r \in W_R$ the moment that t_S arrives,

and t_R and t_S pass the join predicate.

Time-based join A time-based window of Ω_{Time} time units contains all tuples $\{t|t'.ts - t.ts \leq \Omega_{Time}\}$, where t' is the latest received tuple from stream R or S. As discussed in [9] the way the window is updated can give different variants of the 3-step procedure. The Procedure TimeBasedSelfPurgingJoin presents the common 3-step procedure, implementing the semantics of a time-window join. Each incoming tuple is compared with the ones currently existing in the window of the opposite stream, possibly allowing the distance of the new tuple from the older ones to be more than Ω_{Time} time units.

Procedure TimeBasedSelfPurgingJoin(t_R)

```
Compare $t_R$ with all $t_S \in W_S$;
Add $t_R$ to $W_R$;
Remove all $t_i \in W_R : t_i.ts < t_R.ts - \Omega_{Time}$;
```

Alternatively, one can update the window the moment a new tuple arrives (cf. Procedure TimeBasedCrossPurgingJoin), comparing each incoming tuple with the ones received from the opposite stream that are not farther than Ω_{Time} time units. In the rest of this paper, we focus on the variant TimeBasedSelfPurgingJoin. Nevertheless, our model covers both variants.

Procedure TimeBasedCrossPurgingJoin(t_R)

```
Remove all $t_i \in W_S : t_i.ts < t_R.ts - \Omega_{Time}$;
Compare $t_R$ with all $t_S \in W_S$;
Add $t_R$ to $W_R$;
```

Figure 1 presents a sample execution for a time-based join. In the example, Ω_{Time} is set to 30 minutes and it is presented how the incoming tuple ⟨08:25,3⟩ is compared with the tuples stored in W_S. In the example, one matching output is produced.

Tuple-based join. The 3-step procedure can be leveraged for tuple-based joins too. A tuple-based window of size Ω_{Tuple} contains the last Ω_{Tuple} tuples received. The Procedure TupleBasedJoin implements the semantics of a tuple-based join.

Procedure TupleBasedJoin(t_R)

```
Compare $t_R$ with all $t_S \in W_S$.;
Add $t_R$ to $W_R$.;
Remove the earliest $t_i \in W_R$ if $|W_R| > \Omega_{Tuple}$.;
```

Deterministic and parallel stream joins. Our model aims at covering features of joins such as deterministic and parallel execution. In the following, we report useful definitions and propositions originally presented in [9].

Definition 3.1. A join implementation is *deterministic* if, given the same sequences of input tuples, the same sequence of output tuples is produced, independently of the inter-arrival times of tuples delivered by different physical streams.

The inter-arrival time of tuples plays an important role in the deterministic execution of a join. Building on the example in Figure 1, we observe that tuple ⟨08:24, 4⟩ is received from S before tuple ⟨08:25, 3⟩. When received, tuple ⟨08:24, 4⟩ is matched against tuple ⟨07:54, 3⟩ (in W_R) and an output tuple is produced. If, nevertheless, tuple ⟨08:24, 4⟩ is received after ⟨08:25, 3⟩ because of a delay, tuple ⟨07:54, 3⟩ is prematurely purged and one matching comparison is missed. The following proposition states how this can be prevented.

PROPOSITION 3.2. *The processing of a sequential join, by means of the TimeBasedSelfPurgingJoin (resp. TupleBasedJoin) procedure, is deterministic if tuples from R and S are processed after they are ready and in timestamp order,*

where a *ready* tuple is defined as follows:

Definition 3.3. Let t_i^j be the i-th tuple from timestamp-sorted stream j. t_i^j is *ready* to be processed if $t_i^j.ts \leq merge_{ts}$, where $merge_{ts} = min_k\{t_x^k.ts\}$ is the minimum timestamp among the timestamps in the set of tuples that includes the latest received tuple t_x^k from each timestamp-sorted stream k.

Definition 3.3 assumes the tuples delivered by each physical stream to be timestamp-sorted. This happens when data sources

[1] We modify the semantics definition to capture both tuple-based and time-based windows, instead of only the latter as in [16].

Variable	Description
Model	
Δt [sec]	Length of the time interval.
K_i [sec]	Time required for the join to carry out the work of time i.
Θ	Quota of Δt Available for the join to carry out the work at any time i.
$w_{i+h,i}$ [sec]	Work that should be carried out at time i but is postponed to time $i + h$.
$\rho_{i+h,i}$ [sec]	Residual of work that should be carried out at time i but is still to be done at the end of time $i + h$.
Join	
W_R, W_S	Windows storing R and S tuples, respectively.
Ω_{Time} [sec]	Size of a time-based window.
Ω_{Tuples} [tup]	Size of a tuple-based window.
ω_i^R, ω_i^S [tup]	Tuples in W_R and W_S at time i, respectively.
α [sec/comp]	Time to perform a comparison.
σ [tup/comp]	Selectivity (# tuples produced per comparison).
β [sec/tup]	Time to output a tuple.
o_i [tup/sec]	Output tuples' rate at time i.
n	Number of processing units.

Variable	Description
Input	
R, S	Logical input streams.
$\lvert R\rvert$, $\lvert S\rvert$	Number of physical R and S streams, respectively.
r_i, s_i [tup/sec]	Arrival rates of logical streams R and S at time i, respectively.
r_i^j, s_i^j [tup/sec]	Arrival rate of the j-th physical stream R or S at time i, respectively.
Output	
c_i [comp]	Number of comparisons that needs to be performed due to tuples that arrived at time i.
y_i [comp]	Throughput, number of comparisons performed at time i.
ℓ_i^{in} [sec]	Latency incurred to process input tuples deterministically at time i.
ℓ_i^{join} [sec]	Latency incurred to match input tuples and produce output tuples at time i.
ℓ_i^{out} [sec]	Latency incurred to produce output tuples deterministically at time i.
ℓ_i [sec]	Overall latency, given by the sum of ℓ_i^{in}, ℓ_i^{join} and ℓ_i^{out}.

Table 1: List of main variables used to model the throughput and latency behavior of time-based and tuple-based joins.

or other upstream streaming operators produce timestamp-sorted streams of tuples [9] or when sorting mechanisms (preceding the join itself) such as [10, 12, 15, 16] are used.

Similarly as in [8], we see that a parallel join implementation remains deterministic if its processing is equivalent to that of a sequential one (as in Proposition 3.2).

PROPOSITION 3.4. *Let J_S be a deterministic sequential join and J_P a parallel join sharing the same predicate and window size Ω_{Time} or Ω_{Tuple}. If J_P, given the same input R and S tuples, (1) runs the same set of comparisons run by J_S and (2) produces the same timestamp-sorted stream of output tuples, then J_P's processing is equivalent (and thus deterministic) to that of J_S.*

In the following, we consider the common parallelization approach used for time-based and tuple-based joins [9, 15, 16], in which multiple *processing units* execute the 3-step procedure running an approximately equal share of the overall comparisons and seeing a certain portion (possibly all) of the tuples delivered by R and S. As discussed in [8, 9, 15], deterministic execution for such a parallelization approach does not only require each processing unit to process *ready* tuples in timestamp order, but also to merge the output tuples produced by each processing unit into a single timestamp-sorted stream of *ready* tuples.

