Impatience is a Virtue: Revisiting Disorder in High-Performance Log Analytics

Badrish Chandramouli, Jonathan Goldstein, Yinan Li
Microsoft Research
Email: {badrishc, jongold, yinan.li}@microsoft.com

Abstract—**There is a growing interest in processing real-time queries over out-of-order streams in this big data era. This paper presents a comprehensive solution to meet this requirement. Our solution is based on *Impatience sort*, an online sorting technique that is based on an old technique called Patience sort. Impatience sort is tailored for incrementally sorting streaming datasets that present themselves as *almost sorted*, usually due to network delays and machine failures. With several optimizations, our solution can adapt to both input streams and query logic. Further, we develop a new *Impatience framework* that leverages Impatience sort to reduce the latency and memory usage of query execution, and supports a range of user latency requirements, without compromising on query completeness and throughput, while leveraging existing efficient in-order streaming engines and operators. We evaluate our proposed solution in Trill, a high-performance streaming engine, and demonstrate that our techniques significantly improve sorting performance and reduce memory usage – in some cases, by over an order of magnitude.**

I. Introduction

We are witnessing a growing demand for real-time analysis of logs that are produced in today's distributed data processing systems and applications. *Stream processing engines* (*SPEs*) are used to analyze such streams of logs and produce incremental results in real time. An inherent characteristic of today's logs is that they are *out-of-order*, i.e., events may not be collected for processing in strict timestamp order. As a result, we are witnessing an increasing interest in processing real-time queries over such out-of-order streams [1]–[4].

For example, suppose a user wants to develop an online *dashboard* that displays accurate aggregate statistics as close to real time as possible. Events are collected from many distributed servers, and are transmitted to an SPE that computes these statistics. However, due to network delays, intermittent machine failures, and race conditions, events may be delivered to the SPE in a non-deterministic order. A fundamental challenge in SPEs is to handle these out-of-order streams without sacrificing performance significantly, in terms of latency, throughput, accuracy, and memory usage.

The most widely-used solution in today's SPEs is a sort-based method called "buffer-and-sort". With this mechanism, an SPE tolerates a specified amount of disorder, buffering an incoming out-of-order stream for a specified amount of time. Then, the engine sorts the buffered data, and feeds the sorted data to subsequent operators in the execution workflow. As a result, all operators besides the sorting operator in the SPE are free from handling out-of-order streams, and always process in-order streams. The buffering time, called *reorder latency*, is a key parameter as it introduces a lower bound on latency. Despite its simplicity, this mechanism raises several concerns in latency, completeness, memory usage, and throughput.

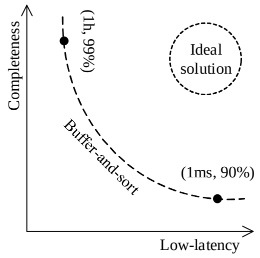


Fig. 1: Buffer-and-sort: Latency vs. completeness tradeoff.

A. Challenges

Latency and Completeness. A fundamental pitfall of the buffer-and-sort mechanism is that users are forced to make a tradeoff between completeness and latency by setting reorder latency. Note that events that arrive after the specified reorder latency have to be either discarded or adjusted (on timestamps), thus reducing the accuracy of query results on the input data. For example, as illustrated in Figure 1, if the dashboard application specifies a lower latency, e.g., 1 millisecond, the engine executes queries in a timely manner, but risks losing events which arrive more than 1 millisecond late. In contrast, if the dashboard application specifies a higher latency, e.g., 1 hour, the engine is more tolerant of late arriving data, but receives aggregate statistics with an unacceptable latency. Many find that in practice, such a tradeoff between latency and completeness is not acceptable in many scenarios.

In order to address this challenge, previous work has proposed a different execution strategy based on *speculation* [4], where operators produce output before receiving all the data, and on receiving late events, are responsible for retracting incorrect outputs and adding the correct revised outputs. Thus, an SPE with speculation is able to deliver early inaccurate results without losing late arriving data. However, introducing speculation into each operator makes operator logic highly complex, requiring considerable effort to implement each operator correctly. For example, a simple disordered aggregate operator in StreamInsight [5] consisted of more than 3000 lines of high-level language code. Even worse, such operator complexity results in significant inefficiencies (often by orders-of-magnitude [6]) compared to their in-order counterparts. More recently, Dataflow [1], a model that supports out-of-order processing, was developed at Google. Here, users can add late events to prior windows and output based on triggers. However, this proposal focuses on the lower-level APIs for users to implement their own disorder handling and operator logic, and does not describe or advocate a particular solution, where the issues addressed in this paper would surface.

Memory Usage. The streaming engine has to buffer all the data in input streams during the specified reorder period, leading to high memory usage. To address this problem, previous work [2] proposed to offer all operators the capability to process out-of-order events. Instead of buffering original events, these out-of-order operators only keep track of states associated with a given query (e.g., aggregate results for the dashboard application) and thus reduce memory consumption in many scenarios. However, the latency issue remains and, similar to speculation, this method suffers from design effort and code complexity. Consequently, this solution is rarely implemented in commercial SPEs.

Throughput. Finally, existing sorting algorithms fall short in incrementally sorting out-of-order events in SPEs. To support incremental sorting, traditional SPEs often use a priority queue to sort out-of-order events [5]. However, the new generation of SPEs like Trill [6] adopt techniques such as columnar batching to increase throughput by orders of magnitude compared to traditional SPEs. As a result, an inefficient sorting operator quickly becomes the bottleneck of the engine and dominates query execution time.

B. *Our Solution*

In this paper, we propose an end-to-end sort-based solution to overcome all the pitfalls described above. As a sort-based solution, it also relies on a sorting operator to handle out-of-order events, keeping all other operators free from handling out-of-order streams directly. In addition, unlike existing sort-based methods, our solution brings the benefits of out-of-order operators [2] and speculation [4] into the sort-based method, allowing us to use high-performance in-order operators unmodified. More specifically, the proposed solution focuses on three key aspects, shown as follows.

How to sort streams efficiently? We first adapt a long-forgotten sorting technique called Patience sort [7]–[9] to an online sorting algorithm. This sorting algorithm, called *Impatience sort*, is tailored to fit the demand for incrementally sorting out-of-order streams in SPEs. Impatience sort takes advantage of existing order in input streams, and is especially efficient for the common out-of-order patterns appearing in logs produced by distributed systems. In addition, unlike its offline counterpart (i.e., Patience sort) that has to be performed after receiving all data, Impatience sort incrementally sorts events and outputs sorted partial results based on progress indicators (i.e., punctuations) in SPEs. This sorting algorithm is a key building block in our solution.

How to produce good streaming query plans with sorting operators? We next present a sort-as-needed execution strategy that sorts data to the extent that is necessary for a given query, and avoids an unnecessarily complete sorting of the input data stream. More specifically, as order-insensitive operators can be performed earlier than the sorting operator without violating their semantics, we defer the sorting operator in an execution workflow until the input data has been reduced by other operators. Interestingly, we observe that the early execution of order-insensitive operators usually result in better performance of the subsequent sorting operator, by reducing the disorder, the number of events, or the size of events in the input stream. To leverage these opportunities, we developed programming APIs to allow users to exploit this execution strategy.

How to cope more flexibly with the latency-completeness tradeoff? We propose a programming framework called the *Impatience framework*, that is inspired by the basic idea behind Impatience sort. Impatience framework enables users to specify a set of different reorder latencies, rather than a single reorder latency value. Based on the specified reorder latencies, we partition an input out-of-order stream into multiple in-order streams. Taking the online dashboard application as an example, the application can subscribe to multiple output streams associated with specified reorder latencies, e.g., $\{1\text{ms}, 1\text{sec}, 1\text{min}\}$, and thus illustrates early but inaccurate aggregate statistics as close to real time as possible, but also updates the refined and more accurate results 1 second and 1 minute later. More interestingly, users can optionally provide query logic functions that are embedded into this framework. For instance, the dashboard application can perform partial aggregations on each partition, and then quickly combine the partial aggregations on early events and late events. Thus, instead of buffering original events as in the buffer-and-sort method, the framework enables the engine to reduce early arriving events to aggregate results, resulting in low memory usage. Note that the latency-completeness tradeoff is a fundamental one: what the Impatience framework provides is (1) a way to expose this tradeoff as a user specification; and (2) a design to meet this specification at low memory usage and high throughput. *To summarize, the use of our Impatience framework reduces both latency and memory usage of query execution, while meeting the desired completeness goals at high throughput.*

We prototyped our solution in Trill [6], a high-performance query processor for streaming analytics. We conducted an evaluation of the proposed techniques using both synthetic and real workloads. Our evaluation shows that the combination of Impatience sort and sorted-as-needed execution significantly improves the sorting performance, and in some cases produces an order of magnitude in performance improvement. The use of our Impatience framework enables low-latency query results while preserving late events. In addition, it also reduces the memory usage by up to $31.5\times$, while sustaining comparable throughput to the method with a single reorder latency.

