PECJ: Approximate Window Join on Disorder Data Streams with Proactive Error Compensation

Anonymous Author(s)

ABSTRACT

Stream Window Join (*SWJ*), a vital operation in stream analytics, struggles with achieving a balance between accuracy and latency due to common out-of-order data arrivals. Existing methods predominantly rely on adaptive buffering, but often fall short in performance, thereby constraining practical applications. We introduce *PECJ*, a solution that *proactively* incorporates unobserved data to enhance accuracy while reducing latency, thus requiring robust predictive modeling of evolving data streams. Central to *PECJ* is a mathematical formulation of the *posterior distribution approximation* (**PDA**) problem using *variational inference* (**VI**), which sidesteps error propagation while adhering to the low-latency requirements of *SWJ*. We detail the implementation of *PECJ*, striking a balance between complexity and generality, and discuss both analytical and machine learning based **VI** instantiations. Experimental evaluations reveal *PECJ*'s superior performance. The successful integration of *PECJ* into a multi-threaded *SWJ* benchmark testbed further establishes its practical value, demonstrating promising advancements in enhancing data stream processing capabilities amidst out-of-order data.

ACM Reference Format:

Anonymous Author(s) . 2023. *PECJ*: Approximate Window Join on Disorder Data Streams with Proactive Error Compensation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Stream Window Join (*SWJ*) is a procedure for merging two input streams across discrete, finite subsets, referred to as 'windows', of infinite streams. As a cornerstone of data stream analytics [38], *SWJ* differs from traditional relational join operations. Instead of waiting for the complete input data to become available, *SWJ* faces the challenge of generating join results instantaneously. This real-time processing is a consequence of *SWJ*'s pivotal role across various sectors, such as financial markets [12], fraud detection systems [2], and sensor networks [25].

The emergence of machine learning applications like online decision augmentation (OLDA)[1, 37] has amplified the importance of *SWJ*. These applications leverage *SWJ* to amalgamate dynamic features, such as short-term user behavior, within time-bound windows, thereby facilitating quick downstream feature computations and continual model updates. Especially in OLDA, certain banking applications impose strict end-to-end latency as low as 20*ms* [37], stretching the limits of latency requirements. Such stringent constraints underscore the insufficiency of traditional batch-based approaches to handle *SWJ*, as they require the full input data to be present before commencing processing, resulting in increased latency.

SWJ is complicated by the disorderly arrival of input tuples from streams, primarily due to factors like network delays and parallel processing [5, 6, 8]. The management of these disordered data streams typically involves buffering input data [15, 16], providing a more comprehensive view of *in − window* data, thereby facilitating higher accuracy results from running *SWJ* directly on potentially disordered data streams. However, the extended buffering time needed to gain this comprehensive view frequently incurs substantial latency costs. These costs become particularly pronounced when waiting for straggling tuples, a situation exacerbated by the non-linear nature of *SWJ* [15, 38].

We propose a novel solution to these issues: the *PECJ*[1] algorithm, designed to proactively manage disordered data streams. Contrasting with existing methods that solely depend on already-arrived data (i.e., in-window data), *PECJ* actively takes into account the contributions of future, disordered data to enhance join accuracy. This innovative approach to disorder management introduces a promising avenue for achieving significant accuracy enhancements without corresponding increases in latency. Notably, while subjects such as disorder handling parallelization [18, 24, 39] and efficient buffer structures [10] have been thoroughly explored in prior studies, these aspects are orthogonal to our work.

Aware of the inherent challenge in *PECJ*'s approach—predicting the evolution of data streams—we designed *PECJ* to follow three conceptual steps of mathematical formulation and implementation. In the *first step*, we abstract disorder *SWJ* handling into the *posterior distribution approximation* (**PDA**) problem, formulating its probability model under data stream scenarios. This method, in contrast with individual prediction of each unseen data point as in traditional time series [32, 34], estimates the overall contributions of all unobserved data, bypassing error propagation inherent in individual predictions.

In the *second step*, we optimize the parameterization process of the probabilistic model. Instead of applying brute-force parameterization—which is not only infeasible but also compromises *SWJ*'s low latency requirements—we employ the 'variational inference (**VI**)' approach [14, 32] to minimize overhead. **VI** enables efficient and continual model parameterization throughout **PDA**. For the *third step*, we implement the previously mentioned steps of mathematical formulation through both analytical and machine learning-based **VI** instantiations. The analytical instantiation (*analytical*) suited to simpler stream

[1] *PECJ*: Proactive Error Compensation-Join

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

dynamics scenarios, offering a low-overhead solution of linear equations. The machine learning (ML) based instantiation (*learning*), conversely, addresses more complex situations, using neural networks to approximate the posterior distribution.

We assess the viability of the *PECJ* algorithm by executing a series of experiments that underline its superior performance compared to several existing solutions [8, 15]. Additionally, we validate its effectiveness in AllianceDB, a multi-threaded *SWJ* benchmark testbed [38], showcasing an enhancement in managing out-of-order processing errors without compromising scalability. In summary, this paper provides the following contributions:

- Section 3 introduces the *PECJ* algorithm, tailored to balance both accuracy and latency in *SWJ* operations amid disordered data. The distinct advantage of *PECJ* lies in its proactive approach of incorporating the impact of yet-to-be-seen data for join error compensation.
- In Section 4, we delve into the mathematical formulation of how *PECJ* addresses the challenge of forecasting the future of evolving data streams. The disorder *SWJ* handling is initially abstracted into a posterior distribution approximation (**PDA**) problem, which is followed by optimizing the parameterization of its probability model via variational inference (**VI**).
- Section 5 presents two practical implementations of *PECJ*, demonstrating its adaptability. We begin with a straightforward, analytical **VI** instantiation (*analytical*) suitable for less dynamic streams and gradually progress to a more generalized form (*learning*) that employs machine learning for handling complex stream dynamics.
- Our experimental results, highlighted in Section 6, offer a comprehensive comparison between *PECJ* and the existing state-of-the-art methods. We provide data from both standalone tests and system integration tests, underscoring the superior performance of *PECJ*.

2 PRELIMINARY

This section provides a detailed introduction to Stream Window Join (*SWJ*), including the buffering mechanisms for handling disorder prevalent in existing research. We underline the fundamental distinction that *PECJ* introduces and briefly touch upon the technical challenges posed by our approach.

2.1 Stream Window Join and Key Definitions

Table 1 summarizes the notations used in this paper. For the purposes of this paper, we define a *tuple* y as $y = \tau_{event}, \kappa, v, \tau_{arrival}, \tau_{emit}$, where τ_{event}, κ, and v represent the event timestamp, key, and payload of the tuple, respectively. The tuple's arrival time at a system is denoted by $\tau_{arrival}$, while τ_{emit} signifies the moment the final result incorporating y is released to the user. An input stream, referred to as R or S, is a sequence of tuples arriving at the system (e.g., a query processor), which may arrive out-of-order with respect to their event timestamp.

We adopt the *windows* concept from [38] to perform infinite stream joins over limited subsets. A ***window*** is an arbitrary time range ($t1 \sim t2$), represented as $\mathbb{W} = [t1, t2]$. A tuple y belongs to $\mathbb{W}$ if its t_e falls within the $\mathbb{W}$ range. To denote the length of the window, we use $|\mathbb{W}|$. There are various types of *SWJ* operations, such as intra-window join [38], online interval-join [37], and sliding window join [27, 30] (e.g., sliding, tumbling, interval, etc.). In this paper, we use the intra-window join operation as an example for evaluating *PECJ*. However, *PECJ* can be readily adapted for other types of *SWJ*.

Table 1: Notations used in this paper

Notations	Description
R, S	Two input streams to join
$\mathbb{W}$	A bounded subset of data stream to join
κ	Key of a tuple
v	Payload of a tuple
τ_{event}	The time of event occurrence of an input tuple
$\tau_{arrival}$	The input tuple arrival time
τ_{emit}	The time to emit an output tuple
δ	The delay from event occurrence (τ_{event}) to event arrival ($\tau_{arrival}$) of an input tuple
Δ	Maximum delay among all events from the time of occurrence (τ_{event}) to the time of arrival ($\tau_{arrival}$). $\Delta = \max_{\forall i}(\tau_{arrival} - \tau_{event})$
O	The aggregated results of $R \bowtie_{\mathbb{W}} S$
l	The processing latency
ϵ	The relative error of output
α	The average payload of joined tuples
ω	The assumed time point of window completeness
n	The number of tuples
σ	The join selectivity, as defined by [15]
μ_w	A global variable for describing window-averaged contribution
φ_w	A variable for describing other global information of a window
$\bar{r}_n$	Window-averaged tuple rate corresponding to n
U	The set of global variables, including the interested μ_w and φ_w
X	The set of observations made on acquired tuples
Z	The set of latent variables

For given input streams R and S and a window $\mathbb{W}$, the intra-window join, hereafter simply refer to as *SWJ*, is denoted as $R \bowtie_{\mathbb{W}} S = (r \cup s) | r \in R, s \in S, r \in \mathbb{W}, s \in \mathbb{W}$. The results of $R \bowtie_{\mathbb{W}} S$ are condensed into a scalar value O, either by tallying the number of joined $(r \cup s)$ tuples or by computing a summation or average over the $R.v$ and $S.v$. When O is finally dispatched to the user at the time point τ_{emit}, we consider the following two performance metrics for stream window join:

- **Accuracy**: This metric is evaluated by investigating O and measured by its relative error ϵ. Specifically, $\epsilon = \frac{|O^{opr} - O^{exp}|}{O^{exp}}$, where O^{opr} is the aggregated value generated by an algorithm and O^{exp} is the expected value.
- **Latency**: For all tuples involved in generating O, we assign their τ_{emit} as when O is generated and compute the latency l for each tuple as $l = \tau_{emit} - \tau_{arrival}$.