Modeling goal. Our goal is to define a dynamic model to estimate the time-evolving throughput and latency of a time-based or tuple-based join.

Definition 3.5. The throughput y represents the number of comparisons run by the join.

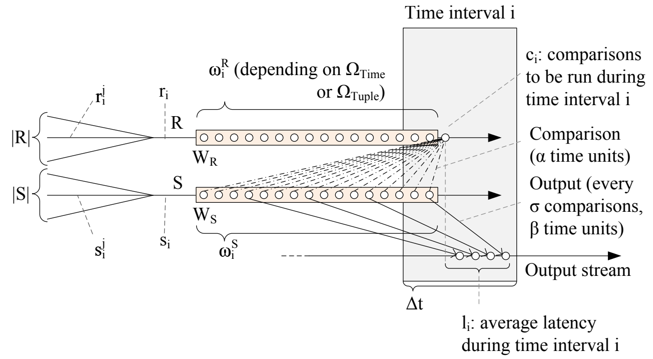

Time interval i
c_i: comparisons to be run during time interval i
ω_i^R (depending on Ω_{Time} or Ω_{Tuple})
r_i^j
r_i
R
|R|
W_R
Comparison (α time units)
|S|
S
Output (every σ comparisons, β time units)
W_S
s_i^j
s_i
ω_i^S
Output stream
Δt
l_i: average latency during time interval i

Figure 2: Visual representation of the variables of a time-based or tuple-based join used in our model.

Definition 3.6. The latency ℓ represents the average time elapsed between the arrival of an input tuple and, for each matching comparison, the output of the resulting tuple.

Based on the dynamic behavior of a join, both the throughput and the latency are to be modeled for a given time period i of duration Δt time units. We thus refer to y_i and ℓ_i as the throughput and latency during the i-th time period, respectively. In the following all the quantities that are functions of time are indicated with subscript i.

Table 1 and Figure 2 present the main variables used in our model. We introduce also the processing quota $\Theta \in (0,1]$, that describes the fraction of the time interval Δt that is dedicated to the join processing. It will be used for modeling the overutilization conditions, as described in the next section. For simplicity, and without loss of generality, we assume that Δt equals 1 second (sec) and Ω_{Time} is the same for R and S when time-based windows are used (similarly Ω_{Tuple} is the same for R and S). This can be easily extended if different window sizes are allowed for W_R and W_S for a time-based or tuple-based join. We also assume that Ω_{Time} is an integer multiple of Δt. In Section 4.5, regarding the case of parallel executions with n processing units, we use the superscript k to refer to a quantity related to the k-th processing unit.

4 THE MODEL

In this section, we build the model for time-based and tuple-based joins. In a modular fashion, we analyse how the throughput and latency behaviors described by the model depend on whether the join is exceeding or not its processing quota, enforcing deterministic execution or not, fed by one physical R stream and one physical S stream or multiple physical streams and run in a centralized or parallel fashion. In the proposed model the overall latency is given by three different contributions as:

$$\ell_i = \ell_i^{\text{in}} + \ell_i^{\text{join}} + \ell_i^{\text{out}} \tag{1}$$

where ℓ^{join} is the latency introduced by the actual execution of the join, as described in Sections 4.1, 4.2 and 4.5, ℓ_i^{in} is the contribution related to the deterministic execution of the join as described in Section 4.3 and 4.4, and ℓ_i^{out} is the contribution related to the parallel execution as described in Section 4.5. Table 2 presents the different setups taken into account in the remainder of the section.

	Sec 4.1	Sec 4.2	Sec 4.3	Sec 4.4	Sec 4.5
Possibly exceeding the processing quota	✗	✓	✓	✓	✓
Determinism	✗	✗	✓	✓	✓
Multiple physical streams	✗	✗	✗	✓	✓
Parallel execution	✗	✗	✗	✗	✓

Table 2: Setup of the join modeling in Sections 4.1-4.5.

4.1 Centralized, non-deterministic join fed by single physical streams, not exceeding the processing quota

In this first section, we model a centralized non-deterministic join that is fed by exactly one physical R and one physical S stream. In order to compute its throughput (Definition 3.5), we begin by modeling the number of tuples contained in windows W_R and W_S at time interval i, namely ω_i^R and ω_i^S. As we mentioned in Section 3, time-based windows define a concrete period of time but do not specify how many tuples they contain. The latter can be computed as the sum of the rates in the last Ω_{Time} time units:

$$\omega_i^R = \sum_{h=i-\Omega_{Time}}^{i} r_h \Delta t, \qquad \omega_i^S = \sum_{h=i-\Omega_{Time}}^{i} s_h \Delta t. \tag{2}$$

On the other hand, tuple-based windows define a concrete number of tuples, but can span an arbitrarily long period of time. They can be modeled as:

$$\begin{aligned} \omega_i^R &= \begin{cases} \sum_{h=0}^{i} r_h \Delta t, & \text{if } \sum_{h=0}^{i} r_h \Delta t < \Omega_{Tuple}. \\ \Omega_{Tuple}, & \text{otherwise.} \end{cases} \\ \omega_i^S &= \begin{cases} \sum_{h=0}^{i} s_h \Delta t, & \text{if } \sum_{h=0}^{i} s_h \Delta t < \Omega_{Tuple}. \\ \Omega_{Tuple}, & \text{otherwise.} \end{cases} \end{aligned} \tag{3}$$

Notice that their size is fixed to Ω_{Tuple} as soon as Ω_{Tuple} tuples have been received from each logical stream.

Given ω_i^R and ω_i^S (both measured in tup), we can compute the total number of comparisons c_i to be run by the join during time interval i (measured in comp). Each incoming tuple from R is compared with all the tuples in window W_S. Similarly, each incoming tuple from S is compared with all the tuples in window W_R. Hence, we can calculate the new comparisons that are introduced in the workload of the join at time interval i as:

$$c_i = \left(\omega_i^S r_i + \omega_i^R s_i\right) \Delta t \tag{4}$$

Notice that the same equation applies both for time-based and tuple-based joins. The time K_i (measured in sec) needed to run all the comparisons introduced at time interval i, is then

$$K_i = c_i \, (\alpha + \sigma\beta) \tag{5}$$

and the assumption that the join does not exceed its processing quota translates to

$$K_i \leq \Theta\Delta t. \tag{6}$$

If $K_i \leq \Theta\Delta t$, the total number of comparisons y_i that are performed in the time interval i coincides with c_i. That is, $y_i = c_i$.

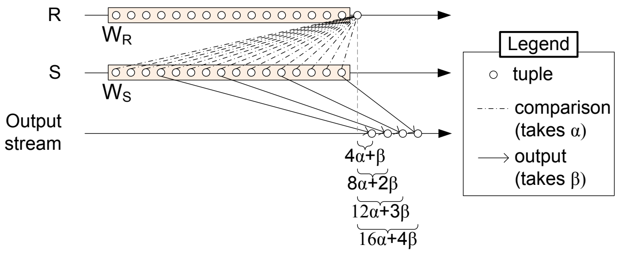


Figure 3: Sample execution showing how an incoming tuple from R is compared against the tuples in W_S and the resulting latency for the output tuples. In the example, σ is equal to 0.25 (i.e., one out of four comparisons results in an output tuple).