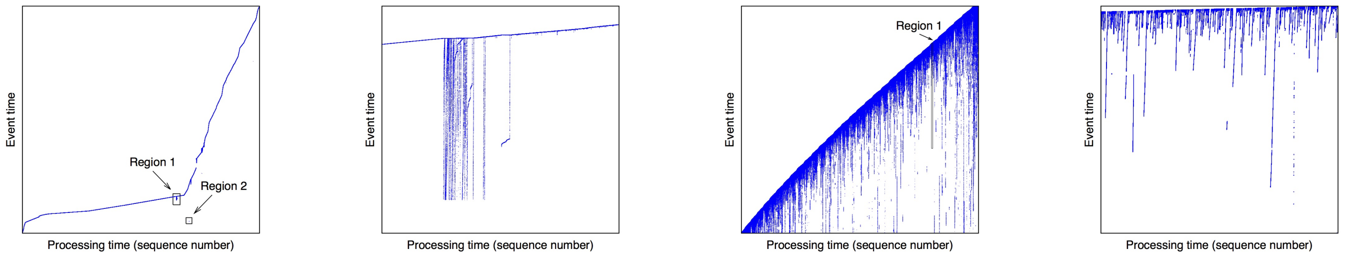
To summarize, we make the following contributions in this paper: 1) we demonstrate that Patience sort is a promising starting point for incrementally sorting out-of-order streams, and adapt it for SPEs; 2) we study the impact of query plans on the performance of the sorting operator in a streaming setting, and propose an approach to leverage this observation; 3) we introduce the Impatience framework, based on the realization that Impatience queues can be exposed to the rest of the query processing pipeline, to cope more flexibly with the latency-completeness tradeoff, without scarifying memory usage or throughput; and 4) by putting it all together, we introduce a complete solution to enable out-of-order processing in an SPE, without modifying each operator.

II. Workload analysis

Disordered streams are increasingly common in today's real-time stream engines. In order to better understand the nature of disorder in such out-of-order streams, we conduct empirical analysis on real-world workloads in this section.

There are two different notions of time for each event in a stream, shown as follows:

- **Event time**: the logical time at which the event occurs (also referred as *application time* in literature).
- **Processing time**: the time at which the event is ingested into a streaming engine. According to this definition, an incoming stream is always ordered with respect to processing time.



(a) CloudLog (b) CloudLog (Region 1) (c) AndroidLog (d) AndroidLog (Region 1)

Fig. 2: The relationship between event time and processing time in the CloudLog and AndroidLog datasets.

In an ideal world, an event can be processed in a streaming engine as quickly as the event occurs. However, from a practical standpoint, there is always a delay between event time and processing time, resulting in an event-time out-of-order stream for several reasons. First, network latency of sources is different and is dynamically changing over time. Second, data can be significantly delayed due to failure or disconnection of sources. Lastly, there might be no guarantee of ordering over the transmission channel.

Though out-of-order events are the norm rather than the exception in stream processing, it is also clear that events tend to be nearly sorted in practice. The intuitive notion of "nearly sorted" generally means that a stream is very close to being sorted, but it loosely refers to several related but distinct types of patterns. For example, a stream consists of two concatenated sorted streams and a stream produced by interleaving two sorted streams are both, in some sense, nearly sorted. In order to distinguish disorder in many different ways and quantify the sortedness in a stream, we introduce four common measures of disorder [10] as follows:

- **Inversions**: the number of inversions in the stream. An inversion is a pair of positions where the events on these positions are out-of-order (in event time). The number of inversions is likely the best-known measure of sortedness.
- **Distance**: the maximum distance between the positions associated with an inversion.
- **Runs**: the number of increasing runs (ordered by event time) in the stream.
- **Interleaved**: the minimum number of sorted runs that can be interleaved to produce the stream.

In the remaining of this section, we present our analysis on disorder in two example datasets. We believe that these two datasets represent two common types of log streams.

- **CloudLog dataset**: this dataset is a log of a large-scale cloud application deployed at Microsoft. The events are generated at multiple distributed application servers, and are then sent to a central server immediately, which aggregates these events to produce a log stream.
- **AndroidLog dataset**: this dataset is collected in the device analyzer project [11] at University of Cambridge. An application on each user's smartphone is used to record phone activities as events, and periodically upload these events to a server when the phone is attached to a charger. Due to this uploading strategy, events in this dataset are often delivered to the server hours or even days later.

Figure 2 plots the relationship between event time and processing time (here we simply use the sequence number as the processing time), for both datasets. We also list statistics on the four disorder metrics in Table I.

TABLE I: Statistics on disorder in the two datasets.

Measure of disorder	CloudLog dataset (20M events)	AndroidLog dataset (Region1:20M events)
Inversions	53,541,688,892	73,004,914,227,284
Distance	13,635,714	19,990,056
Runs	7,382,495	5,560
Interleaved	387	227

For the CloudLog dataset, we see a nearly increasing curve with minor spikes in Figure 2(a). This curve has a steeper slope in the right side of the figure, as there is a sharp reduce in event frequency at a certain point of time (note that in x-axis we show the sequence number, not logical time). Though the curve looks relatively smooth at this plotting scale, there are actually numerous minor spikes along the curve, each of which corresponds a group of late events. This is confirmed by statistical results shown in Table I: there are as much as 7.3 million natural runs in 20 million events, meaning that each run consists of only 2.7 events on average. Hence it is clear that there are a significant number of out-of-order events in this dataset, though a majority of them are not very far from their correct positions. In some sense, this dataset is well-ordered at a coarse granularity, but is chaotic at a fine granularity.

In addition to minor spikes, some unusual and more pronounced pikes occurs when a system failure happens. In Figure 2(b), we zoom in the Region 1 in Figure 2(a) to show the patterns of late-arriving events caused by a system failure. In the worst case, as shown in Table I, the most delayed events in this dataset (in Region 2 of Figure 2(a)) need to be moved over 13.6 million events to reach their sorted positions.

For the AndroidLog dataset, there is a nearly straight line from the lower left corner to the upper right corner, with numerous spikes of various lengths, as shown in Figure 2(c). Each spike represents a sequence of events from a same phone that are uploaded to the server in a batch. Hence the length of a spike is proportional to the time period between two consecutive uploads from the same phone. Though the number of inversions in this dataset is several orders of magnitude more than that in the CloudLog dataset, it is still very close to being sorted, because it consists of only 5,560 natural runs. In contrast to the CloudLog dataset, the AndroidLog dataset is well-ordered at a fine granularity, but is chaotic at a coarse granularity.

III. Impatience sort

In this section, we focus on the first of the three key questions in this paper: *How to sort out-of-order streams efficiently?* To answer this question, we present Impatience sort, a sorting algorithm based on Patience sort, that is tailored to meet the increasing demand on *incrementally* sorting out-of-order events in stream processing engines.

A. Problem Definition and Requirements

Problem Definition. A sorting operator in a Stream Processing Engine (SPE) takes as input a continuous data stream that consists of data events and *punctuations*. A punctuation is a special control message embedded in the stream with a timestamp T and indicates that there are no more events whose timestamp is less than or equal to T. Once a sorting operator receives a punctuation with a timestamp T, it must flush out all buffered events whose timestamps are less than or equal to T in an ascending timestamp order.

$$2 \quad 6 \quad 5 \quad 1 \quad 2^* \quad 4 \quad 3 \quad 7 \quad 4^* \quad 8 \quad \infty^*$$

As an example, we show a data stream above that consists of eight events and three punctuations (marked with asterisks). In this example, we use timestamps to represent events/punctuations and ignore other fields. When a sorting operator receives the first punctuation 2, it outputs a run [1, 2]. For the second punctuation 4, it generates [3, 4]. And finally, it pushes out [5, 6, 7, 8] when the last punctuation arrives.

