Enabling Signal Processing over Data Streams

Milos Nikolic*
University of Oxford
milos.nikolic@cs.ox.ac.uk

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Jonathan Goldstein
Microsoft Research
jongold@microsoft.com

ABSTRACT

Internet of Things applications analyze the data coming from large networks of sensor devices using relational and signal processing operations and running the same query logic over groups of sensor signals. To support such increasingly important scenarios, many data management systems integrate with numerical frameworks like R. Such solutions, however, incur significant performance penalties as relational data processing engines and numerical tools operate on fundamentally different data models with expensive intercommunication mechanisms. In addition, none of these solutions supports efficient real-time and incremental analysis.

In this paper, we advocate a deep integration of signal processing operations and general-purpose query processors. We aim to reconcile the disparate data models and provide a common query language that allows users to seamlessly interleave tempo-relational and signal operations for both online and offline processing. Our approach is extensible and offers frameworks for quick and easy integration of user-defined operations while supporting incremental computation. Our system that deeply integrates relational and signal operations, called TRILLDSP, achieves up to two orders of magnitude better performance than popular loosely-coupled data management systems on grouped signal processing workflows.

1. INTRODUCTION

An increasing proportion of today's data comes from networks of sensors and devices, commonly known as the Internet of Things (IoT). IoT applications typically run the same logic over a large collection of sensor devices using queries that combine relational and signal processing operations. Data analysts use relational operators, for example, to group signals by different sources or join signals with historical and reference data. They also use domain-specific algorithms such as Fast Fourier Transform (FFT) to do spectral analysis, interpolation to handle missing values, or digital filters to recover noisy signals. Reconciling these two seemingly disparate worlds, especially for real-time analysis, is challenging.

*Work performed during internship at Microsoft Research and later on at EPFL where his work was supported by ERC Grant 279804.

The database community has recognized the need for a tighter integration of data management systems and domain-specific algorithms. Numerical computing environments like MATLAB and R provide efficient domain-specific algorithms but remain unsuitable for general-purpose processing involving relational operations such as joins, filtering, or group-by aggregation. To enable the use of specialized routines in complex data processing, increasingly many data management systems integrate with numerical frameworks, R in particular: MonetDB and SQL Server support queries that can invoke R code; SciDB [34, 17], Spark [38], and Pivotal (Greenplum) [37] provide R packages that allow the user to interact with these systems directly from R. All these integrations benefit from re-using unmodified MATLAB/R scripts.

However, the existing integration mechanisms between database systems and numerical frameworks are suboptimal performance-wise as they treat both sides as independent systems. Such *loose system coupling* comes with significant processing overheads – for instance, executing R programs requires exporting data from the database, converting into R format, running R scripts, converting back into a relational format, and importing into the database. Sending data back and forth between the systems might dominate the execution time, for example when running (sub)linear R operators; it also increases the latency of processing, which makes this approach particularly unsuitable for real-time processing.

In this paper, we advocate a *deep integration* of digital signal processing (DSP) operations with a general-purpose query processor. Our approach aims to bring signal processing closer to data, not the other way around, and eliminate the need for expensive communication with external numerical tools. Integrating DSP operations into a query engine empowers users to express end-to-end workflows more succinctly, inside one system and using one language.

This tight integration poses several requirements and challenges:

1. *Query and data model reconciliation.* General-purpose query engines and numerical tools use different query and data models. The former support relational and streaming queries over relational or tempo-relational data; the latter support domain-specific, mostly offline, computations on arrays. The key challenge is how to seamlessly unify these disparate models instead of layering them on top of each other, and yet provide experts from both relational and signal processing worlds with familiar abstractions.

2. *Performance.* High performance is always a critical requirement for analytics. The deep integration approach brings more expressiveness to the query language but also carries a risk of throwing the baby out with the bathwater, that is, completely giving up on performance. Previous results suggested that simulating array computations on top of a relational database can yield orders of magnitude worse performance, which has motivated the development of array-based database systems [34, 17]. In order to be

TRILLDSP

```
1 var q = stream
2   .Map(s => s.Select(e => e.Value), e => e.SensorId)
3   .Reduce(s => s.Sample(100, 0, p => p.FirstOrder())
4     .Window(512, 256, true,
5       w => w.FFT().Select(a => f(a)).InverseFFT(),
6       a => a.Sum()));
```

R

```
1  groups <- lapply(split(x,x[,1]),
2      function(y) matrix(y,ncol=3))    # group by id
3  z <- vector(``list'', length(groups))
4  for (x in groups) {
5    t <- x[, 2]; v <- x[, 3]           # times and values
6    qt <- seq(t[1], t[length(t)], by = 100) # probes
7    y <- interp1(t, v, qt, method = 'linear')
8    frames <- window(y, 512, 256)      # create frames
9    Y <- mvfft(frames)
10   Y2 <- f(Y)
11   y2 <- mvfft(Y2, inverse = TRUE)
12   z[[i]] <- unwindow(y2, 512, 256)   # merge frames
13 }
```

SPARKR

```
1  grouped <- groupByKey(flatMap(rdd, function(x) {
2    groups <- lapply(split(x,x[,1]),     # group by id
3        function(y) matrix(y,ncol=3))
4    lapply(groups,                       # key-value pairs
5      function(y) list(y[1, 1], y[, 2:3]))
6  }, cores)
7  sorted <- lapply(grouped, function(x) {
8    merged <- matrix(rbind(x[[2]]), ncol = 2)
9    merged[order(merged[, 1]),]          # sort by time
10 })
11 result <- lapply(sorted, function(x) {
12   t <- x[, 1]; v <- x[, 2]             # times and values
13   qt <- seq(t[1], t[length(t)], by = 100) # probes
14   y <- interp1(t, v, qt, method = 'linear')
15   frames <- window(y, 512, 256)        # create frames
16   Y <- mvfft(frames)
17   Y2 <- f(Y)
18   y2 <- mvfft(Y2, inverse = TRUE)
19   unwindow(y2, 512, 256)               # merge frames
20 })
```

Figure 1: Spectrum analysis in TrillDSP, R, and SparkR

welcomed by data scientists and DSP experts, a deeply integrated system should preserve the performance of existing relational operators while being competitive with MATLAB and R on pure domain-specific tasks. For workflows mixing relational and DSP processing, a tightly-coupled system should capitalize the potential of much better performance than the existing loosely-coupled alternatives. For achieving this goal, the key challenge is how to efficiently integrate DSP operators inside a query processor.

3. *Extensibility.* A query processor with DSP support should allow domain experts and practitioners to implement custom operators in a way that feels natural to them – by writing algorithms against arrays without worrying about the format of the underlying data. Exposing arrays to operator writers enables easy integration of existing highly optimized DSP algorithms, for instance, implementations using SIMD instructions. The system should seamlessly integrate new operators with the existing query language.

4. *Online and incremental computation.* Stream processors loosely coupled with MATLAB or R cannot incrementalize signal processing tasks that operate over hopping (overlapping) windows of data. The stateless nature of the DSP routines in MATLAB and R leaves no choice for stream engines but to redundantly compute over overlapping subsets of data. On the other hand, deep integration opens up the opportunity for incremental DSP computation through the use of stateful operators. The operator writer should decide on which state to maintain and how to incrementalize computation using deltas provided by the system.

We have built a query engine, named TRILLDSP, that deeply integrates relational and signal processing while satisfying the above requirements: (1) it provides a unified query language for processing tempo-relational and signal data; (2) its performance is comparable to numerical tools like MATLAB and R and orders of magnitude better than existing loosely-coupled systems; (3) it provides mechanisms for defining custom DSP operators and their integration with the query language; (4) it supports incremental computation in both offline and online analysis. The following example compares TRILLDSP against other commonly used systems.

1.1 Example: Spectral Analysis of Signals

An IoT application receives a stream of temperature readings from different sensors in time order. Each reading has the same format `<SensorId,Time,Value>`. The application runs the same query logic on every signal coming from a different source. The query consists of several processing stages, which are discussed next from the viewpoint of three different systems: R, SPARKR, and TRILLDSP. Table 1 shows their relevant code excerpts.

- *Stage 1: Grouping* The DSP routines in R cannot process multiple time-intertwined signals at once. The initial phase in R and SPARKR has to disentangle readings by their source (`SensorId`), while preserving the time order inside each group. The grouping operations in R (lines 1-2) and SPARKR (lines 1-10) are CPU- and memory-intensive tasks that involve copying the entire input. SPARKR consolidates the input data in parallel but requires local sorting of each group to restore the time order (line 9).

  TRILLDSP natively supports grouped processing through its data model and group-aware operators that internally maintain the state of each group. Grouping in TRILLDSP (line 2) is an in-place `Map` operation that associates a group identifier to each event to enable grouped processing in the downstream operators. Avoiding data copying brings orders of magnitude better performance than R and SPARKR, especially with many groups.

The workflow continues with processing each group either sequentially (R) or in parallel (SPARKR and TRILLDSP).

- *Stage 2: Interpolation* The values within one group might appear at irregular time intervals due to network delays or never appear due to message losses. To leverage DSP algorithms that mostly operate on equally-spaced sequences, the next stage transforms each group into a uniformly-sampled signal with a given sampling period and offset using first-order (linear) interpolation. TRILLDSP hides from the user the burden of explicit timestamp management, in contrast to R (line 6) and SPARKR (line 13).

  Sampling and interpolation in TRILLDSP is a group-aware operator with full online support (line 3). The operator's ability to simultaneously process intertwined signals can lead up to 146x better performance than R and SPARKR, seen in the experiments.