The latency (Definition 3.6), is computed based on the output tuples produced during the time interval. As exemplified in Figure 3 for an incoming tuple from R (the discussion holds for tuples from S too), on average σ of the ω_i^S comparisons will result in an output tuple. Before being produced, each output tuple's latency grows proportionally to the number of comparisons run for the preceding tuples stored in the window and the production of output tuples. In the example, the first output tuple incurs a latency of $4\alpha + \beta$, the second output tuple incurs a latency $8\alpha + 2\beta$, and so on.

The sum of all the latency values can be computed as:

$$\sum_{m=1}^{\sigma\omega_i^S} m\left(\frac{\alpha}{\sigma} + \beta\right) = \left(\frac{\alpha}{\sigma} + \beta\right) \frac{\sigma\omega_i^S(\sigma\omega_i^S + 1)}{2} \tag{7}$$

Given that we are interested in modeling the average of (7) with respect to the produced $\sigma\omega_i^S$ output tuples, then:

$$\begin{aligned}\ell^{\text{join}}_{\sigma\omega_i^S} &= \frac{1}{\sigma\omega_i^S}\left(\left(\frac{\alpha}{\sigma}+\beta\right)\frac{\sigma\omega_i^S(\sigma\omega_i^S+1)}{2}\right)\\ &= \frac{\left(\sigma\omega_i^S+1\right)\cdot(\alpha+\sigma\beta)}{2\sigma}\end{aligned} \qquad (8)$$

Equation 8 defines the average latency experienced by tuples produced upon reception of a tuple from stream R. Similarly, $\ell^{\text{join}}_{\sigma\omega_i^R}$, represents the average latency experienced by tuples produced upon reception of a tuple from stream S. The overall latency can be then estimated as the weighted average for the tuples received during the time interval, which leads to:

$$\widehat{\ell^{\text{join}}_i} = \frac{r_i}{r_i+s_i}\ell^{\text{join}}_{\sigma\omega_i^S} + \frac{s_i}{r_i+s_i}\ell^{\text{join}}_{\sigma\omega_i^R}. \qquad (9)$$

The latency is computed as per (1), with $\ell_i^{\text{join}} = \widehat{\ell^{\text{join}}}_i$ and the other two terms equal to zero, since we are here considering no determinism, and no parallelism. The ℓ_i^{join} term will be slightly different in the case of a join exceeding its processing quota (as described in the next section) and this justifies the "hat" notation.

4.2 Centralized, non-deterministic join fed by single physical streams, possibly exceeding the processing quota

In this section we investigate how the model presented in the previous section can be extended in the case of a join exceeding its processing quota. That is, when for some time interval i, the time needed to run the comparisons c_i exceeds the available one:

$$K_i > \Theta\Delta t. \qquad (10)$$

In order to compute the actual amount of work that is performed in a time interval Δt, we shall introduce some quantities. Let us denote with $\rho_{i+h,i}$, the residual work that is still to be done at the end of time $i+h$, due to the comparisons of time i. We can define the work performed at time $i+h$, due to the comparisons of time i as:

$$w_{i+h,i} = \begin{cases} K_i - \rho_{i,i}, & h = 0\\ \rho_{i+h-1,i} - \rho_{i+h,i}, & h > 0\\ 0, & h < 0\end{cases} \qquad (11)$$

Assuming that the tuples are processed in FIFO order, at a generic time $i+h$, the residual work $\rho_{i+h,i}$ that is left to perform due to the comparisons of time i can be computed as the residual work $\rho_{i+h-1,i}$ that was left at time instant $i+h-1$, minus the budget of time that can be used at time $i+h$ to perform the comparisons of time i. The budget of time can be computed as the whole time interval Δt multiplied by the quota Θ minus the time that is needed to perform the comparisons of time $m < i$, that are still pending. Formally:

$$\rho_{i+h,i} = \begin{cases} \left(K_i - \left(\Theta\Delta t - \sum_{m=0}^{i-1} w_{i+h,m}\right)\right)^+ & h = 0\\ \left(\rho_{i+h-1,i} - \left(\Theta\Delta t - \sum_{m=0}^{i-1} w_{i+h,m}\right)\right)^+ & h > 0\\ 0 & h < 0\end{cases} \qquad (12)$$

where $(x)^+ = \max(x{,}0)$.

Defining the total work performed at the end of time i as:

$$w_i = \sum_{m=0}^{i} w_{i,m} \qquad (13)$$

we can compute the latency as:

$$\ell_i^{\text{join}} = \sum_{m=0}^{i}\frac{w_{i,m}}{w_i}\left(\widehat{\ell_m^{\text{join}}} + (i-m)\Delta t\right) \qquad (14)$$

where $\widehat{\ell_m^{\text{join}}}$ is the latency computed as (9). Notice also that when the processing quota is not exceeded, (14) and (9) coincide.

Finally, the number of comparisons performed at time i can be computed as:

$$y_i = \sum_{m=0}^{i}\frac{w_{i,m}}{\alpha+\sigma\beta} = \frac{w_i}{\alpha+\sigma\beta} \qquad (15)$$

As shown by the sample execution presented in Figure 4, the effect of exceeding the processing quota is disruptive for the latency observed by the join.

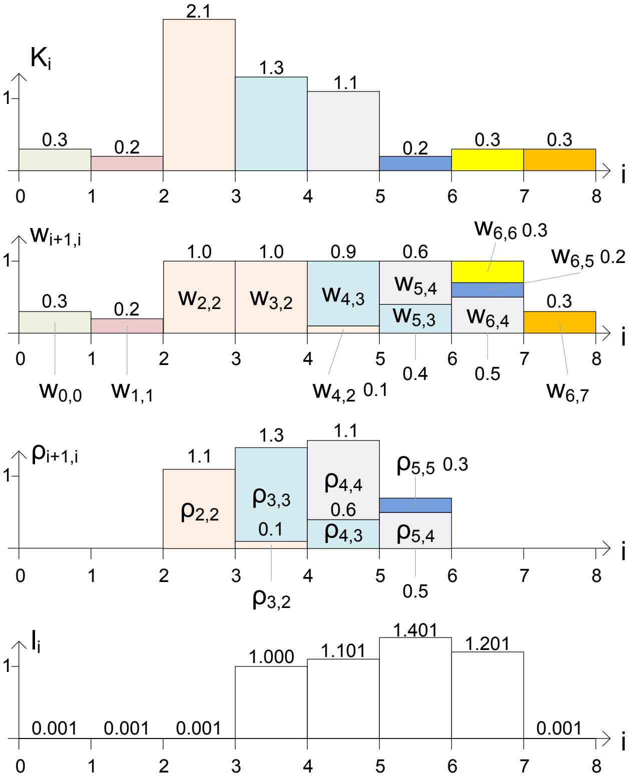
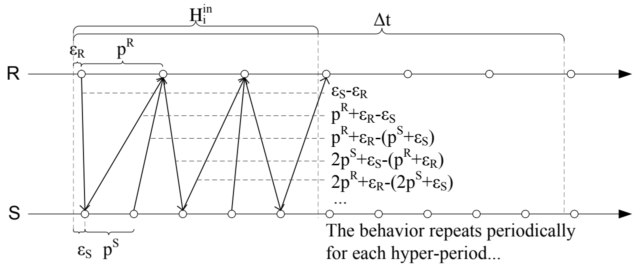


Figure 4: Sample execution showing the values K_i, $\rho_{i+h,i}$, $w_{i+h,i}$ and ℓ_i^{in}. The processing quota 1 is exceeded during the time intervals 2, 3 and 4. In the example, $\widehat{\ell_m^{\text{join}}}$ is equal to 0.001 sec $\forall m$.

