
# *Erebus*: Explaining the Outputs of Data Streaming Queries

Dimitris Palyvos-Giannas
Chalmers University of Technology
Gothenburg, Sweden
palyvos@chalmers.se

Katerina Tzompanaki
ETIS, CY Cergy-Paris University, ENSEA, CNRS, UMR8051
Cergy, France
aikaterini.tzompanaki@cyu.fr

Marina Papatriantafilou
Chalmers University of Technology
ptrianta@chalmers.se

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

## ABSTRACT

In data streaming, why-provenance can explain why a given outcome is observed but offers no help in understanding why an expected outcome is missing. Explaining *missing answers* has been addressed in DBMSs, but these solutions are not directly applicable to the streaming setting, because of the extra challenges posed by limited storage and by the unbounded nature of data streams.

With our framework, *Erebus*, we tackle the unaddressed challenges behind explaining missing answers in streaming applications. *Erebus* allows users to define *expectations* about the results of a query, verifying at runtime if such expectations hold, and also providing explanations when expected and observed outcomes diverge (missing answers). To the best of our knowledge, *Erebus* is the first such solution in data streaming. Our thorough evaluation on real data shows that *Erebus* can explain the (missing) answers with small overheads, both in low- and higher-end devices, even when large portions of the processed data are part of such explanations.

**PVLDB Reference Format:**
Dimitris Palyvos-Giannas, Katerina Tzompanaki, Marina Papatriantafilou, and Vincenzo Gulisano. *Erebus*: Explaining the Outputs of Data Streaming Queries. PVLDB, 16(2): 230 - 242, 2022.
doi:10.14778/3565816.3565825

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/dmpalyvos/erebus.

## 1 INTRODUCTION

Data streaming [1, 32] allows users to query unbounded datasets in a continuous manner with Stream Processing Engines (SPEs) [4, 10]. SPE users can connect query outputs to their contributing inputs using *why-provenance* [23], which, however, cannot explain why an expected output is *missing* since, by definition, it only records information related to generated outputs. Such expected but *missing answers* can indicate problems in the query, the input data, or both. Knowing *if* and *why* answers are missing can be critical in validating query correctness and understanding issues, unanticipated during the query design, which now need to be addressed [12].


Proceedings of the VLDB Endowment, Vol. 16, No. 2 ISSN 2150-8097.
doi:10.14778/3565816.3565825

In DBMSs, missing answers can be explained through *provenance of missing answers* (also known as *why-not provenance*) [9]. Since DBMSs manage bounded datasets, though, existing why-not provenance solutions are not applicable to the streaming setting. First, such solutions assume that the query can be replayed with the exact same (bounded) data. This is not necessarily the case in streaming queries, whose one-pass analysis is commonly motivated by the inefficiency of maintaining all the data to be analyzed [1]. Second, streaming queries produce results continuously [32], introducing the issue of discerning missing from yet-to-be-produced answers, which lies outside DBMSs' closed-world assumption.

Example - Part 1 introduces a scenario of missing answers and their explanations in a streaming use-case from our evaluation.

**Example - Part 1**

An analyst runs the query of Figure 1 over household power data to find faulty plugs. The figure shows the operators, tuple schemas, and attribute transformations, discussed in the next sections. At 23:30, the analyst is notified by a customer about a broken plug: since 22:00 the display of that plug has been showing seemingly random power loads between 0 and 29 watts, but nothing is connected to it. The analyst checks the mean usage of that whole customer household and it has been above 34 watts since 22:00. The analyst believes the query should have produced an alert in this situation, but no alert has been produced yet. Is the alert going to be delivered in the immediate future or is the query ignoring the faulty plug's data? To understand the problem of the query (if any), the analyst would benefit from system-generated information on whether relevant alerts are finally generated or if tuples that could have contributed to the alerts were pruned by an operator.

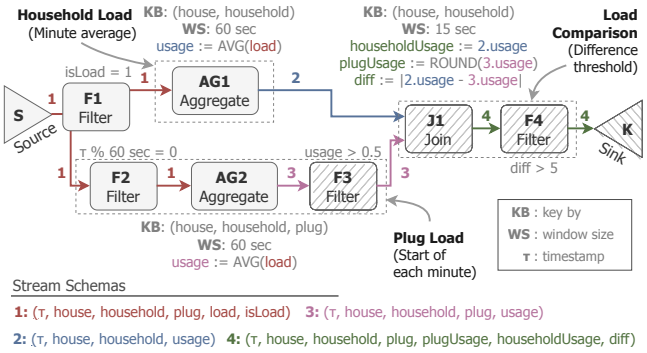


**Figure 1: Example streaming query from *Erebus*' evaluation.**

*Contribution.* Considering the above, we ask: *How can we efficiently monitor user-defined expectations about query results and explain missing answers in a streaming manner?* We propose a new framework, called *Erebus*,[1] which allows users to define *expectations* about query results at runtime. *Erebus* notifies users about whether their expectations were met and, if they were not, it explains *"why not"*, relieving users from the necessity of manual query debugging or replaying of the data. We make the following contributions:

- We formally define the problem of explaining missing answers in streaming. We propose query-based explanations, which identify operators pruning off tuples that could have contributed to missing results together with such pruned-off tuples.
- We formalize the notion of an *expectation predicate* for a streaming query as boolean conditions describing user expectations about the query outputs at certain time intervals.
- We describe *Erebus*, the first framework that can provide explanations for already processed and yet to be processed streaming data, augmented with *explanation markers* indicating if the produced explanations are finished and complete, at a specific time.
- *Erebus* can work as a holistic why- and why-not provenance solution by seamlessly integrating with why-provenance tools to connect its answers to their contributing source tuples (both for met expectations and for pruned-off tuples).
- We implement *Erebus* [27] on top of Apache Flink [10] and evaluate it with a wide range of predicates on real-world and synthetic workloads, showing that its low overheads allow it to run alongside streaming queries, both on low- and higher-end devices.

We describe streaming in §2, introduce the expectations and explanations problem in §3, our methodology for translating predicates in §4, *Erebus*' design and implementation in §5, our evaluation in §6, related work in §7 and we conclude in §8.

## 2 PRELIMINARIES

In the DataFlow model [1, 10], *streams* are unbounded sequences of *tuples* $t$ defined as lists of attribute-value pairs, i.e., $(\tau : v_0, A_1 : v_1, \ldots, A_n : v_n)$, where $t.A_i$ denotes the value of $t$'s $i$−th attribute (omitting $t$ when clear from the context). The first attribute is the *timestamp* $\tau$. The *type* of tuple $t$ in stream $S$ is $t$'s attribute list excluding $\tau$ and denoted as $type(t)$ (or $type(S)$). We write $dom(A_i)$ to denote $A_i$'s domain. Attribute names are unique in each *type*.

A *streaming query* (or simply query) is a Directed Acyclic Graph composed of *Sources*, *operators* and *Sinks*. Sources produce *source tuples* (e.g., events from IoT sensors) and deliver them as *streams* to operators. Sources set the timestamp $\tau$ to the time when the corresponding event took place, i.e., the *event time*. Operators manipulate tuples through user-defined functions (UDFs) and can forward (potentially new) tuples or prune off tuples. Query results reach the Sinks as *sink tuples* and are delivered to downstream applications or end-users. For simplicity, we refer to Sources/Sinks as operators when there is no need to differentiate them. Operators assign a value to $\tau$ according to their semantics, and the values of the other attributes according to their UDFs. Given operators $O_1$, $O_2$, a *path* $g = [O_1, \ldots, O_2]$ is a list of consecutive operators connecting $O_1$ to $O_2$ in the query, with length $|g| \geq 2$. Two operators can be connected by multiple paths. Operator $O_1$ is *upstream* of operator $O_2$ (and $O_2$ is *downstream* of $O_1$), if there exists a path from $O_1$ to $O_2$.

Modern SPEs [2–4, 10] provide *native operators* like *Map* and *Filter* (*stateless*), and *Aggregate* and *Join* (*stateful*). Stateless operators do not maintain a state based on the tuples they process. A **Filter (F)** checks a user-defined condition on each input tuple $t$, forwarding $t$ if the condition holds. A **Map (M)** transforms each input tuple $t$ into an arbitrary number of output tuples by applying a UDF to $t$ and copying $t.\tau$ into each output.

Each stateful operator executes its computation over groups of tuples organized in time *windows*, represented as intervals $[L, R)$, where $L$ is the *left* and $R$ the *right boundary* of the window. A stateful operator is characterized by its window *size* $WS = R - L$ and *advance* $WA$ (the difference between the $L$ of consecutive windows). Left boundaries are at timestamps $L = nWA$, $n \in \mathbb{N}$ and right boundaries at $R = nWA + WS$, $n \in \mathbb{N}$. Here, we study the usual case in which $WA \leq WS$ [1]. A tuple $t$ *falls* in the window $[L, R)$ if $t.\tau \in [L, R)$.

An **Aggregate (AG)** applies a UDF to emit aggregated tuples for windows, optionally using a Key-By ($KB$) function to split tuples of different keys into different (aligned) windows (similarly to `GROUP BY` in DBMSs). A **Join (J)** joins two input streams by windowing and optionally keying each stream with a $KB$ function, and applying a UDF to each matching tuple pair to produce an output. Conditions on the outputs of AG/J (e.g., `HAVING`/`WHERE` clauses) can be enforced by adding a Filter after the respective AG/J. AG and J set the timestamp $\tau$ of each output $t$ to a fixed distance from the right boundary $R$ of the window from which the output is emitted [10], i.e., $t.\tau = R - \epsilon$, $0 \leq \epsilon \leq WS$, where $\epsilon$ is SPE-specific.

The event time of operators progresses in discrete, SPE-specific increments $\delta$ (e.g., 1 millisecond [10]), and thus $\tau \in \mathbb{N}$. Event time is usually tracked through *watermarks*. Paraphrasing from [30]:

Definition 2.1. *The watermark $W_O^\omega$ of operator $O$ at* wall-clock time $\omega$ *is the earliest possible* event time *of tuples processed by $O$ from $\omega$ onward: $t_o.\tau \geq W_O^\omega, \forall t_o$ processed at (wall-clock) time[2] $\omega' \geq \omega$.*

Sources periodically propagate watermarks. An operator's watermark is the minimum watermark of its input streams. A stateful operator $O$ emits timestamp-sorted outputs for its windows with right boundaries up to $R$ after $O$'s watermark becomes $W_O^{\omega'} \geq R$.

## 3 PROBLEM ANALYSIS AND FORMALIZATION

Our goal is to allow users to identify and debug potential problems of a (running) query $\mathbb{Q}$ by 1) *validating* that $\mathbb{Q}$ is producing the expected results and 2) *explaining missing answers*, i.e., why (some) expected results were not produced by $\mathbb{Q}$ within a given time interval. Expressing user expectations as a boolean *expectation predicate* $P$ on the attributes of a sink stream of $\mathbb{Q}$, the former can be achieved by checking all sink tuples against $P$ and reporting any matches. For the latter, we focus on *query-based* explanations [23], pinpointing culprit operators. To compute such explanations, we use the notions of successors and pruned tuples, defined below:

Definition 3.1 (Successor). *We say that tuple $t'$ is a* successor *of tuple $t$ (and write $t' \in succ(t)$) if $t'$ is produced by an operator $O$ when $O$ processes $t$ or a successor of $t$.*

[1] Erebus is the personification of darkness in Greek mythology.