The remaining steps describe the fundamental technique of windowing in DSP [33]: (1) decompose the signal into simple components, (2) process each of the components in some useful way, and (3) recombine the processed components into the final signal.

- *Stage 3: Windowing* Most digital signal processing algorithms operate over windows of data defined by two parameters: the window size and the hop size. Using R (or SPARKR) misses the opportunity for incremental computation over hopping (overlapping) windows. Furthermore, invoking R routines for every window accumulates their startup costs. In offline analysis, DSP experts often choose to copy windows out of the array and stack them into a matrix for batch processing. The `window()` function from Table 1 forms such a matrix (details omitted for clarity) using twice as much memory for the given arguments.

  The `Window` operator in TRILLDSP allows users to express a hopping window computation as a series of transformations of fixed-size arrays. One such pipeline transforms all input windows, thus amortizing the startup overhead, while supporting incremental computation. Based on the window specification, the operator manages the data on behalf of users using circular buffers to avoid any redundancy in the input.

- *Stage 4: Spectral Analysis* The processing pipeline starts with the Fast Fourier Transform (FFT) that computes the frequency representation of 512-sample windows at each hopping point. A user-defined function f modifies the computed spectrum (e.g., retains top-k spectrum values with the highest magnitudes, zeros out others) before invoking the inverse FFT. The output stream has complex 512-size arrays at each hopping point.

- *Stage 5: Unwindowing* To restore the signal form, the final phase projects the output arrays back to the time axis and sums up the overlapping values. The `unwindow()` procedure from Table 1 carries out this task (7 lines omitted for clarity). In TRILLDSP, the framework performs this task on behalf of the user using the provided aggregate function (line 6).

TRILLDSP has much lower code complexity than R and SPARKR, as evidenced in Table 1. The declarative query model of TRILLDSP allows expressing complex workflows using high-level operators, whereas R and SPARKR force users to write low-level operations.

All three compared systems support offline analysis, but only TRILLDSP offers real-time capabilities. The SPARKR program has hidden performance penalties as the consequence of loose coupling – each RDD function exports its input into R and imports the output back into Spark. Together with the other inefficiencies of loosely-coupled systems described above, SPARKR and SCIDB-R perform orders of magnitude worse than TRILLDSP in our experiments.

1.2 Contributions

To summarize, we make the following contributions:

1. Following the need for a deep integration of DSP operations and a general-purpose query processor, we provide a unified query and data model for relational and signal data. The query language allows the end-user to seamlessly interleave tempo-relational and signal operations when writing relational and streaming queries, without ever explicitly dealing with array data.

2. For DSP experts, we provide frameworks for defining new user-defined window operations and their integration with the query language. The framework internally exposes array abstractions to ease the implementation for DSP experts and enables incremental computation with hopping windows.

3. The unified query model supports both online and offline analysis. Users can build queries from offline data and then put them unmodified in production.

Signal operations	Type	Ref
Relational operators (select, where, join, ALJ)	N+U	§3.2
Arithmetic operations (+, -, *)	N+U	§3.2
Basic signal operation (scale, shift)	N+U	§3.2
Functional signal operations	N	§4.2
Sampling, upsampling, downsampling	N+U	§5.1
Interpolation	U	§5.2
Uniform signal aggregates (sum, power, energy)	U	§3.2
Framework for user-defined digital filters (FIR & IIR filters, correlation, convolution)	U	§5.5
Framework for user-defined window operators (FFT, windowing functions, auto-correlation, cross-correlation, element-wise product)	U	§5.6

Table 1: (N)on-uniform and (U)niform signal operations in TrillDSP with references to the used operators or frameworks.

4. The performance of our system, called TRILLDSP, fortifies the need for a deep integration of DSP and relational query processing. On purely DSP tasks, TRILLDSP is comparable with or better than best-of-breed for signal processing like MATLAB, Octave, and R, and outperforms WAVESCOPE [24, 25], another streaming engine with DSP support. For queries mixing relational and signal processing, TRILLDSP shows up to 146x better performance than loosely-coupled systems, such as SPARKR [38] and SCIDB-R [34, 17].

Table 1 summarizes the supported signal operations in TRILLDSP with references to the operators or operator frameworks used in their implementation. Some signal operations are implemented using purely relational operators. Users can extend the system with custom operations using the frameworks described in this paper.

This paper is organized as follows. We review related work in Section 2, then present our data model and relational operators in Section 3. We introduce signal processing in Section 4 and then discuss processing of uniformly-sampled signals in Section 5. We present experimental results in Section 6 and conclude in Section 7.

2. RELATED WORK

Signal Processing Numerical frameworks like MATLAB [3], R, and LabVIEW [2] provide plenty of highly-optimized algorithms for signal processing but remain unsuitable for general-purpose relational and stream processing. Spark [38] and SciDB [34, 17] provide R packages that allow users to write R code that uses these systems as backend storages with scalable processing capabilities. MonetDB [15] and SQL Server [5] support queries that can invoke R routines. In contrast to all these systems, TRILLDSP uses one query language and one data model, performs in-situ processing of relational and signal data, and provides true online support. Plato [28] applies signal processing models to imprecise or incomplete data to improve the quality and accuracy of query processing inside a DBMS, but provides no support for general signal processing in neither offline and streaming environments.

Stream Processing with DSP Functionality The WAVESCOPE project [25, 24], like our work, recognizes the fundamental need for unifying relational and signal processing operations using one query language and implementing in-situ stream operations against an array abstraction (SigSeg in WAVESCOPE). This system uses multi-stage compilation to translate queries written in the WaveScript language into source code with direct calls to 3rd-party libraries. WAVESCOPE provides a much lower programming experience overall, requiring users to understand multi-stage evalu-

ation and be aware of side effects [10]. WAVESCOPE treats arrays (SigSegs) as first-class entities that are transmitted in streams. This approach, however, ties the size of logical windows specified in queries and the size of physical batches used for processing, meaning that the window specification of a query impacts processing throughput [29]. In contrast, the DSP functionality in TRILLDSP is implemented as a library running on top of the existing relational engine, and the query language benefits from full language integration with C# (e.g., type system, type inference, parser, etc.). This walled-garden library defines clear transitions between streams and signals and integrates seamlessly with other streaming operators as it never exposes arrays outside user-defined signal operators (unless the user explicitly asks for a stream of arrays), thus, decoupling window semantics from system performance. In addition, WAVESCOPE has no built-in support for incremental computation, signals with functional payloads, frameworks for specifying user-defined DSP operators, and group-aware stateful operators.

Conventional stream processing engines [13, 19, 11, 23] have no support for signal processing. We extend Trill [18], a general-purpose incremental query engine, with signal processing functionality. StreamBase [6] enables integration with MATLAB and R but like other loosely-couple systems uses two different query languages and data models, requiring back and forth data exchanging. Gigascope [21] and Tribeca [35] are streaming systems for networking applications that support relational queries but cannot express signal operations. StreamInsight [12] allows domain-specific extensions but provides no high-level signal processing abstractions (e.g., arrays) to the user.

User-defined operations. A straw-man solution to enabling DSP in stream processors is to implement signal operations as user-defined aggregates. Some stream processors, like Spark Streaming [39], Storm [36], and Heron [30] provide frameworks for defining arbitrary aggregation logic executed upon receiving new stream events. Using such a framework, we could maintain arrays of values as aggregated states and use on-new-event functions to map event timestamps to array indexes, expire events falling out of the window range, detect when the window is filled up, and finally invoke signal operations. Such complex logic would be hard to implement, especially by DSP practitioners, who would need to understand the details of the underlying data model. Furthermore, aggregation frameworks typically target commutative operations where the order of processing input events is irrelevant for the final state, which is not the case with DSP operations.

For these reasons, we decided to treat signals as first-class citizens in TRILLDSP's query model and hide the complexity of transiting between relational and signal data, while providing array-based frameworks for DSP experts to implement custom operators.

Sensor networks TinyDB [31] and COUGAR [16] are sensor database systems supporting (tempo-)relational processing. Other specialized systems can handle sensor data [20], but none of them natively supports high-performance signal processing.

3. BACKGROUND: THE TRILL LIBRARY

We implemented TRILLDSP as an extension library for Trill [18], a high-performance incremental analytics engine that uses a tempo-relational model and supports processing of streaming and relational queries. We briefly summarize Trill's design in this section.

3.1 Data Model

Trill uses a tempo-relational data model to uniformly represent offline and online datasets as stream data. Logically, a data stream consisting of events is regarded as a temporal database [27] that is presented incrementally [14, 26, 32]. Each stream event is associated with a data window (or an interval of application time) that denotes its period of validity. Such stream events form snapshots of valid data versions across time. The user query is executed against these snapshots in an incremental manner.

Trill represents a stream of data with payload type `T` as an instance of class `Streamable<T>`. In our introductory example, users can capture the contents of events using a C# payload type:

```
struct SensorReading { long SensorId; long Time; double Value; }
```

The stream type in our example is `Streamable<SensorReading>`.

The `StreamEvent` structure provides static methods for creating stream events, including *point events* with a data window of one time unit. In our example, users may ingest sensor readings as point events at time `t` as:

```
StreamEvent.CreatePoint(t, new SensorReading {..})
```

Physically, a dataset consists of a sequence of *columnar batches*, where each columnar batch holds one array for each column in the event. For example, two arrays hold the start-time and end-time values for all events in a batch. Trill associates a *grouping key* with each event in order to enable efficient grouped operations. It also precomputes and stores the grouping keys and key hashes as two additional arrays in the batch, and includes an *absentee bitvector* to identify inactive rows in the batch.