2.2 Shortcomings of Present Methods

In an optimal scenario for *SWJ*, data arrives *in order*—that is, the sequence based on τ_{event} coincides perfectly with the sequence based on $\tau_{arrival}$, as depicted in Figure 1(a). Here, the window to be calculated is complete, with all data fully visible to the system. However, when the sequence determined by $\tau_{arrival}$ deviates from that defined by τ_{event}, we encounter a *disordered* arrival. In this scenario, guaranteeing window completeness becomes difficult, and usually some data remains unobserved (e.g., the disordered tuples $R1$ and $S2$ arriving significantly later, as shown in red). Neglecting to join such unobserved data inevitably leads to inaccurate results.

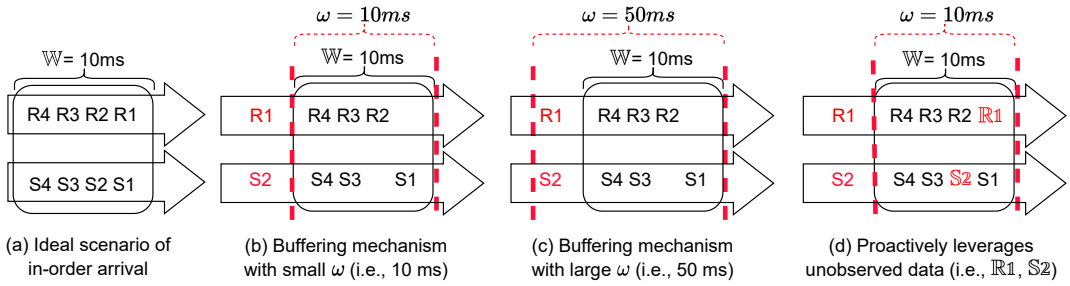


Figure 1: Disorder handling of *SWJ* ($R \bowtie_{|\mathbb{W}|=10ms} S$)

On the other hand, waiting for this tardy data to arrive causes an unpredictable increase in processing latency as the arrival times of these late tuples remain unknown.

Existing methodologies attempt to combat disordered arrivals using a *buffering mechanism*—where observed data is retained in buffers while the system awaits a more complete set of window data. The longer the system waits, the fewer unobserved data points there are. To prevent infinite waiting, these systems often designate a certain point in time, ω, at which they assume the window is complete and all data has been observed, marking the end of data buffering. In essence, these systems assume no more data within the window will arrive later, and the result O is then emitted at τ_{emit}, where τ_{emit} equals ω plus the *processing time*. Given that ω is generally much smaller than the $\tau_{arrival}$ of late tuples, it effectively decreases the overall processing latency. Previous studies [8, 15, 16, 39] have proposed both explicit and implicit methodologies for determining ω.

While these methodologies enable potentially autonomous and flexible determination of ω, they often overlook the influence of *unobserved data*—the data arriving after ω —on the results. For instance, in Figure 1(b), $R1$ and $S2$ are missed with a ω of $10ms$, resulting in an inaccurate output. To rectify this inaccuracy, ω can be extended to ensure $R1$ and $S2$ are observed, as demonstrated by the $50ms$ ω in Figure 1(c). However, extending ω from $10ms$ to $50ms$ leads to significantly increased latency, establishing an unavoidable sub-optimal trade-off between accuracy and latency.

This dilemma—ignoring unobserved data—brings us to propose our solution, *PECJ*. *PECJ* endeavors to enhance *SWJ* by considering unobserved data during processing. By actively integrating unobserved tuples for error compensation prior to their arrival, *PECJ* (as depicted in Figure 1(d)) achieves far greater accuracy under the same ω compared to Figure 1(b). Furthermore, it accomplishes this without needing to increase ω as in Figure 1(c), or worsening the problem of increased latency.

2.3 Challenges of Implementing *PECJ*

The active integration of unobserved data contributions forms a key tenet of *PECJ* and its pursuit of superior results. This integration, however, is not without significant challenges, primarily due to the need to *forecast the evolution of data streams*.

An intuitive approach may suggest the prediction of contributions from each unseen data point, following the principles of time series prediction techniques [34], before their integration. This approach, however, could yield varying degrees of accuracy, as the uncertain count and distinct contributions of tuples within a window can lead to error propagation. Notably, time series prediction, which aims to predict attributes of a *specific and predetermined number* of future data points, becomes increasingly complex with data length, growing super linearly [34]. This complexity translates into substantial prediction overhead, particularly when a significant number of tuples remain unobserved.

Consequently, applying traditional predictive models such as time series analysis directly faces significant challenges. Overcoming these hurdles necessitates a more sophisticated approach — one capable of accurately considering unseen data while simultaneously maintaining computational overhead and latency within acceptable thresholds. This requirement formed the genesis for the development of *PECJ*, a novel strategy engineered to enhance *SWJ* through proactive integration of unobserved tuples. We delve into the design of *PECJ* in the subsequent sections.

3 THE *PECJ* ALGORITHM

This section presents an overview of the *PECJ* algorithm.

3.1 Conceptual Framework of *PECJ*

Designed to actively incorporate unobserved data, *PECJ* compensates for errors that arise in *SWJ* when dealing with disordered data streams. This subsection outlines the conceptual framework of *PECJ*, as depicted in Figure 2.

Abstraction: The first step in the *PECJ* algorithm involves directly abstracting the accurate *SWJ* result by extracting essential information from the disordered data streams to limit error propagation during per-tuple estimation (discussed in detail in Section 4.1). This phase essentially constitutes a *posterior distribution approximation (*`PDA`*)* problem, requiring the development of a probability model that is sensitive to the data streams.

Optimization: Given the inherent challenges of efficient `PDA` parameterization, we turn to the *variational inference* (`VI`) approach for theoretical optimization (explained in Section 4.2). This approach drastically reduces the overhead of `PDA` parameterization, compared to brute force methods that involve possibly unmanageable summations and integrations, and inherently facilitates the evolution of the probability model in parallel with the data streams.

Implementation: Bridging the gap between the mathematical formulations of the previous stages and practical application,

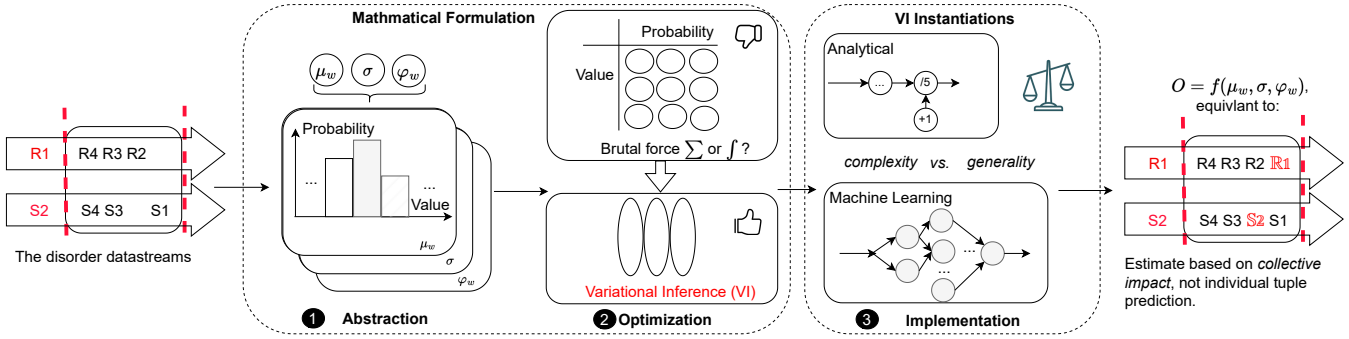


Figure 2: Conceptual framework of *PECJ*.

we provide both analytical (*analytical*) and ML-based (*learning*) implementations of **VI** in Section 5. The *analytical* implementation (Section 5.1) offers ultra-low overhead while accommodating relatively straightforward stream dynamics. Thanks to its linear **VI** solution, we implement it using both Stochastic Variational Inference (SVI) iterations and an Adaptive Linear Filter (ALF) in our code base. For efficiency, we prefer the ALF due to its lower complexity compared to SVI. The *learning* implementation (Section 5.2) seeks to depict various stream dynamics in a more generalized manner. We accomplish this by incorporating **VI** principles with *PECJ*'s parameters of interest to formulate a loss function and use a simple Multilayer Perceptron (MLP) to demonstrate the core ideas.

3.2 Running Example of *PECJ*

To further elucidate the application of *PECJ*, we present a running example. The tuples to be joined are outlined in Figure 3(a), with a window length of $6ms$. These consist of 6 tuples from streams R and S, formatted as 'Key (κ), Payload (v), Event Time (τ_{event}, in ms)'. Intriguingly, tuples $R4$ and $S1$ have not been observed at a certain ω (e.g., $5.1ms$).

Applying *PECJ* to the observed data allows us to enumerate the tuples in R, S, yielding $n_S = 5$ and $n_R = 5$ respectively (as displayed in Figure 3(b)). Additionally, *PECJ* detects 4 matches, of which two are under $\kappa = A$ and the other two fall under $\kappa = B$. This leads to a join selectivity [15] σ computed as $4/25$. In the case of a $JOIN - COUNT()$ query where the payload v is a constant, the result O aligns with the number of matches, resulting in a count of 4. For a $JOIN - SUM(R.v)$ query where the v of the joined R is accumulated, we get $O = 20$. Moreover, the mean v of the joined R results in $\alpha_R = 20/4 = 5$. Nonetheless, these results do not reflect the true outcome as they exclude contributions from $R4$ and $S1$ who have not arrived by the ω.

To bridge this discrepancy, *PECJ* ventures beyond the confines of observed data and endeavors to address the question, 'what would the O appear like if the contributions from unobserved data were factored in?' This is achieved by tackling a **PDA** problem via **VI** (refer to Figure 3(c)). Specifically, *PECJ* tailors the posterior distribution for the estimated values of n_R, n_S, σ, and α_R, integrating unobserved data, and then uses **VI** to approximate these distributions.

This **VI** process proves invaluable in detecting distortions in streaming by delving into concealed patterns or trends within data streams, thereby allowing the reversal of cumulative contributions from distorted observations. For example, *PECJ* could deduce a high probability of distortion of approximately -1 for n_S, n_R and subsequently suggest that the estimated n_S, n_R should comply with a Gaussian Distribution of $\mathcal{N}(6, 0.2)$. This prompts the application of the expected value 6 to estimate n_S, n_R.