4.3 Centralized, deterministic join fed by single physical streams, not exceeding the processing quota

Let us now consider that the centralized join processes tuples deterministically. As we discussed in Section 3, incoming tuples from streams R and S cannot be processed in an arbitrary order (i.e., as soon as they are fed to the join) but need to be processed in timestamp order once *ready* (Definition 3.3).

When enforcing a deterministic execution, we do not alter the number of comparisons y_i run by the join during time interval i, but we expect to observe a higher latency, because of the time each input tuple waits to become *ready*. As we can observe with the help of Figure 5, the time needed for a tuple from R to become *ready* depends on the rate r_s with which tuples are received at S and vice-versa.



H_i^{in}
Δt
ε_R
p^R
R
S
ε_S-ε_R
p^R+ε_R-ε_S
p^R+ε_R-(p^S+ε_S)
$2p^S$+ε_S-(p^R+ε_R)
$2p^R$+ε_R-($2p^S$+ε_S)
...
The behavior repeats periodically for each hyper-period...
ε_S p^S

Figure 5: Example showing the dependencies and latency for R **and** S **tuples to become *ready*. An arrow between two tuples implies the first becomes *ready* once the second is received.**

In the example, 6 tuples are fed from stream R and 10 tuples are fed from stream S. Being p^R and p^S the periods with which R and S tuples are received and ε_R and ε_S (with $\varepsilon_R < \varepsilon_S$, without loss of generality) introduced since R and S streams are not perfectly aligned in a real system, we can observe that the first tuple received from R at ε_R becomes *ready* as soon as the first tuple from S is received, thus incurring a latency of $\varepsilon_S - \varepsilon_R$ time units. The first and second tuples from S, on the other hand, become *ready* as soon as the second tuple from R is received, thus observing a latency of $p^R + \varepsilon_R - \varepsilon_S$ and $p^R + \varepsilon_R - (p^S + \varepsilon_S)$, respectively. Generalizing this behavior, after a certain time interval, the latency pattern becomes periodical, as we discuss in the following. Considering the current arrival rates r_i and s_i, and $p_i^R = {}^1/_{r_i}$ and $p_i^S = {}^1/_{s_i}$, one can compute the current hyper-period H_i of the arrivals as the least common multiple of the two periods:

$$H_i^{\text{in}} = \text{LCM}\left(p_i^R, p_i^S\right) \tag{16}$$

Over a hyper-period, the cumulative latency for stream R's tuples due to the misalignment of arrival time of the tuples is then:

$$\ell_i^{R,\text{in}} = \sum_{m=0}^{H_i^{\text{in}} r_i - 1} \left(p_i^S \left\lceil \frac{mp_i^R + \varepsilon_R}{p_i^S} \right\rceil + \varepsilon_S - \left(mp_i^R + \varepsilon_R\right) \right) \tag{17}$$

where $\lceil x \rceil$ is the ceiling operator for a real number x. Analogously one can define $\ell_i^{S,\text{in}}$, for stream S.

The contribution of the latency ℓ_i^{in} given $\ell_i^{R,\text{in}}$ and $\ell_i^{S,\text{in}}$, can be then computed as the average latency experienced by all the considered tuples in the hyper-period:

$$\ell_i^{\text{in}} = \frac{\ell_i^{R,\text{in}} + \ell_i^{S,\text{in}}}{H_i^{\text{in}}\left(r_i + s_i\right)}. \tag{18}$$

4.4 Centralized, deterministic join fed by multiple physical streams, not exceeding the processing quota

Expanding on Section 4.3, we now assume tuples can be delivered by multiple physical R and S streams. That is, we show how the model can account for multiple physical streams with rates $r_i^{(1)}$, $r_i^{(2)}$, ...,$r_i^{(|R|)}$, and $s_i^{(1)}$, $s_i^{(2)}$, ...,$s_i^{(|S|)}$. As for before, we expect this to have a repercussion on the latency term ℓ_i^{in}.

We denote with r_i and s_i the sums of the incoming tuples for each logical stream, that can be computed as:

$$r_i = \sum_{j=1}^{|R|} r_i^{(j)}, \quad s_i = \sum_{j=1}^{|S|} s_i^{(j)} \tag{19}$$

The number of tuples that fall in the considered window can be just computed as per (2) (for time-based joins), or as per (3) (for tuple-based joins), and the number of comparisons y_i as per (4). The contribution to the latency ℓ_i^{join} is not affected, and can then be computed as per (14). However, the contribution to the latency introduced by determinism must be generalized as follows. Let the set of all the rates of the physical streams be $\Phi_i = \{r_i^{(1)}, r_i^{(2)}, \ldots, r_i^{(|R|)}, s_i^{(1)}, s_i^{(2)}, \ldots, s_i^{(|S|)}\}$. The latency of the j-th physical stream in Φ with rate $\phi_i^{(j)}$ and period p_i^j, can be computed as:

$$\ell_i^{j,\text{in}} = \sum_{m=0}^{H_i^{\text{in}} \phi_i^{(j)} - 1} \max_{x \neq j} \left(p_i^x \left\lceil \frac{mp_i^j + \varepsilon_j}{p_i^x} \right\rceil + \varepsilon_x - \left(mp_i^j + \varepsilon_j\right) \right) \tag{20}$$

Intuitively, the equation chooses for each tuple m from stream j in the hyper-period the stream x whose first tuple coming after m is farther from m than any other physical stream.

The overall contribution is then the average with respect to the total amount of tuples in the hyper-period:

$$\ell_i^{\text{in}} = \frac{1}{H_i^{\text{in}}(r_i + s_i)} \sum_{j=1}^{|\Phi_i|} \ell_i^{j,\text{in}}. \tag{21}$$

Figure 6 shows the dependencies and latency observed for the incoming tuples given the R and S rates in Figure 5 but assuming R tuples are delivered by one physical stream while S tuples are delivered by two.