Trill creates columnar batches during query compilation. For example, the generated batch for `SensorReading` looks like:

```
class ColumnarBatchForSensorReading<TK> {
   long[] SyncTime; long[] OtherTime;       // data window
   TK[] Key;        int[] Hash;    long[] BitVector;
   long[] SensorId; long[] Time;            // payload
   double[] Value;                          //
}
```

Trill shares these arrays with reference counting; in our example, `SyncTime` and `Time` may point to the same physical array. Trill also pools arrays using a global memory manager to alleviate the cost of memory allocation and garbage collection.

3.2 Query Language

Trill's query language is modeled after LINQ [7], with temporal interpretation of the standard relational operations and new operations for temporal manipulation. These operations are exposed using the class `Streamable<T>`. Each operator of the query language is a function from stream to stream, which allows for elegant functional composition of queries. In our example, assuming `s0` is a data source of type `Streamable<SensorReading>`, we can for instance discard invalid reading values using the `Where` operator:

```
var s1 = s0.Where(e => e.Value > 100)
```

The expression in parentheses is called a lambda expression [1]; it is an *anonymous function*, in this case from `SensorReading` to `boolean` specifying the condition for keeping each stream event in the output stream `s1`. The type of `s1` is the same as that of `s0`.

An operator in Trill accepts and produces a sequence of columnar batches. Trill's compiler dynamically generate operators and inlines lambdas (such as the `Where` predicate) in tight per-batch loops to operate directly over columns for high performance. The system provides a rich set of built-in relational operators (e.g., selection, join, and anti-join) as well as temporal operators for defining windows and sessions.

Grouped computation Trill extends the well-known Map and Reduce operations with temporal support to enable parallel execution on each sub-stream corresponding to a distinct grouping key. Consider a shortened version of the query from Section 1.1:

```
var s2 = s1.Map(s => s.Select(e => e.Value), e => e.SensorId)
          .Reduce(s => s.Sample(10, 0),
            (k,p) => new Result { SensorId = k, Temp = p })
```

The first argument to `Map` specifies a sub-query (here, the stateless `Select` operation) to be performed in parallel on the stream, while the second argument specifies the grouping key (`SensorId`) to be used for shuffling the result streams. The first argument to `Reduce` specifies the query to be executed per each group key (`Sample`), and the second argument allows us to combine the grouping key and the per-group payload into a single result.

Temporal join The temporal join operator in Trill allows one to correlate (or join) two streams based on time overlap of their events, with an (optional) equality predicate on payloads. Suppose we wish to augment the filtered `SensorReading` stream `s1` with additional information from another reference stream `ref1` that contains per-sensor location data. We would express such a query as:

```
var s3 = s1.Join(ref1, l => l.SensorId, r => r.SensorId,
  (l,r) => new Result { r.SensorLocation, l.Time, l.Value })
```

The second and third parameters to `Join` represent the equi-join predicate on the left and right inputs (`SensorId`), while the final parameter is a lambda expression that specifies how matching input tuples are combined to construct the result payload. The output stream `s3` is of type `Streamable<Result>`.

Trill's query language is extensible in several ways. First, users can express user-defined aggregation logic by providing lambdas for accumulating and de-accumulating events to and from state. These lambdas are inlined during compilation in a tight loop for high performance. Second, advanced users can write new operators that accept and produce a sequence of (grouped) columnar batches. Note that every operator understands grouping; for instance, `Count` outputs a batched stream of per-key counts. Finally, every operator is transferable between real-time and offline by construction.

4. SIGNAL PROCESSING IN TRILLDSP

Our goal is to enable high-performance signal processing in a streaming engine. We first provide the reader with the intuition behind our design choices, and why and how TRILLDSP differs from other systems capable of signal processing.

Query and data model reconciliation. Our design choice is *not* to extend the tempo-relational data model with arrays as first-class citizens. We observe that input data almost certainly arrives as timestamped relational data, not as a sequence of arrays, and that exposing streams of arrays would burden query writers with details like array sizes and whether array boundaries align with window boundaries. More importantly, having arrays at the stream level would create dependencies between window semantics and system performance, like in other streaming systems such as WAVESCOPE and Spark Streaming in which the size of logical windows specified in queries affects the size of physical batches used for processing.

We decided to use arrays *internally*, exposing them not at the stream level but instead only through designated DSP operators that would abstract away the complexity of on-the-fly transformations between the relational and array models. In that way, we integrate DSP functionality without changing the existing relational API and without affecting the performance of non-DSP queries. At the same time, we provide for DSP experts an extensible "walled garden" with familiar array abstractions and signal operators that can be interleaved with relational operators inside one query language.

Performance. Our initial design was based on a loose integration of our streaming engine with R using R.NET [4] as an interoperability bridge between R and the .NET framework. The impedance mismatch between our system and R quickly became apparent: when performing FFT over hopping windows of data, passing data to R and back took more than 91% of the total execution time and was more than two orders of magnitude slower than a tightly-coupled solution. These results fortified our decision to support in-situ DSP operations using one unified query language.

IoT applications run the same query logic over many sensor devices. For that reason, we design signal operations to natively support grouped processing and implement them as *stateful* operators that internally maintain the state of each group. Coupling these operators with Trill's streaming temporal MapReduce operator is key to efficient grouped signal processing, as seen in experiments.

We design and implement the DSP functionality in TRILLDSP as a layer running on top of the unmodified relational engine described in Section 3. The layer enriches the query model with signals as first-class citizens, carefully separating streams from signals and ensuring type safety of all operations at compile time.

4.1 From Streams to Signals

We consider signals as a special kind of streams in which stream events have no overlapping lifetimes. Thus, representing signals in TRILLDSP requires no changes of the underlying tempo-relational data model. The query model, however, needs to integrate new signal operators and enable their interleaving with the existing tempo-relational operators using one unified query language.

Converting streams into signals needs to ensure that at most one stream event is active at any point in time. This property naturally emerges after applying an aggregate function over the input stream.

```
var s0 = stream.Average(e => e.Value)
```

This query creates a signal by averaging the overlapping stream events on the Value field. When the stream already has the signal form – for example, a sensor generates point events with lifetimes $[T, T+1)$ and increasing timestamps T– users can explicitly obtain a signal using the `stream.ToSignal()` method. If the assertion turns out to be false later on, the system will throw a runtime error.

TRILLDSP considers streams having the signal form as instances of the class `SignalStreamable<T>`, which extends the base class `Streamable<T>` with signal operators. The system leverages the strong type-safety of C# to prevent invocation of signal operations over non-signal streams at compile time.

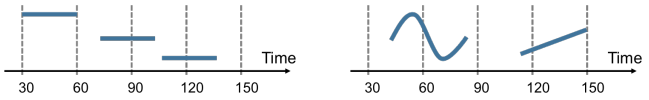
4.2 Signal Payload

Most often signals are discrete-time sequences of real or complex values called samples. Real-valued signals may originate from sensors measuring physical phenomena; complex-valued signals often emerge after processing signals in the frequency domain. Signals can also take more convoluted forms: IoT signals are often sequences of structures carrying multiple sensor measurements; audio and video signals comprise different audio channels or movie frames. Handling these cases using existing signal processing tools requires a careful arrangement of these convoluted values into the matrix form before invoking operations on them.

TRILLDSP can process signals with arbitrary payloads, including real- and complex-valued signals, as long as the payload type `T` implements the interface `TArithmetic<T>`, shown in Figure 3. The interface consists of methods necessary for enabling signal processing on payloads of type `T`: a method defining a neutral element of `T`, an equivalence method, and four basic arithmetic operations over T. Using this interface, users can customize signal payloads for processing in different application domains.

4.2.1 Functional Signal Payloads

The tempo-relational data model defines stream events as having constant payloads over a time interval, as shown in Figure 2a.



(a) Constant payloads (b) Functional payloads

Figure 2: Signal payload types

This model can naturally describe discrete-time signals and step-like continuous signals. In practice, users also want to work with more general continuous signals whose values can be expressed as a function of time, like in Figure 2b[1]. For example, in amplitude modulation, users multiply a signal with a continuous carrier signal, which is usually a sine wave of a given frequency and amplitude.

To support (pseudo-)continuous signals with the tempo-relational data model, one can materialize point events at every timestamp from the function domain. Obviously, such an approach brings huge processing and memory overheads. To capture continuous signals efficiently, TRILLDSP introduces *functional signal payloads* that carry a lambda function for computing signal values at any point in time. Functional payloads delay materialization of signal events until the user explicitly asks for it. TRILLDSP supports two types of functional payloads, which we present next.

4.2.2 *Heterogeneous Functional Payloads*

TRILLDSP allows the user to assign a lambda `Func<long,T>` to each stream event. The lambda expression expresses the payload value as a function of time. TRILLDSP represents such functional payloads as instances of `TFuncPayload<T>`. For example, the user can create the first event from Figure 2b as:

```
var evt1 = StreamEvent.CreateInterval(40, 80,
    new TFuncPayload<double>(t => Sin(2 * Pi * t / 40)))
```

In order to enable processing of signals with functional payloads, `TFuncPayload<T>` implements `TArithmetic<TFuncPayload<T>>`. For example, adding two functional payloads creates a functional payload with a lambda function expressing the addition.

```
TFuncPayload<T> Plus(TFuncPayload<T> b) {
    return new TFuncPayload<T>(t => f(t) + b.f(t)); }
```

Processing signals with functional payloads is then no different than processing any other payload type. For instance, adding two functional signals corresponds to a temporal join in which the matching events yield a summed functional payload as described above. Note that in processing signals with functional payloads, we always maintain the functional representation of values so as to avoid materialization and reduce processing and memory costs.

