The Trill Incremental Analytics Engine

Badrish Chandramouli, Jonathan Goldstein, Mike Barnett,
Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, John Wernsing
Microsoft Research[1]
{badrishc, jongold, mbarnett, rdeline, danyelf, jplatt, jamest, johnwer}@microsoft.com

ABSTRACT

This technical report introduces Trill – a new query processor for analytics. Trill fulfills a combination of three requirements for a query processor to serve the diverse big data analytics space: (1) *Query Model*: Trill is based on a 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 today's comparable streaming engines. For offline relational queries, Trill's throughput is comparable to a major modern commercial columnar DBMS.

Trill uses a streaming batched-columnar data representation with a new dynamic compilation-based system architecture that addresses all these requirements. In this technical report, we describe Trill's new design and architecture, and report experimental results that demonstrate Trill's high performance across diverse analytics scenarios. We also describe how Trill's ability to support diverse analytics has resulted in its adoption across many usage scenarios at Microsoft.

1. INTRODUCTION

Modern businesses accumulate large amounts of data from various sources such as sensors, devices, machine logs, and user activity logs. As a consequence, there is a growing focus on deriving value from the data by enabling timely analytics. In practice, big data analytics requires a diverse range of *types* of analytics, with a variety of *latency settings* over which the analytics is applied:

1) Real-time streaming queries: These include queries on real-time data, which may reference slow-changing data such as social network graphs or data from data markets [14]. For example, notify a smartphone user if any of their Facebook friends are nearby.

2) Temporal queries on historical logs: This includes back-testing streaming queries on historical logs; e.g., compute the average click-through-rate of ads in a 10-minute window, on a 30-day log.

3) Progressive relational queries on collected data: Data scientists perform a series of interactive exploratory queries over logs to better understand the data. Computing *progressively*, i.e., providing immediate early results on partial data and refining as more data is streamed in, allows productive and cost-effective exploration.

These analytics are interconnected: for instance, queries may correlate real-time with historical logs, or real-time data may be logged for progressive analysis using an interactive tool. The diverse and interconnected nature of analytics has resulted in an ecosystem of disparate tools, data formats, and techniques [13]. Combining these tools with application-specific glue logic in order to execute end-to-end workflows is a tedious and error-prone process, with poor performance and the need for translation at each step. Further, the lack of a unified data model and semantics precludes reusing logic across tools, developing queries on historical data and then deploying them directly to live streams.

1.1 Requirements for Diverse Analytics

We identify three key requirements for an analytics engine to successfully serve this diverse environment (here, we focus on the above-mentioned analytics types and settings; other requirements such as graph analytics are interesting areas for future work):

1) Query Model: Existing analytics engines either target a specific point in the diverse analytics space (e.g., DBMS for offline relational) or expose low-level APIs (such as an incremental key-value abstraction [29][38]) that place the burden of specifying non-declarative logic on the application developer.

The *tempo-relational (temporal) query model* [31][1] conceptually unifies the diverse analytics space. Briefly, this model represents datasets as a time-versioned database, where each tuple is associated with a validity time interval. Temporal datasets are presented as an incremental stream to a temporal *stream processing engine* (*SPE*) that processes a query incrementally to produce a result temporal dataset. We can use an SPE to: (1) deploy continuous queries across real-time streams and historical data; (2) back-test real-time queries over historical logs; and (3) run relational or temporal queries over log data. Recently, we have also shown how a temporal SPE can handle progressive relational queries, by using time to denote query progress [2].

Some SPEs such as NiagaraST [9] and StreamInsight [8] support a full tempo-relational algebra, whereas other SPEs such as Spark Streaming [34] and Naiad [35] support limited variants of the model. But today's SPEs fall short as unified analytics engines, because they lack fabric integration and high performance across the diverse analytics space.

2) Fabric & Language Integration: Analytics workflows today are driven by an application, which uses the engine either directly or via a combination of *distribution fabrics* (such as Storm [29], YARN [17], and Orleans [16]) for different parts of the pipeline. To enable integrated execution, an analytics engine must be usable as a library from a hosting *high-level language* (*HLL*). HLLs such as Java and C# provide a rich universe of data-types, libraries, and custom logic that needs to integrate seamlessly with the engine.

[1] Microsoft Research Technical Report MSR-TR-2014-54, April 2014.

Table 1: Desirable features in existing systems and Trill.

Require-ment	Feature	Stream Processing Engines			Columnar Databases	Trill
	Examples →	Stream-Insight, STREAM	Storm	Naiad, Spark Streams	Vertica, Shark, SQL Server	
Query Model	Temporal	Yes	Yes	Yes	No	Yes
	Incremental	Yes	Yes	Yes	No	Yes
Fabric & Language Integration	HLL Integration	Some	No	Yes	No	Yes
	Library on Fabrics	No	No	No	No	Yes
Perfor-mance	Throughput	Low	Low	Mid	High	High
	Batched Ops	No	No	Yes	Yes	Yes
	Latency Spectrum	Yes	No	No	No	Yes
	Columnar	No	No	No	Yes	Yes

DBMSs provide very high performance, but use a server model over a restricted universe of SQL *data-types* (e.g., int and bigint) and *expressions* (e.g., a filter predicate A < 10), with limited support for richer logic via integration mechanisms such as SQL CLR [36]. Spark [28] integrates with Scala, but exposes a multi-node server model. StreamInsight uses *language-integrated queries* (*LINQ*) [32] for seamless query specification from a HLL, but follows a server model and restricts data-types. Naiad [35] uses LINQ and processes arbitrary HLL data-types and expressions, while incremental key-value engines such as Storm expose a low-level key-value-based API with rich data-type support. But these systems lack performance and a declarative query model. One could build a declarative operator layer over such systems, but this layered approach further impacts performance.

3) Performance: High performance is a critical requirement for analytics. Specifically, we need an engine to automatically and seamlessly adapt performance in terms of latency and throughput, across the analytics spectrum from offline to real-time.

Figure 1 depicts single-machine throughput on today's engines, for a simple filter query on an in-memory dataset (see §7 for workload details). SPE-X and DB-X represent a modern commercial SPE and columnar DBMS respectively. We see that today's SPEs have lower throughput (by 500X or more) than modern columnar DBMSs such as Vertica [14], SQL Server [6], and Shark [28] that push the limits of relational performance, approaching memory-bandwidth speeds for common operations. However, these DBMSs lack rich HLL data-type, expression and efficient HLL library support. Further, they use the non-incremental model, which targets a specific (offline relational) point in the analytics space.

To summarize, these capabilities – rich query model, fabric and language integration, and high performance – appear to fundamentally be at odds in today's systems, as seen in Table 1.

1.2 Today's Engine Architectures

To understand why these requirements are not simultaneously addressed by today's systems, we start by classifying existing engine architectures into three categories: event-at-a-time, batch-at-a-time, and offline. These are shown in Figure 2(a)-(c); their throughputs are shown in Figure 1. Low latency motivated the traditional event-at-a-time architecture of SPEs such as SPE-X, but this limits throughput to very low levels. Naiad [35] processes events one batch at a time, which provides better throughput. However, we notice that offline DBMSs still provide significantly higher throughput (by ~500X) than batch-at-a-time SPEs.

The reason for this vast performance difference is that language integration in systems such as Naiad precludes the use of efficient DB-style data organizations such as columnar, i.e., user expressions are evaluated as black-boxes over individual rows. Further, the end user has to manually navigate the latency spectrum by selecting individual batch sizes. Finally, temporal operators have to be written as a layer outside the engine, and thus cannot be optimized for performance. On the other hand, relational engines support only the SQL model over offline data with high latency, and do not provide deep fabric or language integration.

1.3 A New Hybrid System Architecture

We introduce Trill (for a *trillion events per day*), a new analytics engine that addresses all these requirements:

1) Query Model: Trill is based on the temporal logical data model, which enables the diverse spectrum of analytics described earlier: real-time, offline, temporal, relational, and progressive.

2) Fabric & Language Integration: Trill is written as a library in an 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 easily embedding within scale-out fabrics and as part of a Cloud application workflow.

3) Performance: Trill handles the entire space of analytics described earlier, at best-of-breed or better levels of performance (see Figure 1). With temporal queries over streaming data, Trill processes events at rates that are *2-4 orders-of-magnitude higher* than existing commercial streaming engines. Further, for the case of offline relational (non-temporal) queries over logs, Trill's query performance is *comparable to* a modern columnar DBMS, 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 all these requirements using a hybrid system architecture – see Figure 2(d) – that combines novel ideas and key prior ideas from specific points in the analytics spectrum:

1) Support for Latency Spectrum (§3): Trill queries consist of a DAG of operators that process a stream of *data-batch* messages. Each data-batch consists of multiple events carefully laid out in timestamp order in main memory. We find that batching is useful in an SPE to improve throughput, particularly when combined with engineering practices we report here, such as a very careful organization of inner per-batch loops in operators. Critically, unlike other batched streaming systems such as Spark Streaming [34], our temporal 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.

While batching provides high throughput, it may result in low and unpredictable latency which can be unacceptable in a streaming setting. To solve this, Trill supports a new form of *punctuations*, which allow users to control desired latency. Punctuations work alongside batching to transparently *tradeoff throughput for latency*. In Trill, for a user-specified latency, higher input loads result in larger batches that provide better throughput, which in turn allows the system to better handle the increased load.

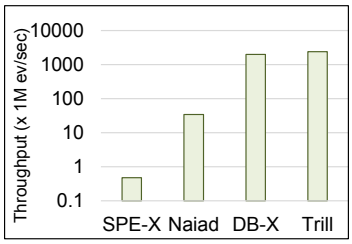

Figure 1: Single-machine filter throughput; different engines.