Upon amalgamating these estimated values of n_R, n_S, σ, and α_R, *PECJ* can compute the rectified O. The calculation for the $JOIN - COUNT()$ query would result in

$$O = \sigma \times n_S \times n_R,$$

and for the $JOIN - SUM(R.v)$ query it would be

$$O = \sigma \times n_S \times n_R \times \alpha_R.$$

These computations integrate the contributions as if $R4$ and $S1$ had been present at the time of computation, as illustrated in Figure 3(d).

4 MATHEMATICAL FORMULATION OF *PECJ*

PECJ applies a posterior distributions approximation (**PDA**) method, optimized via variational inference (**VI**), to address the challenges discussed in Section 2.3. We begin by extracting critical information from the data streams and formulating a streaming-aware probability model to minimize error propagation. We then employ **VI** for efficient model parameterization, ensuring our solution caters to the low-latency demands of *SWJ*.

4.1 Formulating the Streaming-Aware Probability Model

PECJ approximates the posterior distribution of the total contribution from all tuples within a window, encompassing both observed and unobserved data. This strategy diverges from the approach of predicting individual tuples via time-series predictions [34]. Our solution eliminates the need for per-tuple approximation or compensation, thereby reducing potential error propagation. This propagation originates from the interdependent prediction of tuple number (n_S, n_R) and the contribution of each tuple to α_R, σ (as discussed in Section 2.3). Specifically, *PECJ* estimates the parameters of the *window-averaged total contribution* (μ_w) directly, perpetually learning from the data stream observations. μ_w is defined in Equation 1.

$$\mu_w = \frac{1}{|\mathbb{W}|} \sum_{x \in \mathbb{W}}^{\mathbb{W}} \mu(x) \quad (1)$$

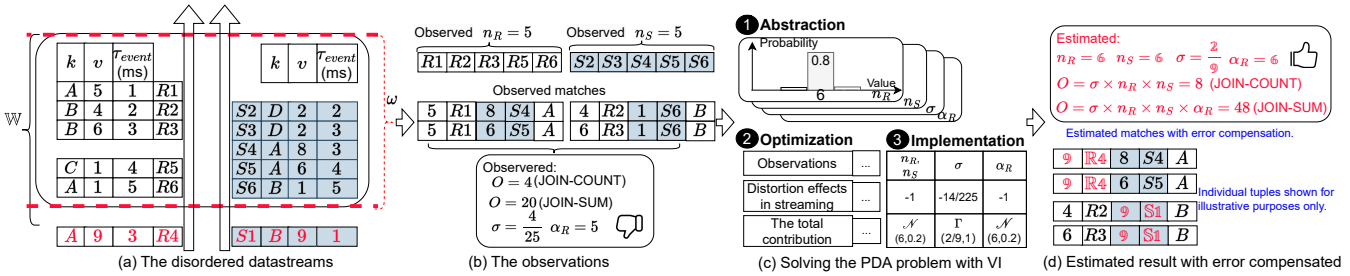


Figure 3: Running Example of *PECJ*.

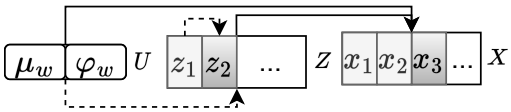


Figure 4: The probability model.

Here, $\mu(x)$ represents a tuple's additive contribution factor, which is summed across all tuples in a window and then normalized by the window length $|\mathbb{W}|$ to define a μ_w. Each μ_w encapsulates a certain type of *averaged* global information within a window, such as join selectivity (σ) or average payload (α) in Section 3.2. For the *accumulated* effects, represented by the n notation, we convert it by the corresponding window average, e.g., $n = \bar{r}_n \times |\mathbb{W}|$, where $\bar{r}_n$ refers to the averaged tuple rate and can also be viewed as a parameter of the window-averaged total contribution. It is crucial to note that σ, α_R, and $\bar{r}_n$ are abstracted in a manner similar to the μ_w notation as each of them describes a certain type of *window-averaged total contribution*. Furthermore, they can be estimated *independently*, avoiding the prediction dependency mentioned in Section 2.3.

PECJ employs specific μ_w variables such as σ to calculate the join aggregation output O (as defined in Section 3.2), thereby facilitating proactive compensation for disorder handling errors. The remaining challenge involves approximating the posterior distribution of μ_w given the corresponding observations $X = x_1, x_2, ..$ from the data streams. Essentially, we aim to determine the posterior distribution $p(\mu_w|X)$, with X evolving as the data stream progresses.

We might also desire additional parameters φ_w, such as the inverse variance of μ_w estimation, which is connected to the credible interval. Both μ_w and φ_w form part of a window's global information U, i.e., $\mu_w, \varphi_w \in U$. For a general illustration, we utilize the $p(U|X)$ notation, as it encompasses both $p(\mu_w|X)$ and $p(\varphi_w|X)$. Our approximation objective can be summarized as follows:

Objective 1. *Approximate the $p(U|X)$, estimating the U by utilizing its expectation given X, i.e., $\hat{U} = \mathbb{E}(U|X)$.*

The inherent dynamics and randomness of data streams [7] can cause the observations X to significantly deviate from the global U. Therefore, unlike approximating a static dataset [19, 35], the dynamic nature of streaming data needs to be specially considered. We, therefore, employ *latent variables* in our model.

As illustrated in Figure 4, our probabilistic model incorporates both the global information U and latent variables $Z = z_1, z_2, ...$ Directed arrows indicate a governing relationship, with X being governed by both U and Z. Notably, Z is situated between U and X, leading to distortions in our observations away from the actual value of U.

Each z_i in Z primarily influences distinct observations in X, symbolizing temporary or local dynamism. For instance, z_1 governs both x_1 and x_2 in Figure 4, while z_2 only governs x_3. To adequately represent a broad spectrum of streaming dynamics, we emphasize the following characteristics related to Z: 1) Z does not necessarily need to have the same length as X, 2) z_i can be a vector of any dimensions, and 3) z_i may be further dependent on U or other latent variables.

By incorporating Z into the model formulation, we can better capture the intricate relationships and dependencies between X and U. In particular, Z can help unveil patterns and trends in the data streams that might not be immediately discernible from X alone, providing invaluable insights into the underlying streaming processes for more precise U estimation.

4.2 Optimizing Model Parameterization with VI

Despite adequately reflecting stream dynamics, it's often challenging to parameterize a probability model when latent variables are involved. Specifically, deriving $p(U|X)$ requires turning Z into constants under the joint distribution $p(U, Z, X)$. This involves computational complexity that grows exponentially when integrating or summing over all potential configurations of Z. Given that low latency is critical for *SWJ*, a large overhead is typically unacceptable. Additionally, we must update the model parameters as new data arrives and carry out inference promptly.

To facilitate efficient parameterization while meeting the low latency requirements, we use the variational inference (**VI**)[7, 14, 32] approach for optimization. **VI** enables approximation on U and Z without resorting to brute force integration or summation. It inherently supports continual learning on data streams by merging posterior distributions into prior distributions when making new observations[11]. We discuss the detailed approach below.

Approximation of $p(U|X)$. We select a manageable family of $q()$ functions, referred to as the *variational family* [14], to approximate the $p()$ distributions. The variational family liberates us from intractable and costly integration computations. A popular choice is the mean-field variational family. Specifically, our $q()$ functions hold the following relationships and constraints. The $\approx$ symbol denotes approximation.

Equation 2 requires that each component in U is considered independent in the approximation function $q()$, and it further designates $q(U)$ as an approximation to our target distribution

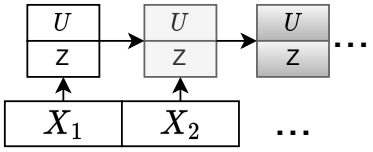


Figure 5: The continual learning process of model parametrization.

$p(U|X)$. Equations 3 and 4 are similar, approximating the conditional prior distribution of $Z|U$ and the joint distribution of U, Z, respectively. When U and Z are independent, the $Z|U$ notation inside brackets can be further simplified to Z. The core idea of Equations 2 to 4 is to decompose each variable into separate distributions during the approximation, e.g., the $q(\mu_w)$, $q(\varphi_w)$, and $q(z_i)$ components. This way, we can apply divide and conquer to each variable, avoiding brute force summation or integration.

$$q(U) = \prod_{\mu_w \in U} q(\mu_w) \times \prod_{\varphi_w \in U} q(\varphi_w) \approx p(U|X) \tag{2}$$

$$q(Z|U) = \prod_{z_i \in Z} q(z_i) \approx p((Z|U)|X) \tag{3}$$

$$q(U, Z) = q(U) \times q(Z|U) \approx p(U, Z|X) \tag{4}$$

Rather than brute force computation, the primary mathematical task of **VI** is to bring $q()$ close to $p()$, which is a simpler *optimization* problem. Specifically, the optimization goal is to maximize the *evidence lower bound* ($ELBO_q$), defined in Equation 5. The $\mathbb{E}_q()$ notation refers to the expectation regarding our approximation functions q. As the name suggests, given that X have already been observed as evidence to support the **PDA**, the $ELBO_q$ should be maximized.

Objective 2.

$$maximize\ ELBO_q,$$
$$s.t., ELBO_q = \mathbb{E}_q(log((p(U, Z, X))) - \mathbb{E}_q(log((q(U, Z))) \tag{5}$$

Continual Learning from Observations. Ensuring accurate results over continuously changing data streams necessitates updating approximations, such as $q(U)$, with newly incoming observations from X. However, considering X are derived from infinite data streams, it is impractical to maintain a complete history of X and execute **VI** each time X expands. As such, we can treat model parameterization as a continual learning process, as depicted in Figure 5.

Assuming we have already extracted insights from previous observations X_1 and have an approximation for U and Z, these approximations can be continuously updated with new observation X_2. This process removes the need for recomputing from the entire $X = X_1, X_2$. Drawing from the approach in [11], we utilize the *prior distribution* of U (i.e., $p(U)$) as the initial conditions, with "starting" not necessarily implying a beginning from scratch.