[2] In the remainder, we refer to wall-clock time as simply *time*, unless it is necessary to differentiate it from event time in that context.

We say that operator $O$ *pruned* tuple $t$ if $t$ is processed by $O$ without producing any successors. To explain missing answers, we want to identify tuples that were pruned by some operator $O$ and *could* have led to a successor satisfying $P$, i.e., are *compatible* with $P$. We want to find all such pruned compatible tuples along with their pruning operator $O$, *in a streaming manner, without replaying the query*. This, in turn, requires the two issues below to be addressed.

First, $P$ is defined on the *type* of the results, which usually differs from the *type* of tuples processed by operators in other parts of $\mathbb{Q}$. Thus, $P$ needs to be *translated* so that it can be applied to tuples with different *type*s. Since it is impractical for users to manually perform this translation for any combination of $\mathbb{Q}$, $P$, and $O$, our solution needs to perform this step automatically for Filters and Joins, the only native operators that can prune tuples. In §5, we outline possible extensions for other pruning operators. Second, $P$ needs to be bounded in event time so that the delivery of explanations eventually terminates. However, $P$'s event-time boundaries can align in arbitrary ways with $\mathbb{Q}$'s state. If $P$ refers to $\mathbb{Q}$'s event-time future, operators can evaluate $P$ each time they prune a tuple. However, if $P$ overlaps with $\mathbb{Q}$'s past, $P$ must also be checked for tuples that have been *buffered*. The size of such a buffer is a trade-off between the explanation completeness and the buffering overheads.

Below, we formalize the concepts necessary to analyze the problem and address the issues highlighted above.

Definition 3.2 (Expectation Predicate). *Given a query $\mathbb{Q}$ and a Sink $K$ fed by a stream $S_K$, an* expectation predicate *(or simply predicate) is a boolean predicate $P_K$ on $type(S_K)$, expressed as:*

$$P_K = c_0(\tau) \wedge c_1(\mathbb{A}_1) \wedge \cdots \wedge c_m(\mathbb{A}_m),\ m \in \mathbb{N} \tag{3.1}$$

*where the* timestamp condition *is $c_0(\tau) = (l \leq \tau < r) \mid l, r \in \mathbb{N}$ and for $i \in [1, m]$, the conditions are: $c_i : \mathbb{A}_i \to \{0, 1\} \mid \mathbb{A}_i \subseteq type(S_K)$.*

We write $t_k \vDash P_K$ if $t_k$ *satisfies* $P_K$, or $t_k \nvDash P_K$ otherwise. Also, we write $c_i(t_k)$ to denote that $c_i$ is applied to the subset of attributes of $t_k$ to which it refers. *Erebus*' predicate can be a disjunction of multiple $P_K$ of Definition 3.2, allowing users to test arbitrarily complex conditions. Below we study one $P_K$ for simplicity, noting that it is straightforward to extend the analysis to multiple $P_K$.

**Example - Part 2**

The query of Example - Part 1 compares the load of each plug at the start of every minute (F2-AG2-F3) with the average load of the household for the same minute (AG1), producing an alert if the difference between the two exceeds a threshold (J1-F4). In Figure 1, stream numbers $(1-4)$ point to the tuple schemas. For brevity, the figure does not show the complete operator UDFs but only how attributes are transformed (e.g., 3.usage rounded into 4.plugUsage by J1). Attributes with the same name are preserved between operators (e.g., house).

In our example, the usage of the broken plug is between $[0, 30)$ watts and the household average usage is 34 watts. Thus, looking at the Sink attributes, the analyst expects relevant alerts to have plugUsage $< 30$ and diff $> 4$. These expectations can be expressed as the predicate:

$$P_K = (22{:}00{:}00 \leq \tau < 00{:}20{:}01) \wedge (\text{diff} > 4) \wedge (\text{plugUsage} < 30)$$

The analyst assumes the alert could be delayed until 00:20 and, to inspect the problem for any customer, poses no condition on the household.

In order to *translate* $P_K$ and evaluate it on tuples pruned by upstream operators of $K$, we will use static information about how operators of $\mathbb{Q}$ transform the timestamps and other tuple attributes. This information is encoded in the *timestamp* and the *attribute mappings*, defined below for an arbitrary operator and path $g$ in $\mathbb{Q}$. For tuples $t_1, t_{|g|}$, inputs of operators $O_1, O_{|g|}$, we write $t_{|g|} \in succ^g(t_1)$ to denote that $t_{|g|}$ is a successor of $t_1$ through path $g$.

Definition 3.3 (Timestamp Mapping). *For any operator $O$, the single-operator timestamp mapping $\hat{T}_O$ is a set of pairs of timestamp values, matching the timestamps of all possible pairs of input and output tuples of $O$. If $O$ is stateless, then $\hat{T}_O = \{(\tau, \tau) \mid \tau \in \mathbb{N}\}$. Otherwise, if $O$ is stateful with window size $WS$ and advance $WA$, then $\hat{T}_O = \{(\tau, nWA + WS - \epsilon) \mid n \in \mathbb{N},\ \tau \in [nWA, nWA + WS)\}$.*

*Extending to a path $g$, the transitive* timestamp mapping *$T^g$ is a set of pairs of timestamp values, such that $t_{|g|} \in succ^g(t_1) \Rightarrow (t_1.\tau, t_{|g|}.\tau) \in T^g$. We construct $T^g$ by combining the timestamp mappings $\hat{T}_{O_i}$ for each $O_i$ in $g$, requiring for each $(\tau_1, \tau_{|g|}) \in T^g$ that:*

$$\exists \tau_1, \ldots, \tau_{|g|} : (\tau_i, \tau_{i+1}) \in \hat{T}_{O_i},\ i = 1, \ldots, |g| - 1$$

For attributes other than the timestamp, we assume that users or static analysis tools provide metadata describing how the attribute value of an input tuple of operator $O$ is transformed to a certain attribute value of its successors through $O$ [12, 30]. Starting with a single operator and extending to a path, we define such metadata:

Definition 3.4 (Attribute Mapping). *Being $S_i$ an input stream of operator $O_i$ and $S_{i+1}$ an output stream of $O_i$ (and an input stream of operator $O_{i+1}$), the single-operator attribute mapping is the set:*

$$\hat{M}_{O_i} = \left\{ (A, A', f) \;\middle|\; \begin{array}{l} A \in type(S_i), A' \in type(S_{i+1}) \\ t_{i+1} \in succ(t_i) \Rightarrow f(t_i.A) = t_{i+1}.A' \end{array} \right\}$$

*Extending to a path $g$, being $S_1$, $S_{|g|}$ an input stream of the first and last operator in $g$, the transitive* attribute mapping *is:*

$$M^g = \left\{ (A, A', f) \;\middle|\; \begin{array}{l} A \in type(S_1), A' \in type(S_{|g|}) \\ t_{|g|} \in succ^g(t_1) \Rightarrow f(t_1.A) = t_{|g|}.A' \end{array} \right\}$$

*$M^g$ can be constructed by combining the attribute mappings $\hat{M}_{O_i}$ of each operator $O_i$ in $g$. For each triplet $(A^1, A^{|g|}, f) \in M^g$, we require:*

$$\exists A^1, \ldots, A^{|g|} : (A^i, A^{i+1}, f_i) \in \hat{M}_{O_i},\ i = 1, \ldots, |g| - 1 \text{ and}$$
$$f = f_1 \circ \cdots \circ f_{(|g|-1)}$$

Joins, whose input streams can have different *type*s, can have two $\hat{M}_{O_i}$. Notice that, depending on each operator $O$'s semantics, it might not be possible to include all of $O$'s input and output attributes in $M^g$. For example, if $O$ computes many-to-one functions (e.g., an Aggregate computing a *mean* of its window), the attribute mapping can describe the attribute corresponding to the aggregation key (which does not change between the input and the output) but not the one of the aggregated value, because the value of the latter depends on multiple inputs. On the other hand, if an Aggregate is computing, e.g., *max*, it can be possible to also include the aggregated attribute (which depends on one input) in $M^g$. We formalize our handling of such attributes absent from $M^g$ in §4.

We can now use the timestamp and attribute mappings to introduce the notion of *potential successors* as follows:

Definition 3.5 (Potential Successors). *For path $g$, tuple $t_1$ an input of operator $O_1$, and $S_{|g|}$ the input stream of operator $O_{|g|}$, the* potential successors *of $t_1$ over $g$ is a superset of the successors of $t_1$, with each element having type$(S_{|g|})$, defined as follows:*

$$psucc^g(t_1) = \left\{ (\tau : v_0, A_1 : v_1, \ldots, A_n : v_n) \;\middle|\; \begin{array}{l} (t_1.\tau, v_0) \in T^g, \\ v_i \in \mu^g(A_i, t_1),\ i > 0 \end{array} \right\}$$

*where $\mu^g$ is defined as:*

$$\mu^g(A,t) = \begin{cases} \{f(t.A')\} & , \text{if } \exists (A', A, f) \in M^g \\ dom(A) & , \text{otherwise} \end{cases}$$

Intuitively, $\mu^g(A,t)$ returns either a singleton containing the converted value of attribute $A'$ from tuple $t$ if $A'$ is mapped to $A$ in $M^g$, or $dom(A)$ if $A$ is not mapped in $M^g$. Potential successors describe what is produced by $\mathbb{Q}$ but also what *could* have been produced due to some pruned tuple $t_1$ if $t_1$ (and $t_1$'s successors) had not been pruned out by some operator(s). Since the set is computed from the static mappings $T^g$, $M^g$, it does not require a replay of $\mathbb{Q}$.

Next, we need to define the tuples compatible with the predicate since these are the tuples we are primarily interested in.

Definition 3.6 (Compatible Tuple). *Given predicate $P_K$ defined at Sink $K$ of query $\mathbb{Q}$ and any input tuple $t_k$ of $K$, we define:*

$$compatible(t_k, P_K) \equiv t_k \vDash P_K$$

*Additionally, for operator $O$ upstream of $K$, connected to $K$ through a set of paths $\mathbb{G}$, for any tuple $t_o$ input of $O$ we define:*

$$compatible(t_o, P_K) \equiv \bigvee_{g \in \mathbb{G}} (\exists t_k \in psucc^g(t_o) : compatible(t_k, P_K))$$

Thus, a tuple $t_o$ is compatible with $P_K$ if it has at least one potential successor at $K$ (over any path) that satisfies $P_K$. Using the above, we define an *explanation* as a pair of a compatible tuple and its pruning operator (or $K$ if the tuple is at the Sink):

Definition 3.7 (Explanations). *Let $K$ be a Sink of a query $\mathbb{Q}$ and $P_K$ a predicate activated at $\mathbb{Q}$ at time $\omega_1$. We define the* explanations *of $P_K$ at time $\omega_2 > \omega_1$ as the set of pairs 1) $(t_k, K)$, where tuple $t_k$ arrived at $K$ before event time $W_K^{\omega_2}$ and for which it holds $compatible(t_k, P_K)$, and 2) $(t_o, O)$, where tuple $t_o$ was pruned by operator $O$ of $\mathbb{Q}$ before $W_O^{\omega_2}$ and for which it holds $compatible(t_o, P_K)$.*

**Example - Part 3**

Taking $P_K$ from Example - Part 2 and supposing F3 pruned tuples $t_1$=($\tau$=23:50, house=14, household=5, plug=7, usage=0.4) and $t_2$=($\tau$=18:59, house=14, household=5, plug=7, usage=0.3), we would like to know if $t_1$ and/or $t_2$ are compatible with $P_K$. F3 is connected to K by path $g$ = [F3, J1, F4, K], containing one stateful operator with $WS = WA = 15$s.

