LifeStream: A High-performance Stream Processing Engine for Waveform Data

Anand Jayarajan
anandj@cs.toronto.edu
University of Toronto, Canada
Vector Institute, Canada

Kimberly Hau
kimberly.hau@mail.utoronto.ca
University of Toronto, Canada

Andrew Goodwin
andrew.goodwin@sickkids.ca
The Hospital for Sick Children, Canada
University of Sydney, Australia

Gennady Pekhimenko
pekhimenko@cs.toronto.edu
University of Toronto, Canada
Vector Institute, Canada

ABSTRACT

Hospitals around the world collect massive amount of physiological data from their patients every day. Recently, there has been increasing research interest to subject this data into statistical analysis for gaining more insights and providing improved medical diagnoses. Enabling such advancements in healthcare require efficient data processing systems. In this paper, we show that currently available data processing solutions either fail to meet the performance requirements or lack simple and flexible programming interfaces. To address this problem, we propose LifeStream, a high performance stream processing engine for physiological data. LifeStream hits the sweet spot between ease of programming by providing a rich temporal query language support and performance by employing optimizations that exploit the constant frequency nature of physiological data. LifeStream demonstrates end-to-end performance up to 7.5× higher than state-of-the-art streaming engines and 3.2× than hand-optimized numerical libraries.

1 INTRODUCTION

In recent years, healthcare industry has been experiencing an increasing trend in the adoption of approaches like data-driven diagnostic methods [12, 41], automated patient monitoring systems [21], and AI-assisted risk prediction models [2, 19, 43]. Advancements in the data collection technologies [14] and recent developments in fields like statistics and machine learning [30, 45] are the major enabling factors for this shift from the traditional methods of healthcare practices. Hospitals collect and store hundreds of gigabytes of physiological data every day with the help of monitoring devices [39] attached to the patients in the intensive care units (ICUs) [7, 15]. The monitors continuously collect physiological signals or waveforms such as arterial blood pressure (ABP), electrocardiogram (ECG), and electroencephalogram (EEG) and produce output at regular intervals in a streaming manner. A single measurement or event in the waveform data typically contains a timestamp and a measurement value at that moment in time.

Traditionally, this data has been monitored and analyzed manually by the clinicians. However, statistical and machine learning based algorithms can provide insights into complex data patterns that can help clinicians to prepare more precise diagnosis and personalized treatment plans [24]. Moreover, statistical models are shown to be capable of accurately predicting short and long term trends in the physiological data such as cardiac arrest [43] and sepsis risk [11]. Even though data analytics on physiological data shows great potential, there are several practical challenges that need to be addressed in order to unleash its full potential.

Unlike other streaming datasets, raw physiological data has a high degree of *noise* and *discontinuities*. Therefore, the data needs to go through a series of data cleaning operations and transformations before it can be used for meaningful analyses. Additionally, in certain cases, researchers need to compute derived variables from the raw data (e.g, measuring heart rate from ECG signal or finding temporal correlation between multiple signals). Although general purpose stream processing engines that can handle these types of computations do exist in both industry and academia (e.g., Apache Spark streaming [48], Apache Beam [3], and Apache Flink [9]), we observe that they fail to be a good fit for processing physiological data for the following reason.

Most contemporary streaming engines provide simple and flexible programming interfaces with an implicit notion of event time, ordering and support for fine-grained windowing strategies that are well suited for building physiological data processing pipelines. However, most of them are designed with a distributed setup in mind and, unfortunately, exhibit poor single machine performance [25]. This is generally compensated by scaling up the computation to large machine clusters that most hospitals neither have the infrastructure nor the required expertise to operate. Moreover, hospitals have to abide by the medical data privacy protection laws [18], which restrict data being moved outside the hospital facilities, eliminating cloud infrastructure as a viable choice for running such computations. This necessitates on-premise computations over limited hardware resources. Our experiments reveal that most of the contemporary streaming engines fail to provide good performance under such hardware resource constrained setup (see Section 3 for more details).

Because of these limitations, data scientists usually prefer to write adhoc data processing pipelines using numerical libraries (e.g., NumPy [33], SciPy [34] and Scikit-learn [13]). Even though, numerical libraries provide a rich set of operations for scientific computing with efficient hand-tuned implementations, the lack of temporal ordering and windowing support, as well as the absence of the unified API specifications and common data abstraction limits data scientists' ability to efficiently program and maintain large

About to appear at ASPLOS '21, April 19-23, 2021, Virtual
.

data processing pipelines. Additionally, as the pipeline gets longer and more complex, the performance of the combined workflow starts to considerably deteriorate due to expensive data movement across the functions and lack of cross-operation optimizations [36].

Our *goal* in this work is to build a physiological data processing system that is both *easy to program* and provides *high performance* even under hardware resource constraints. To this end, we propose LifeStream, a new high performance stream processing engine for physiological waveform data with a rich temporal query language support. LifeStream provides high performance with efficient hardware utilization using optimizations that exploits the *constant frequency nature* of the waveform data. We derive the following two key properties of temporal operations on a constant frequency stream:

Linearity property: *The timestamps of the events produced by a temporal operator is a linear transformation of that of the input events.* This property allows LifeStream to map the events in the output stream of an operator to its parent events in the input stream(s). Since all the temporal operations follow this property, the mapping can be extended to compute the entire lineage of every event produced during query execution. We call this mechanism *event lineage tracking*.

Bounded memory footprint: *The memory footprint of a temporal operator is bounded by the size of its input(s).* Every temporal operator has a fixed interval size with which it operates on its input streams. Given the frequency of the input streams, the total memory required to execute that operation can be calculated statically.

We use the above properties and propose the following three query compile-time and runtime optimizations:

1. **Locality tracing:** LifeStream precisely estimates the data locality of the end-to-end data processing pipeline using the *linearity property*. LifeStream performs static analysis on top of the entire query and prepares an execution plan that maximizes the cache locality for the whole pipeline.
2. **Static memory allocation:** LifeStream estimates the upper bound of the memory required for each operation in the pipeline using the *bounded memory footprint property* and pre-allocates the memory for all intermediate results produced in the stream, thus almost completely eliminating the runtime memory allocation and deallocation overhead.
3. **Targeted query processing:** We observe that the discontinuities in the physiological data are highly uneven across different signals and the number of mutually overlapping events are generally far fewer than the total number of events in the streams. Therefore, joining multiple streams together filters out many events, rendering any prior computation on them wasteful. LifeStream eliminates such redundant computations using *event lineage tracking* mechanism at runtime by selectively targeting only regions of input data that are expected to produce an output.

We evaluate LifeStream against state-of-the-art streaming engines and numerical libraries on real datasets collected at The Hospital for Sick Children, Canada. On a single machine, LifeStream exhibits up to 7.5× higher end-to-end performance compared to the state-of-the-art streaming engine called Trill [10], and 3.2× compared to the hand-tuned numerical libraries such as SciPy [34],

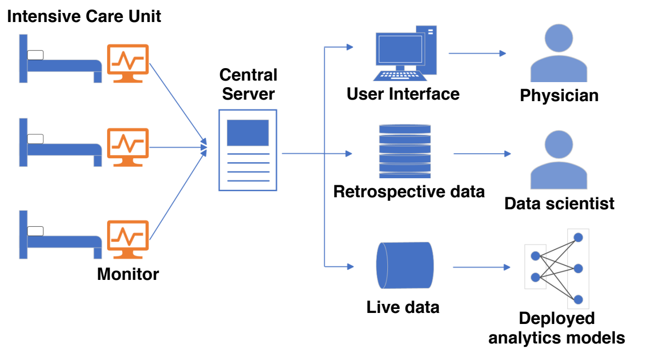


Figure 1: Typical physiological data collection infrastructure

NumpPy [33], and Scikit-learn [13] (while providing much more flexible programming API and corresponding abstractions). LifeStream also extends the traditional query language vocabulary to support certain domain-specific use cases found in the physiological data processing domain (e.g., artifact/shape detection in the signal stream). Finally, we note that even though LifeStream is built for stream processing on physiological data, the ideas behind LifeStream can also be applied to other streaming use cases where data is produced at a constant frequency.