As demonstrated in Equation 6, the $q(U_1)$ derived from old observation X_1 can act as the new prior distribution. We can integrate it with the impacts from the new observation, i.e., $p(X_2|U)$, to update our approximation. The optimization methodologies of continual learning, such as coreset selection [22],

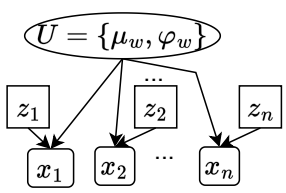


Figure 6: The variable dependency under *analytical* instantiation.

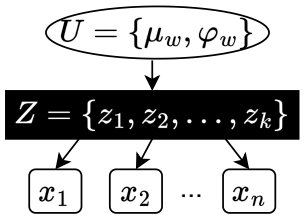


Figure 7: The variable dependency under *learning* instantiation.

are beyond this work's scope due to their complexity, and we earmark them for future exploration.

$$\begin{aligned} p(U|X) &= p(U|X_1, X_2) \\ &\propto p(X_2|U)p(U|X_1) \\ &\approx p(X_2|U)q(U_1) \end{aligned} \tag{6}$$

5 VI IMPLEMENTATIONS FOR *PECJ*

Implementing *PECJ* necessitates the instantiation of the **VI** equations as delineated in Section 4.2. However, the precise organization and interrelationships between U and Z have substantial implications for *PECJ*'s overhead and versatility, requiring a judicious design approach. This section explores two pragmatic implementations, initially focusing on the analytical method (*analytical*) [14], and subsequently examining the machine learning-based approach (*learning*) [7, 32].

5.1 *Analytical* Implementation for Low Overhead

This implementation extends the central limit theorem by incorporating a basic awareness of streaming dynamics. As depicted in Figure 6, Z is positioned between the global (i.e., U) central limit theorem and its sample (i.e., X). There is no local latent variable z_i dependent on the global variable $U = \mu_w, \varphi_w$. Furthermore, μ_w is independent of φ_w here. As a result, we can simplify the $U|Z$ notations in Equation 3 (Section 4.2), given that we have independent U, z_i.

Equation 7 asserts that each observation $x_i \sim \mathcal{N}(\mu_w/z_i, 1/z_i\varphi_w)$. It implies that x_i is influenced not solely by the global mean μ_w and global variance $1/\varphi_w$ of a Gaussian Distribution, but also by the reverse linear distortions represented by the transient dynamics z_i.

When we couple Equation 7 with the prior distribution of μ_w, φ_w, z_i, denoted as $p(\mu_w), p(\varphi_w), p(z_i)$ respectively, we can derive the joint distribution function $p(U, Z, X)$ for all variables in Equation 8, where $U = z_1, z_2, ...z_n$ and $X = x_1, x_2, ..., x_n$.

$$f(x_i|\mu_w, \varphi_w, z_i) = e^{-(z_i \times x_i - \mu_w)^2 * \varphi_w/2} \times \sqrt{\varphi_w} \times const \tag{7}$$

$$\begin{aligned} p(U, Z, X) = const \times \varphi_w^{n/2} \times e^{\sum_{i=1}^{n} (z_i \times x_i - \mu_w)^2 \times \varphi_w/2} \times \\ p(\mu_w)p(\varphi_w) \prod_{i=1}^{n} p(z_i) \end{aligned} \tag{8}$$

When **VI** converges to a mean-field family of $q()$ and Equation 5 is achieved, an analytical solution [9, 14] exists for $q(\mu_w)$, as shown in Equation 9. The notation $\mathbb{E}_{\varphi_w,Z}$ indicates that the approximations of μ_w can be facilitated by the expectations of other variables, specifically φ_w and Z, rather than performing exhaustive computation of their integration or summation. This is a consequence of the decoupling property inherent to the mean-field family.

Moreover, if the prior distribution of μ_w is a Gaussian $\mathcal{N}(\mu_0, 1/\tau_0)$, $q(\mu_w)$ culminates in a Gaussian posterior distribution of μ_w expressed as $\mu_w \sim \mathcal{N}(\frac{\tau_0\mu_0+ng(X)}{\tau_0+n}, \frac{1}{(\tau_0+n)\mathbb{E}(\varphi_w)})$. From this Gaussian posterior distribution, we can deduce two crucial insights:

(1) The **estimated value** of μ_w (denoted as $\bar{\mu_w}$) behaves like a *linear function* of X as shown in Equation 10. Notably, the coefficient vector K correlates with the expectations of each latent variable, represented as $\mathbb{E}(z_i)$.
(2) The **credible interval** for estimating μ_w is related to $\mathbb{E}(\varphi_w)$, as depicted in Equation 11. For example, the 95% credible interval is calculated as $\bar{\mu_w} \pm 1.96\frac{1}{\sqrt{(\tau_0+n)\mathbb{E}(\varphi_w)}}$.

$$
\begin{aligned}
q(\mu_w) &= \mathbb{E}_{\varphi_w,Z}(f(U,Z,X)) \\
&= const \times p(\mu_w) e^{-(\mu_w - g(X,Z))^2 \times (n\mathbb{E}(\varphi_w)/2)} \qquad (9)
\end{aligned}
$$

$$
\text{where } g(X,Z) = \sum_{i=1}^{n} \frac{\mathbb{E}(z_i) * x_i}{n}
$$

$$
\exists \text{vector } K \text{ and scalar } b, s.t., \bar{\mu_w} = \mathbb{E}(\mu_w) = KX + b
$$

$$
\text{where } KX = \frac{ng(X,Z)}{\tau_0+n}, b = \frac{\tau_0\mu_0}{\tau_0+n} \qquad (10)
$$

$$
\forall \text{credible interval } \delta \in (0,1),
$$

$$
\bar{\mu_w} - i(\delta)\frac{1}{\sqrt{(\tau_0+n)\mathbb{E}(\varphi_w)}} \leq \mu_w \leq \bar{\mu_w} + i(\delta)\frac{1}{\sqrt{(\tau_0+n)\mathbb{E}(\varphi_w)}}
$$

where $i(\delta)$ is the δ interval quantile of a standard Gaussian. (11)

The analytical **VI** process can be continued to obtain $\mathbb{E}(\varphi_w)$ and $\mathbb{E}(z_i)$ by employing Stochastic Variational Inference (SVI) [14] and extending Equation9 to φ_w and z_i. Alternatively, given the straightforward linear form, linear filters such as exponential moving average or ARIMA [26] can also be deployed. However, in this scenario, the filter parameters should evolve dynamically with data streams rather than being preset, ensuring accurate on-the-fly approximation of the $\mathbb{E}(z_i)$ parameters.

By default, *PECJ* employs an adaptive linear filter (ALF) under an exponential moving average, with the filter parameter continuously updated based on rule-based learning from the data streams. This choice is motivated by the expectation that an adaptive linear filter will incur significantly less overhead compared to SVI, while also being simpler to design and adjust.

5.2 *Learning* Implementation for Generality

Although more intricate U, Z relationships can be defined to enhance the reflectivity of the *analytical* implementation, this approach demands significant manual effort. Moreover, implementing a more complex *analytical* system may be impractical due to the intricate mathematical relationships involved (see Appendix A for details).

To overcome the challenges of capturing complex stream dynamics, we refer back to the abstract ELBO definition in Equation 5 for a more universal solution. This approach doesn't require knowledge or assumptions about specific relationships between U and Z, nor does it require familiarity with the length of Z or the dimensions of z_i. As depicted in Figure 7, merely creating arbitrary dependencies between some U, Z is sufficient.

First, we remap the entire parameter space of U and Z into another space, $W = w_1, w_2, ..w_m$, i.e., $U, Z \rightarrow W$. Hence, Equation 5 can be rewritten as Equation 12.

Second, we further constrain W by ensuring 1) the independent μ_w and φ_w presented in Equation 7 and Equation 8 are assigned to w_1 and w_2, respectively, and 2) the remaining factors $w_3, w_4, ...w_m$ form an Orthogonal Basis (i.e., they are independent of each other) given w_1, w_2. As a result, the $log((p(W,X))$ term can be decomposed as shown in Equations 13~ 14. Note that $log((p(X|W)))$ is the *log-likelihood* of X in the W space, and $log((p(w_i)))$ is the *log-prior-distribution* of w_i. As both are irrelevant to q, we can conveniently remove the E_q notations.

Third, based on the mean-field property [9, 14], $\mathbb{E}_q(log(q(W)))$ can be further decomposed as per Equation 15.

Finally, by separating $q(\mu_w)$ and $q(\varphi_w)$ from the other $q(w_i)$, we can derive Equation 16. It should be noted that the resulting $\mathbb{E}(\mu_w|X)$ and $\mathbb{E}(\varphi_w|X)$ are the expectations of μ_w and φ_w given X, respectively. They can be directly utilized for the estimated value in *PECJ*'s error compensation, as discussed in Section 4.1.

$$
\begin{aligned}
ELBO_q &= \mathbb{E}_q(log((p(W,X))) - \mathbb{E}_q(log((q(W))) && (12)\\
&= \mathbb{E}_q(log((p(X|W)p(W)))) - \mathbb{E}_q(log((q(W))) && (13)\\
&= log(p(X|W)) + log(p(\mu_w)) + log(p(\varphi_w)) \\
&\quad + \sum_{i=3}^{m} log(p(w_i|\mu_w,\varphi_w)) - \mathbb{E}_q(log(q(W))) && (14)\\
&= log(p(X|W)) + log(p(\mu_w)) + log(p(\varphi_w)) \\
&\quad + \sum_{i=3}^{m} log(p(w_i|\mu_w,\varphi_w)) - (\sum_{i} \mathbb{E}_q(log(q(w_i)))) && (15)\\
&= log(p(X|W)) + log(p(\mu_w)) + log(p(\varphi_w)) \\
&\quad + \sum_{i=3}^{m} log(p(w_i|\mu_w,\varphi_w)) - (\sum_{i=3}^{m} \mathbb{E}_q(log(q(w_i))) \\
&\quad + \log(\underline{\mathbb{E}(\mu_w|X)}) + log(\underline{\mathbb{E}(\varphi_w|X)}) && (16)
\end{aligned}
$$

Equation 16 can further be leveraged to influence the behavior of neural networks (NNs), enabling them to conform to the **PDA** process without requiring knowledge of the relationships between U and Z. Here's a detailed three-step, ELBO-driven solution:

(1) Construct an NN for function fitting, ensuring that the final output is at least seven-dimensional to correspond with the seven scalars depicted in Equation 16.
(2) Conduct supervised pre-training over the entire NN so that each dimension accurately estimates the target scalar, such as $log(\mathbb{E}(\mu_w|X))$. Conventional NN loss functions, such as mean square error, should suffice here.