4.5 Parallel, deterministic join fed by multiple physical streams, not exceeding the processing quota

We now consider the case in which there are $n \geq 1$ processing units. We assume each processing unit runs a fair share of the overall comparisons by comparing each incoming tuple with approximately $1/n$ of the previous tuples stored in W_R or W_S. At the same time, we assume all processing units observe all the tuples delivered by

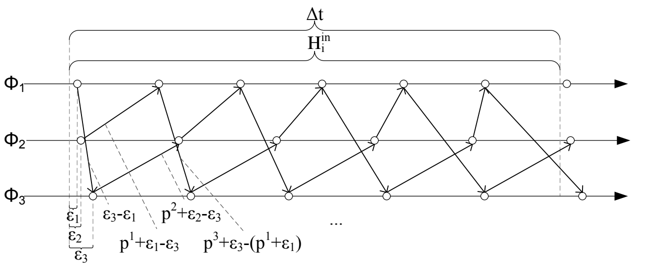

Δt
H_i^{in}
Φ_1
Φ_2
Φ_3
ε_1
ε_2
ε_3
ε_3-ε_1
p^2+ε_2-ε_3
...
p^1+ε_1-ε_3
p^3+ε_3-(p^1+ε_1)

Figure 6: Dependencies and latency observed for the incoming tuples given the R and S rates in Figure 5 but now assuming one physical R stream and two physical S streams. An arrow between two tuples implies the first becomes *ready* once the second is received.

streams R and S (the model can be easily adapted if a portion different from $1/n$ or only a portion of streams R and S is observed by each processing units, as in the case of [8]).

The total number of comparisons c_i^k for the processing unit k at time i can be computed as:

$$c_i^k = \left(\frac{\omega_i^R}{n} s_i + \frac{\omega_i^S}{n} r_i \right) \Delta t. \tag{22}$$

The total number of comparisons to be processed by n processing units is:

$$c_i = \sum_{k=1}^{n} c_i^k = \left(\omega_i^R s_i + \omega_i^S r_i \right) \Delta t,$$

and does not change with respect to the non parallel case.

As for Equation 1, the terms contributing to the latency are:

(1) The latency introduced by the determinism, as per (18), that is not affected by the number of processing units.
(2) The latency introduced by the join, that can be computed as per (14), with $\widehat{\ell_i^{\text{join}}}$ computed as the average latency among the processing units, as:

$$\widehat{\ell_i^{\text{join}}} = \frac{1}{n} \sum_{k=1}^{n} \frac{r_i}{r_i + s_i} \frac{\ell_{\sigma\omega_i^S}^{k,\text{join}}}{n} + \frac{s_i}{r_i + s_i} \frac{\ell_{\sigma\omega_i^R}^{k,\text{join}}}{n} \tag{23}$$

where $\ell_{\sigma\omega_i^S}^{k,\text{join}}$ and $\ell_{\sigma\omega_i^R}^{j,\text{join}}$ are computed based on (8).
(3) The latency introduced by the synchronization of the output tuples ℓ_i^{out}.

This last term, can be computed similarly to (20), considering that each output tuple should be forwarded by the join in timestamp order once *ready* to enforce deterministic processing. We must observe, nevertheless, that the output tuples produced by a processing unit over a certain time interval, $y_i^k\sigma/\Delta t$, when being more than the received input tuples $r_i + s_i$, are not forwarded with a steady period $1/y_i^k\sigma$, but rather in bursts upon the reception of each incoming *ready* tuple.

Based on this observation, we define the output rate of each processing unit as $o_i^k = \min(y_i^k\sigma/\Delta t, r_i + s_i)$, which let us approximate a constant rate for the output stream produced by each processing unit. The hyper-period of the output tuple is then $H_i^{\text{out}} = \text{LCM}(1/o_i^k) = 1/o_i$ (i.e., the period itself). A processing unit k introduces a latency of:

$$\begin{aligned} \ell_i^{k,\text{out}} &= \sum_{m=0}^{H_i^{\text{out}} o_i - 1} \max_{x \neq k} \left(p_x \left\lceil \frac{m p_k + \varepsilon_k}{p_x} \right\rceil + \varepsilon_x - (m p_k + \varepsilon_k) \right) \\ &= \max_{x \neq k} \left(p_x \left\lceil \frac{\varepsilon_k}{p_x} \right\rceil + \varepsilon_x - \varepsilon_k \right) \end{aligned} \tag{24}$$

The average latency can thus be computed as:

$$\ell_i^{\text{out}} = \frac{1}{n} \sum_{k=1}^{n} \ell_i^{k,\text{out}} \tag{25}$$

5 EVALUATION

We study in this section how the proposed model reflects the behavior of a join comparing it with the throughput and latency of a running implementation. We first present the evaluation setup and then compare the model and the running implementation under the different assumptions discussed in Section 4.

Evaluation setup. We run experiments on a server equipped with a 2.0 GHz Intel Xeon E5-2650 (16 cores over 2 sockets) and 64 GB of memory. The 3-step procedure introduced in Section 3 is implemented in Java and run with Java version 1.7.0_95 (IcedTea 2.6.4). Each processing unit runs the 3-step procedure with a dedicated thread. Extra dedicated threads are instantiated to inject input tuples into the input queues shared by the processing units, to collect the output tuples produced by the processing units (once *ready*) and to maintain the statistics later matched against the model. As in [9], when multiple processing units are run in parallel, each stores approximately $1/n$ of the previous tuples in W_R and W_S and thus runs $1/n$ of the overall comparisons. The model simulator has been implemented in Python. The code for the Java implementation and the Python simulator can be found at https://github.com/dcs-chalmers/Join_Model. The presented results are averaged over 10 runs for each experiment. In the following graphs, besides the throughput and latency evolution, we quantify the absolute percentage error between the model and the implementation with a box plot (reporting the value of the median and the lower and upper whiskers).

Evaluation benchmark. We evaluate our model using the common benchmark also used by CellJoin [6], Handshake joins [15, 16] and ScaleJoin [9]. In the benchmark we run, R tuples are composed by attributes $\langle ts,x,y \rangle$, where x,y are of types `int`, `float`. S tuples are composed by attributes $\langle ts,a,b,c,d \rangle$, where a,b,c,d are of types `int`, `float`, `double` and `bool`. A tuple $\langle ts,x,y,a,b,c,d \rangle$ is outputted for each pair of tuples t_R, t_S such that:

$$t_R.x \geq t_S.a - 10 \text{ AND } t_R.x \leq t_S.a + 10 \text{ AND } t_R.y \geq t_S.b - 10 \text{ AND } t_R.y \leq t_S.b + 10$$

We draw values for attributes x,y,a,b from a uniform distribution in the interval $[1 - 200]$. Approximately 1 out of each 100 comparisons results in an output tuple, on average (i.e., $\sigma = 0.01$). The values for α and β used by the simulator are measured and averaged during the implementation run for each experiment.

The time-based join defines a window size Ω_{Time} of 60 seconds while the tuple-based join defines a window size Ω_{Tuple} of 8400 tuples. The sizes have been chosen so that during the initial phase of

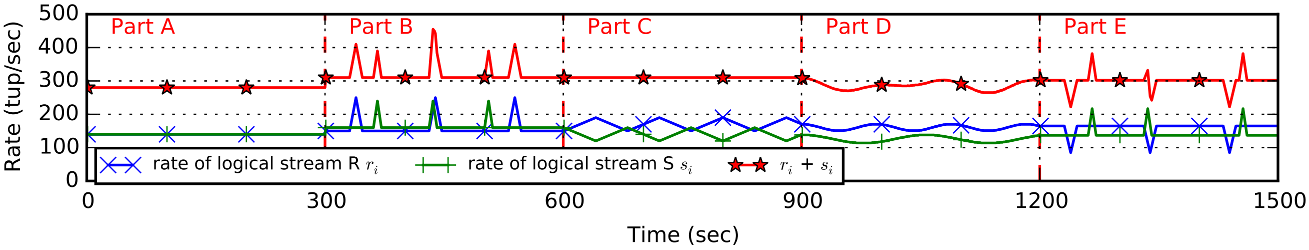


Figure 7: Behavior of the logical streams R's and S' rates during each experiment.

each experiment the time-based join and the tuple-based join observe the same number of tuples in their windows W_R and W_S and allow for their behaviors to be compared.

