# Trill: Engineering a Library for Diverse Analytics

Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, James F. Terwilliger
Microsoft
`{badrishc, jongold, mbarnett, jamest}@microsoft.com`

**Abstract**

*Trill is a streaming query processor that fulfills three requirements to serve the diverse big data analytics space: (1) Query Model: Trill is based on the tempo-relational model that enables it to handle streaming and relational queries with early results, across the latency spectrum from real-time to offline; (2) Fabric and Language Integration: Trill is architected as a high-level language library that supports rich data-types and user libraries, and integrates well with existing distribution fabrics and applications; and (3) Performance: Trill's throughput is high across the latency spectrum. For streaming data, Trill's throughput is 2-4 orders of magnitude higher than comparable traditional streaming engines. For offline relational queries, Trill's throughput is comparable to modern columnar database systems. Trill uses a streaming batched-columnar data representation with a new dynamic compilation-based system architecture that addresses all these requirements. Trill's ability to support diverse analytics has resulted in its adoption across many usage scenarios at Microsoft. In this article, we provide an overview of Trill: how we engineered it as a library that achieves seamless language integration with a rich query language at high performance, while executing in the context of a high-level programming language.*

## 1 Introduction

Cloud applications accumulate data from a variety of data sources, such as machine telemetry and user-activity logs. This accumulation has resulted in an increasing need to derive value in an efficient and timely manner from such data. At Microsoft, we have seen a variety of cloud applications with a diverse range of analytics scenarios:

- An application may monitor telemetry (e.g., user clicks on advertisements or memory usage of a service) and raise alerts when problems are detected.
- An application may wish to correlate live data streams with historical activity (e.g., from one week back).
- Users may wish to develop the initial monitoring query using logs, before deploying it in a real-time system. Conversely, they may want to back-test their live monitoring queries over historical logs, perhaps with different parameters (in a *what-if* style of analysis).
- Analysts may want to run relational analyses (in the form of *business intelligence* queries) over historical logs. Further, they may prefer quick approximate results by streaming the data, as that better fits an exploratory environment.

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

This diverse and interconnected nature of cloud analytics has resulted in an ecosystem of disparate tools, data formats, and techniques [13]. Combining these tools with application-specific glue logic is a tedious and error-prone process, with poor performance and the need for translation at each step. The lack of a unified data model across these scenarios precludes the ability to reuse logic, e.g., by developing queries on historical data and deploying them directly to live streams.

In order to alleviate the complexities outlined above, we built Trill [14], a single analytics engine that can serve a diverse analytics space. Trill simultaneously addresses three requirements present in the scenarios above:

- ***Query Model***: Trill is based on a unifying temporal data model based on application time, which enables the diverse spectrum of analytics described earlier: real-time, offline, temporal [7], relational, and progressive (approximate) [8] queries.

- ***Fabric & Language Integration***: Trill is written as a library in the *high-level-language* (HLL) C#, and thus benefits from arbitrary HLL data-types, a rich library ecosystem, integration with arbitrary program logic, ingesting data without "handing off" to a server or copying to native memory, and easy embedding within scale-out fabrics and as part of cloud application workflows.

- ***Performance***: Trill handles the entire space of analytics described earlier, at best-of-breed or better levels of performance. In Chandramouli et al. [14], we showed that Trill processes streaming temporal queries at rates that are 2 to 4 orders-of-magnitude higher than traditional streaming engines. Further, for the case of offline relational (non-temporal) queries over logs, Trill's query performance is comparable to modern columnar databases, while supporting a richer query model and language integration. Trill is very fast for simple payload types (common for early parts of a pipeline), and degrades gracefully as payloads become complex, such as machine learning models (common on reduced data).

Trill achieves these requirements using a hybrid system architecture that exposes a latency-throughput tradeoff to users. Users specify a latency requirement, and Trill repacks streams into a sequence of batches with a goal of meeting the requirement. Unlike other batched streaming systems, such as Spark Streaming [21], our query model allows batching to be purely physical (not commingled with application time) and therefore easily variable: query results are always identical to the case of per-event processing, regardless of batch sizes or data-arrival rates. The user's query is converted into a directed acyclic graph of streaming operators that each receive and produce streams of data batches. Further, within each batch, Trill uses a columnar data organization when possible, along with new and highly efficient columnar streaming operators that work directly on the columnar batches. Engineering such a query processor as a high-level language library introduced several challenges; this article describes how we addressed these problems as we built a generally usable engine.