In practice, SPEs insert punctuations based on user-specified settings when events are ingested into the engine. The timestamp in a punctuation is set by subtracting the reorder latency from the high-watermark timestamp when the punctuation is produced and emitted. This technique also works in an *Internet-of-Things* setting such as AndroidLog, where devices may get periodically disconnected, as it provides an upper bound on overall disorder relative to the device with the most recent update in the system. The reorder latency Thus, a higher reorder latency often means that the incremental sorting operator sorts only a small portion of the buffered data, on receiving a punctuation.

Performance Requirements. We identify two key requirements for an incremental sorting algorithm to efficiently sort out-of-order events in SPEs:

- **Adaptive to sortedness**. The sorting operator needs to take the advantage of existing order in the input stream, especially for common out-of-order patterns in logs generated in distributed data processing systems (see Section II).
- **Efficient incremental sorting**. The sorting operator should be able to incrementally sort and output a subset of the input based on progress indicators, i.e., punctuations, in an efficient way. The operator needs to tolerate a wide variety of punctuation-related settings, such as high punctuation frequency or high order latency.

Existing sorting algorithms fall short of at least one of the two performance requirements. For instance, Heapsort, a sorting method used in today's stream processing engines [5], naturally supports incremental sorting, but is not adaptive to the sortedness of the input data. In contrast, adaptive sorting algorithms [10] leverage the sortedness of the input data, but are unable to sort data in an incremental way. Quicksort, the most well-known sorting algorithm, meets neither of the two requirements.

B. Background on Patience sort

Patience sort [7], [8] is a comparison-based sorting algorithm inspired by the British card game of Patience (named Solitaire in America). The algorithm sorts an array of elements in two phases shown as follows:

1) **Partition phase**: we partition the array into a sequence of sorted runs, following the steps as follows. Initially, there is no sorted run. Then, we scan over all elements in the input array and append each element into the first sorted run whose last element is less than or equal to the current element, or if such a run does not exist, create a new run with the current element.
2) **Merge phase**: we merge all sorted runs into a single sorted run, in a similar way as Mergesort.

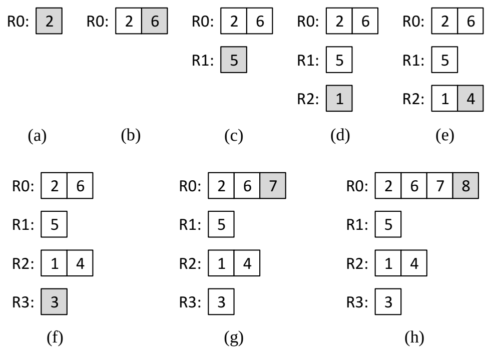


Fig. 3: The steps of Patience sort on a sample array. Shaded elements are the newly added elements in each step.

Figure 3 shows the operation of Patience sort on an 8-element array [2, 6, 5, 1, 4, 3, 7, 8], in the following steps. (a) Initially, there is no run. Thus, the first element 2 creates Run 0. (b) The element 6 is then appended to Run 0 as it is greater than the last element of Run 0. (c)-(d) The elements 5 and 1 are less than the last elements of all existing runs, and are added to new runs. (e) The element 4 is added to the first run whose last element is less than 4 (Run 2). (f) The element 3 creates Run 3. (g)-(h) Run 0 grows to include the elements 7 and 8.

According to the description of the algorithm, the last elements of all runs, called *tails*, are always in strictly descending order. As a result, binary search can be used to find the appropriate run to place a new element. As a result, the performance of the partition phase is largely determined by the number of runs.

Patience sort traditionally uses a heap to merge all produced sorted runs [8]. However, recent work [9] has shown that using binary merges instead of a heap is more efficient on modern computer architecture.

C. Why Patience Sort?

We observe that Patience sort is an ideal starting point for the incremental sorting operator. In this section, we explain two reasons for this observation, corresponding to the two performance requirements discussed in Section III-A.

First, Patience sort is naturally adaptive to the sortedness of the input stream, especially for many common out-of-order patterns appearing in logs. The number of sorted runs created in Patience sort, denoted as k, plays a critical role in how the sorting algorithm leverages the existing order. Intuitively, Patience sort creates potentially much fewer sorted runs on a nearly sorted input. Formally, we give three Propositions to show the upper bound on k using the measures of disorder described in Section II. In the interest of space, the proofs for these propositions are omitted in this paper.

Proposition 3.1: If an input array can be generated by interleaving d sorted runs, the number of sorted runs generated in Patience sort is less than or equal to d.

Proposition 3.2: The number of sorted runs generated in Patience sort is less than or equal to the number of distinct values of timestamps in the input array.

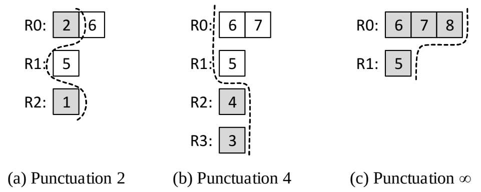


Fig. 4: The steps of Impatience sort on the example stream. Dashed lines mark the punctuation timestamps. Shaded elements are to be merged in each step.

Proposition 3.3: The number of sorted runs generated in Patience sort is less than or equal to the number of natural runs in the input array.

Proposition 3.1 shows a promising property for sorting logs generated in a distributed environment. In these scenarios, events are collected from a set of distributed sources, each of which usually produces ordered events with respect to event time. Delays are introduced when these events are transmitted and combined due to network delays, intermittent machine failures, and race conditions. Thus, the number of interleaved runs in a log is determined by a combination of several hardware factors such as the number of distributed sources, network routers, and failed nodes. As a result, a log usually has a limited number of interleaved runs. For instance, as shown in Table I, the CloudLog and AndroidLog datasets have 387 and 227 interleaved runs, respectively. In general, Patience sort often creates a limited number of runs when sorting these logs.

Proposition 3.2 provides a tight bound on k when there are a limited number of distinct timestamps. This proposition is especially important when a sorting operator is used for evaluating a time-window query on a disordered stream (see Section IV-A2 for more details). Proposition 3.3 is a corollary of Proposition 3.1, and explicitly shows the relationship between k and the number of natural runs in the input stream.

The second reason that we chose Patience sort is related to its merge-based nature, which implies a potential solution for incremental sorting without fundamentally changing the algorithm. A key challenge to support incremental sorting is to find the subset of the buffered data that needs to be sorted on receiving a punctuation, in a way that avoids accesses on all the buffered data. Interestingly, this problem can be naturally addressed using Patience sort, because all the buffered data in Patience sort is stored in the form of a set of sorted runs.

D. Basic Algorithm

Impatience sort is a variant of Patience sort that supports incremental sorting (this explains why the variant is called "Impatience" sort). More specifically, on receiving the i-th punctuation with timestamp T_i, the algorithm sorts all events whose timestamps are between T_{i-1} and T_i.

Impatience sort also has two phases. The partition phase of Impatience sort remains the same as that of Patience sort. In the merge phase, when we receive the i-th punctuation with timestamp T_i, the incremental sorting works as follows. For each sorted run, we cut off a subsequence from the head of the sorted run that contains all events whose timestamps are less than or equal to T_i. The removed subsequence forms a new sorted run called *head run*. Since each run has been in sorted order, this step can be done without accessing all events in the run. Next, we perform a multiway merge on all head runs and emit the merged results for the i-th punctuation. Finally, if a sorted run becomes empty after cutting off its head run, the sorted run is removed from the set of k sorted runs.

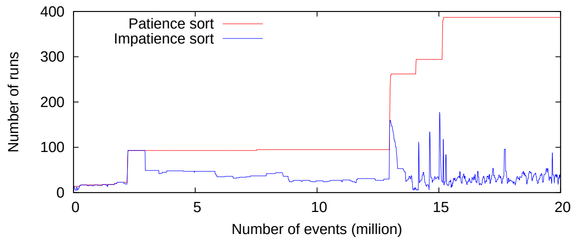


Fig. 5: The number of sorted runs in Patience and Impatience sort when sorting the CloudLog dataset.

Figure 4 demonstrates the steps of Impatience sort on the example stream shown in Section III-A. Each step corresponds to sort a subset of events (marked shaded) for a punctuation. For the first punctuation with the timestamp 2, we remove events 1 and 2 from the sorted runs and merge the two events into a single sorted run as the output for the first punctuation. Run 2 is empty after this step, and is thus removed. Similarly, on receiving the second punctuation, we merge and output the events 3 and 4. Finally, for the last punctuation with an infinite timestamp, a merge is performed on all the buffered events.