Behavior of the logical streams R *and* S. Figure 7 presents the rates for streams R and S and their sum. Each experiment is 25 minutes long and composed of 5 parts of 5 minutes each. The different behaviors observed in each part have been chosen to stress the model under different circumstances. Part A: both streams observe the same constant rate of 140 tup/sec. Part B: both streams observe a constant but distinct rate (150 tup/sec for R and 160 tup/sec for S) with both aligned and non-aligned peaks appearing at different positions. Each peak lasts 30 seconds and increases R's rate by 100 tup/sec and S' rate by 80 tup/sec. Part C: streams R and S follow triangle shapes with opposite phases that sum up to a constant rate. Part D: R and S rates follow a sinusoidal pattern with different periodicities. Part E: streams R and S observe again constant rates with peaks (both non-aligned and aligned) but now negative for stream R.

Centralized non-deterministic join fed by single physical streams not exceeding the processing quota

As in Section 4.1, we assume here that deterministic processing is not enforced, only one R and S physical stream deliver tuples and that the execution never exceeds the quota. The expected throughput is given by Equation 4 while the latency by the term in Equation 9.

Figure 8 presents the throughput in millions of comp/sec for the time-based and tuple-based join, respectively. The throughputs observed for the implementation and the model match perfectly and their behavior resembles that of the overall incoming rate (Figure 7). While the throughput is the same for the time-based join and the tuple-based join during part A, it can be observed that the throughput for the time-based join is then more fluctuating than the one for the time-based join. That is expected, given that a varying rate, for a time-based window, does not only result in a varying number of incoming tuples to compare (Equation 4) but also in a varying number of tuples in windows W_R and W_S to be compared with (Equation 2).

Figure 9 presents the evolution of the latency over time. In this case too, the values reported by the model and the implementation match, despite some minor fluctuations and a slightly higher latency (up to approximately 0.03 ms) caused by other tasks in the running environment of the prototype (e.g., the operating system) which are not captured by the model and result in a median percentage error between 6% and 7%. For both the time-based and tuple-based joins the latency increases in the first 60 seconds since windows W_R and W_S are being filled. Once the windows are full for the tuple-based

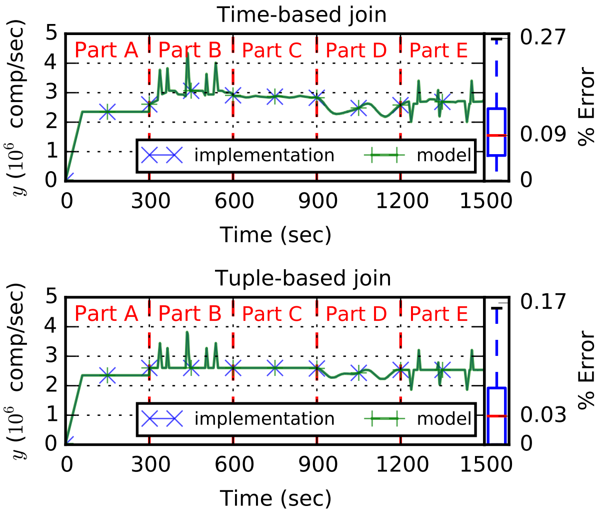


Figure 8: Throughput for centralized, non-deterministic time-based and tuple-based joins fed by one physical R stream and one physical S stream and not exceeding the processing quota.

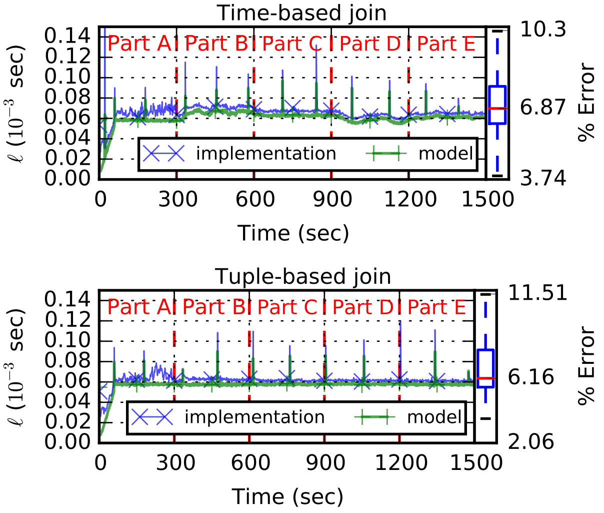


Figure 9: Latency for centralized, non-deterministic time-based and tuple-based joins fed by one physical R stream and one physical S stream and not exceeding the processing quota.

join, the observed latency remains overall constant. This does not happen for the time-based join, since the varying rates result in a varying number of tuples contained in W_R and W_S. As it can be observed for the time-based join, the latency behavior resembles the one of the overall $r_i + s_i$ rate, which is expected, given that the number of comparisons performed upon reception of a tuple depends on the rates observed so far (Equation 4).

Centralized non-deterministic join fed by single physical streams exceeding the processing quota

In this section, we evaluate the join modeled in Section 4.2 by choosing a quota Θ that is exceeded by the execution of the time-based and tuple-based joins. As we show in the following, exceeding the processing threshold has a disruptive effect on the latency. For this reason, we set the quota to approximately 0.04 since such value is only exceeded during the peaks appearing in part B on R's and S' rates (between second 300 and 600).

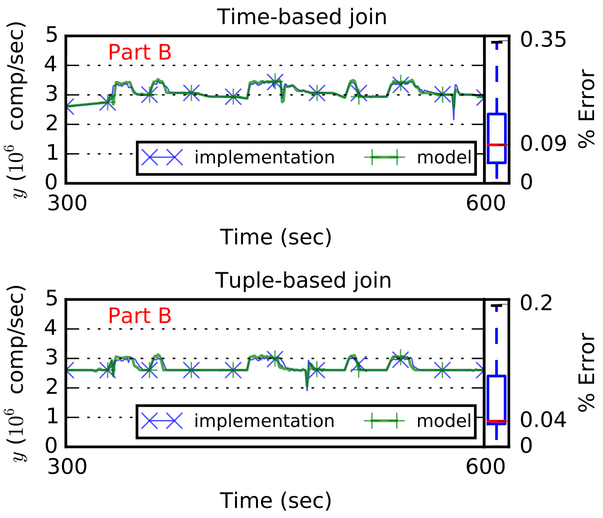


Figure 10: Throughput for centralized, non-deterministic time-based and tuple-based joins fed by one physical R stream and one physical S stream that exceed the processing quota.