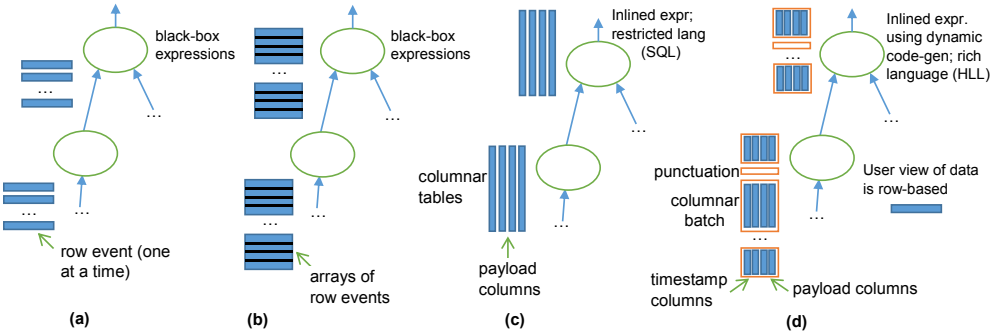

Figure 2: System architectures: (a) Traditional SPE; (b) Naiad; (c) Columnar DB; (d) Trill's hybrid architecture.

2) Columnar Processing in a High-Level Language (§4): Systems like Naiad and Spark Streaming batch data, but in order to reach the performance of modern DBMSs, Trill uses a *columnar data organization* within batches. We adopt and extend columnar techniques [14][12][6] and apply them over temporal data. Our control fields (e.g., timestamps) are also columnar, so we pay the cost of temporality only when necessary.

Critically, in order to benefit from columnar processing (proven by DBMSs) in an HLL, we use a novel *dynamic HLL code generation* technique that constructs and compiles on-the-fly batches and operators in the HLL, all of which operate over columnar batched data. Both columnarization and batching are transparent to users, who program over the usual row-oriented view of data streams. To achieve this, we leverage the abstract syntax trees of *lambda expressions* [21] (available in today's HLLs) to interpret and rewrite user queries (such as select expressions) into inlined columnar accesses inside tight per-batch loops, with few sequential memory accesses and no method calls inside the loop.

Dynamic HLL code generation also enables us to (1) handle strings more efficiently by storing them as character arrays within batches, and rewriting user expressions to operate directly over these arrays; and (2) enable fast serialization by sending columns over the wire without any fine-grained encoding or decoding, which provides a 10X benefit over standard HLL serialization schemes such as Avro.

3) Fast Streaming Operators (§5): Trill exploits the coarse-grained columnar nature of data-batches and the timestamp-order of data via a set of new algorithms for streaming operators. We propose a powerful grouped *user-defined aggregation framework*; it uses an expression-based user API that lets user-defined extensions achieve performance similar to hand-written custom logic. In fact, our built-in aggregates in Trill are written using the user-defined framework.

Trill uses a new *stream property derivation framework* (§5.3) that leverages data characteristics to select from a small set of generated physical operators at compile-time. For instance, operators over progressive queries do not need to handle event removal at input.

4) Library Mode & Multi-core (§6): By default, Trill queries run only on the thread that feeds data to it. This "pure library" mode makes Trill ideal for embedding within frameworks such as Orleans [16] and YARN [17]. For higher performance on multi-core, Trill supports a new two-level streaming *temporal map-reduce* operation, executed using a lightweight optional scheduler.

Detailed experiments (§7) comparing Trill to a commercial DBMS engine and a commercial SPE over real and synthetic data demonstrate Trill's high performance across various settings and its utility for in-memory interactive analytics. Trill is being used extensively within Microsoft – §7.6 overviews the broad range of usage scenarios we have encountered in practice. Finally, we note that while Trill is written in C#, its architecture applies to other HLLs such as Java, which have rich libraries that need to be usable in a big data analytics setting.

2. SYSTEM OVERVIEW WITH EXAMPLE

Consider a stream of user activity in terms of ad clicks, where each event is a HLL data-type:

```
struct UserData {
   long ClickTime; // Time of click on advertisement
   long UserId;    // ID of user who clicked on ad
   long AdId;      // ID of the advertisement
}
```

The application wishes to compute, for each ad, a 5-minute windowed count of clicks on that ad, across a 5% sample of users, with a tolerable latency of 10 seconds.

2.1 User Experience

Users can ingress data into Trill from a variety of sources: real-time push-based sources; datasets cached in main memory; or data streamed from a file or network. As part of ingress, the user specifies a desired latency requirement (time) as an ingress policy. Further, they need to identify the *application time* field in the data for their query logic. For example, the user may create a stream endpoint as:

```
var str = Network.ToStream(e => e.ClickTime, Latency(10secs));
```

Next, the query logic is written in Trill's temporal LINQ language:

```
var query = str.Where(e => e.UserId % 100 < 5)
.Select(e => { e.AdId })
.GroupApply(e => e.AdId, s => s.Window(5min).Aggregate(w => w.Count()));
```

This query first runs a filter (`Where`) to sample users. The argument to `Where` in parentheses is called a *lambda expression* [21]; it represents an abstract syntax tree of the logical operation to be performed for each row (event) of type `UserData` in the stream to determine if it is dropped. Here, events with `UserId % 100 < 5` are retained by the filter. Filter is followed by a projection (`Select`) to drop all columns except `AdId`, and a grouped operation (`GroupApply`) whose first argument is the grouping key (`AdId`) and the second is the per-group operation (windowed count aggregate). Note that the window argument is in application time, i.e., query semantics and results are unaffected by the latency specification.

Finally, the result can be "subscribed" to by any listener as follows:

```
query.Subscribe(e => Console.Write(e)); // write results to console
```

A full description of the Trill programming surface is outside the scope of this technical report, but we note that it supports all the

well-known streaming operators, special operations to manipulate time, as well as a high-performance extensibility framework (see Section 5).

2.2 System Overview and Challenges

We compile the user query using standard techniques [4] into a DAG of streaming query operators. Each operator processes and produces a stream of *messages*, the unit of granularity that flows through Trill. Data is pushed to Trill from external sources, either as fine-grained events or directly as batches of events. Trill batches the data at ingress into messages, if needed, and pushes them through the operators. The output may be consumed directly as messages (for example, to serialize and write to disk or send over the network) or parsed into fine-grained events for human consumption. The key challenges and resulting design decisions are summarized below and discussed in the rest of the technical report.

2.2.1 Support for Latency Spectrum (Sec. 3)

Trill uses an adaptive physical batching model to support the latency spectrum from real-time to offline. We support two kinds of messages: *data-batch* and *punctuation*. A data-batch is simply a variable-sized batch of events, whereas a punctuation is a control message that forces Trill to produce output, terminating batches if necessary. Batching and punctuations help Trill provide high performance across the entire latency spectrum. In our example, the user specification of 10secs latency results in Trill batching events for at most 10secs, at which point a punctuation is inserted into the stream to force batches through the system and generate output. We discuss adaptive batching with punctuations in Section 3.

2.2.2 Enabling Columnar with a HLL (Sec. 4)

As we saw in Section 1, batching alone does not bridge the gap between databases and streaming engines. We need to organize the data within batches in a columnar format, so that only the relevant data is accessed during query processing. In our running example, the first Where should read only the column corresponding to UserId from main memory and write out a bit-vector per batch to indicate tuples that pass the filter. Similarly, the Select should be a constant-time operation per batch that simply drops all payload columns in the batch other than AdId.

The challenge is to leverage a columnar organization in operators, while providing a row-oriented user view of data in a HLL. Section 4 introduces our solution: dynamic HLL code generation. This technique speeds up operators by rewriting users' row-oriented lambda expressions into columnar accesses inside tight per-batch loops. The loops usually have few sequential memory accesses and no method calls. Section 4 also presents several other ways in which we exploit code generation for performance within the HLL.

2.2.3 Fast Grouped Streaming Operators (Sec. 5)

Our running example next needs to compute a windowed count per AdId. Trill supports a *GroupApply* operator [1], which can execute any sub-query for each logical grouping key. We need to support efficient grouping and design new efficient algorithms for temporal operators that fully exploit the batched nature of input and output streams and can be used across temporal and progressive relational operations. We also introduce new stream properties for exploiting data characteristics to select from a small set of generated physical operators at compile-time. For instance, the Count operator in our example can exploit the fact that window sizes are constant, and can therefore expire windows in the order that data is received.

2.2.4 Library Mode and Multi-Core Support (Sec. 6)

We often need Trill to work as a pure library that does not own its own threads; for instance, when Trill is used inside fabrics that manage their own threads. We provide a no-scheduler library mode where processing occurs only on a thread that pushes data to Trill. Other situations require Trill to provide high performance by using all cores – here we use a novel temporal map-reduce operation that is executed on multiple cores using a lightweight scheduler that is configured by the application. In our running example, this facility allows us to scale out both the stateless operations (where and filter) and the grouped windowed count to use all the cores on a machine.

3. SUPPORT FOR LATENCY SPECTRUM

As described earlier, Trill's operators process a stream of messages, which can either be data-batches or punctuations.

Data-batches A data-batch represents a batch of events in Trill. Each event in the data-batch consists of a payload and two timestamps: (1) *sync-time*, the logical time at which the event occurs; (2) *other-time*, an additional timestamp, discussed in Section 5 that indicates the extent of the data window. Sync-time is an important concept in Trill; it denotes the logical instant when a fact about the stream content becomes known. Events in a batch occur in strictly non-decreasing sync-time order.

