User-Defined Aggregates

In addition to Trill's built-in aggregates (i.e., Count, Sum, Average, Max, Min, TopK), Trill provides a framework for users to create their own custom aggregates. The framework only supports computation that adheres to the pattern of a Trill aggregate, which is a reducing operation performed on snapshots in time. As a result, an aggregate can only change its output value in response to a change in the input synopsis (the set of events active at an instance in time). As long as the user's computation meets these requirements, Trill's aggregate framework enables both high-performance and flexibility.

The framework is designed with an emphasis on performance, which is evident in the framework's extensive use of expressions. Expressions use C#'s lamda syntax to express computation that the framework can introspect, manipulate, and optimize. These capabilities allow aggregates to run as fast as hand-coded operators. In fact, all of Trill's built-in aggregates are also written in the aggregate framework. However, expressions can also make calls to normal C# functions to support cases that do not map well to the lamda syntax.

Creating a user-defined aggregate requires writing two pieces of code. The first piece specifies the computation of the aggregate and requires the user to write a class that implements the IAggregate interface. Implementing the IAggregate interface, described in the next section, requires writing five functions that return expressions to perform the computation of the aggregate. The second piece of code specifies how a query writer invokes the aggregate and requires the user to write an extension method on the Window class. The Window class extension method, described in the third section, creates the user-facing interface of the aggregate.

Implementing IAggregate

```
namespace Research.Trill.Aggregates
{
    public interface IAggregate<TInput, TState, TResult>
    {
        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();
    }
}
```

Figure 1. The IAggregate interface.

As detailed in Figure 1, the IAggregate interface is a generic interface that requires three type parameters and specifies five functions. The TInput and TResult type parameters specify the input and output data types for the user-defined aggregate. The TState type parameter specifies the struct or class that the aggregate requires to persist its internal state (called the "state object").

The state object is required to store all mutable internal state of the aggregate. Class instance variables must not be used for mutable state as the framework may share the same instance across several independent aggregations in a Group & Apply. Only state that is constant throughout a Group & Apply may be stored as instance variables. For example, the TopK aggregate persists the integer value 'k' as an instance variable because all groups within a Group & Apply would use the same value of 'k'. However,

the Sum aggregate persists the running sum within the state object because each group within a Group & Apply may have a different running sum.

The five functions of the IAggregate interface all return expressions that read and/or return a state object. The framework invokes the expression returned by InitialState to initialize a new state object. The framework invokes the Accumulate and Deaccumulate expressions in response to an event entering or leaving the synopsis, respectively. For both expressions, the first parameter is the current state object of the aggregate, the second parameter is the current timestamp, and the third parameter is the payload of the event that entered or left the synopsis. Both expressions then return the updated state object, which reflects the addition or removal of the event from the synopsis. The framework invokes the ComputeResult expression to calculate the output value of the aggregate from its state object.

The Difference expression is used by the framework to optimize batch deaccumulations. Specifically, the Accumulate and Deaccumulate expressions specify how to add and remove single events from the synopsis. However, in some circumstances the framework has knowledge of time instances when several events may simultaneously leave the synopsis (e.g., when multiple intervals have the same end timestamp). In these circumstances, the framework optimizes their eventual removal by creating a temporary state (not exposed as an output), invoking the Accumulate expression to accumulate events that will eventually need to be removed at the same timestamp, and then invoking Difference on the actual state, with the temporary state passed-in, to batch deaccumulate all of the events leaving the synopsis. For the Difference expression, the first parameter is the minuend and the second parameter is the subtrahend. As an example, consider a Sum aggregate with state S that has already accumulated active intervals I_0, I_1, and I_2. If I_0 and I_2 end at the same timestamp, then the framework will create a temporary state S_T and accumulate I_0 and I_2 to that state. When the time corresponding to the end timestamp arrives, then the framework will invoke the Difference expression with S as the first parameter and S_T as the second parameter, and persist the result as the updated S.

```
public class MySumAggregate : IAggregate<int, int, int>
{
    public Expression<Func<int>> InitialState()
    {
        return () => 0;
    }
    public Expression<Func<int, long, int, int>> Accumulate()
    {
        return (oldSum, timestamp, input) => (int)(oldSum + input);
    }
    public Expression<Func<int, long, int, int>> Deaccumulate()
    {
        return (oldSum, timestamp, input) => (int)(oldSum - input);
    }
    public Expression<Func<int, int, int>> Difference()
    {
        return (leftSum, rightSum) => (int)(leftSum - rightSum);
    }
    public Expression<Func<int, int>> ComputeResult()
    {
        return sum => sum;
    }
}
```