Figure 10 shows the behavior of the throughput during part B (when the processing quota is exceeded). Also in this case the implementation and the model behaviors match. As expected, for both the time-based and the tuple-based join, the throughput is truncated in correspondence of the highest peaks. The latency behavior (modeled by Equation 14) is presented in Figure 11 (in logarithmic scale to better appreciate it). As shown by the matching curves of the model and the implementation (which also in this case show a median percentage error of 6%, approximately), the latency changes from a fraction of millisecond to whole seconds, thus incurring an increase of 4 orders of magnitude, in correspondence with the truncated peaks. This gives evidence of the disruptive effect observed for the latency when the join load exceeds its available quota. In the remainder of the evaluation, we chose a quota that is not exceeded so that the order of magnitude of the different latency terms contributing to the overall latency (Equation 1) is closer and can be thus compared.

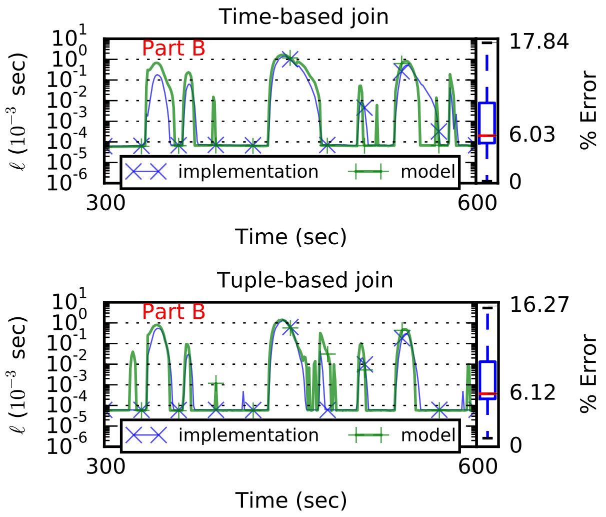


Figure 11: Latency for centralized, non-deterministic time-based and tuple-based joins fed by one physical R stream and one physical S stream that exceed the processing quota.

Centralized deterministic join fed by single physical streams not exceeding the processing quota

In this section, we follow the model described in Section 4.3, which enforces determinism for the centralized join. Since in this model the throughput is not modified (Equation 4), we only discuss the joins' latency. With respect to the latter, we expect to observe an increase due to the introduction of the term ℓ_i^{in} (Equation 18). Figure 12 presents the latency behavior. Given Equation 1, the latency is composed by terms ℓ_i^{in} and ℓ_i^{join}. As it can be observed, term ℓ_i^{in} dominates over term ℓ_i^{join} and makes the behavior of both the time-based and the tuple-based joins practically equivalent, also resulting in a median percentage error lower than 1%. The increase due to determinism is of one order of magnitude, from 0.15 ms to 4 ms, approximately. Notice that the latency behavior of term ℓ_i^{in} is opposite to that of the overall incoming rate $r_i + s_i$. This is as expected, since an increasing rate (resp. decreasing rate) results in a lower time (resp. longer time) for each incoming tuple to become *ready* (Definition 3.3), as modeled by Equation 18.

Centralized deterministic join fed by multiple physical streams not exceeding the processing quota

We relax in this section the assumption that exactly one physical R and S streams are delivering tuples to the time-based and tuple-based joins. In the experiment, we define 3 physical streams for logical stream R, each delivering 1/3 of r_i, and 2 physical streams for logical stream S, each delivering 1/2 of s_i, for a total of 5 physical input streams. As discussed in Section 4.4, the latency term ℓ_i^{in} is now modeled as in Equation 21. Figure 13 presents the evolution of the latency for the time-based and tuple-based join. As it can be seen, the behavior resembles the one in Figure 12 but is shifted up to approximately 15 ms. In these experiments, the mis-alignment between the various physical input streams (modeled by the ϵ variables in

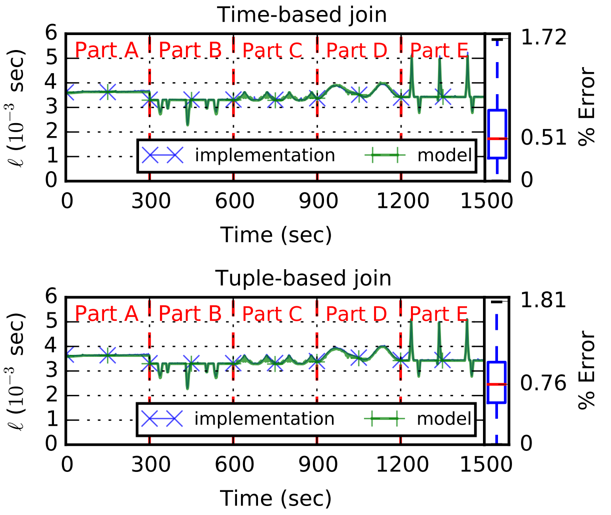


Figure 12: Latency for centralized, deterministic time-based and tuple-based joins fed by one physical R stream and one physical S stream and not exceeding the processing quota.

Equation 20) can result in the model correctly estimating the join behavior but slightly overestimating its absolute value. The mismatch results in a median percentage error of 5%, approximately.

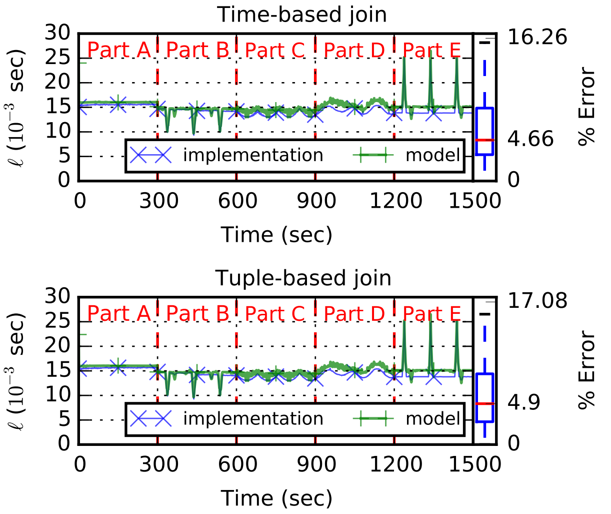


Figure 13: Latency for centralized, deterministic time-based and tuple-based joins fed by multiple physical R and S streams and not exceeding the processing quota.

This section's results highlight an important aspect: the need for properly provisioned operators feeding a join. While resorting to over-provisioning for an operator to avoid it from exceeding its processing quota might seem an intuitive way of coping with the load and observing low latency, the model can quantify the non-negligible price this strategy incurs, especially for an over-provisioned application in which load peaks are observed sporadically.

Parallel deterministic join fed by multiple physical streams not exceeding the processing quota

In this last section, we consider a parallel execution of the join (i.e., with $n > 1$). As modeled in Section 4.5, the execution of the 3-step procedure by multiple processing units in parallel has 2 consequences on the overall latency. On one hand, the term ℓ_i^{join} decreases because each processing unit maintains less tuples and consequently observes a lower latency in scanning them (Equation 23). On the other hand, the term ℓ_i^{out} is introduced because of determinism, since tuples produced by each processing unit are forwarded only when *ready*. How the total latency ℓ_i behaves for a parallel execution depends on which of the two terms dominates the other.

