Trill – A High-Performance Analytical Data Processing Engine

1 INTRODUCTION

This whitepaper introduces and describes the key novel technical innovations introduced in Trill, a high-performance query processing engine for real time and offline temporal-relational data. Trill stands for a trillion events per day, which in many cases is a very conservative estimate of Trill's query processing speeds.

2 THE TRILL DATA AND QUERY MODEL

2.1 BACKGROUND

Each tuple in the system has two associated logical timestamps to denote when the tuple enters computation and when it leaves computation. Users can express a query Q over the timestamped data. Logically, the dataset is divided into snapshots formed by the union of unique interval endpoints of all events in the system. The query Q is logically executed over every snapshot. A result is present in the output at timestamp T if it is the result of Q applied to data that is "alive at" (whose intervals are stabbed by) T. This logical model of streams is called the CEDR model (see [1]). A physical implementation of this model shipped in 2009 as the Microsoft StreamInsight stream processing engine.

2.2 PHYSICAL MODEL IN TRILL

In order to get high performance with the CEDR logical temporal model, Trill makes some very careful and explicit design choices and restrictions (as compared to, for example, Microsoft StreamInsight) in the physical model. These are described next.

2.2.1 Data Events and Timestamps

In Trill, physical data events are annotated with two timestamps: SyncTime and OtherTime. Both timestamps are represented as long values (8-byte integers). A logical event with (logical) lifetime [Vs, Ve) may physically appear in two forms:

- An interval event with SyncTime = Vs and OtherTime = Ve
- A start-edge event and an (optional) end-edge event. The start-edge provides the start time of the interval and has SyncTime = Vs and OtherTime = ∞. The end-edge provides the end time of the interval in case it is not ∞. It has SyncTime=Ve and OtherTime=Vs.

Informally, SyncTime denotes the logical instant when a fact becomes known to the system. In case of an interval [Vs, Ve), it arrives as a start-edge at timestamp Vs and an end edge at timestamp Ve. Hence SyncTime for the start- and end-edge are set to Vs and Ve respectively. OtherTime represents additional information. In case of an interval, it provides some a priori knowledge of the future (i.e., that the event is known to end at some future

time Ve). In case of an end-edge, it provides additional context to the event (i.e., the start time that the event is originally associated with).

All streams in Trill are grouped. In other words, for every payload there exists a key that identifies a logical substream within the original stream. Keys (and their hash values) are materialized as part of events so that they do not need to be re-computed.

Trill constrains data to be ingested and processed in non-decreasing SyncTime order. In other words, all operators in Trill process and produce tuples in this order. This leads to interesting operator algorithm optimizations that we discuss in Section 3. Representing timestamps as SyncTime and OtherTime has several advantages:

(1) It allows us to represent both interval and edge events using two timestamp fields instead of three. This reduces the space overhead of events, as well as reduces memory bandwidth usage during query processing.
(2) Data in Trill is processed in SyncTime order, and thus SyncTime is frequently used in operator algorithms. Have the sync time explicit rather than computed on demand provides performance benefits.

2.2.2 Punctuations

Given that data is timestamped and exists in sync-time order, every event is essentially also a punctuation in terms of guaranteeing that future events will not have a lower sync time. However, Trill still has punctuations. Punctuations in Trill have two uses:

1) They indicate the passage of time in case there are lulls in the input (i.e., periods where there is no incoming data)
2) They may be egressed where data events are filtered out by operators.

2.2.3 Low Watermarks

For partitioned streams, Low Watermarks set a lower bound on all partitions in the stream, and any event ingressed before that lower bound will be considered out of order. This informs downstream operators, and the query consumer, of a global minimum time, allowing cleanup of stale state, etc.

2.2.4 Flushes

Trill internally tries to batch events before sending them to the next operator. Flushes serve to kick the system into producing output; this may involve closing out and pushing partially filled data batches. Flushes may be triggered explicitly using a QueryContainer, or automatically triggered via a FlushPolicy/PartitionedFlushPolicy. Currently, Flushes are represented as a StreamMessage with a StreamMessageKind.Flush.

2.2.5 Grouped Computation as part of the Model

Trill materializes all grouping information as part of events. Every stream is logically grouped. A stream grouped by key K logically represents n distinct sub-streams, one for every distinct value of the grouping key K. An ungrouped streams is represented as a stream grouped by Unit (an empty struct). There is a single timestamp domain across all groups. In other words, passage of time occurs across all groups and not on a per-group basis. The grouping key K and its hash value H are materialized and stored as part of events in Trill. Adding in the concept of multi-streams with per-group progress is part of our future work.

2.3 One Size Fits Many

A key advantage of the Trill model is that it supports a wide range of analytics. Specifically, we have used Trill to handle the following processing models:

1. Real-time temporal queries
2. Offline temporal queries
3. Relational (atemporal) queries
4. Progressive queries

Figure 1 shows how Trill's temporal model enables this wide range of analytics. While a single streaming engine can be used to execute all these models, using a single engine for these diverse use cases imposes a huge requirement on performance on the engine. The Trill project demonstrates that we can build a single engine with a general model that does not sacrifice performance in achieving this goal. We provides a *pay-as-you-go* approach to the engine that adheres to the following principles:

- Simple operations should be very fast.
- One should not pay for temporality unless temporality is necessary for the query.
- One should not pay for progressiveness (approximate results) unless it is needed.
- It should be very fast for simple data types, while degrading gracefully as types become complex.