Data-batches allow Trill to tailor throughput based on desired latencies, exploiting the fact that larger batches lead to better throughputs. Thus, offline relational queries over offline data use larger batches, up to a maximum batch size; whereas, progressive and real-time queries select batch sizes based on the desired interactivity or acceptable result latency (delay), specified by users.

Punctuations A punctuation is a control message with a timestamp T, based on the user-provided latency specification. A punctuation serves two purposes: (1) it denotes the passage of application time until T, in the absence of data, in order to clean up system state; and (2) it enforces a flushing of data-batch messages through Trill, to force processing and output generation until T. Each operator internally batches events (up to the maximum batch size) before sending the batch to the next operator. Punctuations "kick" the system into producing output, which may involve pushing out partially-filled data batches.

Trill injects punctuations based on the user-specified latency (10 seconds in our running example), which allows us to dynamically adapt batch sizes to latency requirements. There is a maximum batch size, and the stream may contain multiple batches between two punctuations. Interestingly, for a given latency specification, a higher input event rate (e.g., during periods of heavy load) results in larger batches, which in turn increases system throughput to better handle the higher load. This form of adaptive batching enables us to use the same engine across a wide range of latency requirements, from real-time to offline.

Finally, we note that our temporal semantics ensure that batching is purely physical: it affects only the physical observed latency and not the logical query results, which depend only on the data (with timestamps) and the query.

4. ENABLING COLUMNAR WITH A HLL

While the end-user view of data is row-based, our data-batches internally store both control fields and payload fields as columns. Specifically, each data-batch contains the following arrays:

1. *SyncTime*: This is an array of all the sync-times in the batch.
2. *OtherTime*: This is an array of other-time values in the batch.

3. *Bitvector*: This is the "event absence" vector – an array with one bit per event. A bit value of 0 (or 1) indicates whether the corresponding data event exists (or is absent). Our micro-benchmarks showed that one can perform more than 1 billion bitvector tests or sets per second per core. The bitvector allows efficient operator algorithms in many cases by avoiding the unnecessary movement of data to/from main memory. For example, a Where operator can apply the predicate and if the predicate fails can set the corresponding bitvector entry to 1.

These fields are organized into a base class as follows:

```
class DataBatch {
      long[] SyncTime;
      long[] OtherTime;
      Bitvector BV;
}
```

The payload in Trill is also organized in columnar format, by generating (and compiling on-the-fly) a new HLL class that extends `DataBatch` and adds one array field for each field in the payload. For example, in case of the `UserData` payload in our running example, we generate a class that looks like:

```
class UserData_Gen : DataBatch {
      long[] col_ClickTime;
      long[] col_UserId;
      long[] col_AdId;
}
```

Trill supports arbitrary HLL types as payloads. If we cannot generate a columnar representation for a given payload type, we revert to a non-generated data-batch with a generic payload field (`TPayload[] Payload`), where `TPayload` is the payload type.

4.1 Generating Operators

In order to process generated data-batches, the operators themselves need to be generated since they compute over columns. The query compiler inspects each operator's input and output types and its user-provided lambda expressions to generate a carefully tailored batch-oriented operator. These generated operators are chained together to form the query DAG to which user data is pushed.

Our transformation, in general, is to replace all references to a field f with references to col_f[i], the i^{th} row in the column corresponding to field f. We describe this process for the initial `Where` and `Select` operators in our example, where we exploit the semantics of the operation and the input lambda expressions to achieve very high performance. The subsequent operations are covered in Section 5.

4.1.1 Where (Filtering)

Consider the first operation in our example query: `Where(e => e.UserId % 100 < 5)`. This filtering operation is compiled into a *custom operator*, a code module that is compiled and loaded dynamically. The argument to Where is a lambda expression as discussed earlier. We convert the body of the function so that it operates over the column-oriented view of the data and construct a Where operator with the resulting code inlined inside a tight loop that iterates over the entire data-batch. For each entry in the data-batch, we check if the bitvector is 0 – if yes, we apply the filter (inlined into the loop) and if the filter does not pass, we set the bitvector entry to 1. A final `On()` call sends the result batch to a downstream operator. The pseudo-code for `Where` for our example is shown below:

```
void On(UserData_Gen batch) {
  batch.BV.MakeWritable(); // bitvector copy on write
  for (int i=0;i<batch.Count; i++)
     if ((batch.BV[i]==0) &&
           !(batch.col_UserId[i] % 100 < 5))
       batch.BitVector[i] = 1;
  nextOperator.On(batch);
}
```

Note that it is not always possible to generate a columnar operator. For example, a filter might invoke a black-box method on each instance of `UserData`. In this case, we transform the data to its row-oriented form using a ColumnToRow() operation, and use the non-generated static (generic) definition of the operator that executes the black-box filter expression directly over elements of the `UserData`[] column in non-generated input data-batches.

4.1.2 Select (Projection)

The argument to `Select` is an expression that transforms a value of type `TPayload` into a value of a new return type `TResult`. Apart from converting the expression into inlined accesses on input and output columns, we optimize the handling of selection predicates that select a subset of input fields, so that they are constant-time operations at the batch level instead of having to iterate over each row. We do this by just assigning the pointer to the column for each input field to the pointer in the output batch. We call this a *pointer-swing*. In our running example, the projection `Select(e => { e.AdId })` is converted into the following generated operator:

```
void On(UserData_Gen batch) {
  var r = new AdId_Gen(); // generated result batch
  r.CloneControlFieldsFrom(batch);
  // constant time pointer swing of AdId column
  r.col_AdId = batch.col_AdId.AddReference();
  batch.Free();
  nextOperator.On(r);
}
```

We create a new result data-batch of payload type long and pointer-swing the control fields. We then pointer-swing the array for AdId from the source batch to the destination batch. We finally free the relevant columns in the input batch and output the result data-batch.

Notice that since `Where` and `Select` are not temporal, we did not have to access the timestamp columns in our operators; they were simply pointer-swung to output batches in constant time. Thus, we do not pay a runtime cost for temporality for these operations.

4.2 Exploiting Columnar Batches

Our columnar batch organization with dynamic code generation of operators enables us to support several common use-cases where traditional HLL engines lose significant performance.

4.2.1 Serialization and Deserialization

Serialization of objects in a high-level language is inefficient due to the need for fine-grained encoding and decoding of rows. Trill data is stored as columnar data-batches, which introduces the potential for transporting arrays directly over the wire. However, traditional serializers encode arrays on a per-element basis. We created a serializer for Trill – called *Trillium* – that can serialize columnar Trill streams 15X to 20X faster than standard row-based serializers such as Avro [19] (see Section 7.5). Trillium uses three techniques for performance: (1) the serializer and deserializer are code-generated to avoid runtime interpretation; (2) generated data-batches are handled by transferring arrays directly without any fine-grained encoding or tests, and using the actual used count of the data-batch to limit how much data is transferred; (3) memory pools

help reuse the memory into which data-batches are deserialized (this is useful when we execute a streaming query over a de-serialized stream).

4.2.2 String Handling using MultiString

Trill supports all HLL types including strings. However, strings in a high-level language such as C# or Java are not optimized for performance. For example, each string in C# is stored as a separate object with a 24-byte overhead per string. Simply using an array of strings causes the creation of a large number of small heap objects, which results in memory and GC overheads. We instead create a *MultiString* data structure per string column in a data-batch that internally stores the individual (true) strings end-to-end in a single large string that is accessible as a character array (as with the columnar data format, users are unaffected by this transformation). The array is augmented with an array of offsets and lengths for the true strings. MultiStrings reduce memory and processing costs for queries over string data: the *string split* and *substring* operations can be done by simply creating a new offset/length array, which is 50X faster than a usual per-string split or substring. Note that a split can generate more rows than its input; we ref-count the character array across these output batches, creating new offset/length arrays for each batch.

Regular expression matching work as follows: we first compile the pattern once for the query, and then execute a standard regular expression matcher directly over the large string. Whenever there is a match that spans true-string boundaries, we re-execute the matching algorithm starting at the specific true string at that location, in order to weed out false positives. This technique allows us to execute the regular expression logic without fine-grained interruptions, which provides very high throughput optimized for cases where matches are infrequent. Upper/lower case conversion also works similarly. *Substring matching* (*contains*) applies the Knuth-Morris-Pratt [22] algorithm directly on the MultiString. We find that these techniques are up to 6X faster than the usual fine-grained string operations.

Arbitrary string operations that cannot be applied directly on the MultiString are executed by copying over each string to a temporary cached string and executing operations on this string; interestingly, we find that even this back-off technique is around 30% more performant than using fine-grained strings directly, since it avoids main memory accesses to randomly located objects. This solution for strings extends to other fine-grained heap object types such as lists.

4.2.3 Columnar Memory Pooling

A critical performance issue in SPEs is the problem of fine-grained memory allocation and release, also called garbage collection or GC. Automatic GC can be expensive and introduce latency in a high level language such as C# or Java. We follow a novel approach to memory management that retains the advantages of the high-level world and yet provides the benefits of unmanaged page-level memory management. The advantage of not using unmanaged memory is that we completely sidestep the problems associated with supporting complex data types.

Trill employs the notion of a *memory pool*, which represents a reusable set of data structures. One may allocate a new instance of a structure by taking it from the pool instead of allocating a new object (which can be very expensive). Likewise, when you no longer need an object, you return it to the pool instead of letting the GC reclaim the memory.