Users can explicitly materialize functional payloads at discrete time points to obtain a uniformly-sampled signal with the desired sampling period and offset. For example, one can sample `fs0` having functional payloads to materialize events at every 10 time units:

```
var fs1 = fs0.Sample(10);        // Materialization
```

Using `TFuncPayload<T>` allows heterogeneity in assigning lambda functions to payloads, that is, each event can take a different function, like in Figure 2b. This function heterogeneity comes at the cost of more expensive memory management during query evaluation. New stream events need to instantiate lambda objects, and since these lambdas have different structures, they cannot be shared among events or re-used through memory pooling. Instead, lambda objects are always allocated on the heap and freed through garbage collection, which increases memory management overheads.

[1]Strictly speaking, continuous-time signals have a discrete nature due to the finite time resolution in TRILLDSP.

```
interface TArithmetic<T> where T : TArithmetic<T> {
    T Zero();                T Plus(T b);
    bool Equals(T b);        T Minus(T b);
    T Scale(double scalar);  T Times(T b);
}
```

Figure 3: Interface for defining custom signal payloads

4.2.3 *Homogeneous Functional Payloads*

To process functional payloads more efficiently, we trade off flexibility and allow users to associate one lambda function to the entire signal rather than to each individual event. In that way, all stream events can share the same lambda function, hence "homogeneous" in the name, and their payloads can represent the function arguments. For example, users can create stream events with payloads being the arguments of a sine function:

```
var event = StreamEvent.CreateInterval(0, 100,
    new SinArgs { Freq = 50.0, Phase = 0.0 });
```

Each signal maintains a property that stores its lambda function. Users can associate a sine function to the signal `s1` as follows:

```
var s2 = s1.setProperty().setFunction(
    (t,e) => Sin(2 * Pi * t / e.Freq + e.Phase))
```

The default signal function is the identity function `(t,e) => e`.

Supporting homogeneous functional payloads requires a change of the signature of the signal type from `SignalStreamable<T>` to `SignalStreamable<TIn,TOut>`, where `TIn` is the payload type (e.g., `SinArgs`) and `TOut` is the result type (e.g., `double`). The lambda function is of type `Func<long,TIn,TOut>`.

Binary signal operations over homogeneous functional payloads use temporal joins to pair matching payloads from both sides as 2-tuple structures. For example, multiplying signal `s2` with a complex-valued constant signal yields `Tuple<SinArgs,Complex>` payloads, which then serve as arguments to the stream-level lambda function multiplying these two signals.

Homogeneous functional payloads enable more efficient signal processing than heterogeneous functional payloads. During query compilation, we can flatten out homogeneous payloads into a sequence of parameters and create their columnar batch representation. The generated batch for the previous example looks like:

```
struct ColBatch {double[] Freq; double[] Phase; Complex[] V; }
```

These columnar arrays are memory-pooled and shared with reference counting, which alleviates the cost of memory allocation and garbage collection. For performance reasons, TRILLDSP uses homogeneous functional payloads as the default payload type.

4.3 Signal Operators

The `SignalStreamable<TIn,TOut>` class encapsulates operators that preserve the signal form of a stream. It redefines such unary operators inherited from the base stream class, like `Where` and `Select`, to return a signal object. Since `Select` can change the input payload type `TIn`, users can also provide a new stream-level function for computing payload values or default to the identity function. The signal class also provides operators that build upon the existing stream operators. For example, `Scale` multiplies signal values with a scalar using the `Select` operator; `Shift` delays or advances a signal by changing the time intervals of its events using the alter-lifetime operator [18].

Users can perform basic arithmetic operations on signals, like addition, subtraction, or multiplication, as these operations are guaranteed to produce at most one event at any point in time. In fact, any

signal operation that uses a temporal join to match events outputs a signal. Users just need to set the stream-level function to decide on how to combine matching events into payload values. But not every binary stream operation can be lifted to the signal domain. For example, a union of two signals can produce a stream with overlapping events, so this operator remains in the `Streamable` domain.

The signal type `SignalStreamable` also provides methods for obtaining uniformly-sampled signals, like the `Sample()` method that we used to materialize functional payloads. In the next section, we cover such operations over uniform signals.

5. UNIFORMLY-SAMPLED SIGNALS

Numerical frameworks like MATLAB and R support mostly DSP algorithms on signals consisting of equally-spaced samples. These tools consider such uniform signals as real- or complex-valued arrays in which the array index act as a measure of time.

TRILLDSP regards uniform signals as streams consisting of point events at regular timestamps. Uniform signals have two defining properties: (1) *sampling period* defines the time difference between two consecutive samples, and 2) *sampling offset* defines the initial time shift of samples. These two properties can help us correlate application times of samples with their positions in the array form.

We represent equally-spaced time series as instances of the class `UniformSignalStreamable<T>`, which extends the signal class `SignalStreamable<_,T>` with uniformly-sampled signal operators. Uniform signals can have only point events at timestamps determined by their sampling period and offset. Regular (non-uniform) signals have no such constraints.

5.1 Sampling

We obtain a uniform signal by sampling a non-uniform signal. The `Sample` operator uses a given sampling period T and an offset O to generate point events at timestamps $kT+O$, where k is integer, at which the source signal has an active stream event. The `Sample` operator invokes the stream-level lambda function to compute the payload value of each materialized event. Since uniformly-sampled signals are discrete-time signals, they have no stream-level lambda function associated to them.

The `Sample` operator supports grouped signal computation. In grouped signals, each stream event carries a group identifier; the example from Table 1 groups stream events by `SensorId`. The `Sample` operator internally keeps track of active events for each group seen so far. Since we ingest stream events in a non-decreasing sync (interval start) timestamp order, each event can move the global stream time forward. In such cases, the operator needs to produce samples for each group up to the current time, update the internal state of each group to remove any inactive events, and update the current group to include the current event. If the operator detects overlapping events in the current state, it throws a runtime error implying that user's assertion about the signal form of the input stream is false.

5.2 Interpolation

Non-uniform signals consist of overlapping-free stream events of arbitrary forms. As shown in Figure 4, a non-uniform signal can have events with different lifetimes, events appearing at irregular time instants (e.g., due to network delays), or missing events (e.g., due to bad communication). The `Sample` operator discussed so far considers only events that are active at predefined sampling beats (e.g., 30, 60, 90, etc.) and ignores in-between and missing events.

To handle such irregular events, we extend the `Sample` operator with an optional parameter – *interpolation policy* – that specifies

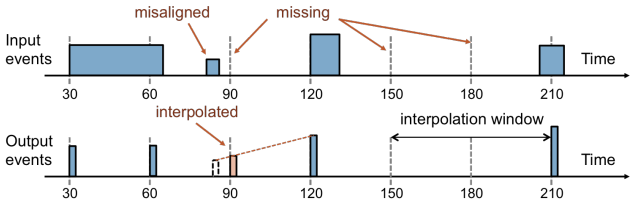


Figure 4: Sampling with interpolation of a non-uniform signal

the rules for computing missing values. An interpolation policy has two defining properties:

1. *Interpolation function* determines how to compute missing samples based on a fixed-size history of observed samples. TRILLDSP provides a set of common interpolation functions: constant, last-valued, step (zero-order), linear (first-order), and second-degree polynomial (second-order), and also supports user-defined interpolation functions.

2. *Interpolation window* defines the maximum time distance among the reference samples used for interpolation. For instance, in linear interpolation, if two consecutive reference samples are more than one interpolation window apart, we consider them as two different signal sequences and disable interpolation in that interval, like in Figure 4.

The user can obtain the uniform signal from Figure 4 as follows:

```
var s1 = s0.Sample(30, 0, p => p.FirstOrder(60));
```

Each uniformly-sampled signal maintains a property that stores an interpolation policy. This policy is used during signal re-sampling, which happens, for instance, in binary operations involving uniform signals with different sampling periods.

5.2.1 Interpolation implementation

The `Sample` operator implements interpolation on grouped signals. For each group, the operator maintains a finite number of observed samples, which serve as reference points for a given interpolation function. Each stream event with an interval $[a,b)$ generates a sequence of reference points: a start point at $[a,a+1)$, an end point at $[b-1,b)$, and all intermediate points at the sampling beats from the interval (a,b).

A per-group interpolator state maintains most recent reference points using a fixed-size circular buffer. The buffer consists of at least two samples, which define the currently active interpolation window; higher-order interpolators may buffer more samples. Each interpolator state also encapsulates the interpolation function that corresponds to the chosen interpolation policy. The `Sample` operator interacts with each interpolator state independently of the interpolation function using the following methods:

- `AdvanceTime(long t)` invalidates points in the buffer that fall outside the interpolation window ending at time `t`.
- `AddPoint(long t, T v)` overwrites the oldest valid entry in the interpolation buffer with a given reference point.
- `CanInterpolate(long t)` returns true if the interpolation buffer is full and `t` is between the last two reference points.
- `Interpolate(long t)` invokes the interpolation function to compute the value at time `t`.

As the global time moves forward with each stream event, the `Sample` operator updates the state of each group to include new

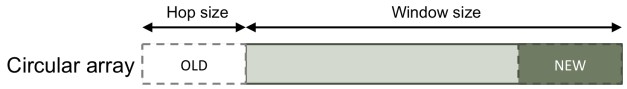


Figure 5: Signal window

and invalidate too distant reference points. When the interpolation buffer of one group is full, it interpolates events at the sampling beats between the last two reference points. Note that the operator interpolates events back in the past. To preserve the time order of output events, the `Sample` operator delays emitting output events by at most the size of the interpolation window.