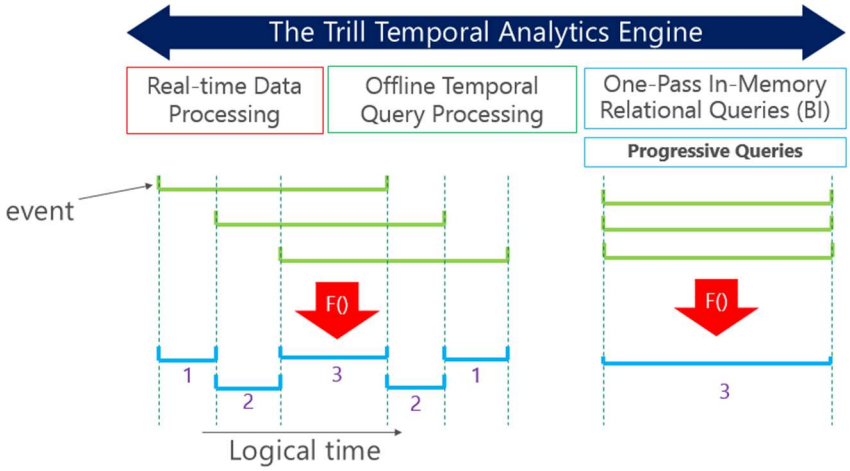

Figure 1: Trill – A One-Size-Fits-Many Architecture

3 An Emergent Design for a Streaming Engine

The goal for Trill was to get the highest possible query processing performance on a single machine. In order to reach this level of performance, it became clear that we need to minimize the amount of data transferred between the main memory subsystem and the CPU.

Trill uses a bottom-up or emergent architecture and design in order to eliminate bottlenecks commonly found in streaming engines. We started with how fast we could make a pass-through query, and added one operator after another while being careful not to degrade existing performance. This approach allowed us to avoid any single point of bottleneck in the system. The result is an architecture that carefully separates out two kinds of work done by a streaming engine:

1) *Fine-grained work*, which is done for every event in the system
2) *Coarse-grained work*, which is performed infrequently so that it gets amortized across multiple events

This led to a novel design where the unit of data processing at the engine level is a *message* – a single unit that can consist of a batch of physical events. The engine operates over a sequence of messages as input.

The query plan is organized as a graph of operators, each of which receives and generates messages. Operators perform fine-grained work in carefully written tight loops that leverage data locality, instruction locality, compiler loop-unrolling, unsafe code, code generation, and other optimizations that we discuss in subsequent sections. Coarse-grained work is performed by the engine and operators at message boundaries. This work includes handing off data across cores, allocating messages, and updating memory pools.

Each operator receives a batch, processes events within the batch, and writes output events into an output batch. The output batch is not handed off to the next downstream operator until the batch is full, or a control event (punctuation) is received. This exposes a nice throughput/latency tradeoff that we can exploit (Section 7 has more details).

3.1 TRILL ARCHITECTURE OVERVIEW

Trill is a .NET technology available as a library (i.e., no heavy-weight server). It allows the use of .NET types and expressions in query specification. Queries are written using LINQ (for *language integrated querying*). Figure 2 shows the overall system architecture and flowchart for Trill. We start with input data sources (for example, as observables) and build the logical query plan, finally connecting it to one or more outputs (for example, observables). During this logical plan creation, query compile-time work such as property derivation is also performed. When you subscribe to an output, the Subscribe() calls propage through the query plan, resulting in the construction of the physical DAG of operators (this includes the creation and instantiation of code-generated operators). Input observers feed content in the form of messages (see Section 4) to Trill via On() messages. This triggers query processing and results in output being pushed as it is produced to result observers.

Trill operators are supported by memory pool management interfaces that allow re-use of messages and data structures within the engine in order to reduce the cost of .NET allocation and GC. It also provides fast data structures such as FastDictionary and blocking concurrent queues to enable high performance. Finally, it incudes techniques to effectively scale out computation to multiple cores using a special version of GroupApply or using a more general scheduler design for operators.

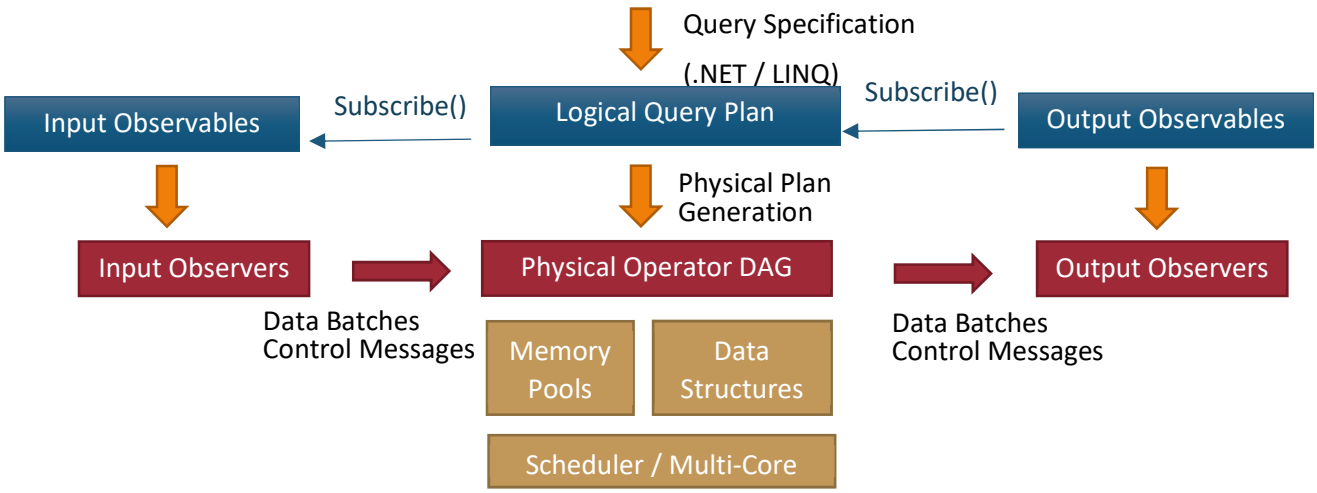

Figure 2: Trill System Architecture

4 Data Organization, Layout, and Memory Usage in Trill

4.1 Messages

As mentioned earlier, data is Trill is organized as a sequence of messages. A message can be of multiple types:

1. DataBatch: This is a bundle of physical data events carefully organized for high performance query processing. Punctuations and LowWatermarks are inlined into the databatch using sentinel OtherTime values.
2. Flush: This is a special control message that forces all operators in a query to output their batched events.
3. Completed: This is a special control message that indicates the end of stream

4.1.1 DataBatch Messages

The stream of incoming data events in Trill is organized into a sequence of batches. Each batch packs a large number of data events (up to BatchSize events).

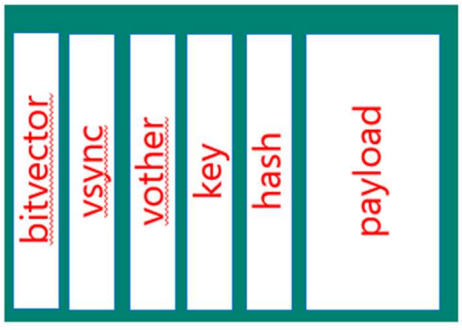

Figure 3: Data organization within a DataBatch

Each DataBatch stores the control parameters in columnar format, as shown in Figure 3. Specifically, each DataBatch contains the following arrays:

1) *SyncTime*: This is an array of the SyncTime values of all events in the batch.
2) *OtherTime*: This is an array of the OtherTime values of all events in the batch.
3) *Bitvector*: This is the occupancy vector – an array with one bit per event. A bit value of 0 indicates whether the corresponding data event exists, while a value of 1 indicates that the event does not exist (is absent). The bitvector allows efficient operator algorithms in some 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, just set the corresponding bitvector entry to 1.
4) *Key*: Recall from Section 2 that streams in Trill are grouped. *Key* is an array of the grouping key values of the data events within the DataBatch.
5) *Hash*: The hash value (4 byte integer) of the key. Keys and their hashes are pre-computed and materialized so that operators do not need to perform the expensive task of computing these.
6) *Payload*: Trill supports arbitrary .NET types as payloads. In the most basic form, Payload is an array of all the payloads within the DataBatch.

```
public class DataBatch<TKey, TPayload>
    {
        long[] SyncTime;
        long[] OtherTime;
        long BitVector;
        TKey[] Key;
        int[] Hash;
        TPayload[] Payload;
    }
```

Figure 4: Payload Organization (row-oriented)

Trill supports two modes for payloads: row-oriented and columnar. In both cases, the user view of data is row-oriented. In other words, users write queries assuming a row-oriented view of data. In row-oriented mode, the payload is simply an array of TPayload (the type of the payload) as shown in Figure 4. Trill also supports a columnar payload organization. To illustrate how this works, consider the following type in Figure 5 that a user is using to write their queries over:

```
public class TitanicPassenger
{
    bool survived;
    int pclass;
    string name;
    string sex;
    int age;
}
```

Figure 5: Example payload data type

In the row-oriented mode, the DataBatch contains an array of values of this type. As shown, each value is an instance of a class, so what is actually kept in the array is the address of a location in the heap where the different fields are stored in contiguous memory (except for the strings, which themselves are heap-allocated objects, so a string-valued field is itself a pointer to somewhere else in memory).

Clearly this does not provide the data locality required for high-performance computing. If the type is a “struct” instead of a class, then the values of the fields for each instance are kept within the array itself. This improves the data locality, but it is often the case that a particular query accesses only a small subset of the fields (and it is also much more common for these types to have **many** more fields than the five shown in this example). So in

executing the query, even though we can access the array of payloads in an efficient manner, the sparse access to the fields within each payload means that we still do not have sufficient data locality.

It is much more efficient to have the data organized as *column-oriented* where there is an array of values for each field, as shown in Figure 6. Now the implementation of a query can access contiguous elements of the arrays of the fields mentioned in the query which provides a much higher level of data locality. Therefore, we generate a column-oriented definition for each type that queries are expressed over.

```
public class BatchTitanicPassenger
{
    bool[] survived;
    int[] pclass;
    string[] name;
    string[] sex;
    int[] age;
}
```

Figure 6: Batched-columnar version of the data type

In the actual implementation, the type of each field in a column-oriented definition is actually a more complicated type: ColumnBatch<T> (where T is the type of the associated field); the array of values is just one part of a ColumnBatch. We discuss the ColumnBatch type in detail in Section 4.2. We also use the type ColumnBatch to store the control parameters.

The new version of Trill's data organization is depicted in Figure 8.

```
public class BatchGeneratedForTitanicPassenger<TKey>
{
    ColumnBatch<long> SyncTime;
    ColumnBatch<long> OtherTime;
    ColumnBatch<long> BitVector;
    ColumnBatch<TKey> Key;
    ColumnBatch<int> Hash;
    ColumnBatch<bool> survived;
    ColumnBatch<int> pclass;
    ColumnBatch<string> name;
    ColumnBatch<string> sex;
    ColumnBatch<int> age;
}
```

Figure 7: The generated columnar batch type in Trill

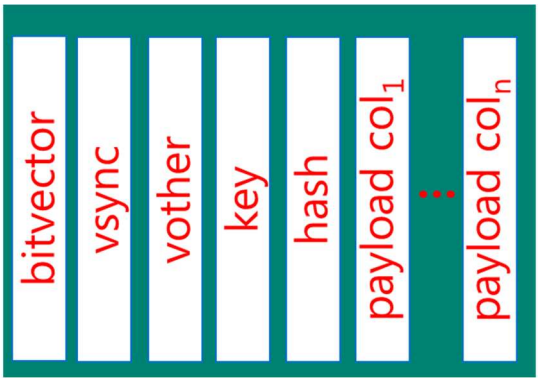

Figure 8: Data organization with columnar payloads

Note that queries are still written with a row-oriented view of the data – we describe how query processing occurs with columnar payloads in Section 5.