Trill has two forms of pools: a *data structure pool* allows you to hold arbitrary data structures such as Dictionary objects. They are used by operators that may need to frequently allocate and deallocate such structures. The data-batch instances (shells) are stored in pools and reused. The second type is a *data pool* for payload and control data inside data-batches. The data pool is generated, and contains a `ColumnPool<T>` or each column type T. Each `ColumnPool<T>` contains a latch-free queue of free `ColumnBatch<T>` entries.

A `ColumnBatch<T>` type is a wrapper for a column (array) of type `T`, and includes a ref-count for the column. `ColumnBatch<T>` instances are ref-counted, and each `ColumnBatch<T>` instance knows what pool it belongs to. When the RefCount for a `ColumnBatch<T>` instance goes to zero, it is returned to the `ColumnPool`. When an operator needs a new `ColumnBatch<T>`, it requests the `ColumnPool` for one. The `ColumnPool` either returns a pre-existing `ColumnBatch` from the pool if any, or allocates a new `ColumnBatch`. Operators use copy-on-write semantics: an operator that needs to update a column with a ref-count more than 1 makes a copy of the `ColumnBatch`.

We use a single shared set of memory pools for each NUMA socket. In a streaming system, we expect to reach a "steady state" where all the necessary allocations have been performed. After this point, there should be very few allocations occurring, as most of the time batches would be freed and reused from the pools.

5. GROUPING & STATEFUL OPERATORS

We next describe Trill's grouped temporal operators using our running example, which computes a per-ad windowed count. The key challenge is to build *stateful* (e.g., maintaining per-ad counter state) operators that operate on batched data and which work well across real-time, offline temporal, and progressive scenarios. Section 5.3 describes our compile-time stream property framework that helps us choose from a small set of such physical operators.

5.1 GroupApply

The GroupApply operation accepts a grouping key selector and a sub-query, and logically executes the sub-query on each sub-stream corresponding to each distinct grouping key. We consider single-threaded query execution for now; multi-core execution is covered in Section 6. GroupApply first creates a stateless *Group* operator that computes grouping keys and adds them to the batches. It adds two columns to each data-batch:

1. *Key*: An array of all the grouping key values.
2. *Hash*: An array of hash values (4-byte) of the keys.

These columns are materialized so that each (grouped) operator does not need to re-compute them. The sub-query (windowed count in our running example) is executed on the resulting grouped stream. In order to benefit from batched columnar execution within the GroupApply, all our operators are designed to accept and produce grouped streams. For example, an aggregate operator that receives data-batches with <group-key, payload> outputs a stream of data-batches with per-group aggregates <group-key, aggregate>.

We then add an *Ungroup* operator to remove the grouping key, and the ungrouped stream is returned to the user. GroupApply can also be nested; *Group* creates a nested key, consisting of the original and the new grouping keys, which gets un-nested at the *Ungroup*.

5.2 Temporal Operator Algorithms

Logically, we view a stream as a *temporal database* (*TDB*) [31] that is presented incrementally, as in CEDR [3], Nile [11], NiagaraST [9], etc. Each event is associated with a *data window* (or *interval*) that denotes its period of validity. This creates *snapshots*, a sequence of TDB versions across time. The user query is logically executed against these snapshots in an incremental manner.

Events may either arrive directly as an *interval*, or get broken up into a separate insert into (called *start-edge*) and delete from (called *end-edge*) the TDB. Internally, events have two timestamps (sync-time and other-time) that are interpreted as follows:

- When other-time is greater than sync-time, it represents an interval with a data window of [sync-time, other-time).
- When other-time is ∞, it is a start-edge that denotes the insertion of an item at sync-time.
- When other-time is less than sync-time, it is an end-edge that occurs at sync-time and deletes an earlier start-edge that occurred at the previous timestamp (other-time).

Consider the stateless Window operator in our running example. It simply sets other-time to sync-time + 5mins in order to make the data have a 5-minute window duration. Further, it drops end-edges by setting their bitvector entry to 1 (since start-edges get converted into intervals when we set other-time as shown above).

5.2.1 User-Defined Snapshot Aggregation

Grouped 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.

User Specification A user implements the following functions:

```
Expression<Func<TState>> InitialState();
Expression<Func<TState, long, TInput, TState>> Accumulate();
Expression<Func<TState,long, TInput, TState>> Deaccumulate();
Expression<Func<TState, TState, TState>> Difference();
Expression<Func<TState, TResult>> ComputeResult();
```

Here, `Func<A, B, ..., X>` denotes a function that takes `A`, `B`, … as input parameter types and outputs a value of type `X`. All these methods are provided as lambda expressions so that Trill can inline them into the generated columnar operator code for performance.

InitialState is a function that takes no input parameters and produces an initial state of type `TState`. Accumulate takes a `TState`, a long timestamp, and an input tuple with payload type `TInput`, and produces a new state of the same type (TState). Deaccumulate works similarly. Finally, Difference allows users to define the notion of subtracting one state from another; this allows users to perform this more efficiently than deaccumulating state one event at a time. Our implementation for Count is shown below:

```
InitialState: () => 0L
Accumulate: (oldCount, timestamp, input) => oldCount + 1
Deaccumulate: (oldCount, timestamp, input) => oldCount - 1
Difference: (leftCount, rightCount) => leftCount - rightCount
ComputeResult: count => count
```

In our running example, the user can compute a streaming count using the Aggregate method, as shown below:

```
var result0 = inp0.Aggregate(w => w.Count());
```

We also support simultaneous application of multiple aggregates in a single snapshot operator, with the ability to combine results on a per-snapshot basis. For example, one could write Average as:

```
inp0.Aggregate(w => w.Sum(), w => w.Count(), (s, c) => s / c);
```

Operator Implementation Given the above specifications, Trill generates a grouped per-snapshot aggregate operator with inlined expressions. Our operator uses three data structures:

1) *AggregateByKey*: This is a hash table that stores, for every distinct key associated with non-empty aggregate state (`TState`) at the current sync-time, an entry with that key and the aggregate state.
2) *HeldAggregates*: This uses a hash table called FastDictionary, that stores – for the current sync-time T – the aggregated state corresponding to keys for which events arrive with sync-time equal to T. This hash table does not support the deletion of individual keys, but handles fast iteration through all the entries, and supports a fast "clear" of the hash table when time moves forward. We describe FastDictionary in Section 5.2.2.
3) *Endpoint Compensation Queue (ECQ):* The ECQ contains, for each future endpoint (due to an interval event), partially aggregated state (*HeldAggregates*) for that endpoint. In general, the ECQ is a priority queue. However, we can often exploit stream properties (cf. Section 5.3) to use a FIFO queue or eliminate the ECQ altogether.

For each data-batch, we iterate through the events in the batch. We first look up each event in HeldAggregates. If not found, we look in AggregateByKey, and if it contains the key, we ref-copy the state into HeldAggregates (and output an end-edge for the old aggregate state). We then update the current state for that key by inlining the appropriate expression: Accumulate for start-edge and interval, and Deaccumulate for end-edge. In case of intervals, we also accumulate state for the (future) end timestamp into the ECQ.
When sync-time moves forward, we inline ComputeResult and output start-edges for the non-empty aggregates in HeldAggregates and clear it. Empty entries are removed from AggregateByKey. We then process the endpoints in ECQ between now and the new sync-time, using the inlined Difference expression to update and output state for each endpoint. Similar processing is performed on receiving a punctuation that moves the current sync-time forward. Our implementation caches the state associated with the current key, so that the common case where many events have the same key can be executed very efficiently without frequent hash lookups.

5.2.2 FastDictionary

The FastDictionary is a lightweight hash table optimized for frequent lookups, small sets of keys, frequent clearing of the entire data structure, no deletes, and frequent iteration over all keys in the table. Briefly, FastDictionary uses open addressing with sequential linear probing. The basic data structure is a prime-number sized array A of <key, value> pairs. An entry to be lookup up or inserted is hashed to an index in A. If that entry is occupied we scan entries in A sequentially until we find the element (or an open slot which indicates lookup failure). The sequential probing is well suited to CPU caching behavior, and with a low load factor (1/16 to 1/8) we get a high likelihood of finding an element very quickly. We resize the hash table when necessary to maintain the load factor – streaming workloads typically reach a stable size quickly, after which hash table resizes become very rare.

The array A is augmented with a bitvector B, which has one bit per array element to indicate whether that entry is used. B allows iteration to be performed very efficiently, and insertion can find an empty slot index without having to access A. Further, clearing the

dictionary is straightforward: we simply zero out the bitvector. Accesses to the bitvector are very fast due to cache locality.

We find that the FastDictionary performs up to 40% better than a carefully designed general hash table, when used to maintain per-key state for the current sync-time in the snapshot operator.

5.2.3 Temporal Join

Trill includes a temporal *symmetric hash join* (SHJ) operator which performs a temporal equi-join by the current grouping key of the stream. SHJ processes input data in sync-time order across its two inputs. When a payload from the two inputs has the same key and overlaps temporally, the join executes a result selector expression to generate output for the pair of payloads. SHJ maintains two hash-tables (also called *synopses*) – one for the left and one for the right side. We optimize the implementation for two special cases:

1) The input has only start-edge atoms: In this case, we know that events will never be removed from the synopsis. All input events are processed, in sync-time order, and inserted into the corresponding hash-table (if the other side has not reached its end-of-stream), with the key and value equal to the grouping key and payload, respectively, of the input event. Additionally, the hash-table for the other side is searched to identify matching active events, apply the (inlined) result selector and add to the current output data-batch. The data-batch is output when it is full, or a punctuation is received.