- Trill operators expect data to be batched in timestamp order for high performance. On the other hand, real-time data may arrive one event-at-a-time, and may have inherent disorder. Section 4 describes our data model that makes batching a purely physical construct, and our ingress-egress design that provides users with control for handling disorder and other requirements.

- Queries in Trill are language-integrated. Users expect a powerful query language capable of both relational-style operations and temporal manipulations such as data-dependent windowing, while staying in the context of a HLL and type system. Section 3 uses a running example to describe several key Trill language elements that enable expressive query specification seamlessly in a HLL.

- Section 4 covers our design of dynamic code generation to enable user-transparent columnar batched execution in a HLL. Further, it discusses Trill's threading choices and features such as checkpointing, which are necessary to use the engine in the context of a distributed fabric for resilient real-time processing.

We conclude the article in Section 5 with a brief overview of the ways Trill is used in practice, and some lessons learned from these scenarios.

### Running Example

As our running example in this article, we consider an advertising platform that tracks advertisement (ad) impressions shown to users, and clicks on the ads. We can use a C# type to capture the event contents as below.

```
struct AdInfo {
   long Timestamp;          long UserId;
   long AdId;               bool IsClick;
}
```

Here, `IsClick` is a boolean value that denotes whether the event is a click (true) or an impression (false). We wish to ingest such a data stream arriving at Trill from diverse sources, execute a variety of temporal queries over the stream, and output results, for example, to a dashboard or console.

## 2 Trill Data Model, Ingress, and Egress

Logically, we view a stream as a temporal database (TDB) [12] that is presented incrementally [3, 11, 14]. Each event is associated with a data window (or interval of application time) that denotes its period of validity. This association creates a sequence of *snapshots* across time, where a snapshot at time $t$ is a collection of events that are valid at time $t$. The user query is logically executed against these snapshots in an incremental manner.

### 2.1 Event Representation

Consider an event with a data window of $[s, e)$. This event may arrive directly as an interval, at application time $s$. We call $s$ the *sync-time* of the interval event. Alternatively, the event may arrive broken up into a separate insert into the TDB (called a *start-edge*) at time $s$, optionally followed by a delete from the TDB (called an *end-edge*) at a later time, $e$. The start-edge and the end-edge have sync-times of $s$ and $e$ respectively. Sync-time is an important concept in Trill; it denotes the logical instant when a fact about the stream content becomes known. Events are always processed by Trill in strictly non-decreasing sync-time order (we discuss the handling of late-arriving events in Section 2.2.1). Because time in Trill is just a `long` (64-bit integer) type, we can, for example, re-interpret time to mean query progress when executing progressive relational queries [8].

`StreamEvent<T>` is a Trill struct that represents an event with payload type `T`, and includes static methods to create interval, start-edge, and end-edge events. We may also create a *point event*, an interval event with an data window of one chronon (the smallest unit of time). In our example, users may ingest clicks and impressions as point events: `StreamEvent<AdInfo>.CreatePoint(timestamp, new AdInfo { ... })`.

Further, users can ingest a special kind of event called a *punctuation*. A punctuation is associated with a timestamp $t$, and serves two purposes: (1) It denotes the passage of application time until $t$, in the absence of data, and allows operators to clean up system state; and (2) Each operator internally batches events (up to the maximum batch size) before sending the batch to the next operator. A punctuation enforces the immediate flushing of batches through Trill, to force processing and output generation until $t$.

### 2.2 Event Ingress

Data is available for querying in Trill by representing the source as an instance of a special generic interface that we call `IStreamable<T>`. This interface is Trill's variant of `IObservable<T>` [17], the standard .NET interface for pushing data. Briefly, `IObservable<T>` provides the ability for a data source to push objects of type `T` to a downstream observer `o` that "subscribes" to the observable via a `Subscribe(o)` call.