Table 2: Server Specification

Component	Description
Processor	Intel(R) Xeon(R) Gold 6252 CPU (24 cores × 2 HyperThreading)
L3 cache size	35.75MB
Memory	384GB
OS & Compiler	Ubuntu 22.04, compile with g++ 11.3.0

(3) During continual learning in a streaming environment, Equation 16 can be employed to optimize NN loss. For example, if gradient descent is implemented via ADAM or SGD [3], the loss function can be designed to decrease monotonically with $ELBO_q$. Note that, if the NN is overly 'confident,' the numerical evaluation of $ELBO_q$ could potentially be ∞. In such instances, we use bounded functions such as $-sigmoid(ELBO_q)$ as the loss function.

In *PECJ*, we implemented a straightforward multilayer perceptron (MLP) to briefly illustrate this concept, leaving more powerful structures like LSTM [7] or transformer [32] for future exploration. Furthermore, given the necessity for NNs to meet low latency requirements, it's critical to efficiently perform their inference and learning processes. As a result, an effective solution for deploying *PECJ* across various dynamic situations is to integrate a well-structured NN with high-performance computing. Pursuing this combination represents an important area of ongoing work.

6 EVALUATION

In this section, we discuss our implementation and evaluation details using a recent multicore server with the Intel Xeon Gold 6252 processors, as specified in Table 2.

6.1 Implementation Details

In our evaluation, we scrutinize the performance of *PECJ* using two distinct setups: standalone and integrated implementations. Each setup facilitates a comprehensive comparison with different existing approaches. Note that, while the automatic determination of suitable ω is orthogonal to this work, it serves as a tuning knob for all mechanisms during the experiments.

Standalone Implementation: In the standalone implementation setup, we're aiming for an algorithmic comparison between *PECJ* and two existing methodologies, namely **K-Slack-Join (*KSJ*)** [15] and **Watermark-Join (*WMJ*)** [8]. For these standalone implementations, we employed the same C++ codebase for *KSJ*, *WMJ*, and *PECJ*.

Our implementation of *PECJ* included three separate approaches for the *analytical* and *learning* instantiations. For the former (discussed in Section 5.1), we utilized both the Adaptive Linear Filter (ALF) and Stochastic Variational Inference (SVI) methodologies. For the *learning* instantiation (Section 5.2), we opted for a simple ML-based approach (i.e., Multi-Layer Perceptron, or MLP). The ALF implementation served as the default configuration for *PECJ*'s *analytical* instantiation.

KSJ employs a k-slack buffer strategy to manage data stream disorder. Upon the preprocessing of data streams by the k-slack buffer, *KSJ* carries out a standard hash-join operation, treating the data as ordered. Specifically, we've linked our tuning knob ω with the control conditions of the k-slack buffer, as previously discussed in Section 2. *WMJ* adopts the watermark mechanism [8] for data preprocessing, bypassing the k-slack buffer. To simplify the process, we generate a watermark only at ω, ignoring the impacts of intermediate watermarks. This assumes the complete arrival of data within a window.

Integrated Implementations: This setup is designed to assess *PECJ*'s performance when incorporated into an existing multi-threaded stream processing system. AllianceDB, which is a recent multi-threaded *SWJ* testbed and serves as our integration platform. In this environment, we selected two representative parallel *SWJ* algorithms, Parallel Radix Join (*PRJ*) and Symmetric Hash Join (*SHJ*), to perform our assessment.

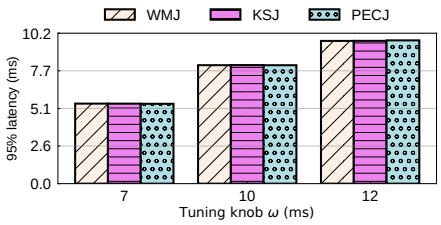
PRJ adopts a 'lazy' approach, delaying the join operation until all tuples have arrived. On the other hand, *SHJ* pursues an 'eager' strategy, initiating the join process as soon as a portion of tuples arrive. Both *PRJ* and *SHJ* operate under the assumption of in-order arrival (i.e., $\tau_{arrival} = \tau_{emit}$), and consider a window complete when the first tuple's arrival timestamp ($\tau_{arrival}$) surpasses the window's limit.

6.2 Experimental Setup

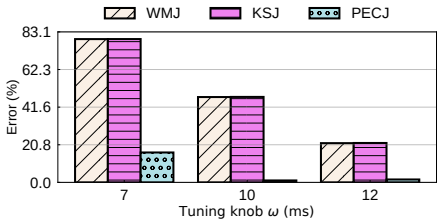
Datasets: The evaluation was carried out using a diverse collection of four widely-used real-world datasets - **Stock**, **Rovio**, **Logistics**, **Retail**, and a synthetic dataset known as **Micro**. The **Stock**, **Rovio**, and **Micro** datasets were adopted from AllianceDB [38], while the **Logistics** and **Retail** datasets were obtained from the resource [37]. To simulate a realistic scenario, we introduced disorder in the data arrival by reordering the arrival timestamps $\tau_{arrival}$ differently from the event timestamps τ_{emit} (as mentioned in Section 2). The δ was set randomly for all tuples. We kept the event rate (controlled by event timestamp τ_{emit}) of both R and S streams consistent at $100Ktuples/s$ unless stated otherwise. We use **Stock** datasets in Section 6.3 and 6.5, and vary the usage of datasets in Section 6.4 and 6.6.

Queries: Three different queries were employed in our evaluation. **Q1**: This query entails a 10-ms *SWJ* aggregated by COUNT (Section 3.2), with a maximum delay among all events from the time of occurrence to the time of arrival, Δ of $5ms$. The small Δ is representative of a scenario where the data stream processing system is geographically close to the data source, such as on the edge of a cloud network [36]. **Q2**: This query modifies **Q1** by changing the aggregation function to SUM (Section 3.2), with all other settings retained as per **Q1**. **Q3**: This query extends **Q1** by altering the disordered arrival pattern of data and setting the Δ to $1000ms$. The significant Δ simulates situations where the stream analytic is situated far from the data source, such as during multiple intercontinental communications within a TOR network [13].

While **Q1** and **Q2** are tailored to require ultra-low latency processing, typically tens of milliseconds or less, **Q3** cannot expect such low latency due to the large arrival delay. Nonetheless, the goal is to achieve a latency below $500ms$, which is half of its Δ.



(a) 95% latency.



(b) Relative Error

Figure 8: Q1 end-to-end comparison.

6.3 End-to-End Comparison

We initiate our analysis by juxtaposing *PECJ*, *KSJ*, and *WMJ* under the conditions stipulated by **Q1** using the Stock dataset. The assumed time point of window completeness ω is fine-tuned to $7ms$, $10ms$, and $12ms$ for each methodology. Subsequently, we elucidate the ensuing 95% processing latency (95% l) and relative error (ϵ) in Figures 8(a) and 8(b).

Three critical insights emerge from this comparative analysis. Initially, it is observed that for the same ω, each strategy incurs a similar latency, as depicted in Figure 8(a). This congruity arises mainly due to the similar overhead incurred from waiting for a more comprehensive window of data. Relative to this waiting overhead, the specific overheads engendered by *WMJ*, *KSJ*, and *PECJ* are marginal. Secondly, as anticipated, the error generated by *WMJ* and *KSJ* exhibits similarity and consistently decreases with larger ω values. Despite their distinct mechanisms for handling disordered data, they have an identical level of data completeness within a given window under the same ω. Consequently, their ignorance extent towards unobserved data also aligns.

The most notable observation, however, is the superior performance of *PECJ* which manifests in significantly lower errors compared to *WMJ* and *KSJ*. For instance, when ω is set to $7ms$, *PECJ* can maintain an error as low as $\leq 16\%$ with a 95% latency 95% l of $\leq 5.5ms$. In contrast, *WMJ* and *KSJ* register an error in excess of $20\% \leq$, even when the 95% latency escalates above $9.5ms$ by setting ω to $12ms$. As expounded earlier, this improved performance is attributed to *PECJ*'s proactive strategy of incorporating the contributions of unobserved data, unlike the passive waiting approach of *WMJ* and *KSJ* (Section 3).

Shifting our focus to **Q2**, we maintain identical settings as in the previous experiment. Given the similar 95% latency (95% l) patterns across these strategies, we primarily present the resulting relative error (ϵ) in Figure 9. Despite **Q2** demanding a more intricate syntax and involving additional parameters compared to **Q1** (Section 3.2), *PECJ* retains its superior performance, evident

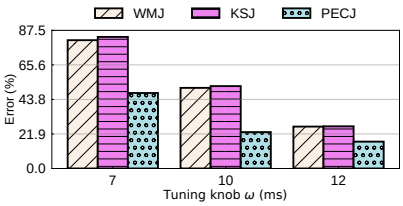
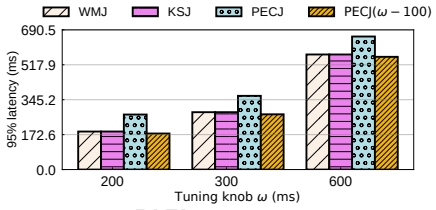
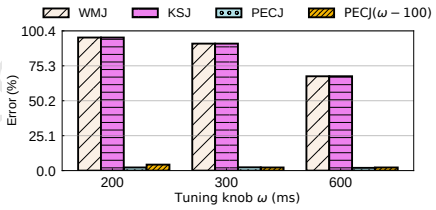


Figure 9: The relative error of Q2 end-to-end comparison.



(a) 95% latency.