In summary, this paper makes the following contributions:

- We showcase the challenges faced in the domain of physiological data processing and propose solutions that are evaluated on real datasets and workloads used in major hospitals.
- We derive two key properties of temporal operations on constant frequency streams, namely *linearity* and *bounded memory footprint*, and leverage them to propose three key optimizations, namely *locality tracing*, *static memory allocation*, and *targeted query processing*, that can significantly improve the hardware utilization and query execution performance compared to the state-of-the-art streaming engines.
- We propose LifeStream, a new high performance stream processing engine with rich temporal query language support. We show that LifeStream can outperform state-of-the-art streaming engines by as much as 7.5× and hand-optimized numerical libraries by as much as 3.2× on the end-to-end data processing pipeline on real physiological datasets.

2 PHYSIOLOGICAL DATA COLLECTION AND PROCESSING

Figure 1 shows a typical physiological data collection process from the patients in the intensive care units (ICUs) to keep track of their health status with the help of multiple monitoring devices [39] attached to patients, each making different measurements such as arterial blood pressure (ABP), electrocardiogram (ECG), and electroencephalogram (EEG). These devices generate a continuous stream of signal events at constant intervals, typically at a rate ranging from 10^{-4} Hz to 10^{3} Hz. Each signal event constitutes a timestamp corresponding to the time of measurement and a signal value which is the magnitude of the measurement at that point in time. We use the term *period* to refer to the shortest time interval

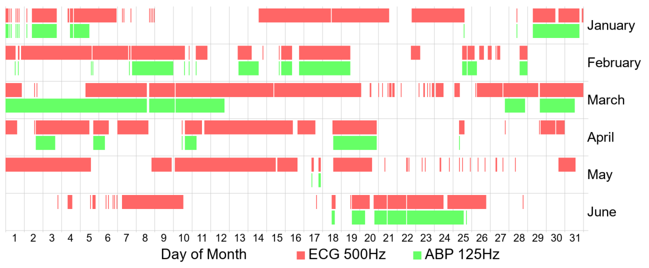


Figure 2: Distribution of ECG and ABP signals collected from a single monitoring device over first six months of 2019

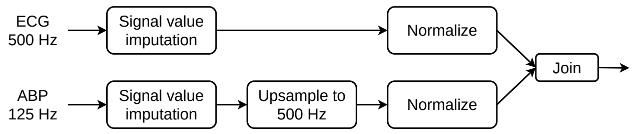


Figure 3: A sample data processing pipeline on ECG and ABP

between consecutive events in a signal. For example, a 500 Hz signal stream would have a period of 2 ms.

Unlike other streaming datasets, physiological waveform data is known to contain a high degree of *noise* and many *discontinuities*, because of external disruptions in the connection between the monitoring devices and the patients. Figure 2 shows the discontinuities in the ECG and ABP signals collected from a single monitoring device over a 6 month period. Such disruptions are common, which make it virtually impossible to run meaningful analyses on top of raw data. Therefore, the data has to go through a series of transformations and data cleaning operations before a data scientist can run analytics algorithms. For example, standard signal processing operations like frequency-based filtering [40] are used for removing the noise from the signals, signal value imputation methods are used to fill small discontinuities in the data stream with dummy values, and data normalization methods are used to convert all signals to a uniform scale. Additionally, different data analytics algorithms might require additional variables that are derived from the raw data (e.g., the systolic and diastolic blood pressure [8], heart rate measured from ECG, and the temporal correlation of different signals).

Figure 3 shows a sample data processing pipeline which joins a 125 Hz ABP signal with 500 Hz ECG signal based on their timestamps. First, the small gaps in both waveforms are filled using signal value imputation. Next, the ABP signal is upsampled to match the frequency of ECG. Finally, the signal values are normalized before joining them together to pair up strictly overlapping events. Even though there are several general-purpose solutions [3, 9, 10, 44, 48] proposed in the big data community to build and process such data flow pipelines, from our experience closely working with the clinicians, data analysts, and machine learning researchers at The Hospital for Sick Children, we recognize several new challenges that make physiological waveform data processing unique.

First, the choice of operations and transformations applied on the waveform data varies considerably based on use cases. Therefore, the data scientists should have *freedom to experiment* with different data processing pipelines and should be able to do so with *minimal effort*. Moreover, most of the operations and transformations applied on the waveform data follows a strict notion of the temporal ordering of the data. This necessitates a flexible and easy to use programming interface with *in-built temporal logic support*. Second, the data analysts and ML researchers usually first perform the experiments and analysis on the retrospective (i.e. historical) data stored in the persistent disks, and then deploy their solutions on real-time data once the algorithms are finalized. It is crucial for this to be a *seamless and error-free deployment*. Finally, as described in Section 1, since the physiological data collected are fully-identified, there are several legal restrictions [18] on moving the dataset outside hospital facilities. Even though there has been some recent efforts (e.g., MIMIC [23]) to de-identify data for public research purposes, the risk of leaking patient information from those datasets still pertains [1]. Hence, most hospitals tend to keep even de-identified data private. As a result, the physiological data processing systems has to perform *computations within limited hardware budget* available in hospitals and still provide high performance.

```
var left = sig500
    .Multicast(s => s
      .Select(e => e.val) // select signal value
      // compute mean and subtract from values
      .Join(s.TumblingWindow(100).Mean(),
        (val, mean) => val - mean));
var right = sig200
    .Select(e => e.val); // select signal value
var output = left
    // join with sig200 values
    .Join(right, (l, r) => new {l, r});
```

Listing 1: Running example of a temporal query

3 A CASE FOR EFFICIENT TEMPORAL STREAMING FOR WAVEFORM DATA PROCESSING

Since physiological waveform data is produced in a streaming fashion, stream processing [6] is a natural choice for building aforementioned data processing pipelines. Modern streaming engines [3, 5, 10, 44, 48] support some type of a temporal query language which has implicit notion of event time, ordering, and fine-grained windowing strategies. To illustrate, Listing 1 shows the query for a simplified version of the pipeline in Figure 3 which joins a 500 Hz signal (`sig500`) and 200 Hz signal (`sig200`) after a series of transformations.[1] First, the signal values of `sig500` is adjusted by taking the mean of signal values on 100 ms tumbling window (fixed-size, non-overlapping and contiguous intervals) and subtracting that from the original values. This transformed signal values are joined with the signal values from `sig200` using temporal *Inner Join*. Through such query languages, modern streaming engines provide simple and flexible programming interface for writing complex data processing pipelines. However, most contemporary streaming engines are built with distributed setup in mind and, unfortunately, exhibit sub-optimal single machine performance as a result of poor hardware utilization [10, 25]. At the same time, streaming engines

[1]We use signal frequencies 500 Hz and 200 Hz to show that LifeStream can handle misaligned signals as well.

Benchmarks	Spark	Storm	Flink	Trill	SciPy
Temporal Join	0.07	0.04	0.09	0.80	-
Upsampling	-	-	-	0.69	15.06

Table 1: Throughput of Spark, Storm, Flink, Trill, and SciPy (in million events/sec)

that are optimized for single machine performance (e.g., Trill [10]) demonstrate orders of magnitude higher performance.

To validate this in the context of waveform data processing, we compare the single-core performance of the temporal *Join* operation in four major state-of-the-art streaming engines (i) Spark streaming [5, 48], (ii) Flink [9], (iii) Storm [44], and (iv) Trill [10] from Microsoft Research. *Join* is one of the most commonly used primitive operations in physiological data processing, since computing derived variables such as aggregates and joining them back with the events in the input stream are frequently performed during data transformations. Therefore, performance of *Join* operation can considerably affect the performance of the entire pipeline in such scenarios.

Table 1 shows the number of signal events joined by different streaming engines per second. We observe that Trill outperforms other streaming engines by more than 10×. Trill's performance benefits come from its better memory management system, improved cache locality using columnar data representation and use of hand-optimized primitive operators that leverages certain characteristic properties of the stream.