Compared to the operation of Patience sort on the same stream (without punctuations) shown in Figure 3, Impatience sort maintains fewer sorted runs in certain steps. For instance, during the second and third punctuations, the number of runs in Impatience sort remains at 2 while Patience sort manages 4 runs. In some sense, Impatience sort is able to gradually clean up sorted runs created by severely delayed events. As the cost of binary search in the partition phase is determined by the number of sorted runs, fewer runs usually result in higher performance.

To demonstrate this effect in more details, we plot the number of sorted runs in Patience and Impatience sort when sorting the CloudLog dataset, in Figure 5. In this experiment, Impatience sort performs incremental sorting on every 10,000 events, while Patience sort sorts the input after receiving all events. Patience sort shows a monotonic increasing curve, as the number of runs never decreases during the partition phase. Events with burst delays, likely caused by server failures (shown in Figure 2(b)), sharply increase the number of runs, and even worse have an unredeemable effect on sorting subsequent events. In contrast, Impatience sort periodically cleans out the sorted runs created by events with burst delays and thus gradually reduces the number of sorted runs. In a short period after receiving these severely late events, Impatience sort brings its internal structure back to a "healthy" status.

E. Optimizations

In addition to the basic algorithm of Impatience sort, several key optimizations play a vital role in achieving better sorting performance in a streaming environment.

1) Huffman Merge: Previous work on Patience sort has shown that the run size distribution on nearly sorted data is highly skewed [9]. When merging the k sorted runs two-at-a-time, it is often more efficient to merge small runs before merging large ones. This observation remains true for Impatience sort, where we merge head runs instead of original sorted runs.

Given a set of head runs of different lengths, there exists an optimal way to merge all head runs into a single sorted run, in terms of the number of accessed events during this process. This problem can be reduced to the Huffman coding problem [12] in the following manner. We use a binary tree to represent the merge process: each leaf node corresponds to a head run; each internal node represents a binary merge between its left and right child nodes, that are either a head run or a sorted run produced by other binary merges. For a head run at the i-th level of the binary tree, all events in this head run need to be accessed i times to produce the final sorted run. As a result, by defining the size of each head run as the weight assigned to the corresponding leaf node, the problem is essentially identical to the Huffman coding problem: to construct an optimal binary tree to minimize the average weighted depth of all leaf nodes.

Following the Huffman coding algorithm [12], the Huffman merge algorithm proceeds as follows. Initially, we put all head runs into a set. Then, we continue to merge the two smallest runs in the set and add the merged run back into the set, until there is only one run left in the set.

2) Speculative Run Selection: In the partition phase, Impatience sort performs binary search on the tails array, in order to find an appropriate run to place a new event. Despite its simplicity, binary search is not the most efficient search method in many situations. To address this concern, we introduce speculative run selection that leverages common characteristics of out-of-order streams to avoid relatively expensive binary search.

Before performing binary search, we first examine if a new event could be directly inserted to the sorted run to which the last event is added. If a new event falls in between the tail of the last-added run and the tail of its immediate previous run, it can be directly inserted into the last-added run, without violating the key property that the tails of all sorted runs must be in decreasing order. It is clear to see that this optimization is particularly beneficial for streams that contain consecutive sorted subsequences (e.g., the AndroidLog dataset).

IV. Query plans

In this section, we shift our focus to the question of *how to produce good streaming query plans with sorting operators*, using the Impatience sort described in Section III as a sorting operator in a SPE. We believe that an efficient solution for handling out-of-order events must go beyond the scope of a sorting operator. Thus, we present a "sort-as-needed" execution strategy to leverage this opportunity in this section.

A. Sort-as-needed Execution

Many SPEs, e.g., [6], [13], perform sorting on incoming out-of-order events as soon as they are ingested into the engine, and stream them to the internal of the engine in event time order. With this mechanism, we have to sort all events in the input stream, regardless of the query logic. In despite of its simplicity of implementation, this method falls short of tailoring the execution workflow for a given query.

To overcome these limitations, we present an execution strategy that sorts data "only as needed" for a given query. More specifically, we defer the sorting operator in an execution workflow until the input data has been tailored by other operators. Streaming operators can be classified into two categories: *order-sensitive operators* (e.g., join and aggregation operators), and *order-insensitive operators* (e.g., selection and projection operators). Order-insensitive operators can be performed earlier than the sorting operator, without violating their semantics. Interestingly, we observe that the early execution of order-insensitive operators usually results in better performance of the deferred sorting operator, by reducing the number of events, the size of events, or the disorder in the input stream. We discuss three such operators in more details below.

1) Selection and projection operators: Both selection and projection operators are order-insensitive operators. They can process events in an arbitrary order without violating their semantics or sacrificing performance. More importantly, a sorting operator often benefits from the early execution of the two types of operators: a selection operator can reduce the number of events that are needed to be sorted; a projection operator can decrease the size of each event by removing unnecessary fields for a given query. Consequently, it is always beneficial to execute selection and projection operators earlier.

2) Window operator: Time-based window operators are also stateless, order-insensitive operators. In Trill, in addition to event time, each event contains another timestamp, called *other time*, to indicate the extent of the time interval. A window operator is performed by adjusting both event time and other time, and thus controlling the time interval over which an event contributes to query computation. Subsequent order-sensitive operators are then logically executed against events whose (adjusted) time intervals are overlapped with a given timestamp.

As an example, consider a hopping window (i.e., sliding window) query that computes over an one-minute window for every second. To execute this query, the hopping window operator sets event time to `eventTime - eventTime % 1000`, and other time to `eventTime - eventTime % 1000 + 60000` (suppose that event time is in milliseconds), and then streams all events with the adjusted timestamps to downstream operators.

As illustrated by this example, the window operator sets event time of all events in a time interval of 1000 to an identical value, eliminating disorder within each time interval. In general, a time-based window operator reduces disorder in event time, resulting in a stream that is closer to being completely sorted. As a result, it is always beneficial to perform early window operators before a sorting operator. Note that unlike existing systems, where the window is a property of existing stateful operators, Trill uses a separate Window operator that transforms timestamps in the data; this allows us to uniformly apply the push-down optimization to windowing as well. Taking Impatience sort as an example sorting operator, it usually creates fewer runs on the stream produced by the window operator (see Proposition 3.2), resulting in better performance of the partition phase. Furthermore, the window operator likely increases the lengths of natural runs in the stream. Thus, the speculative run selection optimization of Impatience sort (see Section III-E2) is able to leverage this fact to further speedup the partition phase.

B. Programming Interfaces

Trill provides a functional programming API [14], which is a variant of LINQ [15] with temporal operators such as window operators. Each stream is represented as an instance of the immutable Streamable abstraction. Users can execute a query over Streamable by chaining a sequence of operators, each of which produces a new Streamable instance that is then fed into the next operator. As an example, we show the code for a query that computes the number of events in every second, with a 5% sample of users. Note that sorting is implicitly executed when (out-of-order) data is ingested into Trill.

```
Streamable<> s = File.ToStreamable(...)
  .Where(e => e.UserId % 100 < 5).TumblingWindow(1s);
  .Count();
```

To support sort-as-needed execution in Trill, we need to enable users to manipulate out-of-order streams with order-insensitive operators, and prohibit order-sensitive operations over them. To achieve this goal, we expose a *Disordered-Streamable* abstraction to users, which represents a disordered stream. A DisorderedStreamable instance supports only order-insensitive operators, such as selection, projection, and window operators. The DisorderedStreamable abstraction provides a *ToStreamable()* method to explicitly convert the Disordered-Streamable to a Streamable by performing a sorting operator over the disordered stream.

For the example query, we show the sample code of the sort-as-needed execution below. Unlike the example code shown above, data is ingested into Trill as an instance of DisorderedStreamable, followed by a selection operator and a tumbling window operator over this DisorderedStreamable. A sorting operator is then performed on the DisorderedStreamable to produce a Streamable, that is required to execute an aggregation operator such as Count().

```
DisorderedStreamable<> ds = File.ToDisorderedStreamable()
  .Where(e => e.UserId % 100 < 5).TumblingWindow(1s);
Streamable<> s = ds.ToStreamable().Count();
```

V. Impatience Framework

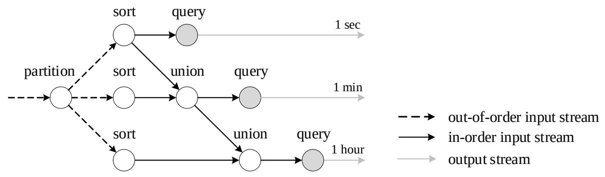
The combination of Impatience sort and sort-as-needed execution provides a high-throughput "buffer-and-sort" solution to process out-of-order streams. However, a fundamental challenge of the buffer-and-sort mechanism is that users are forced to make a tradeoff between latency and completeness. This tradeoff is largely controlled by reorder latency. Even though it is practically feasible to make a reasonable tradeoff in some applications, many find that such a tradeoff is not acceptable in many scenarios, where an user wants to capture the best of both worlds.

