User-Defined Aggregate Operators in **Tutorial D** and *Rel*

Dave Voorhis <dave *at* reldb *dot* org>

Version 1.07 August 2016

As defined by Date & Darwen in *The Third Manifesto*[i] series of documents, the database language **Tutorial D** supports pre-defined aggregate operators. For example, given the following relvar…

```
VAR myvar REAL RELATION {x INT, c CHAR} KEY {x};
```

…containing the following relation…

x *INTEGER*	**c** *CHARACTER*
1	A
2	A
3	B
4	B
5	B
10	D
12	D

…the following expression…

```
SUM(myvar, x)
```

…will return 37.

However, this is not a typical **Tutorial D** operator invocation. The first operand may be any relation, and the second operand may be any attribute expression in the scope of a tuple of the relation. For example, this expression…

```
SUM(myvar, x * 2)
```

…will return 74. Whilst it is reasonable to regard SUM as being an operator that accepts any relation for the first argument (similar to IS_EMPTY() and the like), there is nothing in **Tutorial D** to represent an unevaluated expression as an operand. Therefore, SUM(myvar, x * 2) is perhaps better regarded of as a "pseudo" operator invocation that the compiler translates into some conventional – but otherwise less ergonomic – expression and/or operator invocation.

Like SUM, aggregate operators AVG, MIN, MAX, UNION, XUNION, COUNT – and others – are predefined and "built in", but **Tutorial D** does not define facilities for creating user-defined aggregate operators.

User-defined Aggregation

Ideally, we would like to be able to create user-defined aggregate operators. The remainder of this paper proposes mechanisms for doing so, which have been prototyped[ii] in the *Rel*[iii] implementation of **Tutorial D.**

As a first step, we propose a generic aggregate read-only operator invocation that permits in-line specification a user-defined body similar to an operator body. For example:

```
AGGREGATE(myvar, x);
    RETURN VALUE1 + VALUE2;
END AGGREGATE
```

This expression returns 37, the same as SUM(myvar, x).

It specifies that a hidden accumulator is set to the evaluation of *x* for the first tuple of the first operand to AGGREGATE. For each subsequent tuple in that operand, the accumulator is set to the accumulator + *x.* When there are no more tuples, the accumulator is returned. Its similarity to an operator definition is deliberate, as it is effectively defining an in-line operator similar to (and internally using) another *Rel* extension to **Tutorial D** for defining anonymous operators (aka lambda expressions)[iv].[1]

If there are no tuples in the relation, an exception is thrown.

Similarly, the following…

```
AGGREGATE(myvar, x);
    RETURN VALUE1 * VALUE2;
END AGGREGATE
```

…specifies that an accumulator is set to the evaluation of *x* for the first tuple in *myvar*. For each subsequent tuple in *myvar*, the accumulator is set to the accumulator * x.

The code between *AGGREGATE(…); … END AGGREGATE* may be arbitrarily complex, but must return a value of the same type as the second operand to AGGREGATE, which is also the same type as the generated parameters VALUE1 and VALUE2.

(Aside: The unimaginative parameter names VALUE1 and VALUE2 might benefit from being changed to something more intuitive.)

[1] The syntax above is essentially a shorthand for:
AGGREGATE(myvar, x, OPERATOR (VALUE1 INT, VALUE2 INT) RETURNS INT; RETURN VALUE1 + VALUE2; END OPERATOR))

I.e., it is:
AGGREGATE(myvar, x~~, OPERATOR (VALUE1 INT, VALUE2 INT) RETURNS INT~~; RETURN VALUE1 + VALUE2; END ~~OPERATOR)~~)

There are circumstances where it is desirable to set the accumulator to a default value, and return it if the relation is empty instead of throwing an exception. Another argument can be passed to AGGREGATE to indicate this identity or initial value, which must also be the same type as the second operand to AGGREGATE. For example…

```
AGGREGATE(myvar, x, 0);
    RETURN VALUE1 + VALUE2;
END AGGREGATE
```

…will return 0 if myvar is empty, and…

```
AGGREGATE(myvar, x, 10);
    RETURN VALUE1 + VALUE2;
END AGGREGATE
```

...will return 10 if myvar is empty or 47 if myvar contains the relation described at the beginning of this document.