2) The input has arbitrary data atoms: When the input events can include intervals and end edges, then the operator must handle the case of events being removed. The operator still employs two hash-tables as before, but also employs an *endpoint compensation queue* (*ECQ*) to store away active intervals that need to be removed in the future (i.e., when the current processing time reaches the end time of the interval). A start-edge works as before, while an end-edge outputs a result end-edge for all matching join results. In addition, an end edge results in the entry being removed from the hash-table. When time moves forward, we process the expiring endpoints from the ECQ in a similar manner as end-edges in the stream. More precisely, when events are removed from the hash-table, either by end-edges or the reaching of an interval's ending timestamp from the ECQ, we search the hash-table on the other side to identify payloads which joined previously. For each of those payloads, we outputs a corresponding end edge.

Additionally, within a timestamp, we need to process the end-edges on one side before executing a join with incoming data from the other side. This is because these end-edges may completely remove active events at the current timestamp. Note that this delay in output generation within the current timestamp is only for the case of an event joining against a start-edge, because intervals have known end timestamps. The case of an interval joining with an interval is handled as a special case, because the exact duration of the join is known beforehand, so the operator can directly output an interval for the intersecting duration.

Interestingly, we can use the temporal SHJ operator to perform an *asymmetric* relational hash join (build followed by probe), by simply time-stamping the right input as $[0, \infty)$ and the left input as sync-time values beyond 0. This causes SHJ to first fully process the right input until end-of-stream, which means that we do not need to add atoms from the left side to a hash-table – resulting in a read-only probe phase. We also support a relational merge join (Section 7) to handle cases where inputs are sorted by the join key.

5.2.4 WhereNotExists and Clip

WhereNotExists and Clip are anti-joins that output only those events received on their left input that do not join with an event received on the right. Similar to join, the user provides delegates to determine a mapping key for each payload. Clip is a restricted form of WhereNotExists optimized for the common case of permanently clipping an event received on the left when a future right event successfully joins with it. We optimize the implementations for the two operators differently:

1***) WhereNotExists***: All input events received on the left input are processed, in sync-time order, and inserted into a map data structure with the key and value equal to the mapping key and payload, respectively, of the input event. Similarly, input events received on the right input are processed in sync-time order and inserted into a data structure that counts the number of occurrences of a mapping key on the right input. Any start edge events received on the right input, for which it is the first occurrence of that key, results in a scan for any joining left inputs which require the output of an end edge. Similarly, any end edge events received on the right input, for which the resulting occurrence count drops to zero, results in a scan for any joining left inputs which now require the output of a start edge. Only when time progresses on the right input is a scan performed to search for any newly inserted left events that do not join with any event on the right to output an initial start edge.

2) Clip: Clip is similar but a much more optimized version of WhereNotExists. In Clip, only events received on the right at a later timestamp can join to events received on the left. As a result, no right state must be maintained. Instead, only a map of events received on the left input is required. As events are received on the left, the operator outputs a corresponding start edge and inserts the event into a map. As events are received on the right, the operators performs a scan in the map to locate any joining left events. All joining left events will be removed from the map and output an end edge.

Join, WhereNotExists, and Clip are scaled out by writing them as a GroupApply. The GroupApply operation sets the key of the stream to the join key. The above operators assume that the key of the stream is the equijoin attribute, thus join works efficiently and interoperates correctly in the context of GroupApply.

5.2.5 Alter-Lifetime

Trill supports the notion of altering event lifetimes to support windowed operations and correlating data across time (this is a generalization of the Window operator described earlier). This is accomplished using the alter-lifetime operation. Alter-lifetime accepts two expressions as input: a start-time selector which maps a start-time to a new start-time, and a duration selector, which maps a start-time and end-time to a new duration. An overload allows the duration to be a constant, in which case the `IsConstantDuration` property gets set with the specified duration. Alter-lifetime limits 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 alter-lifetime.

The generated alter-lifetime operator inlines the time-manipulation expressions inside a per-data-batch loop in order to produce new sync-time and other-time values for each output data-batch. The remaining data-batch fields (including payloads) are unaffected and can be added to the output data-batch using pointer swings (i.e., no memory copy). If a duration selector is specified, end-edges need to be removed from the output, since the corresponding start-edge would directly be converted into an interval with the correct

Table 2: Physical operators in Trill.

Physical Operator	Supported Variants
Filter, Project, SelectMany, AlterLifetime	No variants (supports arbitrary HLL expressions)
Temporal Symmetric Hash Join	General, Start-edge, Start-edge + Order-aware
WhereNotExists, WhereExists, Clip	No variants
Snapshot Operator (with custom incremental logic)	General, Start-edge, Constant-duration, Constant-hop
Spray & multicast	General, Order-aware
Group, Shuffle, Ungroup	No variants
Temporal Union	No variants (optimized for batches with same sync-time)

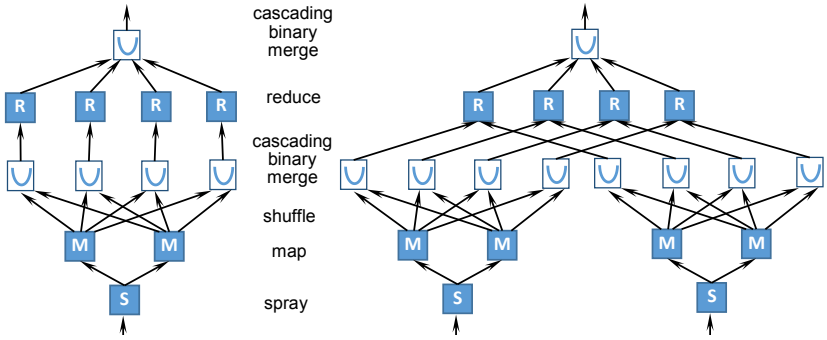

Figure 3: Two-stage streaming temporal map-reduce (one and two inputs).

duration (using the duration selector). We support this by simply using the bit-vector to mask out the end-edges.

5.3 Compile-Time Stream Properties

In order to support temporal and progressive queries efficiently and to optimize performance for common stream characteristics, we use a compile-time stream property derivation framework to help us create customized physical operators. Stream properties define restrictions on the content we expect to see in a given stream. They are specified at stream inputs, and are also inferred at compile-time from query logic at each point in the query plan. Some properties that we support include:

1. `IsIntervalFree(bool)`: This property indicates that the stream contains no intervals; only start- and end-edges. As an example, this property allows us elide the ECQ from our aggregate operator.

2. `IsConstantDuration(bool, long)`: This property is used to indicate that all events in the stream have the same fixed (optionally specified) duration. This property allows us to maintain the future endpoints as a FIFO queue (linear lookup and update) instead of an expensive priority queue. The special case of constant duration=∞ indicates a *start-edge-only stream*. This is common when we execute progressive queries or non-windowed aggregates. For example, a MAX operator can maintain just a single piece of state – the maximum value seen up to now. A related `IsConstantHop` property allows us to optimize for tumbling windows.

3. `IsColumnar`: This property indicates whether we are in columnar or row-oriented mode, and is used to choose between code-gen and normal operators. Operators may need to work in row-oriented mode because (a) some property of the user type prevents it from being used in columnar processing; or (b) an expression in the query is too complex or opaque to allow its transformation. We allow conversions from row to columnar and vice versa.

Trill also includes properties to capture and exploit sort ordering and data-batch packing in the input data; see Section 6.2 for details. We use stream properties to select from a small set of physical operators, as shown in Table 2. We find that these operator variants are sufficient to provide high performance for most queries across the spectrum of analytics that Trill targets.

6. LIBRARY MODE & MULTI-CORE

Trill supports two modes of execution. In the default no-scheduler mode, Trill works as a pure library that does not itself own any threads but performs work on the thread that pushes messages to it.

For efficient multi-core processing, we built a pluggable scheduler framework that allows Trill to parallelize execution on specific application-provided threads or cores. The basic idea is that we take the physical plan and partition it into *query fragments* (described next). The scheduler is given *n* threads; each thread picks up data-batches to push to operators. For progressive queries, we process batches in timestamp order for fair progress across queries. Real-time queries use stimulus-time scheduling [15]. We hold a priority queue of query fragments; each scheduler thread picks the fragment with highest priority to execute next. Note that each query fragment itself may consist of multiple operators, but is executed on the same thread (similar to the no-scheduler mode). Our scheduler works at the batch granularity, which allows its overhead to be amortized.

6.1 Streaming Temporal Map-Reduce

The key building block for multi-core processing in Trill is what we call *Temporal Map-Reduce* – streaming generalizations of the well-known Map and Reduce operations, with temporal support. Users can either use Map and Reduce explicitly to indicate opportunities for parallelism, or use GroupApply which gets transparently rewritten by our compiler into Map and Reduce.

Map takes a query fragment as input, for the purpose of scaling out in a stateless manner by spraying input batches to each instance of the query fragment. Further, it takes a grouping key argument that identifies the key for the subsequent Reduce operation. Reduce takes as parameter a query fragment that is logically executed for each distinct value of the specified grouping key. For example, our running example is rewritten as:

```
inp0.Map(str => str.Where(...).Select(...), e => e.UserId)
.Reduce(str => str.Window(...).Aggregate(...), (g, c) => new { g, c });
```

Here, the first argument to Map specifies the stateless Where and Select operations to be performed in parallel on the input stream, while the second argument specifies the grouping key (UserId) to shuffle the result streams by. The first argument to Reduce computes per-user windowed Count, and the second argument allows us to add the grouping key (UserId) back to the result count stream at the end of the query.