4.1.2 Control Messages

In addition to DataBatch messages, Trill supports control messages such as flushes and completed messages. The semantics of these messages are as described earlier.

4.2 Memory Management in Trill

A critical performance issue in stream processing systems is the problem of fine-grained memory allocation and release (garbage collection). Traditionally, these are very expensive operations and the same is true in .NET. We follow a novel approach to memory management that retains the advantages of the high-level world of .NET and yet provides the benefits of unmanaged page-level memory management. The advantage of not using unmanaged memory for memory management is that we completely sidestep the problem of handling complex .NET types.

In Trill, we introduce 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 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.

In Trill we have 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 second type is a memory pool for events, which is associated with a DataBatch type, and contains a *ColumnPool<T>* for each column type T necessary for the batch. The pool itself may be generated. A *ColumnPool<T>* contains a pool (implemented as a queue) of free ColumnBatch entries.

ColumnBatch<T> instances are ref-counted, and each ColumnBatch<T> instance knows what pool it belongs to. When the RefCount for a ColumnBatch instance goes to zero, it is returned to the ColumnPool. When an operator needs a new ColumnBatch, it requests the ColumnPool (via the MemoryPool) for the same. The ColumnPool either returns a pre-existing ColumnBatch from the pool if any, or allocates a new ColumnBatch.

In a streaming system, we expect the system to reach a “steady state” where all the necessary allocations have been performed. After this point, there should be very few new allocations occurring, as most of the time batches would be available in the pools.

We have experimented with one memory pool per operator and a single global memory pool configuration. We eventually expect to have one memory pool per core (or socket) serving all operators assigned to that core.

5 Query Compilation and Processing

As discussed in Section 4.1 above, queries in Trill are expressed against a row-oriented view of the data. The input data sources are modeled as instances of a type called IStreamable, that can for instance be created from existing data sources such as an Iobservable (see [2]). The application of each logical operator results in another IStreamable, allowing one to compose larger query plans. Some operators such as Join take two IStreamable instances as input and produce a single result IStreamable. The result of the query construction is therefore an IStreamable – note that at this point, no operators are generated and no query is executing.

We bind the query to a execution by creating an observer and subscribing to the query by calling Subscribe() on the IStreamble. This causes the compiler to walk the IStreamable DAG, allowing each logical operator to construct a physical operator instance, thereby generating the physical operator DAG. The data sources, on being subscribed to, start pushing events through the physical operators, producing results that are eventually delivered to the result observers.

For instance, assuming there is a stream *ps* whose payload is of type TitanicPassenger, some example queries are shown in Figure 9. The user has a logical view that the stream consists of values of type TitanicPassenger. Queries are expressed as method calls, in this case to the methods *Where* and *Select*. The argument to each method is a *function*: the notation *x => e* describes a function of one argument *x*. When the function is applied to a value (in this case of type TitanicPassenger), the result is the value *e*.

- *Where* is a *filter*: its value is a new stream consisting of those records (tuples) of the original stream whose field *survived* had a value of *true* and field *age* was greater than 30.
- *Select* is a *projection*: its value is a new stream consisting of records (tuples) of a new type, NewPassenger, distinct from TitanicPassenger, that has two fields. The first field is called *NameSex* and its value is the concatenation of the *name* field and the sex field of the corresponding record from the original stream. Likewise, the second field is called *Age* and its value is the value of the *age* field of the corresponding record.

```
ps.Where(p => p.survived && p.age > 30)
ps.Select(p => new NewPassenger{ NameSex = p.name + “,” + p.sex, Age = p.age })
```

Figure 9: Example queries about passengers on the Titanic.

We will use these queries to illustrate the different aspects of how we compile them into a form that executes against the data organization described in Section 4.1. Recall that in the actual implementation, there is no type TitanicPassenger; Trill has instead produced the type BatchGeneratedForTitanicPassenger. All queries that, at the user's level, involve the type TitanicPassenger instead use the generated type instead. This non-uniformity means that for efficient execution the query submitted by the user is compiled into a *custom operator*. The custom operator is a code module which is loaded (and executed, of course) dynamically in response to a user-formulated query.

5.1 Code-Generation at Query Compile-Time

We create custom physical operators using a combination of the .NET API for *expression trees* together with meta-programming. An expression tree is an abstract syntax tree representing a fragment of code. It uses the type Expression<T> to represent a piece of code of type T. The expression tree API also contains functionality for transforming abstract syntax trees.

For example, the type of the parameter for the method *Select* is Expression<Func<TPayload, U>> where Func<A,B> is the type of a function that takes an argument of type A and returns a value of type B.

Our transformation, in general, is to replace all references to a field, *f*, of a value, *r*, to become references to the i^{th} row in the column corresponding to *f*. There are further optimizations such as *pointer swings* (described in Section 6.2) that are constant-time operations on a batch, instead of having to iterate over each row in a DataBatch.

Once we have created the transformed expression tree, we use the meta-programming capabilities of T4 to create a C# source file which is compiled and dynamically loaded. The transformed expressions are in-lined into the operator.