In our running example, we could create an `IObservable<StreamEvent<AdInfo>>` instance to push a sequence of individual events of type `StreamEvent<AdInfo>` to Trill as follows.

```
IObservable<StreamEvent<AdInfo>> o = Network.CreateObservable<AdInfo>(...);
```

Other ingress mechanisms supported in Trill include efficient bulk-ingress using a stream of arrays of events (`IObservable<ArraySegment<StreamEvent<T>>>`), pull-based sequences (e.g., `IEnumerable<T>`), and generic data-reader formats such as `IDataReader` [1].

We transform an instance of an input such as `IObservable<StreamEvent<T>>` into an `IStreamable<T>` using a special ingress method, defined on the `IObservable` instance, called `ToStreamable(...)`, whose parameters specify policies for ingesting data into Trill. These policies are described next.

### 2.2.1 Ingress Policies

When ingesting data into Trill from the outside world, we need to: (1) specify how to handle disorder in the stream; (2) automate the flushing of data into the system as (columnar) batches; and (3) specify system behavior when the input stream comes to an end. These transformations are driven by three user-defined policies that are provided as part of the `ToStreamable(...)` call:

- **Disorder**: Trill processes data in timestamp order for efficiency. We provide multiple ways of handling disorder, using a disorder policy. We support the policies of **adjust** (modify the late-arriving event to have the current sync-time as its timestamp), **drop** (drop the late event), and **throw** (throw an exception on encountering a late event). Further, each of these policies takes a *reorder latency* argument that is used to buffer and reorder late-arriving events within the provided reorder latency budget. Later-arriving events are handled using the specified policy of drop, adjust, or throw.

- **Flush**: The flush policy allows Trill to automate the injection of punctuation into the stream, in order to flush partially filled batches in the system. Supported policies include (1) **count**, which takes a parameter $c$ and flushes the stream every $c$ events; and (2) **time**, which takes a time duration argument $d$, and flushes the stream every $d$ units of application time.

- **Completed**: When a stream completes, we can (1) halt the query without flushing partial batches in the system; (2) flush partial batches, but not force the current sync-time to move forward; or (3) move the current sync-time to $\infty$ (possibly producing new output) and flush the system.

In our running example, we could reorder late-arriving events within a timespan of `r` units (dropping later events), and issue flushes every 1000 events, while ingesting into Trill, as follows.

```
var s0 = o.ToStreamable(OnCompletedPolicy.EndOfStream(), DisorderPolicy.Drop(r),
                        PeriodicPunctuationPolicy.Count(1000));
```

## 2.3 Query Specification and Egress

An `IStreamable` instance such as `s0` is returned by the `ToStreamable(...)` call. Trill's query specification hangs off this instance in the form of functional method invocations. Each method returns a new `IStreamable` instance, allowing users to chain an entire query plan. We describe query specification in detail in Section 3. Note that query specification itself does not start query execution; this is done by subscribing to a Trill query using a variety of techniques. A common use case is to egress results as an observable sequence of `StreamEvent<T>` instances using a `ToStreamEventObservable(...)` method. We support an optional egress policy called `CoalesceEdges`: when set, this policy indicates that Trill will coalesce start-edge and end-edge pairs into intervals before outputting them. Since Trill emits events in sync-time order, this egress policy can incur latency because output has to be held back when we encounter a start-edge, until a matching end-edge is seen (in order to construct and emit the corresponding interval event). In our running example, we could output all the events to the console (as a pass-through) as follows.

```
s0.ToStreamEventObservable().Subscribe(e => Console.WriteLine(e.ToString()));
```

## 3 The Trill Query Language, by Example

Any values of type IStreamable, such as s0, are stream endpoints over which a Trill query can be written. Trill's query language, called Trill-LINQ, is modeled after LINQ [19], with temporal interpretation of the standard relational operations, along with new operations for temporal manipulation. In this section, we cover several language constructs in Trill using our running example.

### 3.1 Filtering and Projection

Assume that we want to consider only a 5% sample of users in the stream. We use the Where operator in Trill to filter the stream as follows:

```
var s1 = s0.Where(e => e.UserId % 100 < 5);
```

The expression in parentheses is called a lambda expression [10]; it is an *anonymous function*, in this case from the type AdInfo to a boolean value specifying for each row (event) e in the stream that it is to be kept in the output stream, s1, if its UserId modulo 100 is less than 5. Each Trill operator is a function from stream to stream which allows for easy functional composition of queries.