From Definition 3.3, we compute the relevant timestamp pairs for $t_1.\tau$ and $t_2.\tau$ in $T^g$, which are (23:50:00, 23:50:14), (18:59:00, 18:59:14) (assuming $\epsilon = 1$). The attribute mapping $M^g$ (Definition 3.4) can be computed from the attribute transformations of each operator, shown in the Figure 1: $M^g = \{(\ldots, (\text{plug}, \text{plug}, =), (\text{usage}, \text{plugUsage}, \texttt{ROUND})\}$.

Based on $T^g$, $M^g$, $t_1$ is compatible with $P_K$ because the $\tau$ and *plugUsage* of its potential successors satisfy $P_K$ (and the diff of its potential successors can take any value). This leads to the explanation $(t_1, \text{F3})$, which indicates that F3 could be responsible for the absence of the expected alert. Tuple $t_2$ is not compatible with $P_K$ because its potential successors have $\tau$ = 18:59:14, which does not satisfy $P_K$.

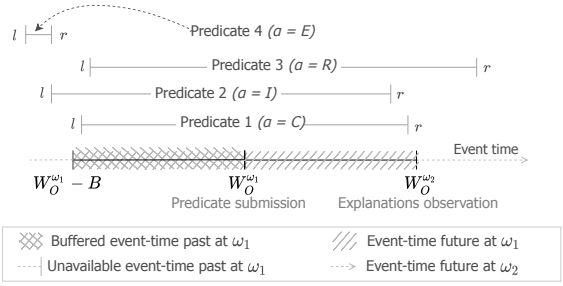


**Figure 2: Examples of the four possible explanation markers for an operator $O$ and predicates with different time intervals.**

When the input data is not maintained persistently, to provide explanations for $P_K$ whose boundaries are before $W_O^{\omega_1}$, it is necessary to temporarily *buffer* pruned data in the past, and retroactively evaluate $P_K$ on that data after $P_K$ is submitted. Supposing that $B$ past event-time units are buffered, explanations at time $\omega_2$ will be in event times $[W_O^{\omega_1} - B, W_O^{\omega_2}]$, as shown in Figure 2. Depending on $P_K$'s boundaries and/or the values of $W_O^{\omega_1}$ and $B$, it is possible that 1) some compatible tuples in the past have left the buffer and will not be considered and 2) some future tuples have not yet been evaluated. We put explanations into a predicate-query time alignment perspective, by defining *explanation markers*, as follows:

Definition 3.8 (Explanation Marker). *An* explanation marker *of operator $O$ at time $\omega_2$, for a predicate activated at time $\omega_1 < \omega_2$ is a pair $(O, \alpha)$, where $\alpha \in \{R, C, I, E\}$ characterizes the explanations associated with $O$ at $\omega_2$:* R *(running) if explanations are still being produced at $O$;* C *(complete) if $O$'s explanations are finished and complete;* I *(incomplete) if $O$'s explanations are finished but more might have been produced given a larger buffer;* E *(empty) if $O$ has no explanations because no (available) tuple could satisfy the predicate.*

Figure 2 shows the time intervals of four example predicates (Predicates 1 – 4) submitted at time $\omega_1$, along with their explanation markers at time $\omega_2$. The x-axis is the event time, marking the time of predicate submission $W_O^{\omega_1}$ and observation of explanations $W_O^{\omega_2}$. Predicate 1's explanation marker is *C* because it has finished producing explanations and was evaluated for all tuples in its boundaries. Predicate 2's marker is *I* as it has finished producing explanations and, when it was submitted, $l < W_O^{\omega_1} - B$ thus, some explanations may have not been possible to observe. Predicate 3's marker is *R* because $W_O^{\omega_2} < r$. Finally, Predicate 4's marker is *E* because all its explanations refer to the non-buffered event-time past.

*Problem Statement.* Given a user-defined predicate $P_K$ for a Sink of query $\mathbb{Q}$, we want to produce explanations and their markers in a streaming manner without replaying the data or requiring manual user intervention. To achieve this, we need to 1) develop a translation technique to identify tuples compatible with $P_K$ at the operators of $\mathbb{Q}$, 2) intercept pruned tuples and result tuples, and 3) handle the buffering of past tuples. In the following, we develop the theoretical methodology to translate predicates, analytically proving its correctness (§4) and describe the algorithmic implementation of our framework, *Erebus*, which includes tuple interception and buffering (§5), showing how it satisfies the goals defined above.

## 4 PREDICATE TRANSLATION

To produce explanations without manual user intervention, we want to define a method to automatically translate $P_K$ for each pruning operator $O$ to a new predicate $P_O$, which can decide whether a pruned tuple is compatible with $P_K$. The desired characteristics of this translation are formalized below:

Theorem 4.1. *Given predicate $P_K$ for Sink $K$ of query $\mathbb{Q}$, for each operator $O$ of $\mathbb{Q}$ there exists a translation $P_O$ of $P_K$, such that, for any tuple $t_o$ in the input stream[3] of $O$ it holds:*

$$compatible(t_o, P_K) \Rightarrow t_o \vDash P_O \tag{4.1}$$

*The translation $P_O$ can be computed statically based only on the set $\mathbb{M}$ of timestamp and attribute mappings $T^g$ and $M^g$ for all paths $g$ from $O$ to $K$. If all attributes of $P_K$ are mapped in $\mathbb{M}$ it also holds:*

$$t_o \vDash P_O \Rightarrow compatible(t_o, P_K) \tag{4.2}$$

The theorem states that the translation identifies all compatible tuples and that if all attributes used by $P_K$ are mapped in $\mathbb{M}$ the translation also returns no false positives, i.e., tuples $t_o$ that satisfy $P_O$ but have no potential successor $t_k$ compatible with $P_K$. Next, we develop such a translation of $P_K$ in two steps, starting with the timestamp and continuing with the rest of the tuple attributes used by $P_K$, combining the two translations to prove Theorem 4.1.

*Translating Timestamp Conditions.* For $P_K$'s timestamp condition $c_0(\tau) = l \le \tau < r$ and path $g$ connecting $O$ to $K$, we want to define a translation $c_0^g$ that satisfies Theorem 4.1 in the time dimension. We will construct $c_0^g$ by applying a pair of functions $\phi_l, \phi_r$ on the boundaries $l, r$ of $c_0$ as follows: $c_0^g(\tau) = \phi_l(l, r, g) \le \tau < \phi_r(l, r, g)$.

We will build a recursive algorithm that computes $\phi_l, \phi_r$, starting from the base case of $g = [O, K]$, i.e., operator $O$ directly connected to Sink $K$. Stateless operators do not require a timestamp translation (since they do not alter the timestamps), so we study stateful $O$ with window size $WS$ and advance $WA$ and include the stateless case in the relevant equations. From the operator definitions in §2, we observe that the outputs of $O$ (i.e., the inputs $t_k$ of $K$) have timestamps that only depend on the window that caused their generation. Thus, to translate $c_0$, we need to find the *first* and the *last* window of $O$ whose output timestamps fall into $c_0$'s range $[l, r)$. Being $l'$ the *left* boundary of the first and $r'$ the *right* boundary of the last such windows, it holds for each input tuple $t_o$ of $O$:

$$(\exists t_k \in psucc^g(t_o) : (l \le t_k.\tau < r)) \Leftrightarrow (l' \le t_o.\tau < r') \tag{4.3}$$

From §2, we know that the left window boundaries of $O$ are at event times $L = nWA$, its right boundaries at $R = nWA + WS$ and output timestamps at $\tau = nWA + WS - \epsilon$, where $n \in \mathbb{N}$ and $0 \le \epsilon \le WS$.

Being $L_1$ the left boundary of $O$'s first window having an output timestamp $\tau \ge l$, it holds that $\tau = nWA + WS - \epsilon \ge l$ and $(n-1)WA + WS - \epsilon < l$, which gives $\frac{l-WS+\epsilon}{WA} \le n < \frac{l-WS+\epsilon}{WA} + 1$. The latter is the definition of *ceil* and thus, the desired left boundary is:

$$L_1(l, WS, WA) = \lceil \frac{l - WS + \epsilon}{WA} \rceil WA \tag{4.4}$$

Tuples in the window starting at $L_1$ are compatible with $c_0$ only if the window's output timestamp falls in $[l, r)$. Denoting $c_0$'s interval length as $I = r - l$ (useful for the recursive algorithm below), this constraint is expressed as $\tau = L_1(l) + WS - \epsilon < l + I$ (we omit $L_1$'s arguments $WS$, $WA$ when clear from the context). Putting it all together, the translation $l'$ for $c_0$'s left boundary $l$ through $O$ is:

$$l'(l, I, WS, WA) = \begin{cases} l & \text{, if } WS = 0 \text{ (stateless)} \\ L_1(l) & \text{, if } L_1(l) + WS - \epsilon < l + I \\ \texttt{null} & \text{, otherwise} \end{cases} \tag{4.5}$$

Being $R_2$ the right boundary of $O$'s last window having an output timestamp $\tau < r$, it holds $\tau = nWA + WS - \epsilon \le r - \delta$ and $(n+1)WA + WS - \epsilon > r - \delta$ (subtracting time step $\delta$ to force the strict inequality), which are combined into $\frac{r-WS+\epsilon-\delta}{WA} - 1 < n \le \frac{r-WS+\epsilon-\delta}{WA}$. The latter is the definition of *floor*, and thus[4]:

$$R_2(r, WS, WA) = \lfloor \frac{r - WS + \epsilon - \delta}{WA} \rfloor WA + WS \tag{4.6}$$

As before, compatibility with $c_0$, requires the output timestamp $\tau \ge l = r - I$. Thus the translation $r'$ for $c_0$'s right boundary $r$ is:

$$r'(r, I, WS, WA) = \begin{cases} r & \text{, if } WS = 0 \text{ (stateless)} \\ R_2(r) & \text{, if } R_2(r) - \epsilon \ge r - I \\ \texttt{null} & \text{, otherwise} \end{cases} \tag{4.7}$$

We can utilize $l'$, $r'$ to translate $P_K$'s timestamp condition for any operator $O$ and any path $g$ from $O$ to $K$, as follows:

Definition 4.1 (Timestamp Translation). *For path $g$ from $O$ to $K$, the timestamp condition $c_0(\tau)$ of $P_K$ can be translated for operator $O$ into a new condition $c_0^g(\tau) = \phi_l(l, r, g) \le \tau < \phi_r(l, r, g)$ where:*

$$\phi_l(l, r, g) = \textsc{Translate}(g, |g|, l, r - l, \text{“LEFT”})$$
$$\phi_r(l, r, g) = \textsc{Translate}(g, |g|, r, r - l, \text{“RIGHT”})$$

Translate() is defined in Algorithm 1. It recursively applies $l'/r'$ upstream through $g$, from $K = g[|g|]$ to $O = g[1]$. It takes advantage of timestamp condition intervals having the same form as operator windows and eventually returns either the translated boundary or `null` if no successor of $O$'s inputs can fall into the given interval through $g$. It computes a new boundary $b'$ using $l'$ ($r'$) on L5-6 and loops trying to find an interval length $I$ starting (ending) at $b'$ that aligns with the upstream windows (L4-16). If it has reached $O$ it returns (L7). Otherwise, if it has found a valid $b'$ for curr (L8), it calls itself on the upstream of curr in $g$, with $I$ = curr.WS if curr is stateful, or with the same $I$ otherwise (L9-12). Then, it checks if the upstream $b'$ is valid. If it is (or if curr is stateless, and thus retrying is not an option) it returns (L13). Otherwise, it increases (decreases) the input boundary $b$ and reduces $I$ by curr.WA (L14-16) until it has found a valid upstream $b'$ or $I \le 0$ (L4).