Unfortunately, despite all these optimizations that made Trill significantly better than its competitors, we observe that the performance of Trill is far from being competitive with the hand-tuned implementations used by the data scientists. Such implementations are usually based on numerical libraries such as SciPy [34], NumPy [33], and Scikit-learn [13] and provide a rich ecosystem of highly efficient data processing operators. Table 1 shows the performance comparison of signal upsampling [47] operation implemented in Trill and the corresponding implementation available in the SciPy library. We observe that Trill is about 22× slower than SciPy. This makes numerical libraries seem like a better choice for building data processing pipelines from a performance perspective. Even though this is the status quo among data scientists, we argue that such an approach has significant drawbacks from a programmability and system maintainability perspective, and below we explain the reasons.

First, the lack of implicit notion of event time and support for flexible windowing strategies make building physiological data processing pipelines using numerical libraries significantly harder and more complicated for data scientists. For instance, writing the data transformations in Listing 1 would require data scientists to manually maintain the temporal ordering of the data at the application level. Moreover, making simple tweaks like modifying the pipeline to use a rolling mean would only take a single line of change from *TumblingWindow* to *SlidingWindow* in a temporal query language. The same change would require a complete redesign of the code base in the typical numerical library-based approaches.

These limitations force data scientists to make one of two undesirable choices. (i) To put considerable engineering effort to implement temporal features on top of the numerical libraries, or (ii) make

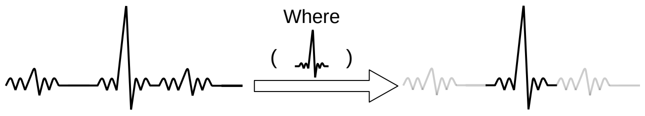


Figure 4: Shape detection using extended *Where* query

compromises in their experimentation and requirements to adjust to the restrictions imposed by numeric libraries. Secondly, a lack of common API specifications and data abstractions across different libraries further complicates building large data processing pipelines, as the data scientists need to make sure the correctness of the input data types of each function, and need to additionally perform type conversions when necessary. Such an approach quickly becomes unmanageable and error-prone when the data pipeline becomes larger and more complex.

Despite all these drawbacks, data scientists still choose to go with such adhoc library-based methods for building data processing pipelines as opposed to more systematic approaches (e.g, using stream processing engines such as Trill) due to the significant performance benefits associated with numerical libraries. As a result, we observe that data scientists end up spending majority of their development time writing peripheral code and extra "glue" logic to wire different numerical libraries together [42], instead of focusing on their primary goal—analysing data and generating insights from it.

On the other hand, even though the library-based approach seems desirable from the performance perspective at first, prior works [35–37] have pointed out that the functions in these libraries may achieve high performance in isolation, but they usually fail to maintain those benefits in a more complex workflow with a combination of functions, because of the overhead associated with intermediate data conversion and lack of cross function optimizations. This makes numerical libraries a poor choice for building physiological data processing pipelines even from a performance standpoint.

Based on the above observations, we conclude that to be efficient and easy to manage, a physiological data processing system must provide a programming interface similar to the ones supported by the major streaming engines with flexible windowing strategies and implicit notion of temporal ordering and event time. Secondly, the system must efficiently utilize the available hardware resources to provide high performance.

4 LIFESTREAM: SYSTEM OVERVIEW

To address the challenges described in the previous section, we introduce **LifeStream**, a temporal query processing engine specifically optimized for physiological waveform data processing. LifeStream lies in the sweet-spot of programmability and performance compared to the alternative approaches. LifeStream provides: (i) superior performance by taking advantage of the constant frequency nature of the waveform data and optimizing the end-to-end pipeline, (ii) a rich temporal query language support (similar to the one provided by Trill) with simple and flexible primitive temporal operations and fine-grained windowing support, and (iii) several extensions on traditional temporal operations that are useful for physiological data processing such as extending *Where* operator to query visual patterns and shapes in data streams as shown in Figure 4 (see Appendix Section A.1 for more details).

LifeStream provides the data abstraction that consists of a stream of events in chronological order. An *event* is a single unit of data with three fields: (i) a user-defined *payload*, (ii) a *sync time* which dictates the time from which that particular event is active, and (iii) a *duration* which defines the active lifetime of the event. LifeStream is exclusively targeting data streams in which events appear at a *constant frequency*. That means the sync time of every event is always at the period boundaries. Since the position of each event in a stream is predictable, we use the symbolic representation of (*offset*, *period*) to describe a stream, where offset is the sync time of the first event in the stream and period is the reciprocal of the frequency. Even though we are primarily targeting physiological waveform data, any streaming data that can be represented in the aforementioned format can take advantage of the benefits of LifeStream. This include constant frequency streaming datasets such as performance counters produced in data centers [17], data collected from wearable devices [16], and real-time sensor data [32].

One of the key design choices we make in LifeStream is to decouple the operator implementation from the data representation used for storing stream data. Most of the streaming engines such as Trill [10] and Spark streaming [5] implement their operators to take an *arbitrary* sequence of events from the input stream data in the form of a batch, and produce another batch as its output. In our work, we realize that this approach has several severe limitations. First, the locality of the computation is highly tied with the batch size. Therefore, usually the system has to trade-off the benefits of large batch processing in favor of preserving locality. Second, we observe that such operator implementations limit different compile-time and runtime optimizations we can perform on LifeStream (see Section 5 for more details).

To avoid such limitations, we introduce a new key construct called *fixed interval sliding window* or *FWindow*. FWindow is an arbitrary interval within a stream. Similar to an event, FWindow also has a sync time corresponding to the starting timestamp of the interval and a fixed size or duration. Since FWindow is an interval on a stream, the size of the FWindow should always be a multiple of the period of the stream. In LifeStream, we implement all the operations based on FWindows—the operators typically take one or two FWindows as input and produce a single FWindow as output.[2] Operators can slide the FWindows to read different parts of the stream at runtime by updating its sync time. The only restriction is that FWindows can only be moved forward in time in order to ensure monotonic progress in query execution.

LifeStream provides a rich set of primitive temporal operations with which data scientists can write queries (similar to Listing 1) on streaming data. The query is compiled into a computation graph composed of FWindows and temporal operators. The size of the FWindows are initially set to the same value as the corresponding stream's period. Figure 6(a) shows the initial computation graph prepared from the example query in Listing 1. The FWindows are represented using a symbolic representation *(offset, period)[dimension]* where the dimension is the FWindow size. This graph is then passed on through a graph transformation process to generate the final executable computation graph. Finally, the input data is streamed through the executable computation graph to generate the result.

[2]Multicast operation is an exception. It outputs only a single FWindow, but passes the same FWindow to multiple operators as input.

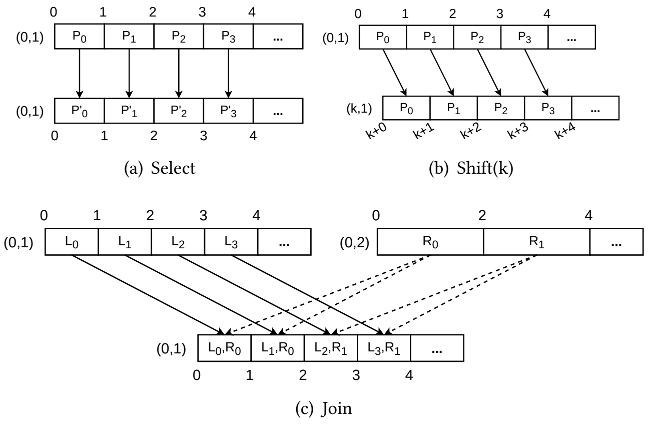


Figure 5: Event lineage tracking

5 LIFESTREAM: KEY OPTIMIZATIONS

LifeStream maximizes resource utilization by (i) improving cache locality, (ii) reducing runtime overhead, and (iii) pruning redundant computations. We achieve this goal by identifying two key properties of the temporal operations on constant frequency streams described in Section 5.1. Using which, we propose three major query compilation and execution time optimizations in sections 5.2 and 5.3.