Such a specification is mapped by Trill into a physical operator graph (shown in Figure 3; left) with multiple query fragments – one for each map and reduce sub-query instance – that can be executed using our scheduler. Let *n* denote the degree of parallelism available to us (e.g., number of cores on the machine).

1) Spray: We first take a stream of batches and perform a stateless spray of the batches to *n* downstream endpoints. Spray performs constant work per batch and introduces negligible overhead.

2) Map, Group, Shuffle: On each of the *n* endpoints, we apply the map sub-query. The result stream enters a generated shuffle operator that computes (inline) the new grouping key and its associated hash on each event. Based on the hash value, we add the event to one of *n* output data-batches (one per hash bucket). There are *n* downstream merge operators in the physical plan. As output batches fill up, they are sent to the corresponding merge.

3) Merge, Reduce, Ungroup: Merge performs a temporal union (described next) and feeds the resulting stream (one per reduce bucket) to the reduce sub-query. We then execute Ungroup to un-nest the grouping key. A final merge operator temporally merges the results from each reduce bucket into a single output stream.

Note that our batched data-flow architecture implies that synchronization occurs only at coarse-grained batch boundaries, where data is handed off from one query fragment to another.

Temporal Cascading Binary Merge The temporal merge in Trill is implemented using a tree of streaming binary merges for performance [26]. Each binary merge reads sync-time values from the left and right input batches, and merges the data in sync-time order into a destination batch. We also check for the special case (common after the map phase and with progressive queries) where one input batch lies ahead of the other in time, in which case we can forward the input batches without doing a fine-grained merge.

Two-Input Reduce Trill also supports two-input reduce. The architecture (see Figure 3; right) is similar, except that there are two separate map phases for each of the inputs, and these map outputs are shuffled and brought together to a single set of n two-input reducers. Trill rewrites binary operators such as temporal joins into a two-input reduce so that they can execute on multiple cores.

6.2 Performance Optimizations

Since the shuffle (repacking data-batches by key) in temporal map-reduce is very memory-intensive, we try to avoid it when possible.

Exploiting Sort-Order and Packing Trill supports a compile-time property to identify whether input snapshots are sorted by a payload field. In addition, it supports a property where a sorted stream is packed according to the following rule: for a given batch B, data with a given sort key value K cannot spill to the next batch B+1 unless all the data in batch B has the same sort key value K. Further, two streams may be packed in a compatible manner, i.e., keys in two different batches in stream 1 do not map to the same batch in stream 2. Sort-order is used, for example, to replace a SHJ by a more efficient merge-join.

If a stream is packed as described above, temporal map-reduce can retain the sort order during spray. Basically, it retains the last key in the current batch B before spraying it to branch *i*. In case the first event in the next batch B+1 has the same key value, that batch is also sprayed to the same branch *i*. Otherwise, the batch B+1 is sprayed round-robin to the next branch *i+1*. Likewise, a two-input map-reduce can retain sort order during spray if the streams are packed in a compatible manner. We can then move the grouped sub-query to the map phase, avoiding the shuffle.

Exploiting Skew in Input Streams Another case where we can avoid a shuffle is when a 2-input reduce is skewed, i.e., the right side is much smaller than the left. We simply broadcast the smaller side to all branches, and spray the larger side round-robin. In this case, we can perform the 2-input operation without a shuffle. A common use of this facility is when doing a temporal join across a high- and a low-rate input stream.

7. EVALUATION & USAGE SCENARIOS

The goal of evaluation is to examine how Trill's hybrid architecture allows it to perform favorably against state-of-the-art specialized engines at different points in the analytics spectrum. We then discuss how our features have enabled a range of usage scenarios.

7.1 Setup and Workloads

All experiments are conducted on a 2-processor 8-core (16 hyper-thread) Intel Xeon CPU E5-2660 machine running at 2.2GHz, with 192GB RAM, and running 64-bit Windows Server 2008 R2.

Workloads We use the following datasets in our experiments:

1. **GenData(n, d, m)**: This is a set of two synthetic tables (T1 and T2). Each table has two 8-byte columns (C1 and C2) and both tables are ordered by C1. T1 has n rows, with d distinct values in C1. Column C2 in T1 has the same d values in random order. T2 has m ($\geq d$) rows whose C1 includes the d distinct values of T2; the remaining values are random. C2 in T2 is random as well.
2. **UserSearch**: This is a dataset of user search phrases from a commercial search engine log. It has two columns: for each 8-byte UserId (ordered by UserId), we store a string (search term). Some experiments use a *hash-tokenized* version of this dataset, where phrases are pre-split into tokens (words) and hashed into 8-byte values. These datasets have two fields: UserId and QueryId. Our temporal experiments use a similar dataset over a 15-day time period, with an additional 8-byte timestamp field (the data is ordered by timestamp in this case). We use real queries over these datasets, with sizes varying from 10M to 600M rows.
3. **SearchURL**: This is another real search dataset (100M rows, 11.6GB) that contains five columns: search phrase, its hash value, number of times issued, URL clicked after search, number of clicks on URL (for that search), and the total URL clicks for that query.
4. **TPC-H**: We experiment with grouped multi-aggregation using the LINEITEM table of TPC-H at a scale factor of 100GB.

Baseline Query Engines For relational queries, we compare Trill's performance against **DB-X**, a modern commercial database system that incorporates a compressed columnar store and supports batched operators. Trill uses a default maximum batch size of 80K tuples whereas DB-X uses a larger fixed batch size (by more than 10X). DB-X does not support progressive, incremental, or real-time temporal processing. We discard the first two runs to warm both engines, and report average performance over the next 5 runs. We ensure that DB-X is operating in columnar mode and running from memory, i.e., not accessing disk or writing to temp DB. Our experiments vary the degree of parallelism (DOP) for both engines from 2 to 32 (DB-X did not support a DOP of 1 in columnar mode).

For temporal streaming and progressive queries, we compare Trill against **SPE-X**, a commercial SPE that is based on the event-at-a-time architecture and a temporal data model. While SPE-X can also be used for relational queries, its performance is lower than both Trill and DB-X by 2-4 orders of magnitude; hence we do not include it in relational experiments. We compare Trill to SPE-X in terms of throughput, memory, and latency for temporal queries. Unless otherwise indicated, all systems use all the available cores on our machine in the experimental results.

7.2 Temporal Stream Processing

7.2.1 Data Ingress

We measure the time it takes to load streaming data into the SPE by executing a pass-through query that drops all tuples (to avoid incurring an egress cost). We use the tokenized search log as input. Loading is performed on a single thread to model real-time ingress. We find that SPE-X can load data at 450K events/sec when the events are pre-created in memory and loaded from an array. In Trill, when the data is pre-loaded into memory in a columnar format, the pass-through query takes trivial time (>1 billion events/sec); no memory copies occur because only pointers to data-batch messages

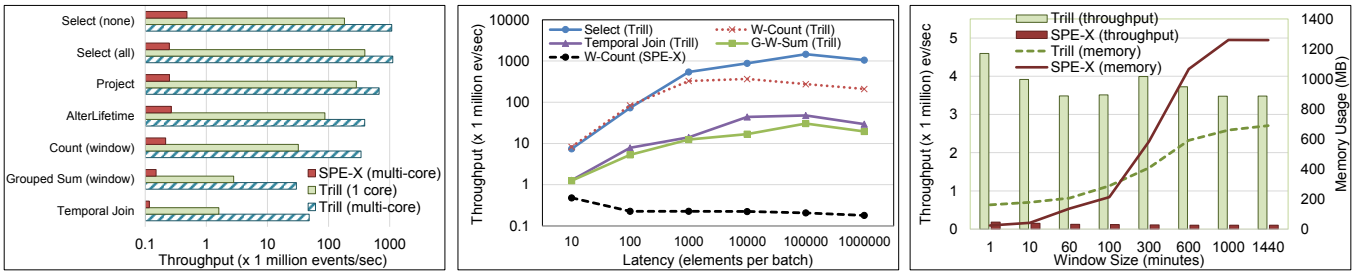

Figure 4: Trill vs. SPE-X: (a) Throughput; (b) Latency vs. Throughput; (c) Single-core temporal join; varying window size.

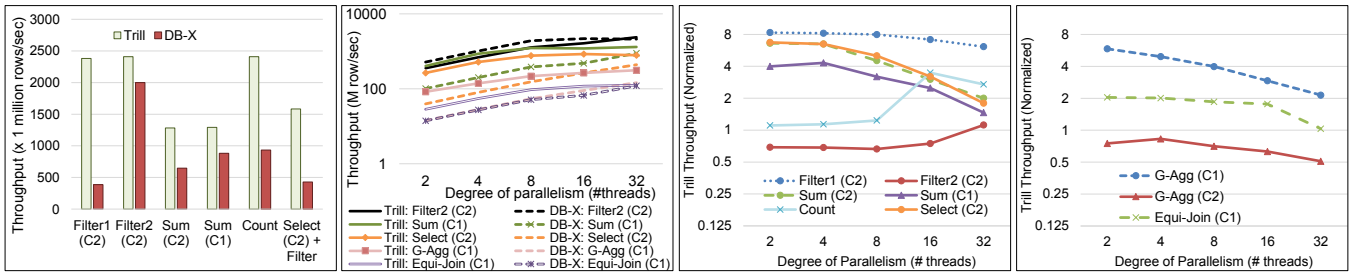

Figure 5: Trill vs. DB-X: (a) Simple queries (DOP=32); (b) Scale-out summary; increasing DOP; (c) Trill throughput (normalized to DB-X); increasing DOP for filter, select, aggregate; (d) Trill throughput (normalized) for grouped-agg & equi-join.

flow through the query plan with no fine-grained work. When the data is ingressed as individual in-memory row events from an array, a conversion to our batched columnar data format is performed in tight loops on-the-fly; here, a pass-through query runs in Trill at 140M events/sec when we ingress data using a single core.