5.3 Uniform-signal Operators

In binary operations with uniformly-sampled signals, we require that both operands share the same sampling period and offset for correctness. *Resampling operations* allow users to change these properties to meet the requirement. `Upsample(n)` increases the sampling rate (i.e., decreases the sampling period) of a signal by an integer factor. The interpolation function computes the values of new intermediate samples. `Downsample(n)` decreases the sampling rate (i.e., increases the sampling period) of a signal by keeping every *n*-th sample and dropping the others. `Resample()` changes the sampling period, offset, and interpolation function of a signal. For instance, we can resample the uniform signal from Figure 4 to halve the sampling period, change the interpolation policy, and double the interpolation window size.

```
var s2 = s1.Resample(30, 0, ps => ps.ZeroOrder(120))
```

Binary arithmetic operations involving uniformly-sampled signals use temporal joins for matching samples. For instance, summing two uniform signals, written as `left.Plus(right)`, when they share the same sampling period and offset is implemented as:

```
left.Join(right, (l, r) => l + r)
```

When these conditions are unmet, binary operators preprocess one of the operands before joining them together: if the sampling periods are the same but offsets are different, we shift one signal by the offset difference; if the sampling periods are different, we resample the signal with a larger sampling period to match the period and offset of the other signal; finally, when one operand is a non-uniform signal, we sample that signal with the sampling period and offset of the other uniform operand to produce a uniform result.

Uniform-signal aggregates are built using the existing relational operators. For instance, summing samples of a uniform signal `s` using a hopping window of size `W` and hop size `H` samples, written as `s.Sum(W,H)`, is implemented as:

```
s.HoppingWindow(W * s.Period, H * s.Period, s.Offset)
 .Aggregate(w => w.Sum(e => e))
 .AlterEventDuration(1)
```

where the `HoppingWindow` macro creates hopping windows by altering event lifetimes, the `Sum` aggregate adds values together, and the `AlterEventDuration` operator restores the uniform-signal form by emitting point events. Note that uniform-signal aggregates and operators use sample-based windows instead of time-based windows to match the expectation of DSP users. Uniform-signal aggregates can also easily integrate any user-defined aggregate (not just `Sum`, `Count`, etc.) created using an extensible aggregate framework, which we support in a similar way to previous work [18].

5.4 Signal Windows

We represent uniformly-sampled signals as data streams built using a tempo-relational model. Numerical tools providing DSP algorithms like MATLAB and R consider uniformly-sampled signals as real- or complex-valued arrays, in which the array index is best thought of as a measure of dimensionless time. To bridge the gap between these two models, we introduce an abstraction that allows DSP users to implement custom operators for offline and online processing in a way that feels natural to them – by writing algorithms against arrays without worrying about the temporal aspect of the underlying data model. Exposing arrays to operator writers enables easy and quick integration of existing highly optimized DSP algorithms, like those exploiting SIMD instructions.

```
SetMissingDataToZero   () => bool
GetFeedForwardSize     () => int
GetFeedBackwardSize    () => int
Compute                (TWindow<T>, TWindow<T>) => T
```

Figure 6: Interface for defining custom digital filters

We introduce a class called `TWindow<T>` that transforms samples of type `T` from the tempo-relational data model into the array data model. In processing uniform signals using windows of data, a signal window maintains an array of active samples of one uniform signal and uses the sampling period and offset of the associated signal to compute the index position of a new sample. The signal window also keeps track of the latest timestamp in the array to detect any missing value when adding a new sample; these missing values are either replaced with zeros or marked as invalid. Each signal window is implemented as a fixed-size circular array to efficiently support offline and online processing with overlapping windows.

Signal windows support incremental computation over hopping windows. Each signal window can maintain a fixed-size history of recently expired samples, as shown in Figure 5. The history size is often the window hop size. The `TWindow<T>` class provides methods for accessing the whole array (`Array`), the old delta (`Old`), and the new delta (`New`). Users can use these deltas to implement operator logic that incrementally updates its state on every hop event.

Signal windows notify registered observers when their arrays are full. A signal window can fire up two types of events: 1) `OnInit` event denotes the array is filled up for the first time, and the old delta is invalid and the new delta is the whole array; 2) `OnHop` event denotes a window hop when both deltas are valid, see Figure 5.

In TRILLDSP, we provide different implementations of signal windows based on their specification. A window specification comprises three parameters: the window size (how many samples each window lasts), the hop size (by how many samples each window moves forward relative to the previous one), and the boolean indicator on how to handle missing samples (if `true` replace them with zeros; otherwise, mark them as invalid). Note that signal windows containing invalid samples cannot fire up any event. The decision on how to treat missing values is operator-dependent. For instance, users might want to use complete arrays when using FFT but zero-padded signals when using digital filters.

5.5 Digital Filters

Digital filters are a cornerstone of digital signal processing. Due to their extraordinary efficiency, digital filters are widely used in practice; for example, filters can separate a signal from noise or restore bad audio recordings [33]. Digital filters are often described in terms of an equation that relates the output signal to the input signal. For example, linear digital filters produce outputs by combining a fixed-size window of inputs and a fixed-size window of previously computed outputs.

$$y[n] = \sum_{i=0}^{N} a[i] * x[n-i] + \sum_{i=1}^{M} b[i] * y[n-i]$$

Here, $x[i]$ and $y[i]$ are sequences of values denoting the input and output signals, $a[i]$ are called feed-forward coefficients, and $b[i]$ are called feed-backward coefficients. The feed-forward window is of size $N+1$, while the feed-backward window is of size M.

We provide a framework for defining custom digital filters that use `TWindow<T>` instances for feed-forward and feed-backward windows. The user needs to implement the interface from Figure 6 for each class of digital filters, which involves specifying the sizes of the feed-forward and feed-backward windows, deciding on how to interpret missing values, and expressing the filtering functionality as a lambda function with a feed-forward and a feed-backward signal windows as parameters.

The digital filter framework supports grouped signal processing. A per-group state maintains two instances of `TWindow<T>`. The framework invokes the `Compute()` method upon receiving `OnInit` or `OnHop` events from the feed-forward window and updates the feed-backward window with the result; upon the `OnInit` event, it also resets the feed-backward window. Note that the framework releases users from the burden of explicitly managing windows in processing grouped signals and allows them to focus on the actual implementation using array abstractions.

An example of a digital filter is a finite impulse response filter, which uses only the feed-forward loop. For a given filter weights $a[i]$, each output is a weighted sum of the most recent inputs. Users can implement such filters as follows:

```
SetMissingDataToZero: () => true
GetFeedForwardSize:   () => a.Length
GetFeedBackwardSize:  () => 0
Compute:              (fw, bw) => DotProduct(a, fw.Array)
```

Signal windows expose array abstractions to users, which enables them to use highly-optimized black-box implementations of DSP algorithms. In this example, users can implement the `DotProduct` method using SIMD instructions of modern processors.

TRILLDSP uses the digital filter framework to implement several signal operators, like signal correlation, signal convolution, and finite and infinite impulse response filters, to name a few.

5.6 User-defined Window Operators

DSP domain experts and practitioners exercise one fundamental workflow when analyzing uniform signals. The workflow consists of three steps [33]: (1) the first step splits the signal into smaller, possibly overlapping components, (2) the second step transforms each component using a sequence of operations, and (3) the third step recombines the processed components into the final signal.

We support such workflows using the `Window` operator, presented next using a shortened version of the query from Section 1.1 as our running example. We assume `s2` is a real-valued uniform signal.

```
var s3 = s2.Window(512, 256, true,
  w => w.FFT().Select(a => f(a)).InverseFFT(), a => a.Sum()));
```

The first workflow step is about windowing the data. The `Window` operator transforms samples from streams into arrays using signal window instances of type `TWindow<T>`. To support grouped signal processing, the operator internally maintains one such instance for each group seen so far. Users can specify sample-based window attributes using the `Window` operator. As before, the window specification includes the window size, the hop size, and the policy on how to handle missing values (`true`: zero, `false`: NaN).

The second workflow step executes a sequence of transformations over fixed-size arrays. The `Window` operator expresses these computations as a pipeline of operators. Each pipeline operator transforms an input window of type `TWindow<U>` into an output window of type `TWindow<V>`. In our example, the `FFT` operator

```
GetOutputWindowSize   () => int
Update                (TWindow<U>, TWindow<V>) => ()
Recompute             (TWindow<U>, TWindow<V>) => ()
```

Figure 7: Interface for defining window pipeline operators

transforms a 512-sample window of type `TWindow<double>` into a 512-sample window of type `TWindow<Complex>` representing the frequency spectrum of the input windowed signal. In general, input and output windows can have different sizes; for instance, the `AutoCorrelation` pipeline operator takes windows of size N and outputs windows of size $2N+1$. The `Window` operator creates exactly one pipeline per signal group, which amortizes the startup overheads of pipeline operators; for instance, `FFT` and `InverseFFT` initialize their spectrum coefficients only once.

Each pipeline operator implements the interface presented in Figure 7. The first interface method defines the output window size of a pipeline operator, which the `Window` operator uses to pre-allocate one output window per operator. The other two methods define lambda functions for incremental computation and re-evaluation, both changing the output window in-place to avoid memory allocations. Note that operator writers can use black-box implementations for these operations. Also, it is their responsibility to enable incremental computation by denoting the delta parts of the output, if possible. Many signal operations, however, completely perturb their outputs making re-evaluation the only option for the downstream pipeline operators. In our example, only `FFT` can incrementally update its output; the remaining operators have to recompute their results from scratch. `InverseFFT` produces results in the output window of type `TWindow<Complex>`.