5.1 Properties of Temporal Operations on Constant Frequency Streams

Linearity of temporal operations: *The sync time of events in the output stream of a temporal operator is a linear transformation of that of the input events.*

This property allows LifeStream to map every output event of a temporal operator to the corresponding parent input event(s). Figure 5 shows how the event times change in the output stream of several common temporal operators such as *Select*, *Shift*, and *Join* when applied on a constant frequency stream. One consequence of this property is that the period and the offset of the output stream is also a linear transformation of that of the input stream and can be computed statically. Moreover, this allows to map the events in the output stream of an operator to the corresponding parent events in the input stream(s). Figure 5 shows how the output events are mapped to the input events after an operator transformation. Since all temporal operators follow this property, the mapping can be extended from the output stream events all the way to the input stream events. We call this mechanism *event lineage tracking* and use in LifeStream to improve both the cache locality and to prune redundant computations at runtime.

Bounded space complexity: *For a constant frequency stream with period p, the maximum number of events that can be present within a given time interval d is bounded by $O(d/p)$.*

One of the key properties of a constant frequency stream is that two events can *not* overlap with each other, which means there can only be at most one event active at any point in time within a stream. Therefore, a maximum number of events in an interval is bounded by the duration of that interval. Moreover, since temporal operations also follow linearity property, all the intermediate streams generated in the query should also have constant frequency, and thus should also satisfy bounded space complexity property. LifeStream uses this observation to estimate the maximum memory footprint of all the intermediate results and preallocate them to minimize the runtime memory allocation and deallocation overhead commonly observed in other streaming engines [10, 38].

5.2 Locality Tracing and Memory Footprint Estimation

One thing that makes stream processing attractive is that even though the data it processes is usually huge (and sometimes can even be potentially infinite), the computations performed on the data are highly local and require only to deal with a small continuous window of events within the stream. Most streaming engines take advantage of this locality property *only* at an individual operation-level and do not optimize or even maintain cross-operation locality. In LifeStream, we introduce a method called *locality tracing* which uses the linearity property of constant frequency streams to precisely estimate the end-to-end locality of the computations in the *entire* pipeline. Locality tracing performs static analysis on top of the computation graph and adjusts the dimensions of all the FWindows to make sure that the input and output dimensions of all the operators match.

Figure 6 shows the locality tracing procedure performed on the example query in Listing 1. The procedure starts from the end of the pipeline, and LifeStream identifies a mismatch in the input and output dimensions of the last *Join* operation ($Join_2$). Since the FWindow sizes has to be a constant multiple of the periods, in order to match the dimensions of $Join_2$, LifeStream sets the FWindow sizes to the least common multiple of the input and output dimensions. In this case, the dimensions are set to 10. Next, the dimensions of the $Join_1$ operation is adjusted similarly. However, this adjustment introduces a mismatch in $Join_2$, which is corrected in the next step. This procedure is continued until all operations have uniform input and output dimensions. This graph transformation ensures that the intermediate results are consumed immediately by the subsequent operation(s), which, in turn, maximizes the locality of the entire query.

Once the dimensions of all the operations are computed, LifeStream uses the bounded space complexity property to determine the maximum memory footprint of all the FWindows. LifeStream then preallocates this memory statically and keep reusing the same memory during runtime in order to minimize dynamic memory allocation overhead, commonly observed in other streaming engines [38].

5.3 Targeted Query Processing

Most streaming engines process queries in an eager fashion where the query computation is initiated at the data ingestion side and each subsequent operators perform the corresponding transformations on the input as soon as it receives the data, and immediately passes it on to the next operator down the pipeline, irrespective of whether the next operation would actually need to process that data or not. In physiological waveform processing, this introduces a lot of redundant computations as the data contains high degree of discontinuity.

One of the most common examples would be the use of *Inner Joins* to pair up overlapping events from multiple signal streams after a series of compute intensive data transformations (Figure 3). Figure 2 shows that the mutually overlapping regions in ECG and ABP signal streams are far fewer compared to the total number of events in the individual streams. In an eager query processing model, all the events from both streams are invariably going to be passed through the intermediate transformations, even though most of them are eventually going to get discarded by the final *Join* operation.

In LifeStream, we address this issue by introducing targeted query processing which uses event lineage tracking to map output FWindows to corresponding parent FWindows in the input. As opposed to eager execution, in LifeStream, the query processing is initiated by the final operator rather than the initial one. This lets LifeStream operators to selectively target regions of its input stream(s) by sliding the FWindows and running the computations only when an output FWindow is expected to be produced. Hence, targeted query processing lets LifeStream skip all those compute-heavy transformations in the presence of discontinuities in the input data stream and focus only on the relevant parts of the data.

6 LIFESTREAM: IMPLEMENTATION

We implement LifeStream as a library using C# programming language and .Net core v3.1 framework [28]. Hence programmers who work with LifeStream can also benefit from high level language features such as arbitrary data types, integration with custom program logic, and rich ecosystem of libraries [29]. They can also create streams from a variety of sources including real time data through networks, retrospective data from files, and cached data from the main memory.

As described in the Section 4, the key building block that temporal operators use to access streaming data is FWindow. In FWindow, events are indexed by their sync time. In addition to the event payload, FWindow contains three extra fields, namely *vsync*, *duration*, and *bitvector*. Vsync and duration fields store the sync time and duration of the events. Bitvector is used to mark the absence of an event. Every event in the FWindow has an associated bit which can either be 0 or 1 to mark the presence or absence of the event. All the fields in the FWindow are stored in columnar format in order to maximize cache locality as most operators only need to read from or write to a subset of the fields.

In the following sections, we describe the details about the extended temporal query language supported in LifeStream and a few implementation challenges we faced and corresponding solutions.

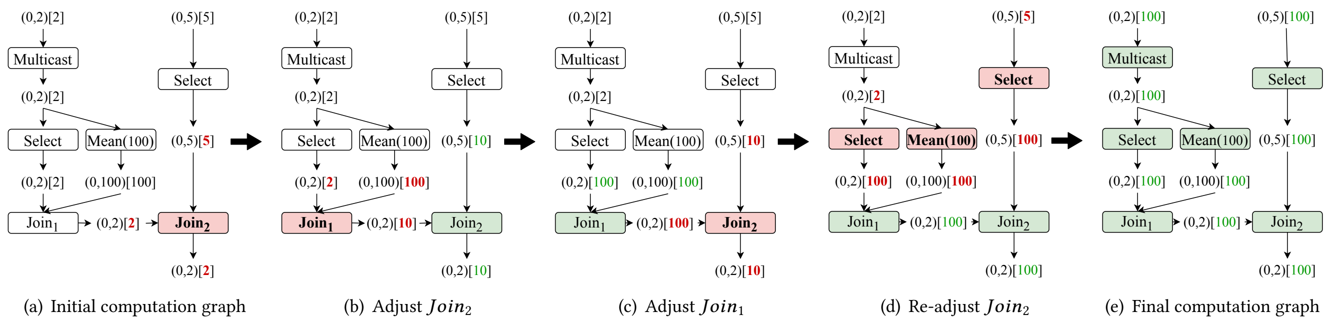


Figure 6: Locality tracing procedure on the example query

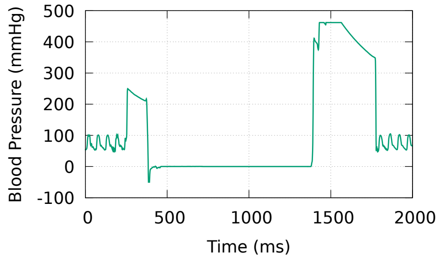


Figure 7: Line zero artifact in arterial blood pressure (ABP) signal

6.1 Temporal Query Language Extensions

Apart from performance benefits, LifeStream also provides several additional important features through the query language extensions. We introduce a generic *Transform* primitive operation which lets users write arbitrary transformations on a fixed interval of events. This operation helps users to easily integrate third-party libraries into the stream processing pipeline.

We also extend the *Where* query primitive operation to support shape-based querying. As shown in Figure 4, users can input arbitrary artifacts or patterns they want to detect in the stream as a list of signal values. We extend the dynamic time warping (DTW) algorithm [46] for a streaming scenario called restricted dynamic time warping (RDTW) algorithm for pattern matching in a data stream.