Lemma 4.1. *Given a path $g$ in query $\mathbb{Q}$ from $O$ to $K$, a timestamp condition $c_0$, and $c_0$'s timestamp translation $c_0^g$ (using Definition 4.1) it holds for any $t_o$ that is an input of $O$:*

$$(\exists t_k \in psucc^g(t_o) : c_0(t_K)) \Leftrightarrow c_0^g(t_o)$$

[3] Note that Joins, which have two input streams with potentially different *type*s, have two such $P_O$s. This is taken into account in our implementation. For simplicity, and without loss of generality, in this section, we discuss only one translated predicate.

[4] When the equations give $n < 0$, i.e., $l < WS - \epsilon$ or $r \le WS - \epsilon$, $L_1$ and $R_2$ must return 0 and `null` respectively, but we omit this edge case from the equations for simplicity.

**Algorithm 1:** Time boundary translation.

```
1  Function TRANSLATE(Path g, Int j, Time b, Time I, String d)
       Data: A path g = [O, ..., K], the operator index j ∈ [1, |g|], the current
             boundary b, the interval length I and string d ("LEFT"/"RIGHT")
       Result: The translated boundary b' or null if no translation exists.
       /* Recursive function called by φ_l, φ_r in Definition 4.1 */
2      b' ← null // Initialize returned boundary
3      curr ← g[j] // Operator in current path position
4      while I > 0 do // Repeatedly try upstream combinations
5          if d = "LEFT" then b' ← l'(b, I, curr.WS, curr.WA)
6          else if d = "RIGHT" then b' ← r'(b, I, curr.WS, curr.WA)
7          if j = 1 then break // Reached O, return computed b'
8          if b' ≠ null then // Found curr's boundary, go upstream
9              if curr.WS > 0 then // Pass WS as upstream I
10                 b' ← TRANSLATE (g, j − 1, b', curr.WS, d)
11             else // Pass same interval I upstream
12                 b' ← TRANSLATE (g, j − 1, b', I, d)
           // Stop if b' found or cannot shift b (i.e., stateless)
13         if b' ≠ null or curr.WS = 0 then break
           // Shift b by WA, reduce I by WA and retry
14         if d = "LEFT" then b ← b + curr.WA
15         else if d = "RIGHT" then b ← b − curr.WA
16         I ← I − curr.WA
17     return b'
```

The lemma is a specialization of Theorem 4.1 for attribute $\tau$ and path $g$. It states that $t_o$ is (timestamp) compatible with $P_K$ (left side, from Definition 3.6) *if and only if* $t_o$ satisfies the timestamp translation $c_0^g$, thus indicating that $c_0^g$ returns no false positives.

Proof Sketch. By induction. Because $t_k \in psucc^g(t_o)$, it holds that $(t_o.\tau, t_k.\tau) \in T^g$. If all operators in $g$ are stateless, $\forall(\tau_o, \tau_k) \in T^g : \tau_o = \tau_k$. Since Algorithm 1 does not alter $l$ or $r$ for stateless operators, the lemma holds. If there is at least one stateful operator in $g$: the base case is handled by Equation 4.3; for the inductive step, if operator $n+1$ is stateless, neither the operator nor the algorithm changes the boundary; if operator $n+1$ is stateful, then the translation is identical to the base case. □

Figure 3 shows a path $g$ of the query of Figure 1 and the operator windows in $g$. Supposing a $P_K$ with $c_0 = 125 \leq \tau < 230$, i.e., time interval $[125, 230)$ of length $I = 105$ (shown next to K), we want to find $c_0^g$ for operator F2. For the windows of J1 and AG2, assuming $\epsilon = 1$ (i.e., the output timestamp of window $[L, R)$ is $R-1$), any inputs of F2 whose successors fall into $[125, 230)$ must be in AG2's window $[120, 180)$ and thus J1's window $[165, 180)$. The gray area shows how Algorithm 1 arrives at this solution. It tests J1's windows ending at 135, 150, and 165 but each upstream call at L8-12 returns null from AG2, as no output of AG2 falls in those intervals. Thus, the algorithm repeatedly sets $b \leftarrow b+15$ and $I \leftarrow I-15$ (L14-16) until reaching J1's window $[165, 180)$ in which the output timestamp of AG2's window $[120, 180)$ falls into. The left boundary of the latter (120) is the translated left boundary eventually returned by $\phi_l$. Similarly, $\phi_r$ eventually returns 180 as the translated right boundary.

*Translating Remaining Conditions.* For the remaining conditions $c_i$, $i > 0$, which use attributes from $type(S_K)$, we utilize the attribute mappings $M^g$ to develop a translation for each path $g$ connecting $O$ to $K$. Our translation is defined as follows:

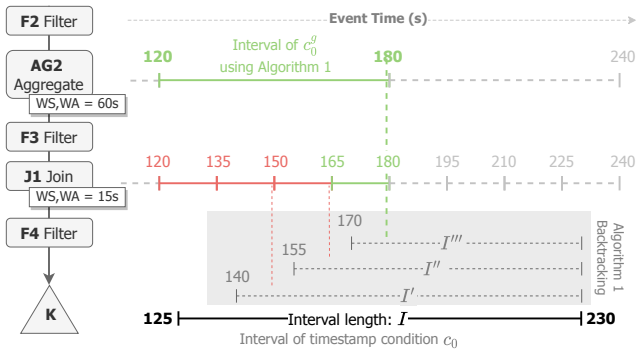


**Figure 3: Timestamp translation for an operator path of the query presented in Figure 1.**

Definition 4.2 (Attribute Translation). *For path $g$ connecting $O$ to $K$, each condition $c_i(\mathbb{A}_i)$, $i > 0$ of $P_K$ can be translated into a condition $c_i^g$ at operator $O$ as follows:*

$$c_i^g(\mathbb{A}_i^g) = \begin{cases} c_i(\mathbb{A}_i^g) & , \text{if } \forall A \in \mathbb{A}_i : \exists (A', A, f) \in M^g \\ 1 & , \text{otherwise} \end{cases}$$

*where $\mathbb{A}_i \subseteq type(S_K)$, $\mathbb{A}_i^g \subseteq type(S_O)$ and:*

$$\mathbb{A}_i^g = \left\{ f(A') \,|\, (A', A, f) \in M^g, A \in \mathbb{A}_i \right\}$$

Intuitively, conditions $c_i$, $i > 0$ have their every attribute $A$ replaced by the respective $A'$ from $M^g$ wrapped by function $f$, or they become 1, if they use attributes not mapped in $M^g$. The second rule is necessary because $M^g$ cannot always describe all attribute translations in a query (see §3). As described in the lemma below, each translated condition satisfies the requirements of Theorem 4.1.

Lemma 4.2. *Given a path $g$ in query $\mathbb{Q}$ from $O$ to $K$, mapping $M^g$ and $P_K$'s condition $c_i$, $i > 0$, it holds for $c_i$'s attribute translation $c_i^g$ (using Definition 4.2) that for any $t_o$ that is an input of $O$:*

$$(\exists t_k \in psucc^g(t_o) : c_i(t_k)) \Rightarrow c_i^g(t_o)$$

*Additionally, if all attributes in $c_i$ are in $M^g$, it holds:*

$$c_i^g(t_o) \Rightarrow \forall t_k \in psucc^g(t_o) : c_i(t_k)$$

Proof. We prove the first part by contradiction. Suppose, for some $t_o$, $\exists t_k \in psucc^g(t_o) : c_i(t_k)$ but $\neg c_i^g(t_o)$. From this, $c_i^g$ cannot be one that was set to 1 by the second rule of Definition 4.2. Thus, looking at the first translation rule there must exist attributes $A \in type(S_K)$, $A' \in type(S_O)$ so that $f(t_o.A') \neq t_k.A$. But from Definition 3.5, we have $t_k \in psucc^g(t_o) \Rightarrow f(t_o.A') = t_k.A, \forall A \in \mathbb{A}_i : (A', A, f) \in M^g$, leading to a contradiction. Thus, the first part of the lemma holds. The above analysis highlights that, for the first rule of Definition 4.2, it holds $\forall t_k \in psucc^g(t_o) : c_i^g(t_o) = c_i(t_k)$. When all attributes of $c_i$ are in $M^g$, this is the only translation rule applied, proving the second part of the lemma. □

*Translating the Whole Predicate.* Having shown correct translations for any condition over a single path, we can define the full translation of $P_K$ for operator $O$ over all paths from $O$ to $K$, by creating a translated sub-predicate for each path and taking the disjunction of all sub-predicates, as described below.

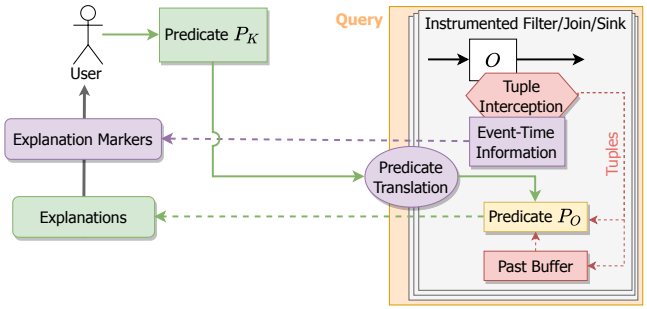


**Figure 4: High-level architecture of *Erebus*.**

Corollary 4.1 (Predicate Translation). *For the set of paths $\mathbb{G}$ connecting $K$ to $O$ in $\mathbb{Q}$, predicate $P_K$ can be statically translated using mappings $\mathbb{M}$ to a new predicate $P_O$ that satisfies Theorem 4.1:*

$$P_O = \bigvee_{g \in \mathbb{G}} (c_0^g(\tau) \wedge c_1^g(\mathbb{A}_1^g) \wedge \cdots \wedge c_m^g(\mathbb{A}_m^g))$$

*with $c_0^g$ from Definition 4.1 and $c_i^g(\mathbb{A}_i^g)$, $i > 0$ from Definition 4.2.*

Below, we prove Theorem 4.1 by showing how the translation described in Corollary 4.1 satisfies both parts of the theorem.