To overcome this dilemma, we propose a programming framework called the Impatience framework, that is inspired by the basic idea behind Impatience sort. This framework can be integrated into existing buffer-and-sort SPEs to provide explicit control over the latency-completeness tradeoff, without changing the implementation of operators in these SPEs.

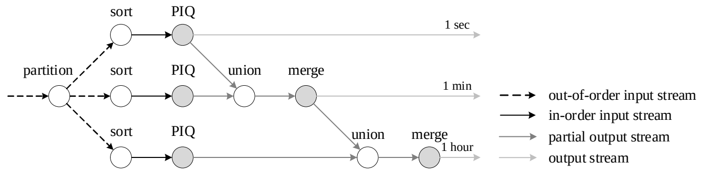
A. Basic Framework

Impatience framework allows users to specify multiple reorder latencies instead of a single reorder latency value. In practice, a latency specification often includes a sequence of logarithmically increasing latency values, e.g., $\{1$ sec, 1 min, 1 hour$\}$. Given a reorder latency setting, Impatience framework incrementally provides multiple output streams, corresponding to the specified reorder latencies respectively, for a given query.

The execution of Impatience framework is inspired by the basic idea behind Impatience sort (c.f. Section III). Recall that Impatience sort partitions an out-of-order stream into a sequence of in-order streams and merges these in-order streams eventually. Following this idea, Impatience framework partitions an input stream into multiple ordered streams based on the time delays of events, and then merges them to produce multiple output streams that are eventually consumed by downstream operators. Each output stream includes *all* events that



(a) Basic framework



(b) Advanced framework with user-defined operators

Fig. 6: Workflow of Impatience framework with three reorder latencies $\{1$ sec, 1 min, 1 hour$\}$.

arrive within the corresponding reorder latency, and therefore contains previous output streams (we assume that the specified reorder latencies are in increasing order).

Figure 6(a) demonstrates the workflow of the basic Impatience framework on a set of example reorder latencies $\{1$ sec, 1 min, 1 hour$\}$. The partition operator partitions the input out-of-order events into three separate streams based on the amount of time delays. All events that arrive less than one second late are buffered by up to one second, and then are incrementally sorted to produce the first partitioned in-order stream. This stream is consumed by the subsequent streaming operators in the execution workflow. Events that arrives between one second and one minute late are sorted every one minute, and are streamed through the second path. These events are then combined with the ordered events coming from the first path using an union operator, which merges and synchronizes two sorted streams into one sorted stream (and thus is a blocking operator). As a result, the second output stream contains all events that arrive less than one minute late, in sorted order. Consumers of the second output stream receive these events with a latency of one minute. Similarly, the third output stream contains additional late arriving events and has a latency of one hour. As can be observed in the example, downstream operators in the execution workflow consume early events as quickly as needed, while receiving more late arriving events with a higher latency.

B. Advanced Framework

Although the basic framework can offer low-latency early results, it falls short in other two aspects: throughput and memory usage. First, a majority of events are duplicated in multiple output streams, and are therefore evaluated redundantly by subsequent operators. This redundancy wastes computation resources and limits system throughput. Second, the (blocking) union operator merges events from two streams with a variety of latency, and therefore may buffer events from the upper stream to synchronize with the lower stream, leading to high memory usage. For instance, in Figure 6(a), the union operator on the third stream needs to buffer all events produced from the second stream, for up to one hour.

To overcome these limitations, we propose an advanced framework that embeds user-defined query logic into the execution workflow, in a way similar to the MapReduce model [16]. With the advanced framework, users can optionally provide a pair of query logic functions: the first function, called *Partial Input Query (PIQ)* function, partially evaluates query logic on a subset of the input stream, i.e., a partitioned in-order stream, and produces an intermediate result stream; the second function, called *merge* function, combines these intermediate result streams and generates a final result stream.

Consider a tumbling window (fixed-sized non-overlapping window) counting query as an example. To evaluate this query with the advanced framework, a user needs to provide a PIQ function that counts events in each tumbling window on a partitioned stream, and a merge function that adds the partial results together for every tumbling window.

Figure 6(b) illustrates the workflow of the advanced framework with the example reorder latencies. In addition to sort and union operators that have appeared in the basic framework, we embed user-defined PIQ and merge operators in this framework. A PIQ operator is applied on each partitioned stream, while a merge operator is added immediately following each union operator. We note that if we use a pass-through operator as both PIQ and merge functions, the advanced framework is reduced to the basic framework.

The embedded query logic in this framework plays a critical role in improving throughput and reducing memory usage. Since PIQ operators are executed on a set of non-overlapping subsets of the input stream, each event in the input stream is evaluated by PIQ operators exactly once. Thus, we avoid redundant computation on the input events, and therefore sustain high throughput. Furthermore, unlike the basic framework that buffers original events in the union operators, the advanced framework executes PIO operators first, and then buffers intermediate results generated by PIQ operators in the union operators. Analytic queries generally involve a large volume of input data, but often report aggregate analysis results, which are usually far smaller than the input data. In these scenarios, an intermediate result stream is much smaller than the original input stream. As a result, the advanced framework usually reduces memory usage.

Taking the tumbling window counting query (described above) as an example again, each PIQ operator produces one event for each tumbling window, that represents an aggregate result on a large number of original events within the window. Thus, in the advanced framework, each union operator only needs to buffer one event per window, whereas in the basic framework it has to buffer a large number of original events.

Previous work on processing disordered streams [2] relies on out-of-order operators to reduce memory usage, which essentially keep track of query states (e.g., aggregate results of all windows) instead of the raw input data. Interestingly, the intermediate results produced by PIQ operators in Impatience framework could be viewed as a form of query states, which are temporarily buffered in the union operators to synchronize with another intermediate result stream (as shown in Figure 6(b)). In that sense, the framework reduces the memory usage in a similar way as the out-of-order operator technique [2].

However, unlike the out-of-order operator technique [2] that needs to carefully adapt each in-order streaming operator to an out-of-order counterpart, our solution copes with the latency-completeness trade-off by simply connecting a network of in-order operators (besides sorting operators). This mechanism keeps all operators besides sorting operators free from handling out-of-order events, allowing us to use our high-performance in-order operators unmodified. As a result, this framework reduces the design effort and code complexity of operators in a streaming engine, and is especially attractive for users who need to implement user-defined operators in addition to the standard operators in a streaming engine.

C. Programming Interfaces

We provide API in Trill to enable users to execute streaming queries using the Impatience framework. More specifically, we create a *Streamables* abstraction to represent a sequence of streams produced by the framework. In this section, we illustrate the usage of the framework using examples.

The first example query is to compute an one-second windowed count of clicks for each ad, with two reorder latencies $\{1 \text{ sec}, 1 \text{ min}\}$. The sample code is shown as follows:

```
DisorderedStreamable<> ds = File.ToDisorderedStreamable()
  .Select(e => e.AdId).TumblingWindow(1s);
var piq = str => str.GroupApply(e => e.AdId,
  s => s.Aggregate(w => w.Count()));
var merge = str => str.Add(e => e);
long[] rl = {1s, 1m};
Streamables<> ss = ds.ToStreamables(rl, piq, merge);
ss.Streamable(0).Subscribe(e => Console.Write(e));
ss.Streamable(1).Subscribe(e => Console.Write(e));
```

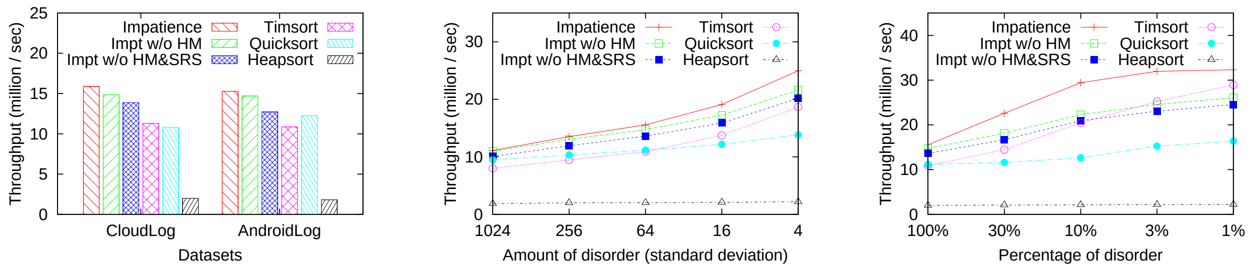
This query first runs a projection operator and a window operator on DisorderedStreamable to leverage the sort-as-needed execution: these operators are pushed down to reduce the disorder in the stream that is fed into Impatience framework (see Section IV-B). Then, we create a PIG lambda expression that computes partial aggregate results on each partitioned stream, and a merge lambda expression that combines partial aggregate results. Note that both PIG and merge functions are specified as normal Trill operators. Given the two lambda expressions, we create an instance of Streamables by calling the *ToStreamables* method on the DisorderedStreamable. Thus, a DAG of operators with the advanced framework is automatically created as shown in Figure 6. Listeners can subscribe to the produced streams in this DAG. Here, the example listener simply prints out the results. In practice, users can write custom code to manipulate these results, e.g., illustrating these results in an online dashboard.