It is not always possible to generate a custom operator. For example, a query might require calling a method on an instance of TitanicPassenger. In such a situation, we may or may not be able (or want) to recreate an instance given the values for its fields. We then transform the data to the row-oriented format and use the non-generated static (generic) definition of the operator (i.e., where each DataBatch has a column of Payload values).

The end result of this transformation is that we have – *in a fully incremental streaming setting* – the following capabilities:

1. The ability to use a columnar data representation when possible
2. The flexibility of using complex types by reverting to non-generated row-oriented code when necessary
3. The ability for users to write their queries using a single row-oriented view of the data

5.2 Transitioning Between Row and Columnar Representations

We use the same meta-programming techniques to generate specialized operators for transforming a row-oriented DataBatch into a column-oriented DataBatch (and vice-versa). This can occur either because of a) some property of the user type prevents it from being used in column-oriented processing, b) some limitation in the operator, or c) an expression in the query is too complex or opaque to allow its transformation.

In any case, data usually enters and leaves the system as row-oriented and so needs to be transformed at the beginning of the processing pipeline into columnar format and then re-created in row format at the end of the pipeline.

5.3 Compile-Time Stream Properties

Trill supports compile-time stream properties that define restrictions on the stream. Each IStreamable carries with it a set of properties that propagate across operators in the logical plan. When translating the query into its DAG of physical operators, we may use the properties to choose specific operator implementations that perform well for the given properties. For example, an interval-free stream may use a version of Aggregate that does not need to remember information about future end-edges (endpoints of intervals).

6 Efficient Temporal Operator Algorithms

All operators in Trill are SyncTime-ordered, and ingest/produce data in non-decreasing order of SyncTime. If data arrives out of order, we need to remove disorder (e.g., by buffering and reordering) before feeding it to the engine.

6.1 WHERE (FILTERING)

In the row-oriented mode, the filtering operator iterates over each row in the DataBatch, and if that row represents a valid tuple in the data, applies the user-supplied predicate to the payload. The BitVector is updated to record the value of the predicate. Note that *predicate* is a compiled form of the function that the user wrote.

```
for (int i = 0; i < input.Count; i++)
  if (BitVector[i])  { BitVector[i] = predicate(input.Payload[i]); }
```

The custom operator for *Select* in the example gets an input value, *input*, of type BatchGeneratedForTitanicPassenger. There are no values of type TitanicPassenger to apply the predicate to. Instead, we make the operator more efficient by inlining body of the (transformed) predicate which just accesses the columns corresponding to the fields *survived* and *age*.

```
for (int i = 0; i < input.Count; i++)
  if (BitVector[i])  { BitVector[i] = input.survived[i] && input.age[i] > 30; }
```

6.2 SELECT (PROJECTION)

The row-oriented version is shown in Figure 10: it is very similar to the filter except that for each row it creates a new value by applying the user-supplied function to the payload. As with the filter, the custom operator for *Select* is much more efficient because it needs access only the columns mentioned in the query. For instance, the value of the NameSex field is set by iterating over each row *i* in the batch:

```
for (int i = 0; i < input.Count; i++)
  if (BitVector[i])  { output.NameSex[i] = input.name[i] + “,” + input.sex[i]; }
```

The assignment is done conditionally because we have to check the BitVector to make sure that the row should be considered valid or not. However, there are even opportunities for further optimization: note that we do not have an assignment for the Age field within the loop. That is because the value of that field for every row in the

```
output.Age = input.age;
```

output is exactly the value of the field for that row in the input. Since the fields are independent objects (of type ColumnBatch<int>), we can just share the same column between the input and output. We refer to this as a *pointer swing*. We do not even need to iterate over each row *i*, but instead have a single assignment:

The generated version of the operator from Figure 10, for a specific project expression in shown in Figure 11.