Proof of Theorem 4.1. We will show that Corollary 4.1 satisfies Equations 4.1 and 4.2. To prove Equation 4.1, assume that the left side holds, i.e., $compatible(t_o, P_K)$, and thus (see Definition 3.6): $\exists g \in \mathbb{G}, t_k \in psucc^g(t_o) : t_k \vDash P_K$. It follows directly from Lemmas 4.1, 4.2 that the sub-predicate(s) for any path $g$ that gave such $t_k$ will return true for $t_o$, and thus $t_o \vDash P_O$ ($P_O$ being a disjunction of all sub-predicates). To prove the second part of the theorem, i.e., Equation 4.2, assume all attributes of $P_K$ are in $\mathbb{M}$ and that $t_o \vDash P_O$. First, from Lemma 4.1 it holds that $c_0^g(t_0) \Rightarrow \exists t_k \in psucc^g(t_o) : c_0(t_k)$. Furthermore, from Lemma 4.2, it holds, for all $c_i \mid i > 0$ and any $t_o$, $c_i^g(t_0) \Rightarrow \forall t_k \in psucc^g(t_o) : c_i(t_k)$. From the above two relations, there is at least one $t_k$ (the one given from Lemma 4.1) that satisfies all $c_i$, and thus leads to $t_k \vDash P_K$. □

**Example - Part 4**

Continuing our running example from §3, the analyst's Sink predicate:

$$P_K = (22{:}00{:}00 \leq \tau < 00{:}20{:}01) \wedge (\text{diff} > 4) \wedge (\text{plugUsage} < 30)$$

can be translated for F3, using Corollary 4.1 (with $\epsilon = 1$), to the predicate:

$$P_{F3} = (22{:}00{:}00 \leq \tau < 00{:}20{:}00) \wedge 1 \wedge (\text{round}(\text{usage}) < 30)$$

$P_{F3}$ can be applied to the inputs $t_1, t_2$ of $F3$ from Example - Part 3 giving $t_1 \vDash P_{F3}$ and $t_2 \nvDash P_{F3}$, as expected from our previous analysis.

# 5 SYSTEM DESIGN AND IMPLEMENTATION

Here, we detail how *Erebus* addresses the challenges described in §3, leveraging the predicate translations presented in §4.

## 5.1 Architecture

Figure 4 outlines the architecture of *Erebus*. For a query $\mathbb{Q}$, users can specify which Sinks, Filters, and Joins are of interest for explanations and *Erebus* instruments them and evaluates predicates on their tuples. While all such operators can be instrumented, *Erebus* offers the flexibility of instrumenting only some of them, based on domain knowledge about $\mathbb{Q}$'s semantics. In this way, we avoid computing uninteresting explanations for the user while also reducing the performance impact of *Erebus* (see §6). The instrumentation, without any changes to the SPE, adds *tuple interception* logic, a *past buffer*, as well as functionality for receiving, translating, and evaluating predicates on intercepted tuples. All of $\mathbb{Q}$'s tuples are also instrumented with metadata necessary for *Erebus*.

*Erebus* intercepts 1) all input tuples of the Sink and 2) all pruned tuples of the instrumented Filters and Joins and sends them to the past buffer and to the predicate (if a predicate is active at that time). The size of the past buffer $B$ is a user-defined parameter. Predicates do not need to be compiled together with $\mathbb{Q}$ and can be submitted at any point during $\mathbb{Q}$'s runtime from a (possibly remote) channel outside the SPE. When a new predicate is submitted, an instrumented Sink activates it directly, whereas instrumented Filters and Joins first translate it as described in Corollary 4.1. Afterward, all intercepted tuples are checked using the predicate. At the same time, the new predicate is (asynchronously) evaluated on the tuples of the past buffer. Tuples satisfying the predicate are transmitted, together with the operator's identifier, as explanations back to the user. Based on the progress of instrumented operators' event time, *Erebus* emits explanation markers. Explanations and their markers, together with watermarks that track their event-time progress, are transmitted as an out-of-band stream, outside the SPE's control.

## 5.2 Operator Instrumentation

Algorithm 2 outlines the main operator instrumentation logic of *Erebus*. The operator is extended with additional state. SQ (Send Queue) and WQ (Work Queue) are bounded queues, the former containing references to explanations and watermarks and the latter "work orders", i.e., function pointers, together with their inputs. These queues are consumed by separate helper threads (not shown) that serialize and transmit the data (SQ) and execute the work orders (WQ). The pastBuffer of event-time size $B$ maintains a sliding window of the intercepted tuples of the operator, ordered by their timestamp. The overlap is a set used to remove duplicate explanations which might be produced during concurrent evaluations of tuples from the pastBuffer and from the present. The previous$P_O$ is the previous predicate evaluated on the tuples of the pastBuffer.

Procedure OnIntercepted handles intercepted tuples of instrumented Filters, Joins, and Sinks. It adds the intercepted tuple to the pastBuffer (L3) and, if the predicate is new (L2) it also adds it to the overlap, in order to prevent duplicate explanations in the case of a concurrent pastBuffer evaluation. Then, if the tuple satisfies the predicate (L4), an explanation is placed in SQ to be asynchronously emitted. We call such explanations *present explanations*, differentiating from *past explanations* coming from the pastBuffer. The instrumented Sink intercepts all its input tuples. Instrumented Filters intercept the tuples for which the Filter's condition returns false. Instrumented Joins, being stateful operators, intercept a tuple when such a tuple is no longer part of any window maintained by the Join if the tuple was not joined with any other tuple.

OnWatermark runs when the watermark of an instrumented operator increases, handling new predicates (L6, L12-20), past explanations (L19-23), explanation markers (L14-18, 24-29), and explanation

**Algorithm 2:** Instrumentation of Filter/Join/Sink.

```
   State: SQ, WQ        Bounded queues of explanations/watermarks and tasks
          pastBuffer    Sliding window of past intercepted tuples of size B
          overlap       Set of tuples that might be evaluated twice
          previous P_O  Previously evaluated predicate
1  Procedure OnIntercepted(Tuple t, Predicate P_O)
2      if P_O ≠ previousP_O then overlap.Add(t) // Prevent duplicates
3      pastBuffer.Add(t)
4      if P_O.Enabled and P_O.Evaluate(t) then SQ.Add((t,O))
5  Procedure OnWatermark(Time W_O^ω, Predicate P_O)
6      if P_O ≠ previousP_O then OnNewPredicate(W_O^ω, P_O)
7      WQ.Add( _ → ExplanationMarker(W_O^ω, P_O))
8      pastBuffer.RemoveBefore (W_O^ω − B)          // Shift pastBuffer
9      if pastBuffer.Empty() then wm ← max(0, W_O^ω − B)
10     else wm ← min(W_O^ω, pastBuffer[0].timestamp)
11     WQ.Add( _ → SQ.Add((wm, O)))                 // Asynchronously
12 Procedure OnNewPredicate(Time W_O^ω, Predicate P_O)
13     previousP_O ← P_O
14     if P_O.l = null or P_O.r = null or P_O.r < W_O^ω − B then
15         P_O.marker ← E                           // Unsatisfiable predicate
16     else
17         if P_O.l < W_O^ω − B then P_O.marker ← I // Incomplete
18         else P_O.marker ← C // Eventually complete
19         Buffer buffer ← pastBuffer.Copy(P_O.l, P_O.r)
20         WQ.Add( _ → EvaluatePast(P_O, buffer))   // Asynchronously
21 Procedure EvaluatePast(Predicate P_O, Buffer buffer)
22     for t ∈ (buffer ∖ overlap) do // Executed in helper thread
23         if P_O.Evaluate(t) then SQ.Add((t,O))
24 Function ExplanationMarker(Time W_O^ω, Predicate P_O)
25     if ¬ P_O.Enabled then return null // Ignore disabled predicate
26     if W_O^ω > P_O.r or P_O.marker = E then
27         P_O.Disable()
28         SQ.Add((O, P_O.marker))                  // Predicate finished
29     else return SQ.Add((O, R)) // Predicate still running
```

watermarks (L9-11). It first checks if $P_O$ was evaluated before (L6) and, if not, it calls OnNewPredicate. The latter sets previous$P_O$ to $P_O$, and then checks if $P_O$'s time condition can be satisfied (L14). If not, $P_O$'s marker attribute is set to $E$ (empty). If $P_O$ is satisfiable, OnNewPredicate checks if the leftmost boundary of $P_O$ is less than the minimum available timestamp (L17), in which case the marker of $P_O$ is set to $I$ (incomplete). Otherwise $P_O$'s marker is set to $C$ (complete). Then EvaluatePast is executed asynchronously, passing a view of the pastBuffer so that all tuples between $W_O^\omega - B$ and $W_O^\omega$ are eventually evaluated while the main operator thread concurrently alters the pastBuffer's contents (L19-23). It ignores tuples in overlap, as these have been already processed by OnIntercepted. Afterward, $O$ produces an explanation marker for the predicate, using WQ to ensure the marker is sent after the results of the past evaluation (L7). More specifically, ExplanationMarker checks if the watermark of $O$ is higher than the $P_O$'s rightmost boundary or if $P_O$ has no explanations to give and, if so, it signals that the explanations are finished (L26-28), returning the marker set by OnNewPredicate. Otherwise, it indicates that $P_O$ is still producing explanations, returning $R$ (L29). Finally, the contents of pastBuffer are shifted (L8) and a new explanation watermark is emitted through a task in WQ (L9-11), ensuring correct ordering.

*Extensions.* Our discussion focuses on Filters and Joins but can be extended to other pruning operators, provided they follow the semantics of our model (§2) and *Erebus* can access their input streams and instrument their UDFs. With these (realistic) assumptions, *Erebus* can alter the UDFs to mark all tuples with a successor and then intercept (as pruned) all input tuples of the operator that have no successor after the operator has finished processing them. This approach is used to instrument Joins in *Erebus*' implementation.

## 5.3 Performance Considerations

To quantify *Erebus*' overheads, we define, for each instrumented operator $O$ and predicate $P$ 1) the *interception ratio* $i_O$: the number of intercepted tuples per processed tuple based on $O$'s semantics (equal to 1 for Sinks and between 0 and 1 for Filters/Joins), 2) the *explanation ratio* $e_{P_O}$: the number of explanations per intercepted tuple based on $P$'s semantics, 3) *$P$'s cost* $c_{P_O}$: the evaluation time of $P$ per intercepted tuple at $O$, 4) the average *explanation size* $s_O$: proportional to the tuple size, and 5) the *rate* of $O$ $r_O$: the number of tuples it processes per time unit. Being $\mathcal{I}$ the set of instrumented operators of $\mathbb{Q}$, the main overheads added to $\mathbb{Q}$ by *Erebus* are described below and evaluated quantitatively in §6:

(1) The *predicate evaluation overhead* captures the computational cost of evaluating the predicate on each intercepted tuple and can be expressed as $\sum_{O \in \mathcal{I}} r_O \times i_O \times c_{P_O}$.
(2) The *explanation overhead* describes the computational and memory cost of serializing and sending explanations for compatible intercepted tuples, expressed as $\sum_{O \in \mathcal{I}} r_O \times i_O \times e_{P_O} \times s_O$.
(3) The computational and memory cost of *maintaining the past buffer* is proportional to the size $B$ of the buffer.
(4) The *metadata overhead* $D$ is the (mostly) computational cost of adding necessary *Erebus* metadata to all tuples of $\mathbb{Q}$.