We can also transform the data to a different type, using the Select operator to perform a projection:

```
var s2 = s1.Select(e => e.AdId);
```

In this case, the lambda expression is a function from AdInfo to long indicating how the input payload type is transformed into a new output payload type by taking the result of the previous query, s1, and dropping the fields other than AdId to form a stream with exactly one field. Thus, stream s2 has the type IStreamable<long>.

### 3.2 Windowing

Trill supports the notion of altering event lifetimes to support windowed operations and correlating data across time. In its most basic form, this is accomplished using the AlterEventLifetime operation. This operation accepts two expressions as input: a start-time selector which maps an interval's start-time to a new start-time, and a duration selector, which maps a start-time and end-time to a new duration. We limit timestamp modifications to those that preserve output sync-time order. Trill also provides macros that allow users to easily create hopping, tumbling, and sliding windows using AlterEventLifetime and its variants such as AlterEventDuration, which serves to alter an event's duration, leaving the start-time unmodified. For example, we can create a 5-minute tumbling window over the (sampled) stream s1 as follows.

```
var s3 = s1.TumblingWindow(fiveMinutes);
```

### 3.3 Aggregation

Aggregation in Trill is done using an operator framework called *user-defined snapshot*, which enables the integration of custom incremental HLL logic into stream processing without sacrificing performance. It handles the class of operations that incrementally compute a result per time snapshot. In fact, all our built-in aggregates (including complex multi-valued aggregates such as top-k) are implemented using this general framework, described in Chandramouli et al. [14]. For example, we can compute a 5-minute tumbling window count of events using s3, as follows.

```
var s4 = s3.Aggregate(w => w.Count());
```

We also support the simultaneous application of multiple aggregates in a single snapshot operator, with the ability to combine results on a per-snapshot basis (see Chandramouli et al. [14] for details).

## 3.4 Grouped Computation

Trill supports a GroupApply operation, where the user specifies a grouping key selector and a sub-query. Logically, GroupApply executes the given sub-query on each sub-stream corresponding to each distinct key, as determined by the grouping key selector. For example, we could compute the five-minute tumbling window count on a per-ad basis as follows:

```
var s5 = s1.GroupApply(e => e.AdId,
                       s => s.TumblingWindow(fiveMinutes)
                             .Aggregate(w => w.Count()),
                       (g, p) => new { AdId = g, Count = p });
```

Here, the first lambda expression specifies the grouping key, and the second lambda expression specifies the query to be executed per key. The final lambda allows the user to combine the grouping key and the per-group payload into a single result payload.

## 3.5 Correlation and Set Difference

The temporal join operator in Trill allows one to correlate (or join) two streams based on time overlap, with an (optional) equality predicate on payloads. Suppose we wish to augment the filtered AdInfo stream s1 with additional information from another *reference* stream ref1 that contains per-user demographics data such as age. We would express such a query in Trill as follows:

```
var s6 = s1.Join(ref1, l => l.UserId, r => r.UserId,
                 (l, r) => new Result { l.AdId, l.UserId, r.Age });
```

The second and third parameters to Join represent the equi-join predicate on the left and right inputs (UserId in this case), while the final parameter is a lambda expression that specifies how matching input tuples (from the left and right) are combined to construct the result events of payload type Result, yields a stream of type IStreamable<Result>. As a more complex example, suppose we wish to join ad impressions to clicks on the same ad, and by the same user, within 10 minutes. This query is written as:

```
var s7 = s1.GroupApply(e => new { e.UserId, e.AdId },
                       s => s.Where(e => !e.IsClick)
                             .AlterEventDuration(tenMinutes)
                             .Join(str.Where(e => e.IsClick), (l, r) => r),
                       (g, p) => p);
```

Trill also support a temporal set difference operator called WhereNotExists. For instance, we can output all clicks that were not preceded by an impression within 10 minutes, as follows:

```
var s8 = s1.GroupApply(e => new { e.UserId, e.AdId },
                       s => s.Where(e => e.IsClick)
                             .WhereNotExists(str.Where(e => !e.IsClick)
                                                .AlterEventDuration(tenMinutes),
                                             (l, r) => r),
                       (g, p) => p);
```