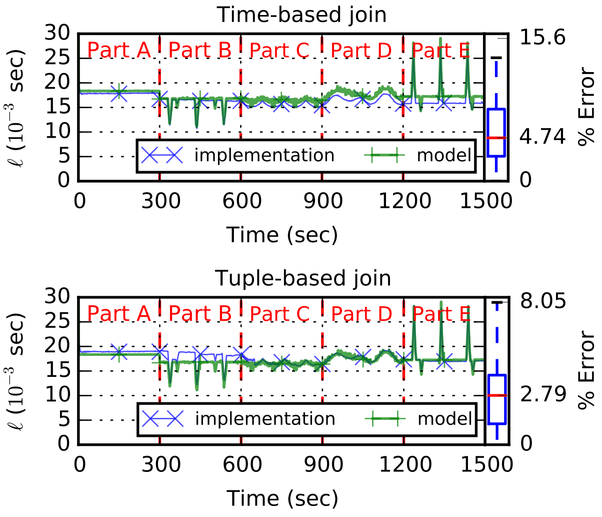


Figure 14: Latency for parallel, deterministic time-based and tuple-based joins fed by multiple physical R and S streams and not exceeding the processing quota.

Figure 14 shows the result for n set to 3 processing units. As it can be observed, we experience a further increase in the overall latency of approximately 2.5 ms. This is because the term ℓ_i^{out} dominates over the term ℓ_i^{join}, as shown in Figure 14, where the behavior of the 3 components ℓ_i^{in}, ℓ_i^{join} and ℓ_i^{out} is presented (in logarithmic scale) for the time-based and tuple-based join. As it can be seen, term ℓ_i^{join} is approximately six times smaller than the one previously observed in Figure 9. At the same time, the latency introduced by ℓ_i^{out} is two orders of magnitude greater than the one of ℓ_i^{join}. As a result, the overall latency grows. Also in this case, the mis-alignment between the multiple physical input streams and, in this case, the one between the physical output streams of each processing unit can result in the model correctly estimating the latency's behavior but slightly overestimating or underestimating its absolute value. The median percentage error is measured between 3% and 5%, approximately.

The results in Figures 14 and 15 highlight a second important aspect: the need for properly provisioned joins. As for the upstream operator feeding a join, resorting to over-provisioning to avoid a join from exceeding its processing quota might seem an intuitive way of coping with the load and observe low latency, but this holds only as

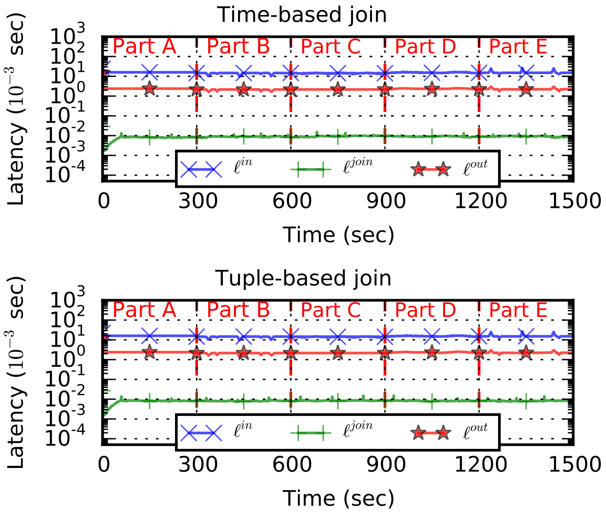


Figure 15: Detail for the different latency terms for parallel, deterministic time-based and tuple-based joins fed by multiple physical streams and not exceeding the processing quota.

long as the higher number of processing units and the improvement over ℓ_i^{join} is higher than the price introduced by ℓ_i^{out}.

6 CONCLUSIONS

In this paper, we presented a model that describes how the throughput and latency of stream joins evolve over time based both on the operator's configuration and its deployment characteristics. Our model covers a wide spectrum of configuration options and deployment characteristics: the possibility for the stream join to exceed its processing quota, its non-deterministic versus its deterministic execution, the existence of multiple physical streams delivering its input tuples and its centralized versus parallel execution. The model identifies the impact of each of the above aspects and discusses its contribution to the overall throughput and latency in a modular fashion. The accuracy of the model is also checked against an extensive empirical evaluation. Interesting future directions include extending the model towards covering worst-case scenarios and tail-latency effects, as well as extending the validation against join implementations in major stream processing engines. While the used join implementation is a parallelization of the 3-step procedure [11] (which is also the case for [6, 9, 15, 16]), extending and validating the model against alternative stream joins is another interesting direction, especially with respect to hardware-aware designs. Nevertheless, we expect factors like determinism to be the dominant costs with respect to latency.

We believe this model can be useful to tune applications leveraging stream joins and study their performance analytically before they are deployed (rather than just empirically once deployed). The model can be also leveraged to replace the empirical measurements and heuristics (which can fail in capturing challenging dependencies such as the ones described in this work between determinism and parallelism) in online adaptive schemes. Finally, analytical models for more operators may provide a framework for a priori studying different deployments of real-world streaming applications.

7 ACKNOWLEDGMENTS

This work was supported by the Swedish Foundation for Strategic Research under the project "Future factories in the cloud (FiC)" with grant number GMT14-0032 and by the Swedish Research Council (Vetenskapsrådet) project "HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures" Contract nr. 2016-03800.

REFERENCES

[1] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. 2013. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 international conference on Management of data*. ACM, New York, NY, USA, 577–588.

[2] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. 2013. Modeling the execution semantics of stream processing engines with SECRET. *The VLDB Journal* 22, 4 (2013), 421–446. DOI:https://doi.org/10.1007/s00778-012-0297-3

[3] Luping Ding and Elke A. Rundensteiner. 2004. Evaluating Window Joins over Punctuated Streams. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*. ACM, New York, NY, USA, 98–107.

[4] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *Proceedings of the VLDB Endowment* 7, 6 (2014), 441–452.

[5] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2016. *Data Stream Mgement: Processing High-Speed Data Streams*. Springer.

[6] Buğra Gedik, Rajesh R Bordawekar, and S Yu Philip. 2009. CellJoin: a parallel stream join operator for the cell processor. *The VLDB journal* 18, 2 (2009), 501–519.

[7] Bugra Gedik, Kun-Lung Wu, S Yu Philip, and Ling Liu. 2007. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding. *IEEE Transactions on Knowledge and Data Engineering* 19, 10 (2007), 1363–1380.

[8] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (2012), 2351–2365.

[9] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2016. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Transactions on Big Data* PP, 99 (2016).

[10] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-Driven Continuous Query Execution over Out-of-Order Data Streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 889–894.

[11] Jaewoo Kang, Jeffrey F Naughton, and Stratis D Viglas. 2003. Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 341–352.

[12] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 311–322.

[13] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable Distributed Stream Join Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 811–825.

[14] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 493–505.

[15] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-Latency Handshake Join. *Proceedings of the VLDB Endowment* 7, 9 (2014), 709–720.

[16] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, New York, NY, USA, 625–636.

[17] Song Wang and Elke Rundensteiner. 2009. Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-slicing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, New York, NY, USA, 299–310.