The final workflow step combines computed output windows into the final signal. This step requires projecting output values from arrays back on the stream time axis as stream events. The `Window` operator projects N output values, which are generated at hop time `t`, backwards in time such that consecutive stream events are one sampling period apart from each other and the last event has timestamp `t`. If the final output size is greater than the hop size, then the projected stream events overlap. To preserve the signal form of the final stream, the `Window` operator allows users to provide an aggregate function (e.g., `Sum`) for merging overlapping output samples. When there are no overlapping events, we can safely lay out stream events on the stream time axis and yet preserve the signal form of the stream.

The last parameter of the `Window` operator is optional. If the aggregate function is omitted, the operator skips the "unwindowing" operation and outputs arrays as stream events defined at hop time points. Omitting the aggregate function in our example would create a signal of type `UniformSignalStreamable<Complex[]>`. Similarly, when the final pipeline operator produces an output of type `T` that is different than `TWindow<_>`, then the output signal is of type `UniformSignalStreamable<T>`.

6. PERFORMANCE EVALUATION

We evaluate how TRILLDSP's deep integration of signal and relational processing compares against state-of-the-art numerical frameworks specialized for DSP operations, data analytics engines loosely coupled with such frameworks, and one tightly-integrated system. Our experiments aim to answer two main questions:

1. What is the price of integrating signal processing operations into a relational query processing system?
2. What is the benefit of having unified data and query models?

Our first set of experiments shows that a general-purpose query engine with tightly-integrated signal operations can bring competitive or even better performance than popular numerical frameworks in pure signal processing tasks. These results come at no surprise as TRILLDSP has relatively low system overheads (smaller than the other tightly-coupled system we compared against) and exposes array abstractions to operator writers for quick and easy adoption of highly-optimized black-box implementations of DSP operations.

Our second set of experiments focuses on a particularly important class of IoT applications that run the same query logic over a large collection of sensor devices. Here, TRILLDSP shows its full potential when processing large numbers of groups using queries that combine relational and signal operations. TRILLDSP's in-situ execution model achieves from 3x to 146x better performance on grouped signal processing than modern relational and array data management systems with loose R integration.

Benchmarked systems We compare TRILLDSP against numerical frameworks that are widely used in practice by DSP experts and practitioners for offline signal analysis.

- MATLAB R2015b – a numerical computing environment offering a plethora of signal processing operations;
- Octave 4.0 – an open-source alternative to MATLAB;
- Microsoft R Open 3.2.3 (R for short) – an enhanced, open-sourced distribution of R used for statistical data analysis.

For workloads combining relational and signal processing, we compare TRILLDSP against the following loosely-coupled systems:

- Spark with R integration 1.5.2 (SPARKR) – an R package providing access to Spark from R;
- SciDB with R integration 14.12 (SCIDB-R)[2].

We also benchmark WAVESCOPE [25, 24], the system closest in spirit to TRILLDSP that allows users to express signal processing operations as declarative queries over streams of data. WAVESCOPE compiles such queries into high-performance execution engines with in-situ processing and direct calls to 3rd-party libraries. We compiled the latest version of the system available in the project repository [9] using PLT Scheme v4.1.1, and benchmarked with the recommended C backend (WSC2) and O3 optimization flag.

TRILLDSP uses a batch size of 80,000 tuples. Appendix B shows the effect of different batch sizes on TRILLDSP's performance.

Experimental Setup We run our experiments on D14 Microsoft Azure instances consisting of 2 Intel Xeon CPU E5-2673 v3 @ 2.40GHz and 112GB of RAM. We use one instance with Ubuntu 12.04.5 LTS for running WAVESCOPE, Octave, R, SPARKR, and SCIDB-R queries, and another instance with Windows Server 2012 R2 Datacenter for running TRILLDSP and MATLAB queries.

[2]The official integration of SciDB and R provides an R package for accessing the SciDB backend from R, same as Spark. However, this loosely-coupled approach suffers from huge overheads when serializing large datasets. For instance, just sending our dataset with 100 million events between SciDB and R without processing them takes more time than any total running time among the other compared systems. We conclude that the official SCIDB-R integration is designed primarily for exchanging small aggregate results between these two systems. In order to make the SCIDB-R integration feasible on our datasets, we use a SciDB plugin [8] that enables running R programs within SciDB queries. In contrast to the official integration, here we write our queries as SciDB (not R) programs and run them from the SciDB (not R) shell.

Query Workload Our query workload includes five queries in total, which we evaluate using seven different systems. Two queries represent pure DSP tasks commonly used in practice, while the remaining three queries combine relational and signal operations on grouped signals using Map-Reduce operators.

Data Workload Our datasets consist of 100 million randomly generated stream events with double-precision values. For the pure DSP tasks, these events represent real-valued uniform signals with equally-spaced samples and the schema `<Time,Value>`. For the grouped computation queries, stream events have the grouping key (`SensorId`) and schema `<SensorId,Time,Value>`. TRILLDSP never explicitly operates on the `Time` column as these values are implicitly encoded in the data windows of events. The query workload is data independent (except the final filtering step in `query3`, which takes negligible time in all the systems), so the reported performance would have been identical if we had used real data.

6.1 Traditional DSP Tasks

We compare TRILLDSP with one streaming engine and three numerical frameworks on tasks commonly used by DSP experts.

FFT over Tumbling Windows Our first query computes the Fast Fourier Transform (FFT) over tumbling windows of different sizes. In TRILLDSP, we write this query as:

```
var query1 = s0.Window(windowSize, windowSize, true,
    w => w.FFT(), a => a.Sum())
```

The input `s0` is a uniform signal; the output is a stream of complex values representing the frequency spectrum of each window. The streaming queries in TRILLDSP and WAVESCOPE consist of three phases: windowing stream events, performing FFT, and unwindowing outputs to obtain a stream of complex values. Appendix A presents the cost of each phase. Both systems invoke the same FFT function in the FFTW library [22].

The numerical tools, MATLAB, Octave, and R, perform only offline computations using preallocated input and output arrays, effectively executing only the second query phase (FFT). We consider two different versions of such offline programs: (1) The 2D versions repack signal values into a matrix form and invoke a two-dimensional FFT only once; the reshaping technique is common among practitioners but obviously applicable only in offline analysis of entire datasets fitting into main memory; (2) The 1D versions invoke FFT repeatedly for each window, thus mimicking a streaming environment but still executing only the second query phase.

Figure 8 shows the running times of these five systems when processing our dataset. In an apple-to-apple comparison, TRILLDSP outperforms WAVESCOPE in all cases by 43 to 60% overall, showing better integration with the FFTW library but being 18% slower on the window-unwindow operation (see Appendix A for more details). In TRILLDSP, windowing-unwindowing dominates by far, from 86% to 89%, in the total execution time. These overheads are missing in the offline tools, nevertheless TRILLDSP outperforms both versions of R and manages to stay competitive with MATLAB and Octave, often being faster than their 'streaming' versions. The running times of the 1D programs decline with larger window sizes due to fewer FFT invocations. TRILLDSP spends on average 0.4s on preparing and executing FFT (see Appendix A), much lower than the numerical tools, which are interpreting their programs.

FIR Filtering Our next query uses a finite impulse response (FIR) filter implemented via the digital filter framework from Section 5.5. The filter convolutes the signal with random filter coefficients:

```
var query2 = s.FilterFIR(coeffs)
```

The query performs a sliding window computation where each output value is a weighted sum of n recent values, where n is the length

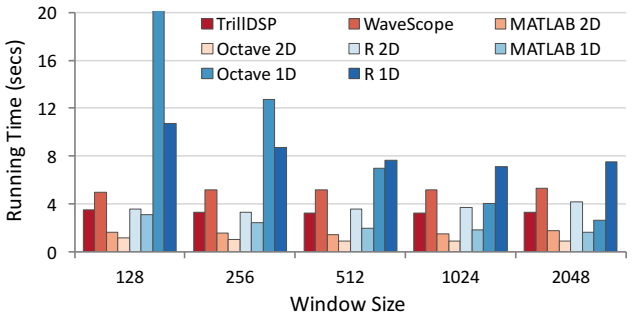


Figure 8: FFT with tumbling window

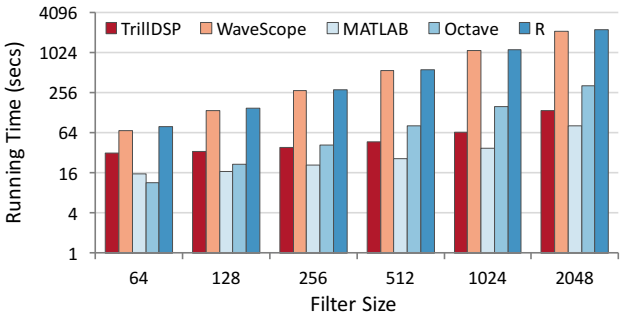


Figure 9: Digital filtering

of `coeffs`. Internally, TRILLDSP uses a `TWindow<double>` instance of size n to buffer recent input values. WAVESCOPE performs similarly[3] although without checking for missing values, like in TRILLDSP. The other systems use the built-in `filter` function.