## 3.6 Data-Dependent Windowing

Trill supports the creation of windows based on data. Such windows can, for instance, be used to create *session windows* that limit an event's influence to the end of the session. For example, suppose we wanted to take impressions and restrict their lifetime to be either 10 minutes or the first click after the impression, whichever comes earlier. We express this query using the ClipEventDuration operator, which *clips* the duration of an event $E$ to end at the start-time of the first matching event on the right-side input that falls within $E$'s time interval.

```
    var s9 = s1.GroupApply(e => new { e.UserId, e.AdId },
                           s => s.Where(e => !e.IsClick)
                                .AlterEventDuration(tenMinutes)
                                .ClipEventDuration(str.Where(e => e.IsClick),
                                                   (l, r) => r),
                           (g, p) => p);
```

## 4 Internal Architecture

### 4.1 Batching with Columnar Organization

As mentioned earlier, we physically batch events before feeding them to Trill, based on the user-specified latency requirement. Batches allows system overhead to be amortized over many events. While batching is advantageous in its own right, it enables us to re-organize data within batches. We store batch content in columnar format. A *columnar batch* (referred to hereafter just as *batch*) is a structure of that holds one array for each column in the event. For example, one array holds the sync-time values for all events in the batch, while another array holds a second timestamp associated with events (called the *other-time*). Internally, every event is associated with a *grouping key* in order to enable efficient grouped operations. We precompute and store the grouping key (and its hash) as two additional arrays in the batch. We also include an *absentee bitvector* to identify which rows in the batch are currently active. The bitvector allows filter operations to logically remove rows without having to physically reorganize the batch. For instance, the `Where` query in Section 3 just sets the bit corresponding to each row for which the function returns false.

Being in a high-level language, we use the generic type system to get strong type safety for batches expressed over the two types `K` and `P` for the key type and payload type, respectively.

```
class Batch<K,P> {
   long[] SyncTime;         long[] OtherTime;
   K[] Key;                 int[] Hash;
   P[] Payload;             long[] BitVector;
}
```

As in database systems, columnar representation results in better data locality, bringing much less data to the CPU. Further, we are able to use a custom memory allocation scheme for the arrays: for instance, the output batch of a selection operator does not modify the sync-time of each event and so can share a reference to that array with the input batch. We aggressively pool arrays using a global memory manager to alleviate the cost of memory allocation and garbage collection. In a streaming setting, the system quickly achieves a steady state with the memory allocated for output batches being reused for succeeding input batches.

Note that the payload of each event above remains a row structure. For instance, the example of Section 3 results in the `Payload` array being of type `AdInfo`[1]. This means that operators accessing very few fields of the payload may not enjoy the data locality that is provided by the columnar layout of the other fields. Each operator in Trill has an implementation that executes against this representation. We call them the *row-based* operators since the payloads exist as an individual instance per row.

[1]In .NET, since that type was defined as a `struct` — a value type — the array is physically laid out in memory as a contiguous sequence of bytes. However, if it were defined as a `class` — a reference type — then the array would be a contiguous sequence of pointers, with the storage for each instance individually allocated somewhere in the heap.

## 4.2 Code Generation

We can adopt a columnar data layout for payload fields as well, by allocating a separate array for each field in the payload. For the type AdInfo, we have three arrays of long and one array of bool:

```
class ColumnarBatchForAdInfo<K> : Batch<K, AdInfo> {
   // Other arrays inherited from Batch<>, Payload array ignored in base class
   long[] Timestamp;       long[] UserId;
   long[] AdId;            bool[] IsClick;
}
```

With this representation, an operator that accesses a single field will result in contiguous memory loads for that field alone. If a payload type cannot be made columnar (e.g., it is a class with private fields), we revert to the data format described in Section 4.1.