Figure 2. Example implementation of a Sum aggregate.

Figure 2 illustrates the code required to implement a Sum aggregate that takes an “int” data type as input, maintains an internal state (which is actually the running sum) as an “int”, and outputs the result also as an “int”. Note that there are no requirements for the input data type, state data type, and output data type to be the same. For example, it would be trivial to modify Figure 2 to have the running sum and output both be “long” data types to lessen the risk of overflow.

As illustrated in Figure 2, when the framework creates a new state, the expression for InitialState initializes it to zero. The Accumulate and Deaccumulate expressions add or subtract the payload of the corresponding event from the running sum. The Difference expression subtracts two running sums. Lastly, the ComputeResult expression simply returns the running sum as the result.

```
public struct AverageState
{
    public long Sum;
    public ulong Count;
}
public class MyAverageAggregate : IAggregate<int, AverageState, double>
{
    public Expression<Func<AverageState>> InitialState()
    {
        return () => new AverageState();
    }
    public Expression<Func<AverageState, long, int, AverageState>> Accumulate()
    {
        return (oldState, timestamp, input) => new AverageState
            { Count = oldState.Count + 1, Sum = oldState.Sum + input };
    }
    public Expression<Func<AverageState, long, int, AverageState>> Deaccumulate()
    {
        return (oldState, timestamp, input) => new AverageState
            { Count = oldState.Count - 1, Sum = oldState.Sum - input };
    }
    public Expression<Func<AverageState, AverageState, AverageState>> Difference()
    {
        return (left, right) => new AverageState
            { Count = left.Count - right.Count, Sum = left.Sum - right.Sum };
    }
    public Expression<Func<AverageState, double>> ComputeResult()
    {
        return state => (double)state.Sum / state.Count;
    }
}
```

Figure 3. Example implementation of an Average aggregate.

Figure 3 illustrates the code required to implement an Average aggregate that takes an “int” data type as input, maintains an internal state using a user-defined “AverageState” struct (which tracks a running sum and count), and outputs the result as a “double”. The expressions are nearly identical to the Sum aggregate but maintain both a running sum and count in the state object. Only when calculating an output value does the aggregate actually divide sum by count to calculate the average.

Creating the Extension Method

After writing the computation of the aggregate, by implementing IAggregate as described above, the next step is then to create the extension method which allows the aggregate to be utilized by the query

writer. It is the extension method that will be listed by Intellisense and actually invoked by the user, as he or she writes a query. The logic of the extension method is relatively simple and only needs to create an instance of the aggregate's class, specifying any required parameters and transformations (described in more detail below).

As a user writes a query, the syntax for invoking an aggregate typically takes a form similar to:

```
var avg = input.Aggregate(a => a.Max(v => v.Field1), b => b.Count(),
                          (c, d) => new { Max = c, Count = d });
```

For the parameters to the "Aggregate" function above, each parameter (except for the last) specifies an aggregate to apply to the input, with the last parameter specifying how to merge the individual results to produce a single output. Aggregates are able to take zero, one, or several input parameters. For example, the Max aggregate above requires a "selector" expression that specifies which field to perform the Max over, but the Count aggregate does not have any parameters. If the TopK aggregate were used, it takes two parameters – one that specifies the value 'k' and another for the selector expression.