The metadata overhead depends on *how Erebus* adds such metadata to $\mathbb{Q}$'s tuples and on the number of tuples serialized between $\mathbb{Q}$'s operators. Our implementation uses an *encapsulation* approach for *Erebus*' metadata, allowing *transparent* instrumentation of $\mathbb{Q}$ by enclosing each tuple *type* into an *Erebus* tuple. This has a low but observable overhead, caused by the extra layer added to the SPE's serialization. $D$ can be reduced if the user manually adds *Erebus*-related metadata to their tuple *type*s, as shown in §6.3.

# 6 EVALUATION

We evaluate *Erebus* for queries running in both low- and higher-end devices. §6.1 covers our setup. §6.2 studies the behavior of the example from §1 and explores the average performance of four real-world queries and eight custom predicates, also showing how *Erebus* performs when why-provenance is included in the explanations. §6.3 analyzes the overheads discussed in §5.3 using synthetic benchmarks and discusses strategies to mitigate them.

## 6.1 Evaluation Setup

*Hardware/Software.* We use Odroid-XU4 [21] devices (or simply *Odroid*) representative of edge devices [18], with Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa-core CPUs, 2 GB RAM, and Ubuntu 18.04.6, and a single-socket Intel Xeon-Phi server with 72 1.5GHz cores (4-way hyper-threading, 32KB L1, 1MB L2 cache), 102 GB RAM, and CentOS 7.9.2009. *Erebus* is implemented using Flink 1.14.0 (artifacts at [27]). Unless otherwise stated, experiments run for at least seven minutes and are repeated at least ten times. Plots show the resulting average and the 95% confidence interval (shaded area).

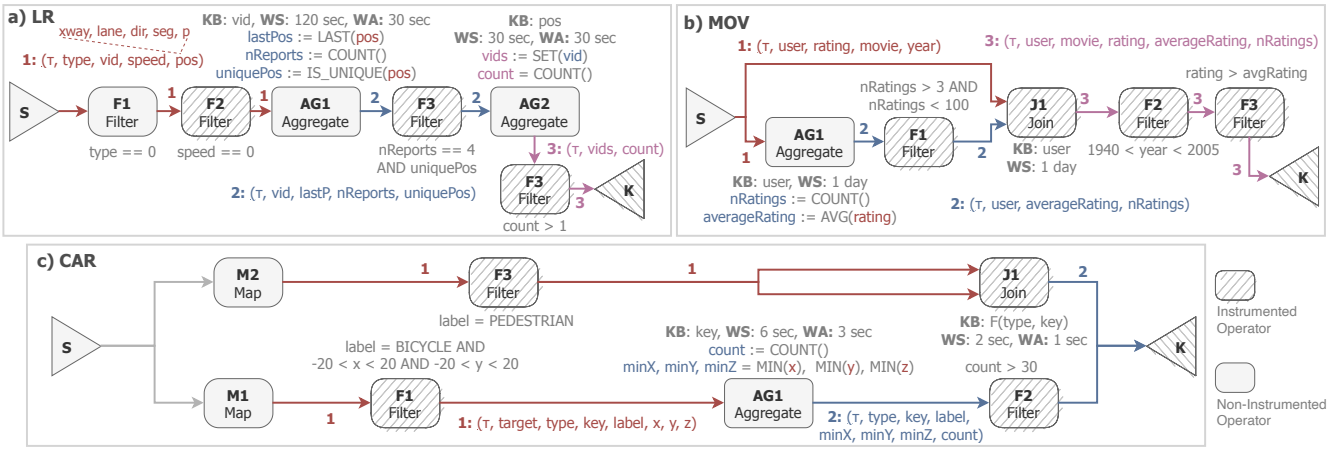


**Figure 5: Queries used in the evaluation (along with SGA, presented in Figure 1).**

*Queries.* To evaluate *Erebus* with different native operators and query graphs, we use four real-world queries (Figures 1 and 5), and a synthetic one. The figures show the queries' operators (marking instrumented ones) and describe operator functions and tuple *types*.

- **SGA**, introduced in our running example, detects anomalies in a Smart Grid, on a per-household basis, comparing, every minute, the average power consumption (AG1) with that reported by each plug at the beginning of the same minute (F2-AG2-F3-J1). It uses real-world data from the DEBS Grand Challenge 2014 [25].
- **LR** is a query from the Linear Road benchmark [6], detecting accidents by identifying vehicles stopped at the same position.
- **MOV** examines movie rating data from the MovieLens [26] dataset to find higher-than-average movies for specific users. It focuses on users with 3 to 100 daily ratings (AG1-F1-J1), outputting ratings from such users, for movies between 1940 and 2005 (F2), if the rating is higher than their daily average (F3).
- **CAR** is an object annotation query [30] that annotates objects for an in-vehicle computer vision system, using the Argoverse Tracking dataset [11], to detect bicycles (M1-F1-AG1-F2) and pedestrians (M2-F3-J1) transiting in front of the vehicle.
- **SYN** is a synthetic query comprised of a Source, Sink, and a synthetic Filter with controllable $i_O$ and $e_{P_O}$ to study the performance overheads discussed in §5.3.

We evaluate SGA, LR on the Odroids and CAR, MOV, SYN on the Server. We evaluate the original, non-instrumented query (**NI**) and the query running *Erebus* (**EB**). In §6.2.2, we also evaluate *Erebus* as a holistic why- and why-not provenance solution (**EB+W**), including why-provenance in its explanations using the technique from [30].

*Predicates and Explanations.* To study the whole range of predicate behaviors, each query is evaluated with four predicates with different explanation ratios $e_{P_O}$: two custom ones denoted as P1, P2, and two synthetic ones, F (always false) and T (always true). Table 1 shows the custom predicates, example explanations from our experiments, and the percentage of overall explanations from each operator over the execution. The predicates are also marked in Figures 7-10, where each line of EB(+W) has four points of increasing $e_{P_O}$, corresponding to predicates F, P1, P2, and T, along with annotations showing the main differences of P1, P2 from NI. To reduce clutter, predicate names are only shown for EB in the first plot of Figure 7. *Erebus* writes the explanations to Apache Kafka [5] (3.1.0) and a secondary query added by *Erebus* (in the same machine) pulls them from Kafka and persists them to disk.

*Performance Metrics.* We evaluate the *throughput*, i.e., the number of tuples a query ingests per unit of time, the *latency*, i.e., the delay in the production of a sink tuple after all its contributing source tuples have arrived at the query, and the *CPU utilization* and *memory consumption* of the SPE. We also measure the query's *interception rate* (*intercepted* in the plots), i.e., $\sum r_O \times i_O$ and *explanation rate* (*explanations* in the plots), i.e., $\sum r_O \times i_O \times e_{P_O}$ (§5.3).

## 6.2 Real-World Query Evaluation

*6.2.1 How Does Erebus Perform Over Time?* Figure 6 compares NI's and EB's performance for the predicate defined in the running example (instrumented operators are marked by a crosshatch pattern in Figure 1). We assume that the analyst (based on domain knowledge about possible issues with the query) is not interested in explanations involving F1 and F2, thus we do not instrument them. For EB, the predicate (SGA-P1 in Table 1) is submitted at $\omega = 180s$, approximately at event time 23:30. No predicate is active before that, and $B = 1$ hour. For $\omega < 180s$ (left of the vertical line), EB's and NI's performance is close: approximately -10% throughput, +7% latency, and +15% CPU. Around $350t/s$ are intercepted. At $\omega = 180s$, the emission of past/present explanations starts, with a transient performance drop (mostly due to past explanations). The CPU jump has a delay, indicating the secondary query converting the (mostly past) explanations to plain text and persisting them. Present explanations are emitted until event time 00:20:01 (when all operator predicates are disabled, see Algorithm 2) with EB's performance being similar to that observed for $\omega < 180s$.

*6.2.2 What Is Erebus' Average Performance During an Execution?* We now study the average query performance of NI, EB, EB+W when the predicate is active since $\omega = 0$ for SGA, LR, MOV, and CAR. The x-axes in Figures 7-10 show the average explanation

**Table 1: Predicates defined in *Erebus* for our evaluation, with example explanations and execution statistics.**

| Predicate | Predicate Definition | Example Explanation | Overall Explanations |
|---|---|---|---|
| SGA-P1 | $(22{:}00 \leq \tau \leq 00{:}20) \wedge (\text{diff} > 4) \wedge (\text{plugUsage} < 30)$ | F3: ($\tau$=22:49:59, usage=0.0, plug=7, household=0, house=2) | F3: 89.1%, J1: 5.3%, K: 5.0%, F4: 0.6% |
| SGA-P2 | $((22{:}10 \leq \tau \leq 02{:}40) \wedge (\text{diff} > 1) \wedge (\text{household} \mod 2 = 0)) \vee ((22{:}05 \leq \tau \leq 23{:}40) \wedge (\text{diff} > 1) \wedge (0 < \text{plug} < 8))$ | F4: ($\tau$=22:05:59, house=34, household=1, plug=1, plugUsage=4, householdUsage=2.1, diff=2.0) | F3: 84.5%, K: 7.4%, J1: 5.2%, F4: 2.8% |
| LR-P1 | $(00{:}10 \leq \tau < 04{:}00) \wedge (\text{pos.lane} \in \{0, 4\}) \wedge (\text{pos.seg} > 30) \wedge (\text{count} > 1)$ | F1: ($\tau$=00:08:30, type=0, vid=9930, speed=40, pos=(xWay=0, lane=0, dir=0, seg=94, p=501418)) | F2: 100% |
| LR-P2 | $(00{:}00 \leq \tau < 01{:}40) \wedge (\forall v \in \text{vids} : v \mod 2 = 0)$ | F2: ($\tau$=01:39:00, vid=8704, lastPos=(0, 1, 0, 19, 100530), nReports=1, uniquePos=1) | F2 > 99.9%, F3 < 0.1%, K < 0.1% |
| MOV-P1 | $(1996\text{-}01\text{-}09 \leq \tau \leq 2002\text{-}01\text{-}07) \wedge (1900 < \text{year} < 1990) \wedge (\text{rating} > 1.5)$ | J1: ($\tau$=2000-08-25, user=260662, rating=4.0, movie=2551, year=1988) | J1: 56.3%, K: 22.0%, F3: 15.5%, F1: 4.2%, F2: 2.0% |
| MOV-P2 | $(1995\text{-}01\text{-}09 \leq \tau \leq 2012\text{-}01\text{-}05) \wedge (\text{nRatings} \times \text{rating} > 33) \wedge (\text{nRatings} > 12) \wedge (\text{movie} < 5000)$ | K: ($\tau$=1996-06-03, user=227882, movie=364, rating=5.0, averageRating=3.5, nRatings=64) | J1: 55.9%, K: 23.4%, F3: 19.8%, F2: 0.6%, F1: 0.3% |
| CAR-P1 | $((00{:}00 \leq \tau < 02{:}40) \wedge (\text{label} = \text{BICYC.}) \wedge (\text{minZ} > 1) \wedge (\text{count} \in [2, 50]) \vee ((00{:}15 \leq \tau < 02{:}20) \wedge (\text{label} = \text{PEDEST.}) \wedge (\text{minX} > 0.25 \times \text{minY}))$ | J1: ($\tau$=00:56:48, target=PEDEST., type=L, key=c4a0.., label=PEDEST., x=378, y=661, z=26) | J1: 45.0%, F1: 44.0%, F3: 6.0%, K: 5.0% |
| CAR-P2 | $((00{:}20 \leq \tau < 02{:}00) \wedge (\text{target} = \text{BICYC.}) \wedge (\text{label} = \text{VEHIC.})) \vee ((00{:}00 \leq \tau < 02{:}00) \wedge (\text{label} = \text{PDEDEST.}))$ | F1: ($\tau$=01:02:32, target=BICYC., type=∅, key=742.., label=VEHIC., x=0.2, y=3.2, z=0.6) | F1: 86.9%, J1: 10.9%, K: 2.1% |

Attributes and predicate conditions are simplified due to space constraints. The attributes of example explanations that match the predicate are underlined.