Note that there is an impedance mismatch between the user's view of the data — the type AdInfo available at compile-time — and the system's view — the type ColumnarBatchForAdInfo — which is not available at compile-time. Since queries and data are dynamic, i.e., a new query expressed over a new schema (payload type) is not predefined, the system must be able to create the generated types and operators that use those types during runtime. We solve this problem using *dynamic code generation* to create new type definitions, e.g., ColumnarBatchForAdInfo for batches, and optimized *columnar operators* that are aware of the columnar representation, and inline operations on the columnar format. Columnar organization also enables optimized serialization and string handling; see Chandramouli et al. [14] for details. These transformations are transparent to users, who continue to operate with their row-based data model. For example, the Select operator generated for the example in Section 3 computes the single payload column in each output batch in the stream s2 in constant time; we simply copy the single pointer to the AdId column from the input batch of stream s1.

We use T4 [15], a text-templating system in Visual Studio, to create the C# source file for batch types and operators. The source file is compiled, and the dynamic loading facilities of the .NET runtime are used to load and instantiate the types. This technique also allows us to put breakpoints and debug generated code easily. We cache and re-use generated types to reduce the overhead of code generation and compilation. Because we use C# source to define the generated code, we need a way to translate user expressions such as Where predicates into inlined C#. An expression is passed to Trill as an *expression tree*, a .NET object model for representing code [10]. Expression trees do not provide a conversion to C# source, since there exist expression trees for which such a translation cannot be done. However, since we are willing to accept a best-effort solution, we wrote our own translator, which is now in use as a stand-alone component for other projects as well.

Columnar execution is best-effort; if we encounter a situation where an operator or type cannot execute in columnar mode, we process the data in row-mode (see Section 4.1). If an expression cannot execute in columnar mode (e.g., it invokes a black-box method), we reconstitute rows on-the-fly to invoke the method. If necessary, we insert a *col-to-row* operator into the query plan. This generated operator converts columnar payloads back into a single column of payload instances for downstream row-based operators. Users are notified when such fallback occurs, so they can try to modify their query or data to remain in the more efficient columnar mode.

## 4.3 Other Details

**Threading** By default, Trill does not create any threads: it accepts data on the thread that pushes the data into Trill's ingress methods and executes all operators on that thread until the output (if any) is produced and the call stack is unwound. The one exception is that, depending on a user-configurable option, Trill will use separate threads for scaling operators across multiple cores on a single processor.

**Checkpointing** We support a client's need for resiliency by offering a synchronous checkpointing service. Under user control, the internal state of a running query can be persisted. The query can later be resumed by loading this state back, possibly on a different machine, and replaying data received since the checkpoint. In conjunction with Trill's threadless library mode, checkpointing allows Trill to fit in with the existing resiliency solutions of distributed fabrics. The fabric can decide whether it replays events exactly from the checkpoint position for correctness, or resumes from a later (e.g., current) stream position, tolerating the resulting inaccuracy.

## 5 Usage Scenarios and Lessons Learned

### 5.1 Usage Scenarios

Trill is being used today in diverse scenarios that serve to illustrate how performance, fabric and language integration, and query model enabled Trill to support a diverse range of use cases.

- Orleans-hosted real-time: Orleans [4] is a programming model and fabric that enables low-latency distributed computations with units of work called grains. Orleans owns threads and manages distribution, while Trill is used as a library to express streaming queries as part of users' grain code.

- Analytics Back-End: Trill is used as a building block for several analytics services. Tempe [9] is a Web-based interactive analytics environment that allows users to author and visualize queries over real-time and offline streams. It uses Trill to run temporal and progressive relational queries. We recently described how the Halo team used Tempe and Trill to quickly analyze large amounts of real-time customer data for hunting down bugs [18]. Azure Stream Analytics [2] is a Cloud service that uses Trill as a query processor [20]. SCOPE [5] is a map-reduce platform that allows arbitrary .NET code as custom reducers. As with Orleans, SCOPE owns threads and schedules reducer code; thus, analysts can embed Trill as a library within their reducers in order to perform temporal analytics [7]. Recently, we also reported on the use of Trill with a streaming version of SCOPE to reduce the latency of Bing Ads reporting [16].

- Monitoring Server: Trill is used to monitor system logs generated by machines in a data center, and visualize real-time performance. Here, Trill is used as a server that processes data from multiple sources in close to real-time (several seconds of latency).