Figure 7 shows an artifact commonly found in the arterial blood pressure (ABP) signal which occurs when the pressure sensors attached to patients are calibrated against atmospheric pressure. In such cases, the ABP signal values produced by the monitors would show this characteristic shape because of the distortion in the measurements. There are several other artifacts found in different signals and data scientists generally want to remove such regions from the physiological data as this could negatively affect their data analysis process. In LifeStream, data scientists can use the extended *Where* query primitive to filter out these artifacts from the stream by providing a representative shape as input to the query in the form of a sequence of signal values. LifeStream subsequently uses the RDTW algorithm do the pattern matching in the input stream. We measure the pattern matching accuracy of RDTW algorithm over a month of ABP signal data from a single device containing 49 line zeroing artifacts. RDTW achieves 0% false negatives and 0.2% false positives. This shows, LifeStream can accurately detect such characteristic shapes in the data streams.

6.2 FWindow Fragmentation

Since FWindow is a continuous interval with a fixed duration, one potential issue on using FWindows for accessing stream data is that the memory might get fragmented if there are small gaps present in the input data. This might lead to low memory utilization, which, in turn, could lead to low query performance. However, as shown in Figure 2 in the main paper, most of the discontinuities in the raw physiological data are generally concentrated on specific time periods rather than being randomly scattered throughout the stream. Hence most parts of the stream has continuous sequence of data. However, in the occurrence of small gaps, we handle them by setting the bitvector field in the corresponding position in the FWindow.

Another possible cause of fragmentation is when *Where* query is used to filter out certain events in the stream, based on an arbitrary predicate defined by the programmer. From our experience building pipelines using LifeStream, *Where* operation is the least commonly used primitive operation. Even when it is used, it is mostly to filter out a large continuous portion of the stream (e.g., removing noisy regions or artifacts from the data). Therefore, the chances for FWindow fragmentation are minimum.

In the use cases that we evaluate in this work (see Section 8), we observe the degree of the FWindow fragmentation to be at most 0.3% which is too small to make any significant negative effect on query performance.

6.3 Stateful Temporal Operators

In certain cases, in addition to the input FWindow(s), some temporal operators need to carry a state throughout the query execution (e.g., rolling aggregate operations or temporal *Join* operation on streams with arbitrary duration). To handle such cases, LifeStream allows operators to create constant size states during initialization, in order to preserve bounded space complexity property and ensure that there is no dynamic memory allocation during operator execution.

For example, Figure 8 shows an example of stateful *Join* operation where a constant duration stream (*Left*) is inner joined with

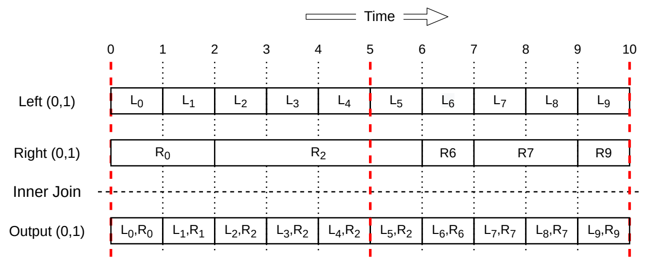

Time
0
1
2
3
4
5
6
7
8
9
10
Left (0,1)
L0 L1 L2 L3 L4 L5 L6 L7 L8 L9
Right (0,1)
R0 R2 R6 R7 R9
Inner Join
Output (0,1)
L0,R0 L1,R1 L2,R2 L3,R2 L4,R2 L5,R2 L6,R6 L7,R7 L8,R7 L9,R9

Figure 8: Stateful Join operation

Operation	Libraries	Description
Normalize	Scikit-learn	Normalize a window of signal values using standard scores.
PassFilter	SciPy	Filter frequencies using finite impulse response [40].
FillConst	NumPy	Fill gaps smaller than the given window size with a constant value.
FillMean	NumPy	Fill gaps smaller than the given window size with the mean of the values in the window.
Resample	SciPy	Up/Down sample the signal using linear interpolation [47].

Table 2: Operation benchmarks and their descriptions

another stream with arbitrary duration (*Right*). The red dotted lines represent the FWindow boundaries. As shown in the figure, event R_2 in the *Right* stream has overlapping events in the *Left* stream in both FWindows. In such cases, in order for LifeStream to produce the output event (L_5, R_2) correctly, the *Inner Join* operation needs to save the event R_2 in its state before moving to the second FWindow. However, the constant frequency property of these streams ensures that there can only be at most one such event in a stream that can cross the interval boundary of the FWindow at any time. Therefore, the state required for temporal Join is always constant size and stateful *Join* operators does not violate the bounded space complexity property.

7 METHODOLOGY

Benchmarks: We evaluate LifeStream on three categories of benchmarks. (i) *Primitive benchmarks*: This benchmark include several primitive temporal operations like *Select*, *Where*, *Aggregate* and temporal *Inner Join*. (ii) *Operation benchmarks*: This include five operations described in Table 2 that we find commonly used by the data analysts to process physiological data. (iii) *End-to-end applications*: We build the data processing pipeline described in Section 2 and Figure 3 using the operations in Table 2.

Datasets: Physiological waveform data we use contain signal events with a 64-bit timestamp and 32-bit floating point value. For the experiments, we use two dataset types. (i) *Synthetic data*: 1000 Hz waveform data generated for 1000 minutes with randomly selected signal values. This dataset contain continuous stream of signal events and no gaps. (ii) *Real data*: A private dataset from a well-known hospital we collaborate with, containing physiological waveform data collected from 6100 patients over the past five years. The dataset contains more than 830,000 patient-hours of data and 250 different signal types. However, for our experiments, we only use ABP and ECG signals sampled at their default rate 125 Hz and 500 Hz respectively [15].

Baselines: We compare the performance of LifeStream with two baselines. (i) *Microsoft Trill*, a state-of-the-art temporal query processing engine specially optimized for single machine performance. (ii) *Numeric libraries (NumLib)* like SciPy, NumPy and Scikit-learn with hand-optimized implementations for data processing operations. For end-to-end benchmarking, we implement the numerical library-based data processing pipeline in Python. In order to make fair performance comparisons, we tried to minimize computations done on native Python as much as possible by offloading the heavy processing to the numerical library functions. However, operations like temporal *Inner Join* required pure Python implementations.

Metrics: We use the total execution time from a single core on a fixed input data size as the primary comparison metric for performance on primitive benchmarks, operation benchmarks, and end-to-end benchmark. For scalability experiments, we use the throughput obtained on multiple cores/machines. Throughput is measured as the average number of signal events processed per unit time. Both execution time and throughput reported is the average of measurements from 10 trials. The standard deviation of the measurements are observed to be less than 1%. For the sensitivity study on cache utilization, we use total number of last level cache (LLC) misses (median over 5 trials) on fixed workload as a comparison metric measured using Intel vTune profiler v2020 [22].

For all the experiments except scalability, we use 8-core (16 hyper-threaded) Intel Xeon CPU E5-2660 machine running at 2.2 GHz, with 16 GB RAM, and running 64-bit Ubuntu 20.04. For scalability experiments, we use up to 16 AWS EC2 m5a.8xlarge [4] machines each with 32 cores and 128 GB DRAM. We use a window size of 1 minute for all the benchmarks unless otherwise specified.