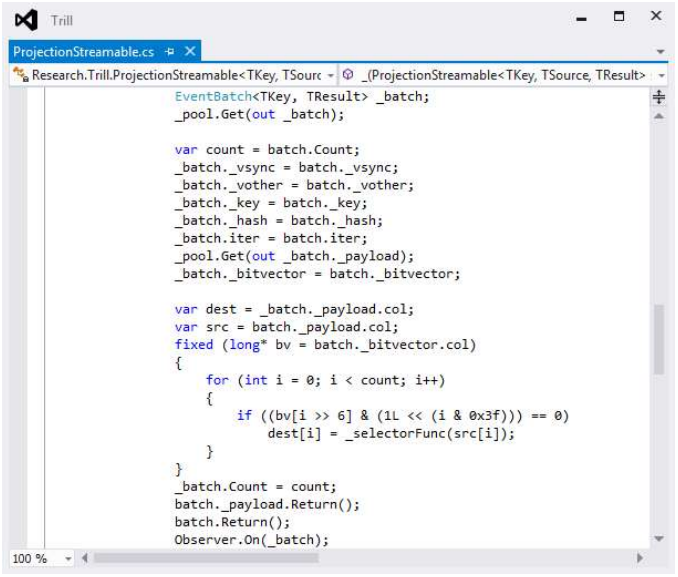

Figure 10: Non-generated code for Projection

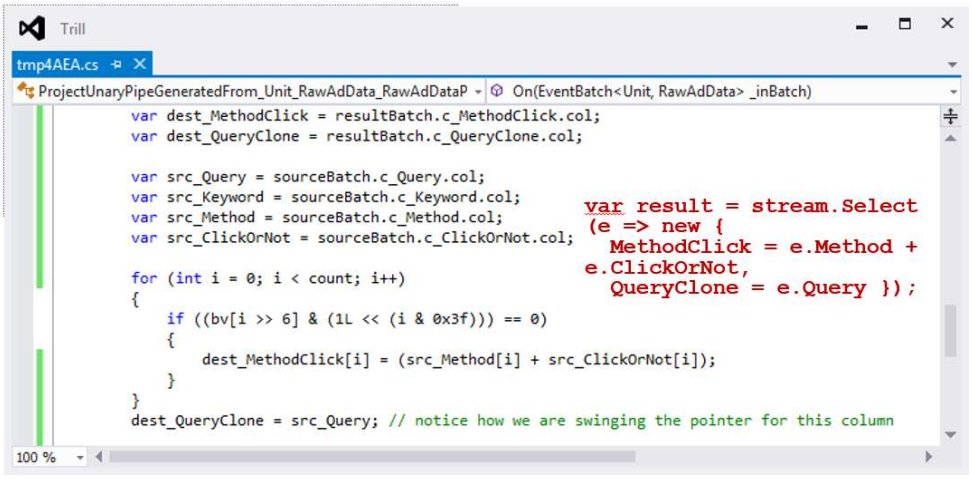

Figure 11: Generated version of the Projection operator from Figure 10

6.3 SNAPSHOT

The snapshot operator is a basic operator in Trill that produces incremental snapshot-oriented results. For a more detailed description of snapshots, see [1] [3]. The traditional technique for computing results for snapshot-oriented operators in described in prior work. In Trill, we propose a novel set of algorithms for the snapshot operator that achieve very high performance while adhering to the temporal semantics of the operation.

Trill distinguishes three cases to build efficient implementations of the snapshot operator:

1) **The stream has only start and end edges**: In this case, we know that an endpoint compensation queue or ECQ (see [1]) is not needed to store endpoints. The algorithm therefore becomes simpler with lesser code and efficient.

2) **The stream has start edges, end edges, and/or intervals of constant duration**: In this case, the intervals provide information for future points, but the information is presented in non-decreasing order. This implies that we can store the ECQ as a FIFO queue that adds items at one end and removes items from the other end.
3) **The stream has arbitrary events**: In this case, intervals may present endpoints in arbitrary order. Therefore we need a data structure that can add items in arbitrary order of time, but allows retrieving them in increasing timestamp order.

We describe the general algorithm for case 3 below. The first two cases simplify/elide certain parts of the code for efficiency.

Recall that every stream in Trill is grouped, i.e., every event has a grouping key. The aggregate operator uses two data structures to store state related to the aggregation operation:

1) **AggregateByKey**: This is a traditional hash table (with a custom implementation for performance) that stores, for every distinct key associated with non-empty state at the moment, an entry in the hash table with that key and the associated state.
2) **HeldAggregates**: This is a special hash table called FastDictionary, that stores – for the current timestamp T – all the partial aggregated state corresponding to keys for which events arrived with sync time equal to T. This hash table is emptied out whenever time moves forward. It does not require the ability to delete individual keys, but requires the ability to iterate through all entries very fast, and clear the hash table very fast when time moves to the next value. We describe FastDictionary in detail in Section 6.3.1.

The snapshot operator works as follows: as we receive events with the same sync time, we build up the current set of partial aggregates in HeldAggregates. When time moves forward, we issue start-edges for these partial aggregates (in case they are not empty) and fold these partial aggregates into the larger set stored in AggregateByKey. Entries in the AggregateByKey that are now empty are removed in order to maintain "empty-preserving semantics". End-edges are also issued as needed for prior aggregates. The HeldAggregates dictionary is cleared for reuse during the next timestamp.

We also maintain an *endpoint compensation queue* that contains, for each future endpoint, partially aggregated state for that endpoint. Whenever time moves forward, we process the endpoints between now and the new timestamp from the endpoint compensation queue. Our aggregate operator implementation also caches the state associated with the currently active key, so that the common case where all events have the same key can be executed very efficiently without hash lookups.

6.3.1 FastDictionary

The FastDictionary is a lightweight .NET dictionary optimized for

1. frequent lookups
2. small sets of keys
3. frequent clearing of the entire data structure
4. frequent iteration over all keys in the table

It tries to minimize:

1. memory allocations during runtime
2. hash computations

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 suitably 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 low load factor.

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. Note that this means that the GC can free older values only when entries are reused later, but this is usually not an issue. Accesses to the bitvector are very fast due to cache locality.

Performance We find that the FastDictionary performs up to 40% better than standard .NET Dictionary for streaming workloads, when used inside the aggregation operator to maintain partial aggregation states for the current timestamp.

6.4 JOIN

Join in Trill is a temporal join operation that works on input data events in sync-time order across its two inputs. The join is implemented as an equijoin with delegates, provided to determine a mapping key for each payload. The join key is the same as the grouping key associated with the stream. When a payload from the two inputs has the same key and overlaps temporally, the join executes another delegate to generate output for the pair of payloads. Outer join is implemented as an equijoin where all payloads have the same mapping key. We optimize the implementation for two special cases:

1) **The input has only start edges:** 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 a map data structure with the key and value equal to the mapping key and payload, respectively, of the input event. Additionally, the map data structure for the alternate input is searched to identify events received with the same mapping key from the alternate input. Any payloads identified with the same mapping key corresponds to a successful join for which the operator will generate output.
2) **The input has arbitrary events:** 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 map data structures similar to the start edge case, but also employs an endpoint compensation queue to remove intervals once time progresses to their ending timestamp. Additionally, the operator cannot determine which newly processed payloads successfully join start edges from the alternate input until time progresses beyond the current timestamp (because any active start edges could always be ended by an end edge). Note that this delay in output generation is only for the case of an event joining against a start edge because intervals have known ending timestamps. When events are removed from the map, either by end edges or the reaching of an interval's ending timestamp, the operator performs a search of the alternate input's map to identify payloads which joined previously. For each of those payloads, the operator outputs a corresponding end edge. The case of an interval joining with an interval is handled specially because the exact duration of the join is known beforehand, so the operator outputs an interval for the corresponding duration.