The second example query is to find users who click ad X followed by clicking ad Y within a one-minute window. Unlike the first example query, it is not straightforward to write a pair of PIQ and merge functions for this query logic. In this case, we can simply use the basic framework by not specifying PIQ and merge lambda expressions:

```
DisorderedStreamable<> ds = File.ToDisorderedStreamable()
  .Where(e => e.AdId == X||e.AdId == Y).Window(1m);
Streamables<> ss = ds.ToStreamables({5m, 1h});
ss.Streamable(0).PatternMatch(...)
            .Subscribe(e => Console.Write(e));
ss.Streamable(1).PatternMatch(...)
            .Subscribe(e => Console.Write(e));
```

In this query, we first run a selection operator on the constructed DisorderedStreamable to retain events that are related to X or Y. After that, we sort the filtered DisorderedStreamable and create a Streamables using the basic framework. Pattern matching operators are executed on each Streamable in the Streamables. However, as the first output stream is a subset of the second stream, the execution of this query involves redundant computation.

It is also feasible to write a more sophisticated program to reduce the redundant computation for this use case, by



(a) Real datasets (b) Synthetic (varying amount of disorder) (c) Synthetic (varying % of disorder)

Fig. 7: Throughput comparison of offline sorting algorithms on real and synthetic datasets.

providing custom PIQ and merge functions. For instance, the user can provide a pair of PIQ and merge functions that combine multiple events into one event, if these events are related to same user and ad, and are overlapped in their validity time intervals. Thus, the subsequent pattern matching operators are performed on smaller streams. Here, we omit the optimized code in interest of space.

VI. Evaluation

A. *Experimental Settings*

We ran our experiments on a workstation with a 3.5GHz Intel Xeon E5-1620 v3 processor, and 16GB of DDR3 main memory. The machine runs 64-bit Windows 10 operating system. In all the results below, we ran experiments using a single thread. We implemented our solution in Trill [6], a high-performance streaming query processor.

We used both synthetic and real-world datasets in our evaluation. Each event has four 32-bit integer payload fields. The details of these datasets are shown as follows.

- Real datasets. We used the CloudLog and AndroidLog datasets presented in Section II. For each dataset, we ran experiments on 20 million events, as the CloudLog dataset contains 20 million events. We also experimented with a larger number of events on the Android dataset, and saw similar performance.
- Synthetic dataset. We also built a data generator to produce synthetic datasets that model out-of-order logs. The generator takes two parameters: percentage of disorder (p) and amount of disorder (d). It starts with a sorted dataset with increasing timestamps, and makes p% of events delayed by moving their timestamps backward, based on the absolute value of a sample from a normal distribution with mean 0 and standard deviation d.

B. *Sorting Performance*

We first evaluated and compared Impatience sort to Patience sort, Quicksort, Timsort, and Heapsort on both offline and online data.

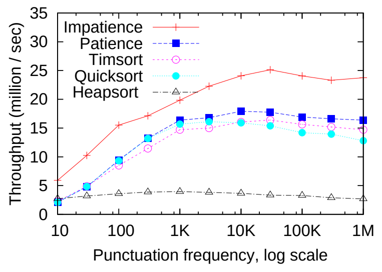
- Patience sort. Patience sort is the sorting algorithm on which Impatience sort is based, but does not originally support incremental sorting.
- Quicksort. Quicksort is a popular sorting algorithm, and is often known as the best practical choice for sorting, because it is remarkably efficient on average cases. Note that although Quicksort is not originally proposed for nearly sorted data, we observed that Quicksort is also adaptive with respect to the existing order of input data. This observation is in line with previous work [17].
- Timsort. Timsort is designed to take advantage of the partial ordering that already exist in real-world data. This algorithm finds subsets of the data that are already ordered, and uses that knowledge to sort the remaining elements more effciently. Timsort has been Python's standard sorting algorithm since version 2.3.
- Heapsort. Although it is well known that Heapsort is generally slower than Quicksort, it naturally and efficiently supports incremental sorting on streaming data.

Although Patience sort, Quicksort and Timsort are adaptive to the existing order of the input data, they are not designed for incremental sorting. In the evaluation below on online data (Section VI-B2), we adapt each sorting algorithm to an incremental sorting method using a general solution. In order to enable an arbitrary sorting algorithm to support punctuations, we maintain a sorted buffer and an unsorted buffer. Newly ingested out-of-order events are added into the unsorted buffer. On receiving a punctuation, we first sort all events in the unsorted buffer using the specified sorting algorithm, and merge these events into the sorted buffer. The unsorted buffer is drained after the merge phase. Finally, we perform a binary search to find the position of the punctuation timestamp in the sorted buffer, and outputs all events whose timestamps are less than the punctuation timestamp. With this method, each event is sorted only once (in the unsorted buffer), but could be written multiple times in a series of subsequent merge phases.

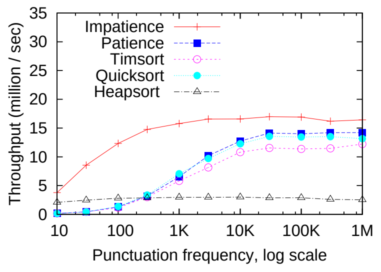
1) Experiments on offline data: We begin with evaluating the performance of sorting algorithms on offline data, i.e., we do not insert punctuations into the input stream, and thus sort the input after receiving all out-of-order events.

Figure 7(a) shows the throughput of the four sorting algorithms on the CloudLog and AndroidLog datasets. Impatience sort outperforms all competitors, and is 36.2% and 24.6% faster than the best competitor on the CloudLog and AndroidLog datasets, respectively. This is largely because the base algorithm of Impatience sort (Patience sort) efficiently leverages the partial ordering that already exists in these logs. As we shown in Section II, even though both real datasets represent two different types of log workloads, the number of sorted runs produced in Impatience sort are relatively low on both datasets, leading to high performance on both the partition and merge phases. Quicksort and Timsort generally have comparable performance on both real datasets. Not surprisingly, Heapsort is the worst, as maintaining a large heap incurs many CPU cache misses and significantly hinders the sorting performance.

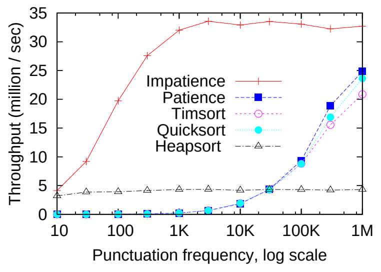
Figure 7(b) and 7(c) plot the throughput of the four sorting algorithms on the synthetic dataset, with varying the amount of disorder (standard deviation) and the percentage of disorder. Impatience sort, Quicksort, and Timsort have similar performance when the standard deviation is large or the percentage of disorder is high. However, when we reduce the amount or percentage of disorder, Impatience sort quickly improves its



(a) Synthetic dataset

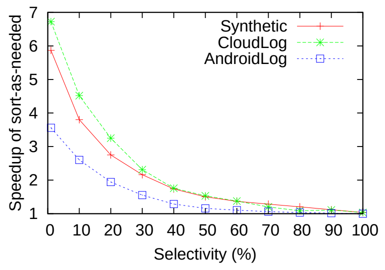


(b) CloudLog dataset

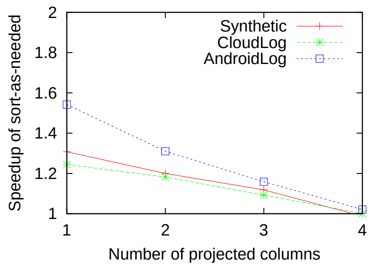


(c) AndroidLog dataset

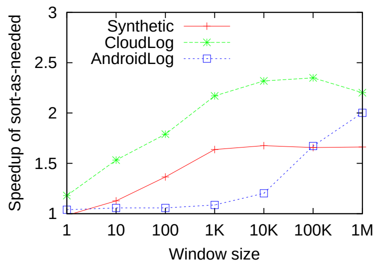
Fig. 8: Throughput comparison of online sorting algorithms on real and synthetic datasets.