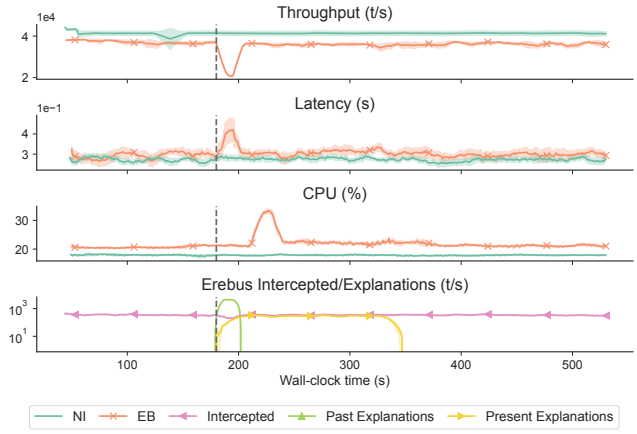


**Figure 6: Performance of SGA over time.**

ratio $e_{P_O}$ of all query operators, the main factor distinguishing the performance of different predicates (§5.3).

For SGA (Figure 7), EB is up to 21% more costly than NI for P1, P2. The metadata overhead $D$ (see §5.3) has a higher performance impact than the explanation ratio $e_{P_O}$, because SGA's source sends a large number of tuples downstream, whose added metadata needs to be serialized, as discussed in §6.3, slowing down the source and lowering the query's performance. EB+W increases the overheads further, due to the additional work of maintaining and transmitting why-provenance metadata and the increased tuple (and thus explanation) size, observable when no explanations are produced (explanation ratio 0). For EB+W, each explanation of *Erebus* contains 130 source tuples as why-provenance (on average), leading to a stronger correlation between $e_{P_O}$[5] and the performance: 41-43% lower throughput and 3.3-3.5x higher latency for P1-P2, compared to NI. The memory increases by up to 2x because of the increased volume of data maintained and the CPU increases due to the higher processing requirements of why-provenance [30].

LR (Figure 8) differs from SGA in that it filters early most (99.9%) of its inputs. Complementing §6.2.1, this implies that 1) the volume of tuples serialized inside the query is very small, 2) the metadata overhead is minimal, as evident by the almost identical performance of NI and EB/EB+W when no explanations are produced, and that 3) the interception rate is now close to the query's throughput (i.e., almost all the input data can be explanations of F2).

For MOV (Figure 9), the metadata overhead $D$ is significant since MOV does minimal filtering and aggregation, serializing a large volume of encapsulated tuples between its operators. For P1 and P2, EB results in up to -34% throughput and +21% latency, with the memory and CPU increasing with the explanation ratio, by up to +14% and +3x, respectively. EB+W, includes on average 30 provenance tuples per explanation, leading to a higher performance impact than EB, with the throughput and latency degrading by up to -37% and +33% respectively (for P1 and P2).

Finally, CAR (Figure 10) behaves similarly to LR, because much of the input data is pruned by F1 and F2 before it is serialized between operators. This, combined with a lower volume of intercepted tuples than MOV, leads EB to have a very small impact on performance for P1, P2: 3% decrease in throughput, 2-3% increase in latency, up to 78% higher CPU, and 66% higher memory. EB+W only slightly differs from EB since the size of its why-provenance is one tuple.

## 6.3 Analysis of *Erebus*' Overheads

Here, we use synthetic loads on the Server to analyze parameters $c_{P_O}$, $i_O$, $e_{P_O}$, $D$, and $B$, which affect *Erebus*' overheads (see §5.3).

*6.3.1 What Affects a Predicate's Evaluation Cost?* In Figure 11, we measure $c_{P_O}$, i.e., the time to evaluate a synthetic predicate $P_O$ based on its complexity. We use a micro-benchmark implemented in JMH [28], with 3 forks, 10 warm-up iterations, and 25 iterations of 10 seconds. The predicate has varying numbers of conditions $n$ and

[5]Because EB+W has a lower throughput than EB, it processes earlier event times during the experiment's execution and thus has higher $e_{P_O}$ than EB for the same predicates.

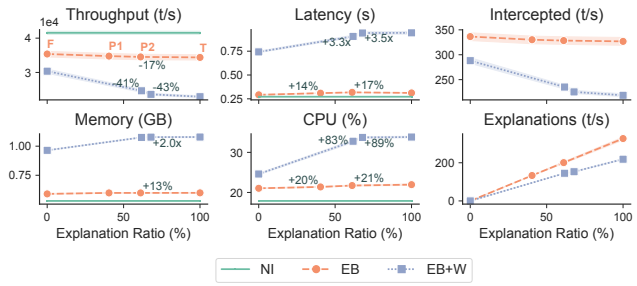


**Figure 7: Performance impact of explanations for SGA.**

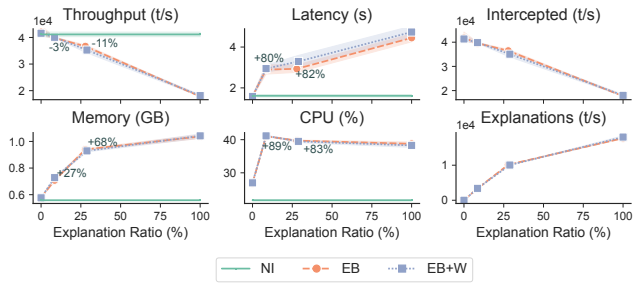


**Figure 8: Performance impact of explanations for LR.**

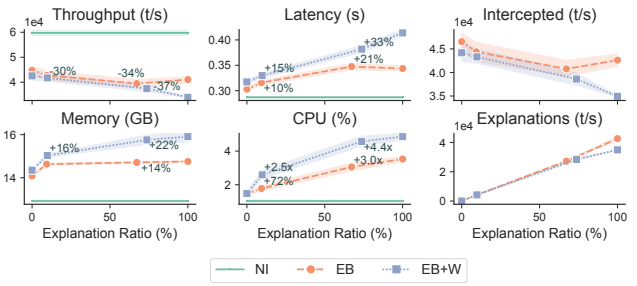


**Figure 9: Performance impact of explanations for MOV.**

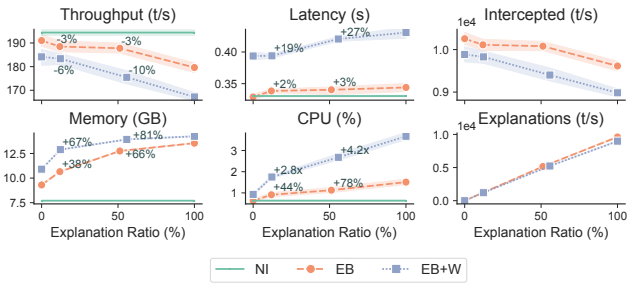


**Figure 10: Performance impact of explanations for CAR.**

(unique) variables $|\bigcup_{i=1}^{n} \mathbb{A}_i|$. The x-axis is the number of unique variables and the y-axis is $c_{P_O}$, in microseconds. Each line is a predicate with a different number of conditions. The 99% confidence interval reported by JMH is shown as a (small) shaded interval. The left part of the plot shows the worst case where all $c_i$ need to be evaluated by the predicate. However, *Erebus*' predicates support *short-circuit evaluation* and can return early if they deem their result final, based on their conditions. This is illustrated in the right part of the plot, where $P_O$ can terminate early after the first condition (best case). As seen in the figure, the predicate evaluation is fast, ranging from 0.3 - 3.4 us. Without early termination (left), $c_{P_O}$ increases with $P_O$'s complexity, with the biggest factor being the number of conditions. With early termination (right), $c_{P_O}$ is almost constant at approximately 0.3 us, regardless of $P_O$'s complexity.

*6.3.2 How do Erebus' Data Costs Affect Query Performance?* In Figure 12, we use a synthetic predicate $P_O$ and the SYN query to measure the impact of the $i_O$, $e_{P_O}$, and $D$ (§5.3). In the green lines (Encapsulated) *Erebus* uses encapsulation for its metadata whereas in the orange ones (Custom) *Erebus* relies on tuples with custom *types* that include its metadata. Each marker represents a different $i_o$ (25%, 75%, 99%) for the Filter of the query. The x-axis is $e_{P_O}$ and the y-axis is the value of the performance metric. The behavior matches §5.3, with higher $i_O$ and $e_{P_O}$ decreasing query performance. Furthermore, encapsulation causes $D$ to have a measurable impact on performance, especially for low $i_O$ and $e_{P_O}$.

Figure 13 evaluates the effect of the buffer size $B$ on SYN's performance. To isolate $B$'s effect, we set $i_o$ = 99% and $e_{P_O}$ = 1%, i.e., most tuples are pruned and stored in the buffer but few are evaluated by $P_O$. As expected, the performance drops for increasing buffer sizes, but the overheads remain low: up to 9% (throughput) and 16% (latency) for 40 million buffered tuples. The memory plot highlights the difference between the total memory used by the JVM (shown in all previous experiments) and the (approximate) size of the buffer: depending on the JVM's configuration, small increases in the buffer size can lead to large jumps in the process memory.