(b) Relative Error

Figure 10: Q3 end-to-end comparison. *PECJ* (ω-100) notation refers to subtracting the ω of *PECJ* by 100.

through its significantly reduced error. For instance, when the ω is adjusted to $10ms$, the error incurred by *PECJ* is as low as 25.0%, compared to a substantial 52% for *WMJ* and 51.5% for *KSJ*.

Additionally, it's worth noting that the partial re-ordering inherent to the k-slack methodology provides slight assistance for *KSJ* in accurately summing the tuples' value compared with *WMJ*. This benefit is illustrated by a minor 0.5% reduction in error when the ω is set to $10ms$.

Lastly, we examine **Q3**, and due to the more intricate disorder arrival pattern and the less stringent real-time requirements of **Q3** (Section 6.2), we adjust *PECJ* from ALF to ML-based implementation ($PECJ_{\text{learning}}$). We set ω to $200ms$, $300ms$, and $600ms$ and present the corresponding results in Figures 10(a) and 10(b). Our findings show that *WMJ* and *KSJ* fall short in adapting to this scenario, where data disordering manifests in an extreme fashion. Notably, even with ω set to a lenient $600ms$, allowing for a latency of around $530ms$, they still yield an unacceptably high error over 70%.

Contrarily, *PECJ* consistently maintains the error within 3%, leveraging its *learning* instantiation of **PDA** to compensate for the error (Section 5.2). It's important to acknowledge that this assistance from ML isn't free, introducing an additional latency of around $90ms$ (Figure 10(a)). However, as this extra latency is a by-product of a constant inference process, it can be circumvented by reducing ω by $100ms$. Consequently, as demonstrated here, the *PECJ* (ω-$100ms$) configuration still manages to maintain the error within 5%. Nevertheless, future studies could explore accelerating

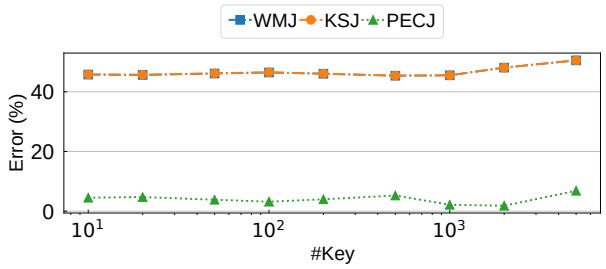


Figure 11: Performance comparison under a varying number of keys.

the current single-thread, CPU-only implementation of *PECJ* using hardware accelerators like GPUs, to optimize performance.

6.4 Workload Sensitivity Study

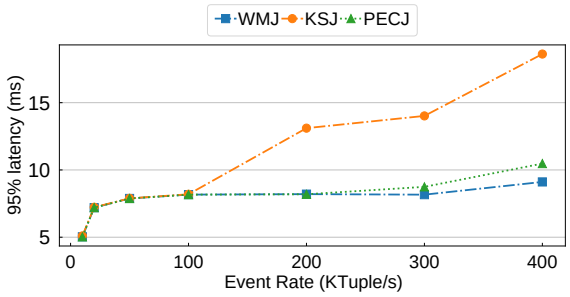
This subsection of the sensitivity study aims to contrast *PECJ* with the baseline models, *WMJ* and *KSJ*, under a range of workload characteristics. These include the number of join keys and the event rate. For the purposes of this study, we fix ω to $10ms$ and operate under a *SWJ* with a window length of $10ms$, followed by SUM. To adjust the workload characteristics, we utilize the synthetic dataset **Micro** [38] and set the Δ as $5ms$.

To assess the impacts of join keys, we distribute the keys of both R and S randomly and vary the number of keys from 10 to 5000, while maintaining the event rate at our default setting of $100Ktuple/s$. Since the key number has virtually no impact on the latency of *PECJ*, *WMJ*, and *KSJ* (with a fluctuation of approximately $\pm 0.6\%$ around $8.25ms$ at most), we present the relative error in Figure 11. In general, *PECJ* outperforms the baseline models across a wide range of key numbers. However, when the key number increases to as high as 5000, the likelihood of encountering a join match diminishes, which leads to fewer observations on join selectivity σ and slightly elevates its error.

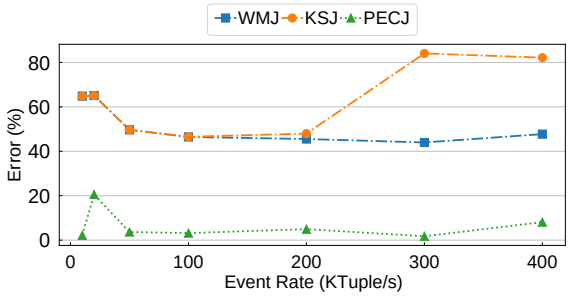
Next, we hold the number of join keys at 10, and adjust the event rate from $10Ktuple/s$ to $400Ktuple/s$. The resulting 95% l and ϵ are displayed in Figure 12. Our findings show that *KSJ* experiences a latency 50% higher than either *WMJ* or *PECJ* when the event rate reaches $200Ktuple/s$, and its ϵ also begins to escalate under such high event rate. This phenomenon occurs because 1) the k-slack overhead swells with a larger number of tuples processed per unit of time (i.e., the higher event rate), causing *KSJ* to overload much more readily than *WMJ* or *PECJ*, and 2) when an overload transpires, the partial reorder in *KSJ* becomes asynchronous, further increasing its error. Compared to *WMJ*, *PECJ* is slightly more prone to overload (i.e., at $400Ktuple/s$) due to the extra overhead involved in making observations and executing compensations. Nonetheless, *PECJ* consistently achieves the smallest error under a non-overload rate, and even under a mild overload.

6.5 Sensitivity Study on the *PECJ* Algorithm

This section is dedicated to a sensitivity study that investigates the accuracy of *PECJ* under various instantiation strategies, specifically the *analytical* instantiation (hereafter referred to as $PECJ_{\text{analytical}}$, as discussed in Section 5.1), and the *learning* instantiation (hereafter referred to as $PECJ_{\text{learning}}$, as discussed in Section 5.2).



(a) 95% latency



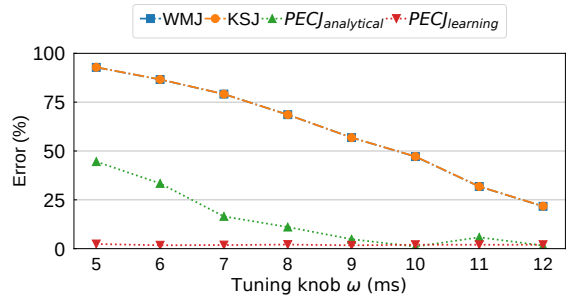
(b) Relative Error

Figure 12: Performance comparison under varying event rate.

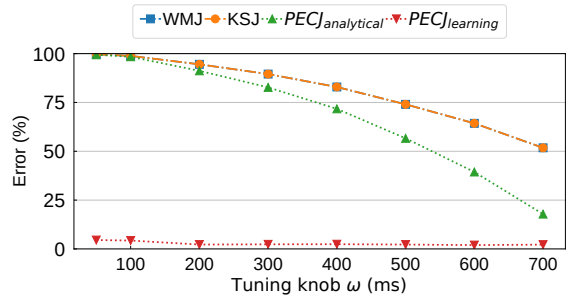
Initially, we assess the **Q1** scenario, which presents relatively simple stream dynamics and observation distortion. As depicted in Figure 13(a), we compare the relative error (ϵ) among $PECJ_{\text{analytical}}$, $PECJ_{\text{learning}}$, and two baseline models, *WMJ* and *KSJ*, while tuning the ω from $5ms$ to $12ms$.

Our study yields several important observations. Firstly, as anticipated from Section 2, the baseline models *WMJ* and *KSJ* display almost identical errors across each ω and consistently present higher errors than either $PECJ_{\text{analytical}}$ or $PECJ_{\text{learning}}$. Secondly, $PECJ_{\text{analytical}}$ exhibits effective error compensation and accurately mirrors the arrival pattern in **Q1**. However, due to its reliance on the central limit theorem (see Section 5.1), its accuracy improves with a larger ω. Lastly, $PECJ_{\text{learning}}$ outperforms $PECJ_{\text{analytical}}$ in terms of error compensation and capturing unobserved data. This capability is notably resistant to degradation, even when fewer observations are available (as a result of smaller ω). This is due to $PECJ_{\text{learning}}$'s design intention for generality (Section 5.2), enabling it to glean more hidden information from the data streams and compensate for errors more effectively than $PECJ_{\text{analytical}}$.

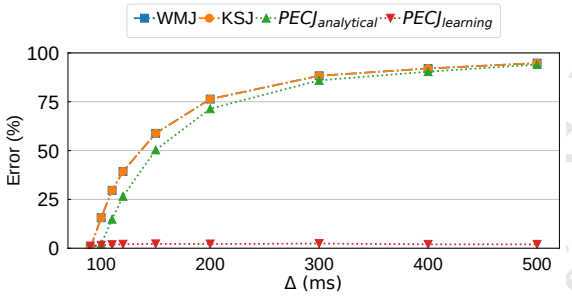
We then proceed to evaluate the **Q3** scenario, which introduces more complexity to the stream dynamics and observation distortion due to a larger Δ. The ω is tuned from $50ms$ to $700ms$, and the relative errors (ϵ) of all methods are illustrated in Figure 13(b). Generally, $PECJ_{\text{analytical}}$ struggles to accurately reflect **Q3**'s arrival pattern and provides sub-optimal error compensation. Each observation on join selectivity or event rate is heavily biased, violating the preconditions for applying the central limit theorem (Section 5.1). While this bias can be reduced with a larger volume of



(a) Q1



(b) Q3



(c) *JOIN* − *SUM* under varying Δ, fix ω at 90

Figure 13: The relative error under different instantiations. We report the minimal error among two implementation ways of $PECJ_{\text{analytical}}$, i.e., ALF and SVI, and $PECJ_{\text{learning}}$.

observations, it necessitates a larger ω. Contrarily, $PECJ_{\text{learning}}$ is equipped to recognize these biases, overcoming the constraints of the central limit theorem, and thus delivers superior error compensations as a general instantiation method.