7.2.2 Throughput Comparisons

We compare Trill's throughput to SPE-X (on all cores) for several stateless and stateful streaming operations: (1) *filter* (no matches); (2) *filter* (all match); (3) *project*; (4) *alter-lifetime* (windowing); (5) *windowed count* (W-Count) with a window size of 1 hour and hop size of 10 minutes; (6) *grouped windowed sum* (G-W-Sum) with QueryId as grouping key and the same window/hop size as before; and (7) *temporal join*, where we find – for each user (join key) – sequences where a user searches for a search term from set A followed by a search term from set B within one hour (A and B are non-overlapping sets of 25% of all terms in the dataset). We use the time-ordered hash-tokenized pre-loaded search log (100M rows) for these experiments, and fix query latency at 80K events per punctuation. Figure 4(a) shows the results. We see that Trill is between 2-4 orders of magnitude faster than SPE-X across the range of queries, due to Trill's superior architecture with features such as columnar batching, generated operators with tight loops, and fast memory-bandwidth-optimized algorithms.

7.2.3 Latency (varying punctuation frequency)

In Figure 4(b), we vary the latency (number of events between punctuations) and measure its impact on throughput. As expected, Trill is able to take advantage of higher latency by using larger data-batch sizes, and performance increases significantly. Interestingly, throughput drops for very large batch sizes because of the need to make large memory allocations and the lower probability of batch reuse with memory pools. We also show the throughput of SPE-X for W-Count. SPE-X is mostly unaffected by latency since it does not take advantage of batched data; in fact, for W-Count, performance degrades due to the inability to clean up internal state and data structures as frequently. Notably, even with a small latency of 100 events per punctuation, Trill benefits significantly from careful columnar batching, providing more than two orders-of-magnitude performance gains over SPE-X for W-Count.

7.2.4 Window Size

We next experiment with Trill's no-scheduler mode in Figure 4(c). For fair comparison, we use SPE-X with only one scheduler thread. We execute a windowed temporal join that correlates searches per user, similar to the query from the previous experiment but looking for searches within a window W. Figure 4(c) shows that the performance falls as W increases because more items need to be retained and joined within the window. Further, while SPE-X has slightly lower memory utilization than Trill for very small W (since it processes one event at a time), Trill uses lower memory when W increases, as it benefits from batched data and sync-time ordering.

7.3 Relational Query Processing

7.3.1 Data Ingress

We compare loading costs for relational data stored in a row-oriented CSV file. DB-X incurs higher loading costs since the data needs to be loaded in a compressed columnar format. To measure this, we use 100M rows of the SearchURL CSV log (11.6GB) and load it into DB-X and Trill. The DB-X data loader is single-threaded and takes 592.8secs of CPU time to load the data in compressed form. Trill takes 179.6secs on a single thread for this data. Trill also supports multi-threaded loading, which takes only 44.6secs. For a generated dataset with 600M rows (each with two 4-byte int columns) and size 12.8GB, DB-X takes 688secs whereas Trill takes 356secs (single-thread) and 33secs (multi-threaded).

7.3.2 Relational Query Performance

Figure 5(a) shows the performance (using all cores) of Trill and DB-X for simple operations such as filter, sum, count, and select over GenData(600M, 20M, 150M). The Filter2 predicate is identical to Filter1, except that Filter2 is pushed to the DB-X storage layer, which explains why DB-X performance improves significantly on Filter2. Overall, Trill performs comparably to DB-X for these operations, faring up to 8X better in case of Filter1.

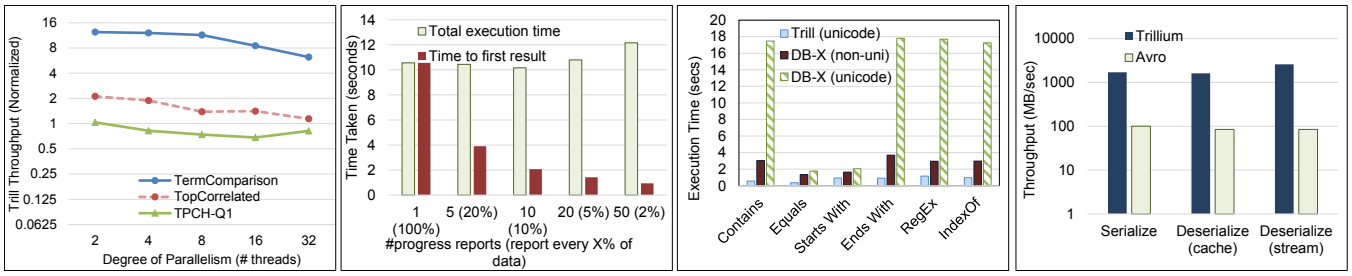

Figure 6: (a) Normalized performance; real search/URL & TPC-H data; (b) Effect of progressiveness in Trill; (c) Performance over strings; (d) Serialization rate: Trillium vs. Avro.

Figure 5(b) summarizes throughput for Trill and DB-X for increasing DOP. The key take-away from this figure is that performance is comparable, and both systems scale well (note that both axes are log scale). We analyze performance in greater detail in the next set of charts, described next.

Figure 5(c) shows Trill's throughput normalized to DB-X, with varying DOP. A value of 1 indicates that throughput of the two engines is identical. We see that Trill has higher performance in all cases (except Filter2, where DB-X is slightly better at low DOP). We also notice that performance tends to converge at higher DOP as both systems hit memory bandwidth limits.

Figure 5(d) shows Trill's throughput normalized to DB-X for two grouped aggregate queries (by C1 and C2) and an equi-join query (on C1). When we aggregate over column C1, Trill leverages the fact that it is sorted and avoids the shuffle, leading to better performance. DB-X is better when we aggregate by C2, since Trill needs to shuffle the data whereas DB-X uses a shared hash table to perform the aggregation. Trill's performance for equi-join is better than DB-X, converging to 1 at DOP=32. Here, Trill leverages the sort-order of C1 to avoid the shuffle and use a merge join (DB-X is superior when Trill uses a shuffle with hash-join). As future work, we plan to investigate improving Trill's shuffle performance and leveraging shared data structures. We note that while shared data structures are superior to shuffle, a shuffle would need to be performed anyway when we need to process data across more than one machine. Further, our techniques to avoid shuffle using sort orders or input skew are applicable even in a multi-node setting.

7.3.3 Query Search, URL Logs, TPC-H LineItem

TermComparison We execute a real query (obtained from a data scientist at Microsoft) that analyzes entities that users compare in searches. We take the SearchURL log, look for searches that contain "versus" or "vs" and use the left and right side substrings as entities. We look up (join) each search phrase against the distinct entities, and sum the total query clicks per entity.

Figure 6(a) shows the results. In DB-X, we use the substring operation to compute a temp table for the entities (which is very slow), whereas Trill's support for HLL strings and *SelectMany* (where one row is converted into zero or more rows by a user-defined function) makes this 10X faster. Trill's join (we use an asymmetric hash join) was slightly faster than DB-X in this case.

TopCorrelated We execute another real query: given a parameter P ("vegas"), compute for each word W in the search log, the ratio of (a) number of distinct users who searched for both W and P; and (b) total number of distinct users who searched for W. This query allows analysts to determine the search terms closely correlated to P, and helps in ad selection/pricing.

The first step is to create a table of (user, word) pairs by splitting search phrases from the log by the space delimiter. Unfortunately, databases are inefficient at split (by more than 10X) since it has to be implemented as a UDF (SQL does not natively support Split or SelectMany). Trill performs the split in 2.6secs (we do not use MultiStrings in this query; they are evaluated in Section 7.5), producing ~31M tokenized results. Figure 6(a) shows performance for the rest of the query pipeline *not including split*, where we see that Trill is slightly faster (up to 2X at DOP=2).

TPC-H Lineitem Figure 6(a) compares Trill against DB-X for TPC-H Q1 which computes 8 grouped-aggregates (we elide the filter on ship-date from Q1). Trill sprays data and uses a hash-table per core with a final aggregation, whereas DB-X uses a shared hash table; we see that performance is close (within 50% at worst).

7.4 Progressive Query Processing

We use the search log query and execute a query that computes the popularity of search terms in the dataset. We vary progressiveness in terms of number of result sets produced (or report at every X% of the dataset), and plot Trill's total execution time in Figure 6(b), as well as time to produce the first result. With one result set, the query produces its result only at the end of the query. We see that increasing progressiveness for this query has a slight impact (~15%) on total execution time, but significantly reduces the time (by 10X) to produce the first result set. A study of progressiveness using a commercial SPE can be found in our recent work [2].

7.5 Code Generation, Strings, Serialization

Code Generation We measured the cost of dynamic code generation in Trill. Trill generates data-batches, memory pools, and operators. We found that on a single core, the average code compilation time was 75ms per operator, and the average time for the remaining parts of code generation (expression transformation, code construction, and assembly loading) was 31ms for memory pools, 25ms for data-batches, and 45ms for operators (on TPC-H queries). The worst case for a complex operator was less than 250ms (using unoptimized code that we believe this can be significantly improved). We also aggressively cache generated types in Trill. Finally, we note that code generation is easily parallelizable on multiple cores.