Figure 9 compares the filtering performance of our systems. Notice the log y-axis. TRILLDSP's SIMD implementation of dot product outperforms the naïve loop implementation in WAVESCOPE and R by a wide margin. TRILLDSP stays competitive with Octave while being up to 1.8x slower than MATLAB. In contrast to the offline tools, TRILLDSP incurs additional overheads from copying stream events into arrays and checking for missing values.

These experiments demonstrate that TRILLDSP can easily integrate with 3rd-party libraries to deliver performance competitive with or even better than the state-of-the-art tools used by practitioners in offline analysis. But most importantly, TRILLDSP offers true online support and a tight integration with relational operators, which none of these numerical frameworks does.

6.2 Grouped Signal Processing

Next, we consider grouped computation where the same query logic is executed for each group identified by the grouping key (`SensorId`). We consider three sub-queries of growing complexity combining signal and relational processing, and we vary the number of groups while keeping the input signal size fixed. We introduce two new systems, SPARKR and SCIDB-R, capable of processing groups in parallel. As both of these are offline engines, we minimize the overhead of their integration with R integration by invoking R only once per group, whereas the amount of communicated data remains unchanged. Noteworthy, WAVESCOPE's built-in functions for grouped signal processing, `deinterleave` and `interleave`, crash due to insufficient memory for our input size; thus, we exclude this system from the discussion.

Grouped Signal Correlation Our next query attempts to correlate a given vector of values of size 32, denoted as a, with a uniform signal of each group identified by `SensorId`. The correlation operation is a sliding window (i.e., the hop size is 1) computation that evaluates a dot product between the window and vector a at every sample point. In TRILLDSP, we express the query as:

```
var query3 =
  s0.Map(s => s.Select(e => e.Temp), e => e.SensorId)
    .Reduce(s => s.Correlation(a),
      (k, p) => new Result { SensorId = k, Temp = p })
```

The `Map` operation selects the value column and tags each stream event with its group identifier (`SensorId`), while the `Reduce` operation executes the sub-query on each group. TRILLDSP supports parallel execution of sub-queries using the specified level of parallelism. Here, we evaluate its performance using 1 and 16 cores.

The other systems implement the grouping operation differently. MATLAB and Octave partition the input signal via the `accumarray` method and then sequentially process each group in a for loop. Due to performance reasons, we partition the signal in R using two methods, `accumarray` for groups of size 10 and `split` for larger groups. SPARKR uses `groupByKey` to partition an input RDD followed by a local sort of each partition to regain the time order, as shown in the example from Table 1. SCIDB-R uses the `redimension` operator to change the layout of array chunks based on the grouping key, also followed by a local sort of each group.

Figure 10 shows the query performance of these systems with different numbers of groups. TRILLDSP with 16 cores (TRILLDSP-16) consistently outperforms all other systems due to its built-in support for grouped processing inside the `Correlation` operator. MATLAB, Octave, and R have no such features in offline mode, let alone online mode, and have to deinterleave the input signal into groups entirely before processing each group in turn. This partitioning operation turns out to be the bottleneck in these systems, incuring from 70% to 97% of the total execution cost and causing 1.9x, 2.1x, and 14.9x worse performance, respectively for each system, than TRILLDSP-16 for 100,000 groups. SPARKR and SCIDB-R pay a high price for their loose integration with R and are 3-20x slower than TRILLDSP-16, despite grouping and executing sub-queries in parallel (interestingly, single-threaded `groupByKey` in SPARK is faster than R's partitioning operation, which reflects in better performance in three cases). Both systems significantly degrade their performance when processing large numbers of groups.

Grouped signal interpolation and filtering Next, we consider two uniformly-sampled signals `s1` and `s2` consisting of real-valued samples from $[0,1]$ and having different sampling periods, 20 and 5. For each group, we want to upsample the first signal to match the sampling period of the second and report events at which both group signals have simultaneously extreme values (close to 0 or 1). TRILLDSP expresses the grouping of `s1` as before (same for `s2`):

```
var u1 = s1.Map(s => s.Select(e => e.Temp), e => e.SensorId)
```

We write the query using a two-parameter `Reduce` operator as:

```
var query4 = u1.Reduce(u2, (l, r) =>
  l.Sample(5, 0, p => p.FirstOrder(20)).Plus(r)
   .Where(e => e < 0.0001 || e > 1.9999),
  (k, p) => new Result { SensorId = k, Temp = p })
```

Figure 11 shows the running times of our systems normalized by the running time of TRILLDSP with 16 cores over varying numbers of groups. Note the log y-axis. The numerical tools spend most of their time interpolating values inside the `interp1` function, up to

[3]FIR filter inspired by `apps/eeg/eeg_static.ws` [9].

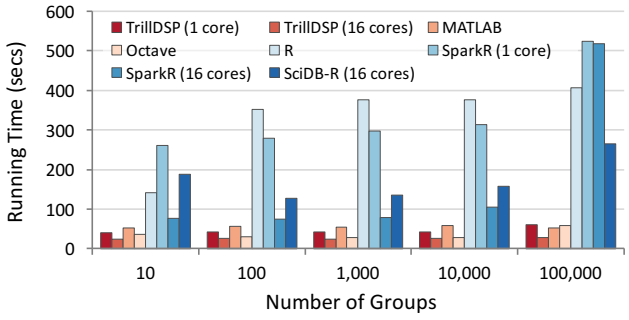


Figure 10: Grouped signal correlation

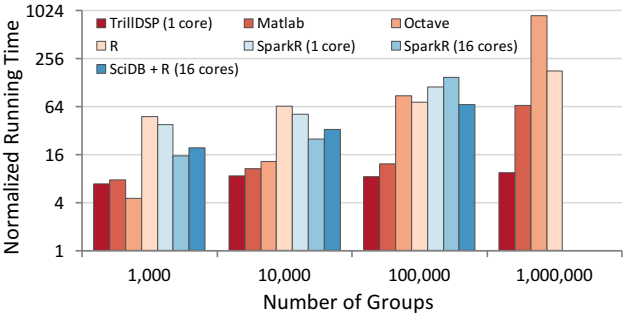


Figure 11: Grouped signal interpolation and filtering

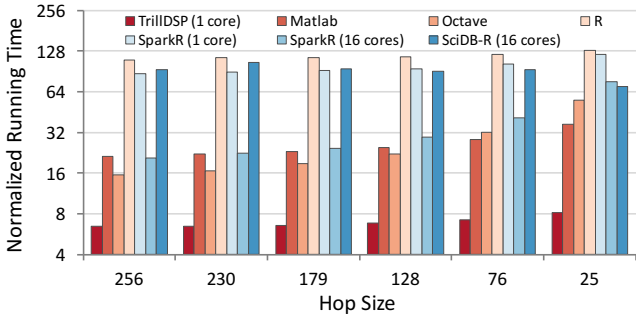


Figure 12: Grouped overlap-and-add method

88% in MATLAB, in contrast to the previous example where the grouping cost was dominant. Octave incurs fairly large interpolation overheads, which significantly degrades its performance as the number of groups (i.e., `interp1` calls) increases. We note that even the single-threaded TRILLDSP program outperforms all the other systems in all but one cases due to its group-aware interpolation operator simultaneously processing multiple subsignals. Running TRILLDSP using 16 cores accelerates the single-threaded performance by up to 9.4 times, which leads to much better performance overall, up to 146x faster than SPARKR and up to 67x faster than SCIDB-R. Both of these systems behave poorly for large numbers of groups, eventually crashing when processing one million groups. Such bad performance was primarily caused by expensive reshuffling operations, which in SPARKR also involve writing to disk.

Grouped overlap-and-add The final query expresses the overlap-and-add method and serves as a blueprint for a large class of DSP workloads. We modify the query from Section 5.6 to use the identity function in the `Select` operator in order to measure the query performance independently of the user-defined function.

```
var query5 =
  s0.Map(s => s.Select(e => e.Temp), e => e.SensorId)
    .Reduce(s =>
      s.Window(windowSize, hopSize, true, w =>
        w.FFT().Select(w => w).InverseFFT(), a => a.Sum()),
      (k, p) => new Result { SensorId = k, Temp = p })
```

Figure 12 shows the performance of our systems normalized by the performance of TRILLDSP using 16 cores. We consider a stream with 100 groups and hopping windows with 256 samples and different hop sizes. Note the log y-axis. The MATLAB, Octave, and R programs are based on the code DSP practitioners in our organization use on a daily basis. Their strategy for dealing with overlapping windows is to replicate them into matrix form with no overlapping data, and then apply fast 2D-FFT. The memory requirements of that approach grow as the hop size becomes smaller, which reflects in worse performance of these tools in our experiments. Combined with their intrinsic grouping overhead, the numerical tools show 37-124 times worse performance than TRILLDSP for the smallest hop size. SPARKR and SCIDB-R exhibit consistently bad performance mostly caused by inefficient grouping and expensive communication with R. TRILLDSP internally uses circular arrays to effectively handle overlapping windows of data and achieve low memory footprint. Since the windowing operator handles 100 subsignals simultaneously, the cost of windowing-unwindowing is more pronounced than in `query1`, which processes only one group. Appendix A discusses these costs in more detail.

7. CONCLUSION

In this paper, we advocate for a deep integration of domain-specific tools and general-purpose query processors to support complex data analysis. Our approach unifies the seemingly disparate worlds of tempo-relational and array data and enables seamless interleaving of relational and signal operators within one query language, and yet provide experts from both relational and signal processing worlds with their familiar abstractions. Our system, called TRILLDSP, builds on top of a commercial streaming engine, natively supports grouped signal processing in both online and offline mode, and provides extensible frameworks for domain experts to integrate new black-box operations. In workflows combining relational and signal processing, TRILLDSP achieves up to two orders of magnitude better performance of grouped signal processing than state-of-the-art numerical tools and data management systems with loose R integration.