8 EVALUATION

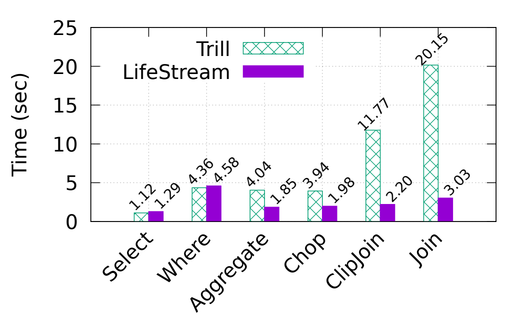
We evaluate LifeStream to answer the following questions:

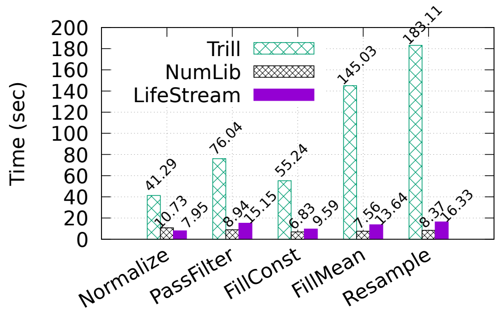
(1) How does the performance of LifeStream compare to state-of-the-art streaming engines and numerical libraries?
(2) Can LifeStream accelerate end-to-end performance of physiological waveform data processing pipelines?
(3) How beneficial are the proposed optimizations?
(4) How well does LifeStream scale on multiple machines?

8.1 Primitive Benchmarks

We compare the performance of LifeStream against *Trill* on 7 most commonly used primitive temporal operations using the synthetic dataset for these experiments. Figure 9(a) shows the execution time taken by both *Trill* and *LifeStream* and based on the results, we make the following two major conclusions.

First, on simple operations such as *Select* and *Where*, performance of LifeStream is within 20% of that of Trill. This shows that LifeStream is not adding any significant overhead over already highly optimized operations in Trill. Second, we observe that





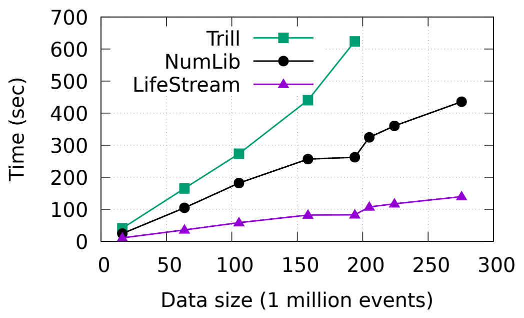


Figure 9: (a) Primitive benchmarks, (b) Operation benchmarks, (c) End-to-end applications

LifeStream shows much higher performance benefits as the operations become more complex. Operators such as *Aggregate*, *Chop*, *ClipJoin*, and *Join* are respectively 2.17×, 1.98×, 5.34×, and 6.65× faster than its Trill counterparts.

We attribute LifeStream's high performance on primitive operations to the introduction of FWindow. FWindow greatly simplifies the operator implementations and eliminates the need for using complex data structures such as hashmaps in temporal *Join* like Trill. Moreover, since the timestamps of the events and their index positions are aligned, operators implemented in LifeStream can calculate the sync time of each event from its index position without doing any memory accesses. Such implementation-level optimization makes LifeStream efficient even at the primitive operation-level.

8.2 Operation Benchmarks

To evaluate the performance of common physiological data transformations, we implement the operations listed in Table 2 on LifeStream by writing queries using the temporal operators and compare their performance against the similar queries written in Trill and the hand-tuned implementations available in the corresponding numerical libraries specified in Table 2. We conduct this experiment on a 500 Hz ECG signal from the real dataset containing 126M events. Figure 9(b) shows the execution time of *Trill*, numerical libraries (*NumLib*), and *LifeStream* on each benchmark. We make three major conclusions from this figure.

First, across all the operations, LifeStream is shown to be 5 – 11.21× faster than Trill. This shows the effectiveness of the optimizations implemented in LifeStream. Second, LifeStream also exhibits comparable performance against highly optimized implementations available in the numerical libraries (within 50% performance of the popular numerical libraries we evaluate). Third, in certain cases such as a very commonly used *Normalize* operation, LifeStream even surpasses the hand-tuned performance provided by Scikit-learn library by 1.35×. This asserts our claim that LifeStream provides ease of programming of a temporal query language without sacrificing performance.

8.3 End-to-end Applications

In order to evaluate whether LifeStream can improve the end-to-end performance on data processing, we build the pipeline shown in Figure 3 over LifeStream, Trill, and in Python using numerical libraries. The data pipeline in all three implementations process 500 Hz ECG and 125 Hz ABP signals from the real dataset stored in CSV format and produces a joined signal stream after running a series of transformations on the data shown in Figure 3. The dataset contains two weeks of data from a single monitoring device with 275M signal events. Figure 9(c) shows the end-to-end execution time by varying the dataset size for all three implementations. We make two major observations from this figure.

First, LifeStream outperforms both Trill and numerical library-based implementation by 7.5× and 3.2× respectively. This reinstates that, even though individual operators in the numerical libraries can exhibit high performance when executed in isolation (as we show in Section 3), it does not necessarily translate into the best end-to-end performance due to data conversion overhead and lack of end-to-end optimizations [36].

Second, in the case of Trill, as the size of the dataset increases, the execution time rises rapidly, and Trill goes out of memory at 200*M* events. Our investigation reveals that this happens because of Trill's implementation issue in the *Join* operation. Trill expects both left and right streams of the Join operation to progress at a similar pace. However, if the two streams diverge considerably, the internal memory allocated for the Join operator keeps accumulating until the available memory is exhausted. Since physiological data contains high degree of discontinuity, it is very common for such divergence to occur during query processing. LifeStream uses targeted query processing optimization (described in Section 5.3) to skip over such non-overlapping parts of the input data.

8.4 Sensitivity Studies

In this section, we conduct experiments to analyze the effectiveness of the optimizations applied in LifeStream.

Batch size	10^5	10^6	10^7
Trill	2.43	4.11	6.73
LifeStream	0.79	0.82	0.96

Table 3: The number of last level cache misses (in millions)

8.4.1 Cache Utilization. In order to analyze how well LifeStream utilizes the cache compared to Trill using optimizations such as locality tracing, we conduct an experiment to measure the last level cache (LLC) misses of both engines on one of the most commonly used operation *Normalize.* In order to avoid the influence of data discontinuities in the measurements, we use synthetic dataset for this experiment. We use Intel vTune [22] to measure the cache misses during the query execution over a constant size input dataset.

Table 3 shows the LLC misses in both Trill and LifeStream on three different batch sizes. For a batch size of 10^5, LifeStream is experiencing 3× lower cache misses as that of Trill. As the batch size increases, the number of cache misses in Trill goes up significantly while LifeStream's miss rate stays relatively constant. The reason being, as we describe in Section 5.2, Trill does not preserve cross-operation locality. The consequence of this severe limitation becomes more pronounced on larger batch sizes. LifeStream, on the other hand, preserves the end-to-end locality of the query using locality tracing irrespective of the batch size.

8.4.2 Targeted Query Processing. In this section, we analyze the effectiveness of targeted query processing while running large data processing pipelines. To perform this analysis, we pick the ECG and ABP signals of several different dates from the real dataset with varying degree of overlapping events between them. Figure 10(a) shows the relative performance speedup of LifeStream over Trill, measured with respect to the percentage of overlapping events in these data subsets. We observe that, as expected, the speedup is smaller when there is near perfect overlap, which is about 7×. The speedup starts to increase as the degree of overlap decreases, because LifeStream can skip increasing number of redundant computations compared to Trill. For example, a day with 10% overlap in ECG and ABP leads to about 38× speedup over Trill, which is about 5× higher than the base performance of LifeStream.

8.4.3 Window size. In this section, we conduct a sensitivity study on LifeStream to measure the effect of window size on its performance. Figure 10(b) shows the execution time of Trill and LifeStream on the end-to-end benchmark over the synthetic dataset with window size varying from 1 minute to 1 hour. The results suggest that LifeStream can maintain its performance benefits compared to Trill even on larger windows.

8.5 Scalability

Physiological datasets generally contain signals collected from thousands of patients, and the data processing pipelines usually process data from different patients separately. That means the data processing can be parallelized across multiple patients. LifeStream takes advantage of this data parallel nature of the physiological dataset to scale up the computation to both (i) multiple cores within a machine and (ii) multiple machines.