We also have specialized versions of Join to handle two cases:

1) The inputs are asymmetric (left side is much smaller/lower throughput than the right side)
2) The inputs are sorted by the join key or its superset

These variants of join are discussed in Section 9.

6.5 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 (described in Section 8) operations. 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.

7 PUNCTUATIONS, LOW WATERMARKS, AND THE PASSAGE OF TIME

7.1 PUNCTUATIONS

Data events in Trill also serve as implicit punctuations until the sync time of the event because of the property that all streams are in non-decreasing sync time order. A novel aspect of the Trill engine is that it retains an explicit notion of punctuations, which signify the passage of time during periods of lulls (i.e., no data). Note that even if the input stream has data events, there may be an operator such as Where that filters out most (or all) events leading to lulls in its output stream.

For partitioned streams, punctuations apply to a specific partition in the stream, indicated by the key in the punctuation event.

7.2 Low Watermarks

For partitioned streams, Low Watermarks set a lower bound on all partitions in the stream, and any event ingressed before that lower bound will be considered out of order. This informs downstream operators, and the query consumer, of a global minimum time, allowing cleanup of stale state, etc.

7.3 Flush

Trill tries to maximize performance by ingressing data in batches. Likewise, operators write output data events to a temporary holding batch, and wait before propagating the batch to the next downstream operator, until one of the following occurs:

1. The batch is full
2. You get a flush control message at the input

Thus, flushes serve to "kick the system" and force it to generate output regardless of the current state of operators in the query plan.

7.4 Batching, Latency, and Memory Tradeoffs

Flushes and the data batch maximum size control the throughput/latency tradeoff in Trill. Trill operators try to fill up a batch before sending it to the next operator, thereby introducing latency. Users may introduce frequent flushes in order to force the engine to output batches early, thereby reducing latency.

However, a very frequent flush strategy may result in many batches being almost empty, which could result in a significant waste of memory.

A simple first step towards solving this problem is to allow users to control the batch size using a global configuration setting. This allows us to avoid some really bad cases, but does not cover the cases where different operators see different flush frequencies due to query and data semantics. For example, consider a query with an ingress policy of FlushPolicy.FlushOnPunctuation, in which each punctuation generated or ingressed will trigger a flush. An input may have a punctuation every 1000 events, but the result of a filter with selectivity 0.2 would have a punctuation every 200 events. The punctuation frequency may also be data dependent and change with time (e.g., how many tuples join with a given input in Join). Thus, we propose a dynamically adaptive batch-size setting algorithm that works as follows. Let DefaultBatchSize denote the "ideal" batch size in the absence of punctuations, where performance does not improve significantly (on expectation) when the batch size is increased further (80,000 by default). Let StartBatchSize be the smallest batch size that gives reasonable performance on expectation, such that a lower batch size degrades performance significantly (250 by default).

For each operator:

1. Start the batch size at MaxBatchSize = StartBatchSize
2. Observe the first k punctuations, or wait until the first DefaultBatchSize events are observed: set batch size to MaxBatchSize = MIN(max. batch fill * 1.25, DefaultBatchSize)
3. Keep a running average of observed actual outgoing batch sizes, and adjust MaxBatchSize periodically.

The memory pools (per operator) are also modified to handle varying batch sizes by allowing us to return batches of the older size to the GC instead of returning them to the pool. As a result, the memory pool at any given moment only contains batches of a fixed size, which may adjust periodically.

8 TWO-STAGE GROUPED SUBQUERY PROCESSING

Trill supports the notion of grouped sub-queries. The basic idea is that given an input stream, the user provides a grouping key and a subquery to be executed for every distinct value of the key. Logically, the sub-query is executed on every substream consisting of events with the same distinct value of the key. The *GroupApply* operator in Trill executes such a query very efficiently on a multi-core machine. We now describe the novel two-stage architecture of GroupApply that makes this possible.

We logically model the GroupApply operator using a DAG as shown in Figure 12. The operation has multiple stages that we describe next. Overall, there are NumBranchesL1 copies of the map/shuffle stage, and NumBranchesL2 copies of the apply/reduce stage

8.1 SPRAY/MAP

The first step takes a stream of batches and performs a stateless spray of the batches to multiple cores (equal to NumBranchesL1) via blocking concurrent queues. This operator performs a constant amount of work per batch and hence introduces negligible overhead to the system.

8.2 MAP, GROUP, SHUFFLE

This operator resides on each of NumBranchesL1 cores, and receives events in a round-robin fashion from the sprayers. The shuffle maintains NumBranchesL2 number of output batches, for each incoming batch, it applies the map subquery and then generates the grouping key and its associated hash function on the resulting stream. Based on the hash value, it stores the tuple in one of the NumBranchesL2 partial output batches. When any output batch gets filled up, the batch is delivered to the corresponding Apply branch via a lock-free blocking concurrent queue.