Lastly, we delve into the scenarios where $PECJ_{\text{analytical}}$ might fail. Specifically, we maintain the SUM aggregation function of **Q1**, fix the ω to 100ms, and increment the Δ from 90ms to 500ms. The resultant error is presented in Figure 13(c). It is observed that the error of $PECJ_{\text{analytical}}$ gradually escalates with Δ, exceeding 50% when Δ reaches 150ms or higher, and eventually matching the high error levels of baseline models *WMJ* or *KSJ* when Δ is sufficiently large. This confirms that the range of Δ is a key contributor to observation distortion, resulting in the unsuitability of the central limit theorem and, hence, the sub-optimal performance of $PECJ_{\text{analytical}}$.

6.6 Integrated Implementation Evaluation

In this evaluation, we contrast the original parallel *SHJ* and *PRJ* in AllianceDB with their corresponding modifications under *PECJ*, namely, *PECJ-SHJ* and *PECJ-PRJ*. It is important to note that the ω doesn't impact *SHJ* and *PRJ*, and we set it to 10ms for both *PECJ-SHJ* and *PECJ-PRJ*.

Using real-world datasets, we compare the 95% l and ϵ under **Q1**, as depicted in Figure 14. We make three key observations. First, both *PRJ* and *SHJ* yield a high error (e.g., 47% on the Stock dataset) under disordered arrival. Second, both *PECJ-PRJ* and *PECJ-SHJ* substantially reduce the error compared to *PRJ* and *SHJ*. Moreover, *PECJ-PRJ* and *PECJ-SHJ* yield a latency similar to their counterparts, *PRJ* and *SHJ*. This is attributable to the efficient model parameterization and instantiation of *PECJ* (discussed in Sections 4.2 and 5). Third, interestingly, *PECJ-SHJ* achieves even lower error than *PECJ-PRJ* (i.e., 1% vs. 13% relative error in Stock), thanks to its additional real-time awareness of data stream trends.

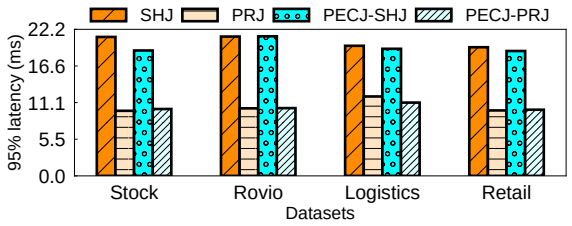
For the scaling-up evaluation, we increase the number of tuples in each window and ensure the event rate of both R and S exceeds $1600KTuples/s$. We adjust the number of threads from 2 to 12 and present the 95% l, ϵ, and system throughput of each mechanism in Figure 15. We observe that the lazy approaches *PRJ* and *PECJ-PRJ* continue to outperform their eager counterparts, i.e., *SHJ* and *PECJ-SHJ* respectively, in terms of latency reduction (majorly caused by the queueing delay of tuples) and throughput improvement. This aligns with prior studies [38] conducted under an in-order arrival scenario, and eager approaches still face challenges due to cache thrashing, especially during scaling up.

Moreover, *PECJ-PRJ* scales up as efficiently as *PRJ* attributed to its constant disorder handling overhead. We also found that *PECJ-SHJ* results in higher errors than *PECJ-PRJ* as shown in Figure 14(b). This is because the additional awareness of eager disorder handling is heavily distorted under data overload, and it even misguides *PECJ* by providing incorrect information to compensate for errors. In conclusion, *PECJ* proves practical for scaling-up *SWJ* algorithms under disorder data arrival and should be implemented with a lazy approach during upscaling.

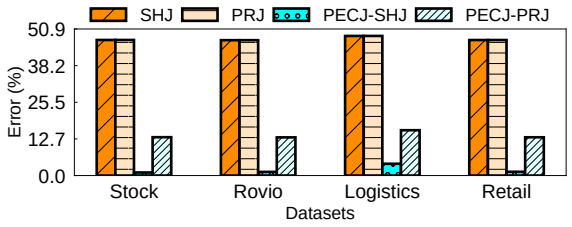
7 RELATED WORK

Stream Window Join. The primary focus of optimizing stream window join operations has been on improving efficiency and incremental processing. For instance, the Handshake Join [33] and Split Join [28] leverage a dataflow model to scale up on modern multicores, while the *IBWJ*[30] employs a shared index structure to accelerate tuple matching. A comprehensive experimental study by Zhang et al.[38] compares these approaches across diverse workload characteristics, application requirements, and hardware architectures. This study also highlights successful attempts to use relational join algorithms for accelerating stream window joins. These approaches typically assume in-order data arrival, obviating the need to concern about accuracy. In contrast, our work explores the balance between accuracy and latency under disordered data conditions.

Buffer-based Disorder Handling. Numerous studies have explored the tradeoff between accuracy and latency using buffers. To avoid potential infinite buffering, existing works implement different mechanisms to control the buffer flushing and assume



(a) 95% latency



(b) Relative Error

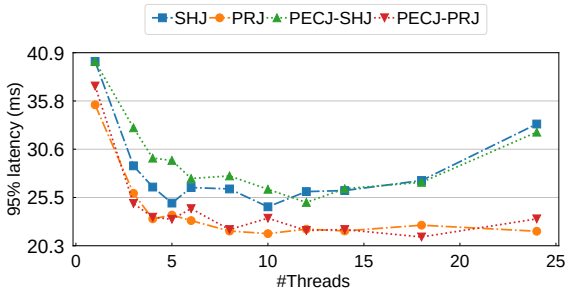
Figure 14: Integrated implementation evaluation using Q1.

temporary completeness of arrived data, such as k-slack [16, 21], watermarks [5, 8, 31], and punctuations [39]. For instance, Ji et al. [15] proposed a k-slack-based disordered stream join, treating a configurable accuracy-latency tradeoff as a fundamental aspect. They emphasize that joins are inherently more complex than single-stream linear operators such as summation or average in disorder handling, due to the mutual and non-linear relationships between multiple streams. Despite variations in specific tradeoff rules and approaches, these methods rely on already acquired data to generate results, neglecting the contributions of unobserved data. Our approach, the *PECJ*, differs by incorporating proactive compensation for this unobserved data.

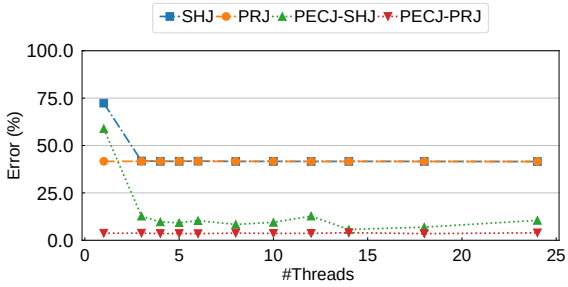
Approximate Query Processing (AQP). The goal of AQP is to reduce computational overhead by selecting a data subset to approximate the result of the whole dataset [17, 20]. As data selection is system-controlled, error compensation can be predefined and is relatively stable in AQP. Compensation can use either linear [29] or non-linear formulas [4], depending on the algorithm's subset selection. More advanced AQP approaches employ machine learning [23] and bootstrap methods [35] to tackle ubiquitous queries under static data, albeit with higher computational costs. To address this issue, the *Wander Join* algorithm [19] applies stochastic and graph optimizations to reduce overhead and optimize online aggregation for joins. Our work, however, addresses a different and more challenging problem—handling disordered *SWJ* where observation distortion cannot be system-controlled. Therefore, we propose a posterior distribution approximation under a streaming-aware model formulation and efficient model parameterization for disordered *SWJ* (Sections 4 and 5).

8 CONCLUSION

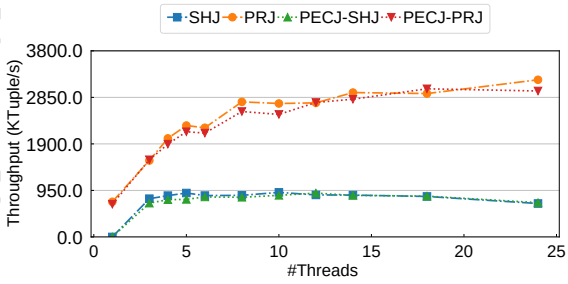
In this paper, we introduced *PECJ*, an innovative approach for stream window join (*SWJ*) on disorderly data. *PECJ* outperforms existing methods by incorporating contributions from unobserved



(a) 95% latency



(b) Relative Error



(c) Throughput

Figure 15: Scaling-up integrated implementation using Stock dataset with an event rate of $1600Ktuples/s$**.**

data into its compensation model, thus improving the accuracy-latency tradeoff. It achieves this through a novel approach of posterior distribution approximation (**PDA**) via variational inference (**VI**), carefully balancing complexity and generality, and shows promising results when integrated into a multi-threaded *SWJ* benchmark testbed. In future work, it will be interesting to explore how our deep learning instantiation, which offers a theoretically general solution for a **PDA** situation distorted by disordered arrival, can be extended and integrated with approximate computing approaches that use algorithm-controlled data distortion to trade off accuracy and latency, such as sampling and compression.

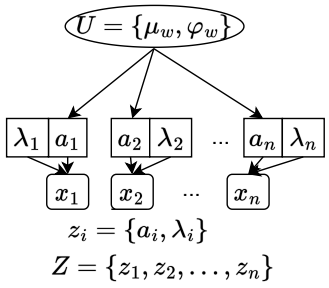


Figure 16: The variable dependency under an embarrassingly failed analytical instantiation example.

A AN IMPRACTICAL COMPLICATED ANALYTICAL INSTANTIATION

Figure 16 demonstrates our attempt at using a more complicated analytical instantiation. Specifically, we treat the long-tail effects of stream data [38] as the first-class citizen in describing the streaming dynamics. Despite its impractical, we present it here to provide a more comprehensive discussion, in order to show the limitations and difficulties of more sophisticated analytical instantiations.