We evaluate the scalability of LifeStream and compare it against Trill and numerical libraries on a single AWS m5a.8xlarge [4] machine with 32 cores and 128 GB DRAM on the end-to-end benchmark using synthetic dataset. Figure 10(c) shows total number of signal events processed per second against the number of parallel threads of data pipeline execution. We observe that LifeStream provides up to 6.02× better scalability than Trill and 1.90× better than numerical libraries. This shows, LifeStream can maintain its performance benefits on multi-core parallel data pipeline execution compared to Trill and provide better parallel processing capabilities than numerical library-based approach.

We also observe that Trill goes out of memory and crashes when we run experiments with more than 12 parallel threads. LifeStream, on the other hand, is much more memory-efficient and can scale up to 32 parallel threads as the memory required for the intermediate results are preallocated are reused throughout query execution. Numerical library-based implementation is observed to scale up to 48 threads, however, the performance gets saturated after 24 threads and exhibits a peak performance that is 44% lower than that of LifeStream.

We also measure the scalability of LifeStream on a multi-machine setup using up to 16 Amazon EC2 m5a.8xlarge [4] machines with each running 12, 24 and 32 parallel threads respectively for Trill, numerical libraries and LifeStream, since these thread counts are observed to provide the peak performance from multi-core experiment. Figure 10(d) shows the throughput measured against the number of machines. On 16 machines, LifeStream can process 473.66 million events per second which is 8.38× higher than the peak performance of Trill and 1.73× higher than that of numerical libraries. This shows that LifeStream can maintain the performance benefits at large scale through efficient data parallel processing.

9 RELATED WORK

Several major solutions were proposed in the past to address large scale stream processing demands [3, 31, 44, 48]. Unfortunately, these solutions fail to satisfy either in terms of (i) programmability or (ii) performance (or even both) due to the unique requirements of physiological data processing. In this work, we show that LifeStream finds a sweet spot between both programmability and performance.

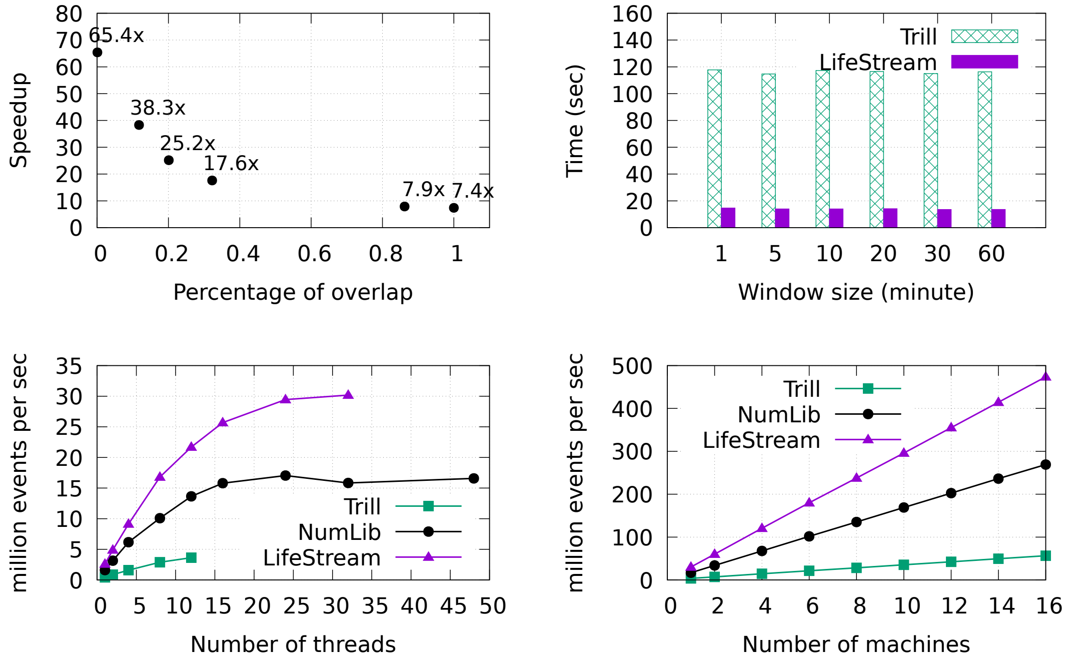


Figure 10: (a) Targeted query processing, (b) Varying window size, (c) Multi-core scaling, (d) Multi-machine scaling

Below we provide the detailed comparison of LifeStream against key prior works based on these two aspects.

Stream processing. Popular stream processing engines such as Apache Spark stream [5], Storm [44], Flink [9], and Beam [3] provide simple declarative programming interfaces for writing complex data processing pipelines. However, most of them fail to support several features essential for physiological data processing. For instance, Storm [44] does not have any implicit notion of event time or windowing. Spark streaming, on the other hand, does not have support for millisecond precision event time as required for many signals in physiological data and lacks several useful temporal primitive operations that are necessary for writing queries on physiological data. Additionally, as we show in Section 3, these solutions are primarily designed for distributed setup and trade-off single machine performance to favor scalability and fault-tolerance.

Trill [10] is the closest streaming engine we could find that provides rich support for temporal operations, high precision event time, and flexible windowing, as well as $1-2$ orders of magnitude higher single machine performance compared to distributed streaming engines. Unlike Trill, LifeStream takes advantage of the end-to-end locality of the entire data pipeline using *locality tracing*. Additionally, LifeStream employs optimizations like *static memory preallocation* and *targeted query processing* to minimize runtime memory allocation/deallocation overhead and pruning redundant computations. These optimizations, as we have shown in Section 8, make LifeStream significantly faster than strong baselines like Trill.

Following the footsteps of Trill, there are two other streaming engines recently proposed, StreamBox [27] and StreamBox-HBM [26] that focus on improving the single machine performance. Both these designs, however, provide a very low-level and generic programming abstraction and lack a rich high-level temporal language support. Additionally, StreamBox-HBM was designed specifically for machines with high bandwidth memory (HBM) which are both very rare[3] and expensive. Compared to these two engines, LifeStream provides much simpler programming interface with high performance on commodity hardware.

Numerical libraries There has been some recent studies [35, 37] to improve the performance of numerical library-based data processing pipelines. Weld [35] proposed a compiler-based approach to optimizes across disjoint libraries and functions with the help of an intermediate representation (IR). The followup work, called Split Annotations [37], eliminates the need for an IR and reimplementation of library functions while potentially providing similar end-to-end performance benefits.

Even though these solutions can improve the end-to-end performance of numerical library-based data processing pipelines by some margin, the lack of temporal logic support and unified API specification still make such approaches less desirable in terms of ease of programming and maintainability. We believe high-level temporal query language provided by LifeStream is more systematic and appropriate approach for doing data processing on physiological data.

[3]In fact, Intel's Knight Landing architecture used in StreamBox-HBM is discontinued now [20].

10 CONCLUSION

In this paper, we showcase the limitations of modern streaming engines and numerical libraries in building complex physiological data processing pipelines in terms of their ease of programming, maintainability, and performance. We subsequently propose LifeStream, which provides a simple and flexible temporal query language as the programming interface, and exploits the constant frequency nature of the physiological data to provide high performance. We propose three key optimizations in LifeStream, namely, (i) locality tracing for improving end-to-end cache utilization of the data pipeline, (ii) memory footprint estimation for minimizing runtime memory allocation and deallocation overhead, and (iii) targeted query processing for pruning redundant computation. We conduct experiments and evaluations on real datasets and use cases, and demonstrate that LifeStream outperforms state-of-the-art streaming engines by as much as 7.5× and numerical library-based approaches by as much as 3.2× on the end-to-end data processing performance.

REFERENCES

[1] Karim Abouelmehdi, Abderrahim Beni-Hessane, and Hayat Khaloufi. 2018. Big healthcare data: preserving security and privacy. *Journal of Big Data* 5, 1 (09 Jan 2018), 1. https://doi.org/10.1186/s40537-017-0110-7

[2] George Adam, Ladislav Rampášek, Zhaleh Safikhani, Petr Smirnov, Benjamin Haibe-Kains, and Anna Goldenberg. 2020. Machine learning approaches to drug response prediction: challenges and recent progress. *npj Precision Oncology* 4, 1 (15 Jun 2020), 19. https://doi.org/10.1038/s41698-020-0122-1

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8 (2015), 1792–1803.