The type of both *a* and *b* above are Research.Trill.Window<TKey, TSource>, which is a construct of Trill to facilitate Intellisense in listing the correct aggregates for a given payload type. Therefore, making a user-defined aggregate available to the query writer requires writing an extension method that accepts Window<TKey, TSource> as the "this" parameter. Once the extension method is written (and assuming the query writer has the class with the extension method accessible via a "using" statement), the new aggregates will then be useable and appear in Intellisense.

```
public static class MyExtensions
{
    public static IAggregate<TSource, AverageState, double> MyAverage<TKey, TSource>(
        this Window<TKey, TSource> window, Expression<Func<TSource, int>> selector)
    {
        var aggregate = new MyAverageAggregate();
        return aggregate.Wrap(selector);
    }
}
```

Figure 4. Example extension method for the average aggregate in Figure 3.

Figure 4 illustrates the extension method required for the average aggregate in Figure 3. The first argument of the function must be a "this" parameter of the "Window" class to enable the extension method semantics, but all of the subsequent arguments are user-defined. As illustrated in Figure 4 (and by most of the built-in aggregates), a good second parameter is to specify a selector expression on the input. Other arguments could include any specific parameters for the aggregate itself (for example, the 'k' parameter for the TopK aggregate).

The purpose of the selector expression is to allow the query writer more flexibility when applying the aggregate to a stream. For example, the selector expression allows the query writer to apply the aggregate to a specific column in a multi-column payload, as illustrated below:

```
var avg = input.Aggregate(a => a.MyAverage(v => v.Field1));
```

The selector expression could also be more complex and even reference multiple columns with arithmetic operations:

```
var avg = input.Aggregate(a => a.MyAverage(v => v.Field1 * v.Field2 + 3));
```

In general, the selector expression specifies a function that maps the data type of the payload (which is the “TSource” type parameter in Figure 4) to the data type expected at the input of the aggregate. For example, the prototype of the extension method in Figure 4 requires the selector expression to produce an integer value because that is the required input of MyAverageAggregate. If the query writer writes a selector expression that returns an incompatible type, then C# will inform the user of a compiler error. It is through this mechanism that type safety is preserved when invoking aggregates.

As C# allows overloads for functions (as long as the input parameters don’t have identical types), it is allowable to have several overloads with the same aggregate name that accept different selector output types. With this mechanism, for example, a user could write several versions of MyAverage with some that operate on “int” input values and others that operate on “long”, “float”, etc. As each extension method is unique, they could also internally use different implementations of the aggregate.

As illustrated in Figure 4, the code inside the extension method is only a few lines. The first line constructs the aggregate, possibly passing-in any required parameters to the constructor (such as ‘k’ for TopK). The second line then performs a “Wrap” function, which combines the aggregate and selector. As the selector provides a mapping from “TSource” to “int”, the Wrap function embeds the selector expression into the expressions of the aggregate, allowing it to create the resulting IAggregate that has an input type parameter of “TSource”. All of the aggregate extension methods must return an IAggregate with “TSource” as the input type parameter.

“Wrap” is one of several helper functions provided by the aggregate framework to help writers of the aggregate extension method. All of the helper functions take in an IAggregate and output another IAggregate (with potentially modified type parameters). Internally, the helper functions apply transformations to the expressions to either manipulate values or specially handle specific cases. Due to the expressiveness of expressions, these transformations actually rewrite the expressions instead of causing nested function calls, thereby minimizing impact on performance.

Table I. List of the Aggregate Framework's helper functions.