String Processing We use the SearchLog dataset with 100M rows, containing 4GB of Unicode search phrases, and execute string operations in Trill and DB-X (with columnar string field). In case of DB-X, we report results for both varchar (one byte per character) and nvarchar (Unicode with 2-byte characters). Trill uses only Unicode strings. We experiment with (1) string containment for "free"; (2) equals for "vegas"; (3) starts with "free"; (4) ends with "download"; (5) regular expression "%free%download%"; and (6) offset of substring for "vegas". Trill uses the MultiString

format described in Section 4.2.2 to store string columns. We see from Figure 6(c) that Trill (Unicode) is up to 5X faster than DB-X (non-Unicode) and up to 30X faster than DB-X (Unicode).

Pavlo et al. [27] report benchmark results that show Vertica and DBMS-X performing "grep" at around 60MB/sec on one node (Hadoop was 2X slower at 25MB/sec) on a 2.4GHz Core 2 Duo processor. Newer results from [28] indicate that Shark (Hive on Spark) executes grep at ~833MB/sec per node (machine specs not mentioned) on memory-resident data. In contrast, grep in Trill operates at 7.2GB/sec on Unicode strings, on our 16-core machine.

Serialization Figure 6(d) compares Avro [19] to Trillium (cf. Section 4.2.1) for a stream of payloads with two 8-byte fields. To avoid the disk bottleneck, we use a memory stream for these experiments. We see that Trillium is around 15X faster, due to the columnar format of Trill data which allows it to avoid fine-grained encoding and decoding. Further, when processing streaming data (that is dropped immediately in this experiment), Trillium is 20X faster than Avro – the speedup beyond 15X is due to memory pooling, which can reuse the streaming data-batch column arrays.

7.6 Current Usage Scenarios

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

1) Orleans-hosted real-time: Orleans [16] is a programming model and fabric that enables low-latency (in milliseconds) distributed streaming computations with units of work called grains. Orleans owns threads and manages distribution. Thus, users use Trill as a pure library (using its no-thread mode) to express temporal streaming queries as part of their Orleans grain code.

2) Analytics within SCOPE: SCOPE [20] is a map-reduce platform for query processing that allows arbitrary .NET code as custom reducers. As with Orleans, SCOPE owns threads and schedules reducer code; thus, users embed Trill as a no-thread library within their reducers in order to perform temporal analytics [1] over search data such as clicks, impressions, and page views. Another such fabric used with Trill is REEF [18], which is built on YARN [17].

3) 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).

4) 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 traces, for example, to detect anomalies or patterns.

5) Back-end for Analytics: Tempe (formerly called Stat! [5]) is a Web-based interactive analytics environment that allows users to author queries and visualize results progressively. It uses Trill as a back-end server to run temporal and progressive relational queries.

8. RELATED WORK

Streaming Engines Starting with the seminal work of STREAM [30] and Borealis [10], there now exist many SPEs; both from research (e.g., NiagaraST [9], Nile [11], Naiad [35]) and industry (e.g., StreamInsight [8], Storm [29], Reactive [4], MillWheel [38]). A detailed feature and architecture comparison of such systems was covered in Table 1 (Section 1). Spark Streaming only targets multi-second latencies and coarsens time for performance, but ties system batching to application time and query semantics: for example, a 1-sec hopping window aggregate forces 1-sec batches, even when executing on an offline log. DataCell [37] follows a different architecture of augmenting a DBMS to support incremental stream processing, but the resulting system provides significantly lower throughputs than Trill and lacks fabric and language integration. In contrast, Trill's library-based hybrid architecture achieves all the features outlined in Table 1, and provides very high performance across the latency spectrum.

Traditional Databases Modern DBMSs leverage techniques such as columnar organization, compression, and SIMD processing for high performance [6][12][14][25]. As depicted in Table 1, DBMSs do not offer rich fabric or language integration, do not handle real-time or temporal analytics, and make choices that favor their particular design point. For example, databases spend significant time reordering and compressing data. Further, query processing is non-incremental and usually involves multiple passes over the data. For example, multiple indexes may be created after data loading. In contrast, Trill provides high performance across the analytics spectrum by processing a stream of varying-sized columnar batches with single-pass algorithms and no compression. Trill also exploits pre-existing sort orders (if any). That said, enabling lightweight online compression schemes is part of our future work. Trill uses temporal operators for relational queries, with timestamps used for scheduling. For example, the SHJ operator turns into an asymmetric relational join if we set the build side to have lower timestamps than the probe side. Finally, unlike most DBMSs, Trill is a library that provides deep fabric and language integration.

Big Data Systems Multiple big data analytics systems have been proposed over the last several years. Map-Reduce was one of the first such systems, and is still popular for non-incremental analytics on disk-based data. The performance of Hadoop is known to be quite low. Phoenix++ [7] is a variant of map-reduce for in-memory analytics; unlike Trill, it is neither temporal nor streaming, and exposes a low-level key-value API. YARN [17] and REEF [18] generalize Hadoop to a distributed resource manager. Storm is a streaming analytics framework that can potentially embed Trill within its spouts. Spark [28] provides a resilient distributed dataset abstraction over which users can write transformations. BlinkDB [33] supports interactive queries over Spark. S-STORE [23] integrates low-latency streaming with OLTP analytics, which is complementary to our goal of high-performance temporal analytics across a wide latency spectrum. Trill, in contrast to these platforms, is a library-based temporal engine that pushes the envelope of performance for a wide range of analytics, and can be embedded within scale-out fabrics. Some comparisons with benchmark results for these systems are given in Section 7.

9. CONCLUSIONS

Trill is a new query processor that fulfills three requirements for an engine to serve the diverse big data analytics space: (1) *Query Model*: Trill is based on a 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 today's comparable SPEs. For relational queries, Trill's throughput is comparable to a modern commercial columnar DBMS. This technical report describes and experimentally validates Trill's new hybrid system architecture and design that has enabled the above combination of features, and has

resulted in Trill's usage as a library within Microsoft, across a number of fabrics and scenarios ranging from real-time to offline.

REFERENCES

[1] Badrish Chandramouli, Jonathan Goldstein, Songyun Duan. Temporal Analytics on Big Data for Web advertising. In ICDE, 2012.

[2] Badrish Chandramouli, Jonathan Goldstein, Abdul Quamar. Scalable Progressive Analytics on Big Data in the Cloud. In VLDB, 2014.

[3] R. Barga et al. Consistent Streaming Through Time: A Vision for Event Stream Processing. In CIDR, 2007.

[4] Reactive Extensions for .NET. http://aka.ms/rx.

[5] M. Barnett et al. Stat! - An Interactive Analytics Environment for Big Data. In SIGMOD, 2013.

[6] P. Larson et al. Enhancements to SQL Server Column Stores. In VLDB, 2013.

[7] J. Talbot et al. Phoenix++: Modular MapReduce for Shared-Memory Systems. In Intl. Workshop on MapReduce and its Applications, 2011.

[8] Microsoft StreamInsight. http://aka.ms/stream.

[9] D. Maier, J. Li, P. Tucker, K. Tufte, V. Papadimos: Semantics of Data Streams and Operators. ICDT 2005: 37-52.

[10] D. Abadi et al. The design of the Borealis stream processing engine. In CIDR, 2005.

[11] M. Hammad et al.: Nile: A Query Processing Engine for Data Streams. ICDE 2004: 851.

[12] Actian Vectorwise DBMS. http://www.actian.com/.

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

[14] Vertica. http://www.vertica.com/.

[15] B. Chandramouli et al. Accurate Latency Estimation in a Distributed Event Processing System. In ICDE, 2011.

[16] P. Bernstein et al. Orleans: Distributed Virtual Actors for Programmability and Scalability. MSR Technical Report (MSR-TR-2014-41, 24). http://aka.ms/Ykyqft.

[17] Apache Hadoop 2.3.0 (YARN). http://aka.ms/Quslzk.

[18] B. Chun et al. REEF: Retainable Evaluator Execution Framework. PVLDB 6(12): 1370-1373 (2013).

[19] Microsoft Avro Library. http://aka.ms/Nxbdwg.

[20] R. Chaiken et al. SCOPE: easy and efficient parallel processing of massive data sets. PVLDB, 1(2), 2008.

[21] Expression Trees. http://aka.ms/K0fzli.

[22] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. SIAM Journal on Computing (1977).

[23] U. Cetintemel et al. S-Store: A Streaming NewSQL System for Big Velocity Applications. In VLDB, 2014.

[24] M. Stonebraker et al. C-Store – A Column-Oriented DBMS. In VLDB, 2005.

[25] P. A. Boncz, M. Zukowski, and N.Nes, MonetDB/X100: Hyper-pipelining query execution. CIDR, 2005, 225-237.

[26] M.-C. Albutiu et al. Massively parallel sort-merge joins in main memory multi-core database systems. In VLDB, 2012.

[27] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In SIGMOD, 2009.

[28] C. Engle et al. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In SIGMOD, 2012.

[29] Apache Storm. http://storm.incubator.apache.org/.

[30] B. Babcock et al. Models and issues in data stream systems. In PODS 2002.

[31] C. Jensen et al. *Temporal Specialization*. In ICDE, 1992.

[32] The LINQ Project. http://aka.ms/rjhi00.

[33] BlinkDB. http://blinkdb.org/.

[34] M. Zaharia et al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In SOSP, 2013.

[35] D. Murray et al. Naiad: A Timely Dataflow System. In SOSP, 2013.

[36] SQL Server CLR integration. http://aka.ms/Bbtg44.

[37] E. Liarou et al. Enhanced Stream Processing in a DBMS Kernel. In EDBT, 2013.

[38] T. Akidau et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In VLDB, 2013.