Regarding the applicability of our approach to other systems, one could implement our signal "walled garden" inside any system that exposes streaming event-at-a-time API, for instance inside spouts in Storm or Heron. Notice, however, that in our data model, each event has an interval (lifetime) associated with it, while most other systems regards events as single points in time. In that sense, our interpolation operator has richer semantics than an equivalent operator considering only point events. All other uniformly-sampled operators from this paper consider only point events, so implementation of these operators in a general-purpose streaming system would be relatively straightforward. While one could use our language and model for signal processing in the context of existing systems such as Storm and Spark Streaming, what is less clear is whether our model would interoperate and compose well with the semantics of the existing operators in these systems.

8. REFERENCES

[1] Expression Trees. https://msdn.microsoft.com/en-us/library/bb397951.aspx. Retrieved 05/29/2016.

[2] LabVIEW homepage. http://www.ni.com/labview.

[3] MATLAB Signal Processing Toolbox homepage. http://www.mathworks.com/products/signal.

[4] R.NET. https://rdotnet.codeplex.com. Retrieved 23/10/2016.

[5] SQL Server R Services. https://msdn.microsoft.com/en-us/library/mt604845.aspx. Retrieved 05/29/2016.

[6] StreamBase homepage. http://www.streambase.com.

[7] The LINQ Project. http://tinyurl.com/42egdn. Retrieved 05/29/2016.

[8] The r_exec plugin for running R code within SciDB queries. https://github.com/Paradigm4/r_exec. Retrieved 05/29/2016.

[9] The WaveScope/WaveScript project on GitHub. https://github.com/rrnewton/WaveScript. Retrieved 23/10/2016.

[10] WaveScript Rev 3657 User Manual. http://www.cs.indiana.edu/~rrnewton/wavescope/wsman.pdf. Retrieved 23/10/2016.

[11] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[12] M. H. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The Extensibility Framework in Microsoft StreamInsight. In *ICDE*, pages 1242–1253, 2011.

[13] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. *Book chapter*, 2004.

[14] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR*, pages 363–374, 2007.

[15] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, pages 479–490, 2006.

[16] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Mobile Data Management*, 2001.

[17] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, pages 963–968, 2010.

[18] 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*, 8(4):401–412, 2014.

[19] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, 2003.

[20] C.-Y. Chong and S. P. Kumar. Sensor Networks: Evolution, Opportunities, and Challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.

[21] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, pages 647–651, 2003.

[22] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[23] B. Gedik, H. Andrade, K. Wu, P. S. Yu, and M. Doo. SPADE: The System S Declarative Stream Processing Engine. In *SIGMOD*, pages 1123–1134, 2008.

[24] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. The Case for a Signal-Oriented Data Stream Management System. In *CIDR*, pages 397–406, 2007.

[25] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. XStream: A Signal-Oriented Data Stream Management System. In *ICDE*, pages 1180–1189, 2008.

[26] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Y. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A Query Processing Engine for Data Streams. In *ICDE*, page 851, 2004.

[27] C. S. Jensen and R. T. Snodgrass. Temporal specialization. In *Encyclopedia of Database Systems*, pages 3017–3018. 2009.

[28] Y. Katsis, Y. Freund, and Y. Papakonstantinou. Combining Databases and Signal Processing in Plato. In *CIDR*, 2015.

[29] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *SIGMOD*, 2016.

[30] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, pages 239–250, 2015.

[31] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *TODS*, 30(1):122–173, 2005.

[32] D. Maier, J. Li, P. A. Tucker, K. Tufte, and V. Papadimos. Semantics of Data Streams and Operators. In *ICDT*, pages 37–52, 2005.

[33] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[34] M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR*, pages 173–184, 2009.

[35] M. Sullivan and A. Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *USENIX*, 1998.

[36] 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. Ryaboy. Storm@Twitter. In *SIGMOD*, pages 147–156, 2014.

[37] F. M. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *BIRTE*, 2008.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[39] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *HotCloud*, 2012.

APPENDIX

A. COST BREAKDOWN ANALYSIS

In this section, we breakdown execution costs of TRILLDSP and WAVESCOPE to gain deeper understanding of their performance.

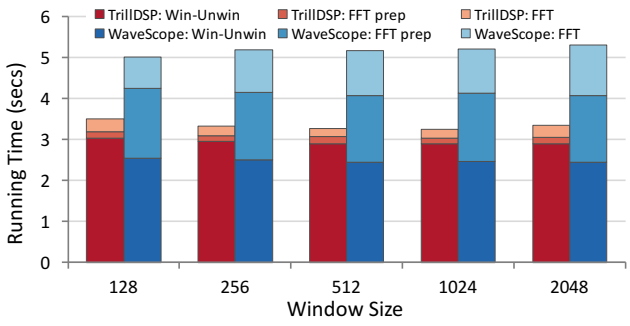


Figure 13: Cost breakdown of FFT with tumbling window

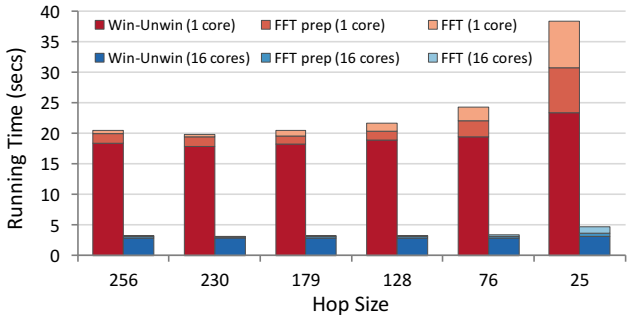


Figure 14: Cost breakdown of grouped overlap-and-add method

Figure 13 compares the cost of TRILLDSP and WAVESCOPE for executing `query1` from our experiments, FFT with tumbling windows of different sizes. Both systems use the same FFTW library [22] although on different platforms, TRILLDSP on Windows and WAVESCOPE on Ubuntu. The goal of this analysis is to estimate the execution costs of the following three phases:

1. *Win-Unwin* refers to the windowing operation of stream events into arrays followed by the unwindowing operation to restore the input stream;
2. *FFT prep* denotes the time spent in preparing for FFT calls after the windowing phase; this phase copies arrays into buffers;
3. *FFT* refers to the time spent inside the FFTW library.

Since measuring the exact running time of each phase is hard in these streaming systems, we estimate their costs by running the following three queries:

1. A query with a dummy windowing function. In TRILLDSP, we write this query as:

```
var query1 = s0.Window(windowSize, windowSize, true,
    w => w, a => a.Sum())
```

   In WAVESCOPE, we call `window()` and `dewindow()` one after the other. These running times form the basis of Figure 13.

2. A modified original query where the actual call to the FFT function (`fftw_execute_dft_r2c`) is commented. These running times subtracted from the windowing-unwindowing times estimate the cost of FFT preparation in each system.
3. The results of the original query are sufficient to estimate the FFT execution times.

Figure 13 shows that TRILLDSP outperforms WAVESCOPE in all cases by 43-60% overall. The *Win-Unwin* cost in TRILLDSP is 18% higher than in WAVESCOPE due to mapping of timestamps into array indices and checking for missing values, which is not performed in WAVESCOPE. The overhead of preparing for FFT is almost negligible in TRILLDSP, while it represents 30-35% in WAVESCOPE. Finally, FFT on Windows is by up to 5x faster than FFT on Ubuntu, despite using the same FFTW library. This experiment shows that the window-unwindow operation in TRILLDSP is the bottleneck, which explains fairly flat performance in Figure 8.

Our next analysis (Figure 14) considers `query5` from our experiments, the overlap and add method over a group of signals. We use the same methodology as before to estimate the cost breakdown in TRILLDSP running 1 and 16 cores. We consider a grouped signal computation over 100 groups, and the window size is 256 samples.

For the single-threaded implementation, we have a similar cost distribution as before for hopping windows with little overlap. As the hop size decreases (overlapping increases), the running times of *FFT prep* and *FFT* also increase as we perform more FFT calls. The multi-threaded version exhibits from 6.4x to 8.2x better performance, while almost completely attenuating FFT-related costs.

B. EFFECTS OF BATCH SIZES

Our next experiment measures the effect of different batch sizes on TRILLDSP's performance. Batching size is used as an internal configuration parameter of our engine, and batching is completely transparent to query writers.

Figure 15 shows the performance of `query1` from our experiments, FFT with tumbling windows, for different batch sizes. As expected, small batch sizes create negative cache effects causing bad performance. For batch sizes with more than 1,000 stream events, TRILLDSP's performance remains mostly flat with some minor degradation for extremely large sizes. In all our experiments, we used a batch size of 80,000 tuples.

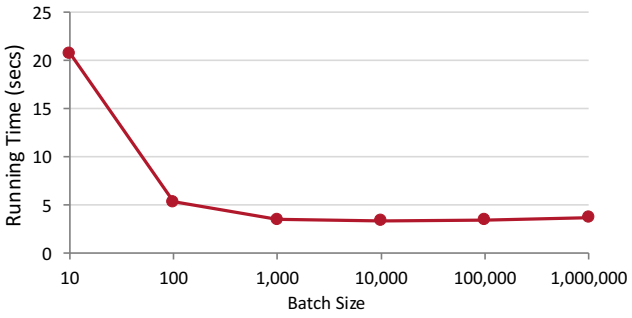


Figure 15: The effect of different batch sizes on the performance of FFT with tumbling windows