- Trace-Log-Analysis Tools: A large number of time-oriented traces are generated by applications and operating systems. Trill is used as part of stand-alone tools and Cloud services, to allow users to analyze such offline traces, for example, to detect anomalies or complex patterns.

### 5.2 Lessons Learned

We have learned several things from building Trill and interacting with its users. Our prior work [7, 8] showed that a single model could, in theory, handle a diverse range of analytics scenarios. However, users chose to use specialized systems for performance reasons, which led us to re-examine streaming engine architectures with a goal of achieving best-of-breed or better performance across the latency spectrum.

In addition, a key design decision was to create a *library* instead of a *server*. Implementing Trill as a HLL library meant that it could be immediately integrated into diverse environments, each of which had its own policies on thread management, distribution, scheduling, resiliency, and resource utilization. By default, Trill is passive and performs work only on the thread that feeds data to it. This choice also simplified Trill's implementation considerably, since we could focus on efficient query processing. Subsequently, we created a lightweight scheduler that takes a user-specified set of threads to efficiently use multiple cores on a machine. With this scheduler, we made it easy to build servers using the Trill library as well.

Another crucial aspect was to directly support the HLL data model that users wish to analyze in. For instance, users often wish to stream complex data-types such as dictionaries and machine learning models through Trill. We extended LINQ to make query specification and execution a seamless part of user programming, and our powerful query language is able to express a wide variety of data processing tasks. Further, columnar batching and code generation needed to be automated and done under the hood, to avoid complicating the user experience.

Finally, none of these decisions would have induced users to adopt Trill as enthusiastically as they have, if it did not work at extremely high speeds. Getting high performance meant starting with a simple `for` loop with an inlined predicate, and working our way out, ensuring that performance was not lost at any step along the way. Once the overall system architecture was decided, it was crucial to observe the resulting design patterns throughout all system components. For example, using custom memory management for the strategic data allocations of batches and columns, restricting the operations performed in the tight loops within each operator, and creating custom data structures (such as hash tables) for optimizing the memory-usage of stateful operators, were all critical to achieving and retaining high performance.

## References

[1] ADO.NET DataReader. `http://aka.ms/datareader`. Retrieved 10/14/2015.

[2] Azure Stream Analytics. `https://azure.microsoft.com/en-us/services/stream-analytics/`. Retrieved 10/14/2015.

[3] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.

[4] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical report, Microsoft Research, 2014.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.

[6] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4), 2014.

[7] B. Chandramouli, J. Goldstein, and S. Duan. Temporal Analytics on Big Data for Web Advertising. In *ICDE*, 2012.

[8] B. Chandramouli, J. Goldstein, and A. Quamar. Scalable Progressive Analytics on Big Data in the Cloud. *PVLDB*, 6(14), 2013.

[9] R. DeLine, D. Fisher, B. Chandramouli, J. Goldstein, M. Barnett, J. F. Terwilliger, and J. Wernsing. Tempe: Live scripting for live data. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, 2015.

[10] Expression Trees. `https://msdn.microsoft.com/en-us/library/bb397951.aspx`. Retrieved 10/14/2015.

[11] M. A. Hammad et al. Nile: A query processing engine for data streams. In *ICDE*, 2004.

[12] C. Jensen and R. Snodgrass. Temporal specialization. In *ICDE*, 1992.

[13] H. Lim et al. How to fit when no one size fits. In *CIDR*, 2013.

[14] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of data streams and operators. In *ICDT*, 2005.

[15] Microsoft Visual Studio T4 Template System. `http://aka.ms/eeg4w5`. Retrieved 10/14/2015.

[16] Now Available in Bing Ads: Campaign Performance Data in Under an Hour. `http://aka.ms/bing-trill`. Retrieved 10/14/2015.

[17] Reactive Extensions for .NET. `http://aka.ms/rx`. Retrieved 10/14/2015.

[18] The high-tech research behind making Halo 5: Guardians multiplayer better for gamers. `http://aka.ms/fenfxy`. Retrieved 10/14/2015.

[19] The LINQ Project. `http://tinyurl.com/42egdn`. Retrieved 10/14/2015.

[20] Trill Moves Big Data Faster, by Orders of Magnitude. `http://aka.ms/w6y2kt`. Retrieved 10/14/2015.

[21] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*, 2013.