(a) Selection



(b) Projection



(c) Tumbling window

Fig. 9: Speedup of sort-as-needed execution on real and synthetic datasets.

performance, and achieves larger speedup over Quicksort and Timsort. When the percentage of disorder is remarkably low, e.g., 1%, the sorting performance of both Impatience sort and Timsort is largely dominated by the sequential scan over the input data. As a result, the throughput of Timsort is close to that of Impatience sort in this scenario. In contrast to these algorithms, Heapsort is not adaptive to the existing order in the input data and shows a nearly flat line in both figures.

In Figure 7, we also show the performance of Impatience sort without the Huffman Merge (HM) optimization (labeled as "Impt w/o HM") and without both the Huffman Merge and the Speculative Run Selection (SRS) optimizations (labeled as "Impt w/o HM&SRS") that we described in Section III-E. "Impt w/o HM&SRS" is identical to the Patience sort on offline data. As can be seen from the figure, HM and SRS improve the performance of Impatience sort by up to 30% and 15%, respectively. SRS is especially effective on the Android dataset where SRS can leverage many long natural runs to avoid binary searches.

2) Experiments on online data: We next evaluate Impatience sort on online data, where the sorting operator has to incrementally perform sorting on the input stream based on the progress information provided by punctuations. In experiments below, we inserted punctuations into datasets based on the specified punctuation frequency. When a sorting operator receives a punctuation, it has to immediately sort and emit all events that are prior to the timestamp specified in the punctuation.

Figure 8 plots the throughput of the five sorting algorithms on the synthetic dataset ($p = 30\%, d = 64$), the CloudLog dataset, and the AndroidLog dataset, with varying the punctuation frequency from 10 to 1,000,000. The value of punctuation frequency represents the number of events between two continuous punctuations. We tuned the reorder latency for each dataset independently, to ensure that the sorting operator can tolerate a majority of late events and only drop events that arrive noticeably late.

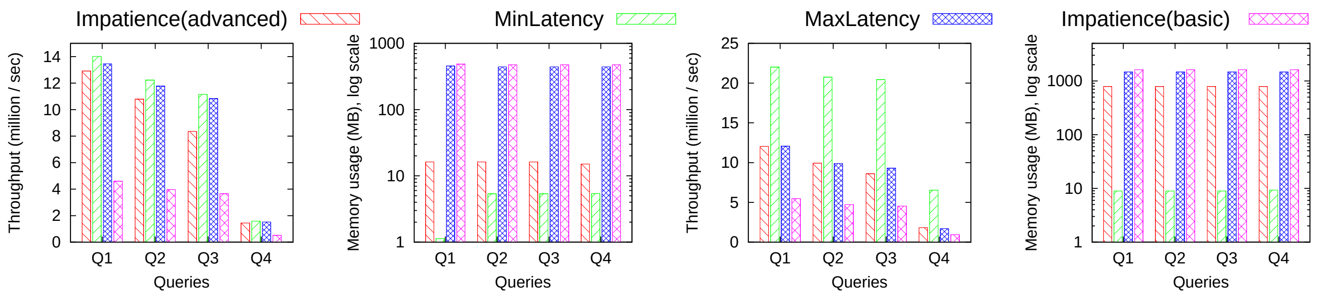
As can be seen from Figure 8(a), Impatience sort is 1.3X-2.1X faster than the best competitor across all punctuation frequency, on the synthetic dataset. Patience sort, Quicksort and Timsort achieve reasonable throughput, and are always faster than Heapsort on this dataset. This is mainly because they only needs to buffer around 1000 events and their throughput is not significantly hurt by the merge phase on the sorted and unsorted buffer.

However, Impatience sort yields far higher speedup factors on real datasets, where the sorting operator has to buffer a large number of events to tolerate severely late events. Figure 8(b) and Figure 8(c) illustrate that Impatience sort achieves 1.3X-4.4X and 1.3X-7.9X performance improvement over the fastest competitor on the CloudLog and AndroidLog datasets, respectively. On receiving a punctuation, Patience sort, Quicksort and Timsort have to read and write all the buffered data during the merge phase, and therefore result in significantly lower throughput. In contrast, as Impatience sort internally maintains a set of sorted runs, it can quickly find all required events without accessing all the buffered data. Consequently, the sorting performance is only related to the punctuation frequency, largely regardless of the number of buffered events. For Heapsort, although it is also insensitive to frequent punctuations, its throughput is nevertheless significantly lower than that of Impatience sort in most of the cases.

C. *Impact of Other Operators*

In order to understand the effect of sort-as-needed execution, we evaluate the impact of order-insensitive operators on the sorting performance of Impatience sort. In the evaluation below, we compare the throughput of Impatience sort with and without early execution of selection, projection, and tumbling window operators.

Figure 9(a) plots the performance improvement on Impatience sort caused by early execution of a selection operator, with varying the selectivity of the selection operator. In this case, sort-as-needed execution achieves up to 7X speedup on the sorting performance. In Trill, a selection operator is performed by simply marking corresponding bits in a bitmap for unmatched events. Thus, Impatience sort still needs to access all bits in the bitmap, and might bring other fields (e.g.,



(a) CloudLog: throughput (b) CloudLog: memory (c) AndroidLog: throughput (d) AndroidLog: memory

Fig. 10: Throughput and memory usage comparison with and without Impatience framework.

timestamps and payloads) of filtered events into the processor due to locality issues. As a result, the sort-as-needed execution does not achieve an ideal speedup of $1/s$ in Trill, when the selectivity is s.

As shown in Figure 9(b), a projection operator improves the sorting performance by up to 1.5X. This speedup factor is lower than the ideal speedup of 4X, when we project only one payload field from the four payload fields. The reason for this behavior is because in addition to the payload fields, each event in Trill also contains two 64-bit timestamps, a 32-bit key, and a 64-bit hash value. These metadata fields reduce the impact of the projection operator on event size.

Interestingly, a window operator also improves the throughput of Impatience sort by up to 2.4X, as shown in Figure 9(c). In Trill, a window operator aligns event timestamps to window boundaries, and thus reduces the disorder in the input data. Compared to the synthetic and CloudLog datasets, the performance improvement is decreased on the AndroidLog dataset. This is because the AndroidLog dataset contains long runs of ordered events (see Table I), which means that there is a limited opportunity to improve sorting performance by further reducing the disorder in the input data.

D. Evaluation of Impatience Framework

Finally, we compare the throughput of query execution with and without Impatience framework, in terms of completeness, latency, throughput, and memory usage. The tag "Impatience (advanced)" refers to the execution with a set of three reorder latencies using the advanced Impatience framework (Section V-B). The tag "MinLatency" and "MaxLatency" refer to the execution with the minimum and maximum latency values used in the Impatience method. The tag "Impatience (basic)" refers to the method that uses the basic Impatience framework (Section V-A) and repeatedly runs the same query multiple times with different reorder latencies.

Table II summarizes the latency and completeness of the four methods tested in this evaluation. The MinLatency and MaxLatency methods make a different trade-off between latency and completeness: the MinLatency method delivers low-latency results but loses 1.9% and 79.5% of events on the CloudLog and AndroidLog datasets, respectively; the MaxLatency method delivers high-latency but more accurate results. On the other hand, both the advanced and basic Impatience methods capture the best of both worlds: both methods deliver early query results while preserving late data.

Figure 10 compares the throughput and memory usage of the four methods with four queries on the two real datasets. The four test queries are shown as follows: Q1 is a simple tumbling window counting query; Q2 is to compute the number of events in 100 distinct groups, on tumbling windows;

TABLE II: Latency and completeness of various methods.

	CloudLog		AndroidLog	
	Latency	Completeness	Latency	Completeness
Impatience (advanced)	{1s, 1m, 1h}	100%	{10m, 1h, 1d}	92.2%
MinLatency	{1s}	98.1%	{10m}	20.5%
MaxLatency	{1h}	100%	{1d}	92.2%
Impatience (basic)	{1s}+{1m}+{1h}	100%	{10m}+{1h}+{1d}	92.2%

Q3 is similar to Q2, but with 1000 groups; Q4 computes the top 5 results for 100 groups. All tested queries were implemented using unmodified Trill operators with the advanced Impatience framework. In this experiment below, we set the punctuation frequency to 10,000.