To cover the long-tail effects, the dependency between the U and Z should be considered. Specifically, each local latent variable z_i has two components independent of each other, i.e., a_i and λ_i. The a_i controls observation x_i to concentrate on a certain value, which is further determined by global variable $U = \{\mu_w, \varphi_w\}$. We further assume that a_i is independent of each other when given U. Different from a_i, λ_i is independent of U, and it controls the long-tail skewed distribution of x_i. More clearly, Eqn 7 is changed into the following.

$$f(a_i|\mu_w, \varphi_w) = e^{-(a_i-\mu_w)^2 \times \varphi_w/2} \times \sqrt{\varphi_w} \times const \quad (17)$$

$$f(x_i|a_i, \lambda_i) = \lambda_i \times e^{-\lambda_i(x_i-a_i)} \times const \quad (18)$$

Eqn 17 enforces that $a_i \sim \mathcal{N}(\mu_w, 1/\varphi_w)$ given the global variable μ_w, φ_w, and a_i is i.i.d to each other, and Eqn 18 models x_i to follow an exponential distribution concentrated on a_i, and have a tail assigned by λ_i. As a result, we rewrite the joint distribution function from Eqn 8 into Eqn 19. Similar to Eqn 9, the $q(\mu_w)$ under this setting is given as Eqn 20. The posterior Gaussian distribution of μ_w is still Gaussian as $\mu_w \sim \mathcal{N}(\frac{\sum_{i=1}^{n}(\mathbb{E}(a_i))\mathbb{E}(\varphi_w)+\mu_0\tau_0}{\mathbb{E}(\varphi_w)n+\tau_0}, 1/(\mathbb{E}(\varphi_w)+\tau_0))$, and we can also acquire the estimated value and credible interval of μ_w as Section 5.1. However, the key difference is that $\mathbb{E}(\mu_w)$ is no longer linear to U (i.e., comparing Eqn 20 and Eqn 9), due to the involved $\mathbb{E}(\varphi_w)$ item, which can be further expanded as Eqn 21 by continuing [14] derivation. It is worth noting that $\mathbb{E}(\lambda_i)$ and x_i are further contained in $\mathbb{E}(a_i)$, and we omitted its details.

$$f(U, Z, X) = (\varphi_w)^{n/2} e^{-\varphi_w \times \sum_{i=1}^{n}((a_i-\mu_w)^2)} \times \prod_{i=1}^{n}(\lambda_i) e^{-\sum_{i=1}^{n}(\lambda_i(x_i-a_i))} \times p(\mu_w) \times p(\varphi_w) \times p(z_1 \cap z_2 \ldots \cap z_n|U) \times const \quad (19)$$

$$q(\mu_w) = \mathbb{E}_{\varphi_w, Z}(f(U, Z, X)) = const \times e^{-\frac{\mathbb{E}(\varphi_w)+\tau_0}{2} \times (\mu_w - \frac{\sum_{i=1}^{n}(\mathbb{E}(a_i))\mathbb{E}(\varphi_w)+\mu_0\tau_0}{\mathbb{E}(\varphi_w)n+\tau_0})^2} \quad (20)$$

where the prior knowledge $p(\mu_w)$ is given as $\mu_w \sim \mathcal{N}(\mu_0, 1/\tau_0)$

$$\mathbb{E}(\varphi_w) = (n/2 + \alpha_\tau)/(\frac{\sum_{i=1}^{n}(\mathbb{E}(a_i) - \mathbb{E}(\mu_w))^2}{2} + \beta_\tau) \quad (21)$$

where the prior knowledge $p(\varphi_w)$ is given as $\varphi_w \sim \Gamma(\alpha_\tau, \beta_\tau)$

It seems that the $\bar{\mu_w} = \mathbb{E}(\mu_w)$ estimation follows some analytical restrictions, and we can still theoretically let analytical VI converge into mean-field equilibria [14]. However, we found that the ELBO optimization (Eqn 5) under this instantiation can not be well supported by common optimizers, such as the ADAM and SGD in *Pytorch* [3]. **Specifically, a full unfold of putting Eqns 20 and 21 into Eqn 5 requires a catastrophically complicated tensor graph, which prevents the common optimizers from automatically computing the gradients**. Considering the extreme difficulty of implementing a custom optimizer from scratch or further reshaping the Eqn 5 under this instantiation, we give up the attempt in this case.

REFERENCES

[1] [n.d.]. A Benchmark for Real-Time Relational Data Feature Extraction. https://github.com/decis-bench/febench. Last Accessed: 2023-01-03.

[2] [n.d.]. OpenMLDB Use Cases. https://openmldb.ai/docs/en/main/use_case/index.html. Last Accessed: 2022-09-23.

[3] 2023. *Pytorch homepage, https://pytorch.org/*.

[4] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European conference on computer systems*. 29–42.

[5] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. *Watermarks in Stream Processing Systems: Semantics and Comparative Analysis ofApache Flink and Google Cloud Dataflow*. Technical Report. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. (2015).

[7] Abdullah Alsaedi, Nasrin Sohrabi, Redowan Mahmud, and Zahir Tari. 2023. RADAR: Reactive Concept Drift Management with Robust Variational Inference for Evolving IoT Data Streams. In *Proceedings of the 39th IEEE International Conference on Data Engineering (ICDE2023)*. IEEE.

[8] Ahmed Awad, Jonas Traub, and Sherif Sakr. 2019. Adaptive Watermarks: A Concept Drift-based Approach for Predicting Event-Time Progress in Data Streams.. In *EDBT*. 622–625.

[9] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

[10] Savong Bou, Hiroyuki Kitagawa, and Toshiyuki Amagasa. 2021. Cpix: real-time analytics over out-of-order data streams by incremental sliding-window aggregation. *IEEE Transactions on Knowledge and Data Engineering* 34, 11 (2021), 5239–5250.

[11] Tamara Broderick, Nicholas Boyd, Andre Wibisono, Ashia C Wilson, and Michael I Jordan. 2013. Streaming variational bayes. *Advances in neural information processing systems* 26 (2013).

[12] Badrish Chandramouli, Mohamed Ali, Jonathan Goldstein, Beysim Sezgin, and Balan Sethu Raman. 2010. Data stream management systems for computational finance. *Computer* 43, 12 (2010), 45–52.

[13] Roger Dingledine, Nick Mathewson, Paul F Syverson, et al. 2004. Tor: The second-generation onion router.. In *USENIX security symposium*, Vol. 4. 303–320.

[14] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *Journal of Machine Learning Research* (2013).
[15] Yuanzhen Ji, Jun Sun, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2016. Quality-driven disorder handling for m-way sliding window stream joins. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 493–504.
[16] 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*. 889–894.
[17] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data*. 631–646.
[18] Nikos R Katsipoulakis, Alexandros Labrinidis, and Panos K Chrysanthis. 2020. Spear: Expediting stream processing with accuracy guarantees. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1105–1116.
[19] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.
[20] Kaiyu Li, Yong Zhang, Guoliang Li, Wenbo Tao, and Ying Yan. 2018. Bounded approximate query processing. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2018), 2262–2276.
[21] Ming Li, Mo Liu, Luping Ding, Elke A Rundensteiner, and Murali Mani. 2007. Event stream processing with out-of-order data arrival. In *27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07)*. IEEE, 67–67.
[22] Yiming Li, Yanyan Shen, and Lei Chen. 2022. Camel: Managing Data for Efficient Stream Learning. In *Proceedings of the 2022 International Conference on Management of Data*. 1271–1285.
[23] Qingzhi Ma and Peter Triantafillou. 2019. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*. 1553–1570.
[24] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2017. Streambox: Modern stream processing on a multicore machine. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)* (Santa Clara, CA, USA) *(Usenix Atc '17)*. USENIX Association, Berkeley, CA, USA, 617–629. http://dl.acm.org/citation.cfm?id=3154690.3154749
[25] Adrian Michalke, Philipp M Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. 2021. An energy-efficient stream join for the Internet of Things. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. 1–6.
[26] Douglas C Montgomery, Cheryl L Jennings, and Murat Kulahci. 2015. *Introduction to time series analysis and forecasting*. John Wiley & Sons.
[27] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 493–505. https://www.usenix.org/conference/atc16/technical-sessions/presentation/najafi
[28] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 493–505.
[29] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. Streamapprox: Approximate computing for stream analytics. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 185–197.
[30] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel Index-Based Stream Join on a Multicore CPU. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2523–2537. https://doi.org/10.1145/3318464.3380576
[31] Yang Song, Yunchun Li, Hailong Yang, Jun Xu, Zerong Luan, and Wei Li. 2021. Adaptive watermark generation mechanism based on time series prediction for stream processing. *Frontiers of Computer Science* 15 (2021), 1–15.
[32] Binh Tang and David S Matteson. 2021. Probabilistic transformer for time series analysis. *Advances in Neural Information Processing Systems* 34 (2021), 23592–23608.
[33] 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* (Athens, Greece) *(Sigmod '11)*. Acm, New York, NY, USA, 625–636. https://doi.org/10.1145/1989323.1989389
[34] Sifan Wu, Xi Xiao, Qianggang Ding, Peilin Zhao, Ying Wei, and Junzhou Huang. 2020. Adversarial sparse transformer for time series forecasting. *Advances in neural information processing systems* 33 (2020), 17105–17115.
[35] Kai Zeng, Shi Gao, Barzan Mozafari, and Carlo Zaniolo. 2014. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 277–288.
[36] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf
[37] Hao Zhang, Xianzhi Zeng, Shuhao Zhang, Xinyi Liu, Mian Lu, Zhao Zheng, and Yuqiang Chen. 2023. Scalable Online Interval Join on Modern Multicore Processors in OpenMLDB. (2023).
[38] Shuhao Zhang, Yancan Mao, Jiong He, Philipp M Grulich, Steffen Zeuch, Bingsheng He, Richard TB Ma, and Volker Markl. 2021. Parallelizing intra-window join on multicores: An experimental study. In *Proceedings of the 2021 International Conference on Management of Data*. 2089–2101.
[39] Shuhao Zhang, Yingjun Wu, Feng Zhang, and Bingsheng He. 2020. Towards concurrent stateful stream processing on multicore processors. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1537–1548.