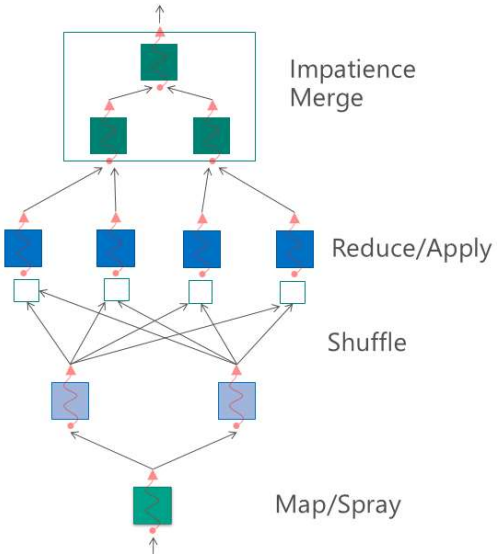

Figure 12: Two-stage streaming GroupApply architecture

8.3 Merge, Apply, Ungroup (Streaming Reduce)

At each of the NumBranchesL2 cores, we first perform a union of data from NumBranchesL1 upstream inputs. The union is a temporal union (maintaining timestamp order) in order to retain the invariant that all streams in Trill are in strict sync-time order. This is followed by an application of the reduce sub-query on each of the cores. This is followed by an ungroup operation that unpeels the grouping key (nesting is allowed). The results of these are fed to the final merge operation.

8.4 Final Merge

The result batches from the reduce operations need to be merged into a single output stream in temporal order. This operation is performed by the same merger from Section 8.3 that merges multiple streams into one. We use a sequence of cascading binary merges in order to scale out the merge across multiple cores (each binary merge can potentially reside on a different core). Further, the use of binary merges provides significant performance benefits over using a priority queue to perform a single n-way merge. We typically use three cores to perform the merge: one for the root, and two more for the left and right sub-trees respectively, although other configurations are also possible.

8.5 Multi-Input Apply/Reduce

Trill also supports an apply branch with two inputs. The basic idea and architecture are similar to the description above, 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 two-input reducers. The details are omitted for brevity.

9 DISCUSSION

9.1 SORTED DATA AND EXPLOITING SORT ORDERS

In case we are performing offline relational queries (including progressive queries), we may have the ability to pre-sort the data by some key in order to optimize query processing using Trill. Trill supports a compile-time property to identify whether the data is snapshot-sorted (i.e., whether each snapshot is sorted by some payload key). In case of progressive data, snapshot-sorted implies a global sort order for the data.

Trill also supports the notion of sort-order-aware data packing. The basic idea here is that sorted data is packed into batches 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*

The spray phase of GroupApply can exploit this packing scheme to retain the sort order during spray. Basically, it retains the last key in the current batch B before spraying it to core *i*. In case the first element in the next batch B+1 has the same key value, that batch is also sprayed to the same core *i*. Otherwise, the batch B+1 is sprayed as usual to the next core *i+1*. This choice allows the sort ordering and packing property to be retained within each downstream branch.

Further, if the GroupApply key happens to be equal to or a subset of the sorting key, we move the apply sub-query into the spray phase, thereby completely avoiding the shuffle.

9.2 ASYMMETRIC JOIN

Trill supports a variant of join called the asymmetric join. The basic idea is that if the left side of the join is such smaller than the right side, we multicast the left side to all the cores and simply spray the right side round-robin across the join operator instances. This allows us to completely avoid the shuffle phase of the map-reduce, at the expense of having a duplicate copy of the (smaller) left side at all cores. The user is allowed to choose this variant of Join by specifying an optional parameter to the Join operator. The asymmetric join operation is supported by an asymmetric two-input reduce phase that multicasts the smaller side and sprays the larger side of the reducer.

9.3 SORT-ORDER-AWARE JOIN

In case the data is sorted by same key (or a superset) as the join key, we can use a more efficient version of the join that does not perform any hashing of tuples. This is similar to the merge-join operator in databases. Trill automatically chooses between the traditional hash join and the merge join based on compile-time stream properties.

9.4 SCHEDULERS

Our current implementation of Trill uses the GroupApply operator as the building block for scaling out on to multiple processor cores. We have also designed a more general scheduler mechanism to allow more effective sharing of resources across multiple queries and operators. The basic idea is to create and assign a scheduler per core, and allocate operators across these schedulers. The key novelty is that the scheduler logic operates purely at batch boundaries, thus avoiding a performance penalty. Communication across schedulers (in case a

downstream operator is assigned to a different scheduler) is accomplished using efficient lock-free queues of batched messages.

10 CONCLUSIONS

Trill is a very fast lightweight stream processing engine. It executes temporal streaming queries at speeds between 10 and 1000 times faster than engines such as StreamInsight and Rx. The key innovations in Trill that enable this level of performance include a bottom-up emergent system design, along with techniques such as aggressive physical batching, columnar data storage with row-oriented data access, code-generation for highly optimized operations, a carefully restricted physical model with new sync-time-ordered operator algorithms, and a careful separation of fine-grained work performed with each event and coarse-grained work performed at boundaries of groups (or batches) of events. Trill enables *one-size-fits-many*: a single engine can be effectively and efficiently used for real-time and offline temporal queries, as well as relational queries and progressive relational queries.

11 BIBLIOGRAPHY

[1] R. Barga, J. Goldstein, M. Ali and M. Hong, "Consistent Streaming Through Time: A Vision for Event Stream Processing," in *CIDR*, 2007.

[2] Microsoft, Reactive Extensions for .NET.

[3] B. Chandramouli, J. Goldstein and S. Duan, "Temporal Analytics on Big Data for Web Advertising," in *28th International Conference on Data Engineering (ICDE '12)*.

[4] B. Chandramouli, J. Goldstein and A. Quamar, "Scalable Progressive Analytics on Big Data in the Cloud," in *40th International Conference on Very Large Data Bases (VLDB '14)*.

[5] M. Barnett, B. Chandramouli, R. DeLine, S. Drucker, D. Fisher, J. Goldstein, P. Morrison and J. Platt, "Stat! - An Interactive Analytics Environment for Big Data," in *2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*.