Function Name	Description
Wrap(transform)	Returns a new aggregate where all input values will go through the transform function before calling the underlying aggregate.
SkipNulls()	Returns a new aggregate that will not call accumulate and deaccumulate on null values – this includes both Nullable<T> nulls and reference nulls, depending on the input data type. If the input data type is not nullable, then this function returns the aggregate unchanged.
MakeInputNullableAndSkipNulls()	This function only supports aggregates with an input data type that is valued-type and non-nullable. If so, it returns a new aggregate with a nullable-version of the same input data type and will automatically skip all null values.
MakeOutputNullableAndOutputNullWhenEmpty()	This function only supports aggregates with an output data type that is valued-type and non-nullable. If so, it returns a new aggregate with a nullable-version of the same output data type and will automatically output null instead of invoking ComputeResult on an empty snapshot.
OutputDefaultWhenEmpty()	Returns a new aggregate that will automatically output default(output data type) instead of invoking ComputeResult on an empty snapshot.

Table I lists the helper functions provided by the aggregate framework. Many of the helper functions are meant to provide transformations that allow aggregates to support null values. Handling nullable values is a common case for overloading aggregates. For example, if the MyAverage aggregate had a selector that returned an input type of nullable int's ("int?"), C# would not use the extension method in Figure 4 because "int?" and "int" are different types (and there is no implicit cast from "int?" to "int"). Instead, another extension method would need to be written that allows for the selector output type to be "int?". In addition, the writer of the extension method would also need to determine what the behavior is for handling nulls.

While there are many different schools of thought for what is the correct behavior for nullable values, the semantics provided by the helper functions and used by Trill's built-in aggregates is to ignore null values. This semantic essentially considers a "null" value to mean "don't include this value in the aggregate's computation". Note that the built-in Count aggregate is one exception as it does count null values. A side effect of ignoring null inputs is that an aggregate must also be prepared to return a value for cases where a snapshot consists of only null values. For example, in the case of the MyAverage aggregate, if a snapshot consisted of only null values, then the aggregate would have no choice but to calculate an average performed over a snapshot with no integer values. As there is no meaningful average in this case, the most appropriate choice is for the aggregate to output null, which also requires making the output type of the aggregate nullable. With these changes, the resulting behavior of the

MyAverage aggregate would then be to output the average of only the non-nullable integer values in a snapshot, and output null only for the case when the input consists entirely of null values. The framework's helper functions provide this behavior without needing to rewrite the aggregate's computation.

```
public static class MyExtensions
{
    public static IAggregate<TSource, NullOutputWrapper<AverageState>, double?>
        MyAverage<TKey, TSource>(this Window<TKey, TSource> window,
        Expression<Func<TSource, int?>> selector)
    {
        var aggregate = new MyAverageAggregate();
        var aggregate1 = aggregate.MakeOutputNullableAndOutputNullWhenEmpty();
        var aggregate2 = aggregate1.MakeInputNullableAndSkipNulls();
        return aggregate2.Wrap(selector);
    }
}
```

Figure 5. Example extension method that uses helper function to support null inputs.

Figure 5 illustrates how the helper functions allow a user to transform an aggregate that does not support nulls into one that does. Notice that the extension method's return type now has a state object wrapped by a "NullOutputWrapper" and outputs a "double?", despite the extension method still internally creating an instance of the same "MyAverageAggregate" defined in Figure 3. The "MakeOutputNullableAndOutputNullWhenEmpty" and "MakeInputNullableAndSkipNulls" helper functions change the type parameters of the aggregate and transform the aggregate's expressions to support null values. Specifically, the "MakeOutputNullableAndOutputNullWhenEmpty" transforms the output type to "double?" and modifies the behavior to output null automatically in lieu of invoking the ComputeResult expression on an empty snapshot. Empty snapshots are possible because the snapshot could consist entirely of null values, which the "MakeInputNullableAndSkipNullValues" transformation would remove. As the transformation requires tracking the count of active events (to determine when it is empty) the transformation additionally modifies the state type parameter by wrapping it in a "NullOutputWrapper" struct. This change is completely invisible to the underlying aggregate. The second transformation is then to apply "MakeInputNullableAndSkipNulls" which will change the input type of the aggregate from "int" to "int?" with the behavior of automatically dropping null values. The last transformation is the "Wrap" function that embeds the selector expression, which provides a mapping from TSource to "int?", so that the resulting aggregate has an input type of TSource.