*6.3.3 Best Practices for Erebus.* The evaluation results indicate some best practices for using *Erebus*. First, the different performance of SGA and LR shows it is best to *only instrument necessary operators* to reduce the number of (irrelevant) explanations. Second,

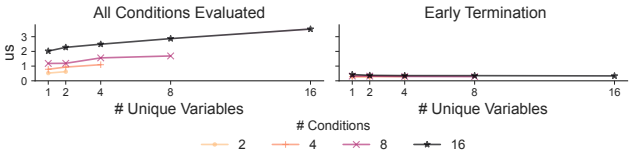


**Figure 11: Time to run a single evaluation of the predicate.**

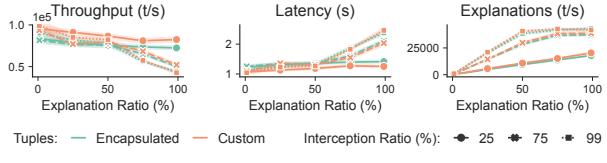


**Figure 12: Effect of *Erebus*' overheads $i_O$, $e_{P_O}$, $D$ on SYN.**

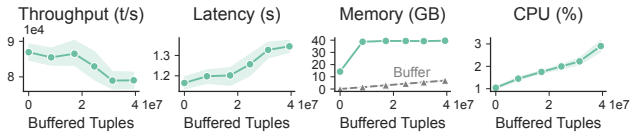


**Figure 13: Effect of the buffer size $B$ on SYN.**

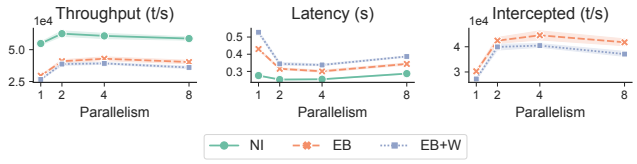


**Figure 14: Scalability of MOV query.**

as illustrated by Figure 11 the user should *place the strictest conditions first*, to take advantage of early termination and minimize the predicate's cost $c_{P_O}$. Third, if the interception and explanation ratios $i_O$, $e_{P_O}$ are low, it might be beneficial to alter the tuple *type*s and avoid encapsulation, to reduce the metadata overhead $D$, as highlighted by Figure 12. The last practice, which we have already used in §6.2 is to *scale-up bottleneck operators* when needed. Modern SPEs can deal with such bottlenecks by increasing operator parallelism, and because *Erebus* keeps all instrumentation logic local in each operator task, instrumented operators can be parallelized without complications. An example of this possibility is illustrated in Figure 14. In this experiment, we run the MOV query with predicate P2 with different levels of parallelism for all operators except the source. As seen in the figure, the query instrumented with *Erebus* scales almost identically to the non-instrumented query.

*Evaluation Summary.* We evaluated *Erebus* on real-world and synthetic workloads and showed that it can produce expectation explanations with low to moderate overheads over the original query, for a variety of custom predicates, both in low-powered devices and a higher-end server. For the custom predicates we defined, *Erebus* led to 3-35% lower throughput and 2-82% higher latency while delivering tens of thousands of compatible tuples per second. Given that explaining missing answers is generally a heavy task with potential slowdowns of more than one order of magnitude [7, 17], *Erebus* offers an acceptable trade-off for the provided functionalities.

## 7 RELATED WORK

*Why-provenance* has been studied extensively in databases [13, 15, 23]. Streaming approaches such as Ariadne [20] and GeneaLog [29] collect backward why-provenance using tuple metadata and instrumented operators. Ananke [30] extends such tools to deliver a live graph of forward provenance. Such techniques, orthogonal to our work, can be combined with *Erebus* to include why-provenance in the explanations of produced and missing answers.

*Provenance of missing answers* has been studied for relational queries, for (reverse) top-k [19, 22], skyline queries [14, 24], and more general systems [33]. Explanations can be identified in the data (instance-based), the query (query-based) or both (hybrid), or expressed as query modifications (modification-based) [23]. We discuss here works closest to *Erebus*, i.e., query-based explanations referring the reader to the survey [23] for more details. In "Why Not?" [12], the authors explain missing answers in workflows, assuming the user cannot alter the query and/or inspect the input data. They search for compatible input tuples not part of the lineage [15] of any result, replaying the query and returning the manipulations closest to the data sources responsible for pruning the last successors of such tuples. NedExplain [9] adopts a similar technique, focusing on SPJUA database queries, proposing an algorithm that returns more detailed and correct answers by more precisely identifying the source data to trace. Similarly to *Erebus*, in NedExplain compatible tuples are allowed to belong to the lineage of some query results. *Ted* and *Ted++* [7, 8] focus on returning the same, and complete missing answer explanations regardless of the query plan, presenting such explanations in the form of *why-not polynomials*. The work in [17] uses reparameterizations of query operators to compute query-based explanations in complex analytical queries, focusing on nested data and operators that modify the schema (e.g., projections). In contrast to the above, *Erebus* focuses on streaming and produces explanations without replaying the query or requiring persistent storage of the inputs.

To the best of our knowledge, *Erebus* is the first work explaining missing answers in streaming. In Complex Event Processing (CEP), Song et al. [31], use temporal networks [16] to find approximate missing-answer explanations based on event time with respect to the query or the data. Their solution studies only time constraints and focuses on a subset of CEP operators, whereas *Erebus* targets general-purpose queries and query-based explanations based on all tuple attributes, not just the timestamps.

## 8 CONCLUSIONS

We formally defined the problem of explaining *missing answers* in streaming and presented *Erebus*, a framework that allows users to validate and debug streaming queries by defining boolean *expectation predicates* on the query outputs. *Erebus* verifies whether expected results are produced and explains the absence of expected results (missing answers). *Erebus*' predicates can be submitted at any point during the query runtime and can refer to any attribute of the results. *Erebus* produces query-based explanations comprising pruned compatible tuples that could have contributed to an expected — but missing — result along with their eliminating operator. We evaluated *Erebus* in real and synthetic workloads, showing it can run alongside queries with small overheads. Future work directions include computing why-not polynomials [7] for query-plan-independent explanations, and instance-based explanations to identify problems in the input streams [23].

## ACKNOWLEDGMENTS

Partially supported by the French Agence Nationale de la Recherche, under the grant for the "EXPIDA" project; the Swedish Research Council (Vetenskapsrådet) grants "EPITOME" project (2021-05424), "HARE" grant nr. 2016-03800, the Chalmers AoA frameworks Energy and Production, (WPs INDEED, and "Scalability and Big Data") and the Swedish Government Agency for Innovation Systems VINNOVA, project "AutoSPADA" grant nr. DNR 2019-05884, in the program FFI: Strategic Vehicle Research and Innovation.

# REFERENCES


[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fer Andez-Moctezuma, Reuven Lax, Sam Mcveety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle Google. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *VLDB* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[2] Apache. 2021. Beam. Retrieved November 5, 2021 from https://beam.apache.org/

[3] Apache. 2021. Heron. Retrieved November 5, 2021 from https://heron.incubator.apache.org/

[4] Apache. 2021. Storm. Retrieved November 5, 2021 from https://storm.apache.org/

[5] Apache. 2022. Kafka. Retrieved March 24, 2022 from https://kafka.apache.org/

[6] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, Toronto, Canada, 480–491. http://dl.acm.org/citation.cfm?id=1316689.1316732

[7] Nicole Bidoit, Melanie Herschel, and Aikaterini Tzompanaki. 2015. Efficient Computation of Polynomial Explanations of Why-Not Questions. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 713–722. https://doi.org/10.1145/2806416.2806426

[8] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Immutably Answering Why-Not Questions for Equivalent Conjunctive Queries. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*. USENIX Association, Cologne.

[9] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *Extending database technology (EDBT)*.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[11] Ming-Fang Chang, John Lambert, Patsorn Sangkloy, Jagjeet Singh, Slawomir Bak, Andrew Hartnett, De Wang, Peter Carr, Simon Lucey, Deva Ramanan, et al. 2019. Argoverse: 3D Tracking and Forecasting With Rich Maps. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 8740–8749.

[12] Adriane Chapman and H. V. Jagadish. 2009. Why Not?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 523–534. https://doi.org/10.1145/1559845.1559901

[13] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2007. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2007), 379–474. https://doi.org/10.1561/1900000006

[14] Sean Chester and Ira Assent. 2015. Explanations for Skyline Query Results. In *EDBT*. 349–360.

[15] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems* 25, 2 (June 2000), 179–227. https://doi.org/10.1145/357775.357777

[16] Rina Dechter, Itay Meiri, and Judea Pearl. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49, 1 (May 1991), 61–95. https://doi.org/10.1016/0004-3702(91)90006-6

[17] Ralf Diestelkämper, Seokki Lee, Melanie Herschel, and Boris Glavic. 2021. To Not Miss the Forest for the Trees - A Holistic Approach for Explaining Missing Answers over Nested Data. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 405–417. https://doi.org/10.1145/3448016.3457249

[18] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. 2019. Edgewise: A Better Stream Processing Engine for the Edge. In *USENIX Annual Technical Conference (ATC) 19*. USENIX, WA, USA, 929–946.

[19] Yunjun Gao, Qing Liu, Gang Chen, Baihua Zheng, and Linlin Zhou. 2015. Answering Why-Not Questions on Reverse Top-k Queries. *Proceedings of the VLDB Endowment: 41st VLDB 2015, August 31 - September 4, Kohala Coast, Hawaii* 8, 7 (Sept. 2015), 738–749. https://doi.org/10.14778/2752939.2752943

[20] Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. 2014. Efficient Stream Provenance via Operator Instrumentation. *ACM Trans. Internet Technol.* 14, 1, Article 7 (Aug. 2014), 26 pages. https://doi.org/10.1145/2633689

[21] HardKernel. 2020. Odroid-XU4. Retrieved November 12, 2020 from http://www.hardkernel.com

[22] Zhian He and Eric Lo. 2014. Answering Why-Not Questions on Top-k Queries. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2014), 1300–1315. https://doi.org/10.1109/TKDE.2012.158

[23] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A Survey on Provenance: What for? What Form? What From? *VLDB Journal* 26, 6 (2017), 881–906. https://doi.org/10.1007/s00778-017-0486-1

[24] Md. Saiful Islam, Rui Zhou, and Chengfei Liu. 2013. On Answering Why-Not Questions in Reverse Skyline Queries. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 973–984. https://doi.org/10.1109/ICDE.2013.6544890

[25] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. Association for Computing Machinery, New York, NY, USA, 266–269. https://doi.org/10.1145/2611286.2611333

[26] MovieLens. 2022. MovieLens. Retrieved March 24, 2022 from https://www.kaggle.com/rounakbanik/the-movies-dataset

[27] Open Source. 2022. Erebus Implementation. Retrieved August 5, 2022 from https://github.com/dmpalyvos/erebus

[28] OpenJDK. 2021. Java Microbenchmark Harness. Retrieved February 24, 2022 from https://github.com/openjdk/jmh

[29] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. 2019. GeneaLog: Fine-Grained Data Streaming Provenance in Cyber-Physical Systems. *Parallel Comput.* 89 (Nov. 2019), 102552. https://doi.org/10.1016/j.parco.2019.102552

[30] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. 2020. Ananke: A Streaming Framework for Live Forward Provenance. *Proceedings of the VLDB Endowment* 14, 3 (2020), 391–403.

[31] Shaoxu Song, Ruihong Huang, Yu Gao, and Jianmin Wang. 2021. Why Not Match: On Explanations of Event Pattern Queries. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1705–1717. https://doi.org/10.1145/3448016.3452818

[32] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *ACM Sigmod Record* 34, 4 (2005), 42–47.

[33] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2014. Diagnosing Missing Events in Distributed Systems with Negative Provenance. *ACM SIGCOMM Computer Communication Review* 44, 4 (Aug. 2014), 383–394. https://doi.org/10.1145/2740070.2626335