As can be observed in Figure 10(a) and Figure 10(b), the advanced framework is 2.8X, 2.7X, 2.3X, and 2.8X faster than the basic framework and saves the memory usage by 29.8X, 29.2X, 29.2X, and 31.5X for the queries Q1-Q4, respectively. Compared to the MaxLatency method, the advanced framework can deliver low-latency early results and save the memory consumption by 27X-29X, at the expense of 4-22% overhead in throughput. Although the MinLatency method produces less accurate query results, Impatience framework is only 4-22% slower than the MinLatency method in throughput. To summarize, the advanced Impatience framework offers query execution with low latency, high completeness, high throughput, as well as low memory usage.

On the AndroidLog dataset, as shown in Figure 10(c) and Figure 10(d), the advanced Impatience framework achieves 1.9X-2.2X throughput improvement over the basic framework and is slightly (1%-7%) slower than the MaxLatency method. In addition, the advanced framework uses 1.9X less memory than the MaxLatency and the basic framework methods on this dataset. The reduction in the memory usage is less than that on the CloudLog dataset, because a majority of events are significantly delayed on this dataset. Note that the MinLatency method is especially fast and efficient in memory usage on this dataset. However, this is because with the reorder latency of 10 minutes, the sort operator only processes 20.5% of the input data and produces far less accurate results on this dataset.

VII. Related work

Disorder in Streams. The first generation of streaming systems assumed that events arrive in timestamp order at the SPE [18]. One initial solution to handle disorder was k-slack [13], [19], where the stream is assumed to be disordered by at most k tuples or time units, with reordering performed before stream processing. Such an approach can lead to potentially uncontrolled latency.

Later systems provided low latency by processing events out-of-order and issuing *compensations* on the receipt of late-arriving events. Examples of such proposals include revision tuples [20] and lifetime modifications [5]. Execution strategy based on speculation is proposed to support retracting incorrect outputs on receiving late arriving data [4]. NiagaraST [2] proposes operators that manage out-of-order state, but produce output only on punctuation. While this strategy helps with memory, it does not provide the low latency that we desire. Handing out-of-order natively inside an operator has been explored in the context of more complex operators such as pattern detection as well [21]–[23]. Generally, however, adding native support for disorder in operators can be complex, resulting in low performance. With systems such as Trill [6] significantly raising the bar for throughput compared to the first generation of streaming systems (by two to four orders-of-magnitude), modifying operators to natively handle disorder often results in significant inefficiencies. Even worse, there can be a non-trivial amount of revision traffic (e.g., in the form of compensating events that correct an early result) due to late-arriving events, which is exacerbated by query composition. The overall result is unacceptable throughput with such solutions.

In contrast, we allow users to specify a reorder latency, and further accept a set of reorder latencies to provide explicit control over the latency-completeness tradeoff, while allowing us to use our high-performance operators unmodified. Further, we leverage punctuation [24] to bound disorder in the stream, and use impatience sort to reorder data based on the specified latency to retain high performance at data ingestion as well.

Big Data Systems. Big data systems such as Storm [25] and Spark Streaming [26] do not provide support for late-arriving events; users have to implement logic to handle such events. Google Dataflow [1] is a streaming model that supports out-of-order processing, where an event can trigger prior windows based on a triggering mechanism. Apache Flink [27] uses watermarks to reorder events before processing them. In all these systems, the user is responsible for providing logic to handle late events — the proposals in this paper can be applied in such settings at the user layer.

Sorting and Query Optimization. There is a large body of work in designing adaptive sorting algorithms (see [10] for a summary of adaptive sorting algorithms). These algorithms take advantage of existing order in the input data, but are unable to sort data in an incremental way, and thus fall short of one performance requirement for a streaming sorting operator. BSort, an incremental sorting algorithm used in the Aurora streaming engine [28], is essentially a variant of insertion sort, and therefore is not efficient in sorting a large number of events.

Sorting operator has also been well studied for relational query optimization, e.g., [29], [30]. We adapted these techniques to a streaming setting where events are streamed into a non-blocking sorting operator, and are incrementally sorted based on their timestamps. Unlike relational query optimization, we developed programming interfaces to provide users the flexibility to appropriately place a sorting operator in an execution plan, as users often have comprehensive understanding of these long-running streaming queries. In addition to classic relational operators like selection and projection operators, we also observed that this technique is very attractive for some streaming-specific operators such as window operators.

VIII. Conclusions

This paper proposes a technique for processing real-time queries over out-of-order streams in high-performance streaming engines. This solution relies on Impatience sort, an adaptive sorting algorithm tailored for streaming data, to handle out-of-order events, keeping all other operators free from processing disordered streams. To overcome the pitfalls associated with the traditional sort-based methods, the solution combines the sorting algorithm with Impatience framework, a new method that reduces latency and memory usage of query execution while preserving high completeness and high throughput. We have empirically demonstrated the performance and usability aspects of the proposed solution in a streaming engine, and show that our methods significantly improve the sorting performance and reduce memory usage, and in some cases by over an order of magnitude.

References

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.

[2] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *PVLDB*, vol. 1, no. 1, pp. 274–288, 2008.

[3] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre, "Continuous analytics over discontinuous streams," in *SIGMOD 2010*, 2010, pp. 1081–1092.

[4] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," in *CIDR 2007*, 2007, pp. 363–374.

[5] "Microsoft Streaminsight," http://aka.ms/stream.

[6] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A high-performance incremental query processor for diverse analytics," *PVLDB*, vol. 8, no. 4, pp. 401–412, 2014.

[7] C. L. Mallows, "Problem 62-2, patience sorting," *SIAM Review*, vol. 4, no. 2, pp. 148–149, 1962.

[8] ——, "Patience sorting," *Bull. Inst. Math. Appl.*, vol. 9, pp. 216–224, 1973.

[9] B. Chandramouli and J. Goldstein, "Patience is a virtue: revisiting merge and sort on modern processors," in *SIGMOD 2014*, 2014, pp. 731–742.

[10] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Comput. Surv.*, vol. 24, no. 4, pp. 441–476, 1992.

[11] D. T. Wagner, A. C. Rice, and A. R. Beresford, "Device analyzer: largescale mobile data collection," *SIGMETRICS Performance Evaluation Review*, vol. 41, no. 4, pp. 53–56, 2014.

[12] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, Sept 1952.

[13] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *PODS 2004*, 2004, pp. 263–274.

[14] B. Chandramouli, J. Goldstein, M. Barnett, and J. F. Terwilliger, "Trill: Engineering a library for diverse analytics," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 51–60, 2015.

[15] "LINQ (language-integrated query)," https://msdn.microsoft.com/en-us/library/bb397926.aspx.

[16] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI 2004*, 2004, pp. 137–150.

[17] G. S. Brodal, R. Fagerberg, and G. Moruz, "On the adaptiveness of quicksort," *ACM Journal of Experimental Algorithmics*, vol. 12, 2008.

[18] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," *PODS*, 2002.

[19] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams," *ACM Trans. Database Syst.*, vol. 29, no. 3, pp. 545–580, Sep. 2004.

[20] E. Ryvkina *et al.*, "Revision processing in a stream processing engine: A high-level design," in *ICDE*, 2006.

[21] B. Chandramouli, J. Goldstein, and D. Maier, "High-performance dynamic pattern matching over disordered streams," *PVLDB*, vol. 3, no. 1, pp. 220–231, 2010.

[22] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool, "Sequence pattern query processing over out-of-order event streams," in *ICDE 2009*, 2009, pp. 784–795.

[23] T. Johnson, S. Muthukrishnan, and I. Rozenbaum, "Monitoring regular expressions on out-of-order streams," in *ICDE 2007*, 2007, pp. 1315–1319.

[24] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 555–568, 2003.

[25] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," in *SIGMOD 2014*, 2014, pp. 147–156.

[26] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *SOSP 2013*, 2013, pp. 423–438.

[27] "Apache Flink," https://flink.apache.org/.

[28] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[29] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass, "Bringing order to query optimization," *SIGMOD Record*, vol. 31, no. 2, pp. 5–14, 2002.

[30] A. Lerner and D. E. Shasha, "Aquery: Query language for ordered data, optimization techniques, and experiments," in *VLDB*, 2003, pp. 345–356.