[4] Amazon. [n.d.]. *EC2 m5a.8xlarge*. https://aws.amazon.com/ec2/instance-types/m5/

[5] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Madison, Wisconsin) *(PODS '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/543613.543615

[7] Valentina Baljak, Adis Ljubovic, Jonathan Michel, Mason Montgomery, and Richard Salaway. 2018. A scalable realtime analytics pipeline and storage architecture for physiological monitoring big data. *Smart Health* 9 (2018), 275–286.

[8] Jan N. Basile. 2002. Systolic blood pressure. *BMJ (Clinical research ed.)* 325, 7370 (26 Oct 2002), 917–918. https://doi.org/10.1136/bmj.325.7370.917 12399325[pmid].

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[10] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401–412. https://doi.org/10.14778/2735496.2735503

[11] Calvin J. Chiew, Nan Liu, Takashi Tagami, Ting Hway Wong, Zhi Xiong Koh, and Marcus E. H. Ong. 2019. Heart rate variability based machine learning models for risk prediction of suspected sepsis patients in the emergency department. *Medicine* 98, 6 (Feb 2019), e14197–e14197. https://doi.org/10.1097/MD.0000000000014197 30732136[pmid].

[12] Ambika Choudhury and Deepak Gupta. 2019. *A Survey on Medical Diagnosis of Diabetes Using Machine Learning Techniques: IC3 2018.* 67–78. https://doi.org/10.1007/978-981-13-1280-9_6

[13] David Cournapeau. [n.d.]. *Scikit-learn.* https://scikit-learn.org/

[14] C. M. Furse, R. Harrison, and F. Solzbacher. 2007. Recent Advances in BioMedical Telemetry. In *2007 International Conference on Electromagnetics in Advanced Applications*. 1026–1027.

[15] Andrew J Goodwin, Danny Eytan, Robert W Greer, Mjaye Mazwi, Anirudh Thommandram, Sebastian D Goodfellow, Azadeh Assadi, Anusha Jegatheeswaran, and Peter C Laussen. 2020. A practical approach to storage and retrieval of high-frequency physiological signals. *Physiological Measurement* 41, 3 (apr 2020), 035008. https://doi.org/10.1088/1361-6579/ab7cb5

[16] Kyeonghye Guk, Gaon Han, Jaewoo Lim, Keunwon Jeong, Taejoon Kang, Eun-Kyung Lim, and Juyeon Jung. 2019. Evolution of Wearable Devices with Real-Time Disease Monitoring for Personalized Healthcare. *Nanomaterials (Basel, Switzerland)* 9, 6 (29 May 2019), 813. https://doi.org/10.3390/nano9060813 31146479[pmid].

[17] Chuanxiong Guo. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM* (sigcomm ed.). ACM. https://www.microsoft.com/en-us/research/publication/pingmesh-large-scale-system-data-center-network-latency-measurement-analysis/

[18] HHS. [n.d.]. Your Rights Under HIPAA. https://www.hhs.gov/hipaa/for-individuals/guidance-materials-for-consumers/index.html

[19] Melanie Hilario, Alexandros Kalousis, Markus Müller, and Christian Pellegrini. 2003. Machine learning approaches to lung cancer prediction from mass spectra. *PROTEOMICS* 3, 9 (2003), 1716–1719. https://doi.org/10.1002/pmic.200300523 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/pmic.200300523

[20] HPCWire. [n.d.]. Requiem for a Phi: Knights Landing Discontinued. https://www.hpcwire.com/2018/07/25/end-of-the-road-for-knights-landing-phi/

[21] A. F. Hussein, N. A. kumar, M. Burbano-Fernandez, G. Ramírez-González, E. Abdulhay, and V. H. C. De Albuquerque. 2018. An Automated Remote Cloud-Based Heart Rate Variability Monitoring System. *IEEE Access* 6 (2018), 77055–77064.

[22] Intel. [n.d.]. *Intel vTune profiler.* https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html

[23] Alistair E.W. Johnson, Tom J. Pollard, Lu Shen, Li wei H. Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G. Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific Data* 3, 1 (24 May 2016), 160035. https://doi.org/10.1038/sdata.2016.35

[24] Jared L. Katzman, Uri Shaham, Alexander Cloninger, Jonathan Bates, Tingting Jiang, and Yuval Kluger. 2018. DeepSurv: personalized treatment recommender system using a Cox proportional hazards deep neural network. *BMC Medical Research Methodology* 18, 1 (26 Feb 2018), 24. https://doi.org/10.1186/s12874-018-0482-1

[25] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[26] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2019. StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 167–181. https://doi.org/10.1145/3297858.3304031

[27] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 617–629. https://www.usenix.org/conference/atc17/technical-sessions/presentation/miao

[28] Microsoft. [n.d.]. .NET Core. https://dotnet.microsoft.com/

[29] Microsoft. [n.d.]. .NET Libraries. https://en.wikipedia.org/wiki/List_of_numerical_libraries#.NET_Framework_languages_C.23.2C_F.23.2C_VB.NET_and_PowerShell

[30] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley. 2018. Deep learning for healthcare: review, opportunities and challenges. *Brief. Bioinformatics* 19, 6 (11 2018), 1236–1246.

[31] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[32] Silvia Nittel. 2015. Real-Time Sensor Data Streams. *SIGSPATIAL Special* 7, 2 (Sept. 2015), 22–28. https://doi.org/10.1145/2826686.2826691

[33] Travis Oliphant. [n.d.]. *NumPy.* https://numpy.org/

[34] Travis Oliphant, Pearu Peterson, and Eric Jones. [n.d.]. *SciPy.* https://www.scipy.org/

[35] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (May 2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[36] Shoumik Palkar, J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, Saman P. Amarasinghe, M. Zaharia, and Stanford InfoLab. 2016. Weld : A Common Runtime for High Performance Data Analytics.

[37] Shoumik Palkar and Matei Zaharia. 2019. Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 291–305. https://doi.org/10.1145/3341301.3359652

[38] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. 2018. TerseCades: Efficient Data Compression in Stream Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 307–320. https://www.usenix.org/conference/atc18/presentation/pekhimenko

[39] Phillips. [n.d.]. *Phillips Patient Monitoring*. https://www.usa.philips.com/healthcare/solutions/patient-monitoring

[40] Donald Reay. 2015. *Finite Impulse Response Filters*. 97–162. https://doi.org/10.1002/9781119078227.ch3

[41] Jonathan G. Richens., Ciarán M. Lee, and Saurabh Johri. 2020. Improving the accuracy of medical diagnosis with causal machine learning. *Nature Communications* 11, 1 (11 Aug 2020), 3923. https://doi.org/10.1038/s41467-020-17419-7

[42] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2503–2511.

[43] Sana Tonekaboni, Mjaye Mazwi, Peter Laussen, Danny Eytan, Robert Greer, Sebastian Goodfellow, Andrew Goodwin, Michael Brudno, and Anna Goldenberg. 2018. Prediction of Cardiac Arrest from Physiological Signals in the Pediatric ICU.

[44] Jan Sipke van der Veen, Bram van der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J. Meijer. 2015. Dynamically Scaling Apache Storm for the Analysis of Streaming Data. In *Proceedings of the 2015 IEEE First International Conference on Big Data Computing Service and Applications (BIGDATASERVICE '15)*. IEEE Computer Society, USA, 154–161. https://doi.org/10.1109/BigDataService.2015.56

[45] Jenna Wiens and Erica S Shenoy. 2017. Machine Learning for Healthcare: On the Verge of a Major Shift in Healthcare Epidemiology. *Clinical Infectious Diseases* 66, 1 (08 2017), 149–153. https://doi.org/10.1093/cid/cix731 arXiv:https://academic.oup.com/cid/article-pdf/66/1/149/24265881/cix731.pdf

[46] Wikipedia. [n.d.]. Dynamic Time Warping. https://en.wikipedia.org/wiki/Dynamic_time_warping

[47] Wikipedia. [n.d.]. *Linear interpolation*. https://en.wikipedia.org/wiki/Linear_interpolation

[48] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664