Just as SUM and the other predefined aggregate operators may be used in the SUMMARIZE operator, AGGREGATE may be used too. For example, the following…

```
SUMMARIZE myvar BY {c}: {
  total := AGGREGATE(x); RETURN VALUE1 + VALUE2; END AGGREGATE
}
```

…returns:

c ***CHARACTER***	**total** ***INTEGER***
A	3
B	12
D	22

As with AGGREGATE when used outside of SUMMARIZE, an optional second argument can be passed to set an initial value. For example, the following…

```
SUMMARIZE myvar BY {c}: {
  total := AGGREGATE(x, 100); RETURN VALUE1 + VALUE2; END AGGREGATE
}
```

…returns:

c *CHARACTER*	total *INTEGER*
A	103
B	112
D	122

User-defined Aggregate Operators in *Rel*

Whilst AGGREGATE provides an effective way to create user-defined aggregations on a one-off basis, we'd ideally like to be able to define new general-purpose aggregate operators that can be reused.

In *Rel*, every aggregate operator <*op*> is defined as a corresponding operator with the generic signature:

AGGREGATE_<*op*>(ARRAY TUPLE {AGGREGAND <type>, AGGREGATION_SERIAL}) RETURNS <type2>

For example, the operator invoked by SUM(myvar, x) is defined with the specific signature[v]:

> *AGGREGATE_SUM(ARRAY TUPLE {AGGREGAND INT, AGGREGATION_SERIAL}) RETURNS INT*

Likewise, the operator invoked by AVG(myvar, x) is defined with the signature:

> *AGGREGATE_AVG(ARRAY TUPLE {AGGREGAND INT, AGGREGATION_SERIAL}) RETURNS RATIONAL*

Tutorial D defines ARRAYs of TUPLEs, but limits their use to a cursor-like facility for accessing a collection of tuples in a specified order. In *Rel*, ARRAYs of TUPLEs can be used as an operand type.

The *Rel* compiler converts an expression like SUM(myvar, x * 10) to an invocation of AGGREGATE_SUM(ARRAY TUPLE {AGGREGAND INT, AGGREGATION_SERIAL INT}) RETURNS INT by:

1. Extending myvar with two new attributes:
    a) A new attribute named AGGREGAND whose value is the result of evaluating x * 10;
    b) A new attribute named AGGREGATION_SERIAL that has a unique integer for each tuple. The presence of this attribute ensures that duplicate values of AGGREGAND are retained.
2. Projecting away all attributes except AGGREGAND and AGGREGATION_SERIAL before converting the result to an ARRAY; and
3. Invoking AGGREGATE_SUM with the result.

This means AGGREGATE_SUM can be a conventional operator, and new operators like it can be defined by the user. For example, we might want to define an aggregate operator so that an invocation like SUM(myvar, c) will work. To do so, we'll define a new aggregate operator:

```
OPERATOR AGGREGATE_SUM(r ARRAY TUPLE {AGGREGAND CHARACTER, AGGREGATION_SERIAL
INTEGER}) RETURNS CHARACTER;
        RETURN
                AGGREGATE (r ORDER(ASC AGGREGAND), AGGREGAND);
                        RETURN VALUE1 || VALUE2;
                END AGGREGATE;
END OPERATOR;
```

We've used AGGREGATE, described above, to implement this new aggregation operator.

Once implemented, we can use the new operator via expressions like the following…

```
        SUM(myvar, c || ' blah ')
```

…which evaluates to the string:

```
        A blah A blah B blah B blah B blah D blah D blah
```

Note the use of *ORDER* in the above. ORDER is a *Rel* extension that converts a given relation to an ARRAY with a specified TUPLE ordering. It is based on – and employs exactly the same syntax – as the ORDER clause of the LOAD … FROM construct for creating an ARRAY in **Tutorial D**. Here, it ensures that the result of invoking SUM on a character attribute expression produces a deterministic result, which would otherwise not be possible because RELATIONs specify no tuple ordering and || is not commutative.

User-defined Aggregate Operator Invocations

Imagine that we wanted to define an aggregate operator for calculating the population standard deviation of a set of integers. We might define the aggregate operator as:

```
OPERATOR AGGREGATE_STDEV(data ARRAY TUPLE {AGGREGAND INTEGER, AGGREGATION_SERIAL
INTEGER}) RETURNS RATIONAL;
        RETURN WITH (
                mean := AVG(data, AGGREGAND),
                squarediffs := EXTEND data UNORDER(): {
                        squaredifference :=
                                WITH (difference := CAST_AS_RATIONAL(AGGREGAND) - mean):
                                        difference * difference
                }
        ): SQRT(AVG(squarediffs, squaredifference));
END OPERATOR;
```

Note the use of UNORDER() – a Rel extension that is the counterpart of ORDER(…), which converts an ARRAY to a corresponding relation – so that we can use the relational operator EXTEND on the source data.

We can invoke this new aggregate operator directly with an appropriate operand type:

```
AGGREGATE_STDEV(
    REL {
        TUP {AGGREGAND 1, AGGREGATION_SERIAL 1},
        TUP {AGGREGAND 2, AGGREGATION_SERIAL 2},
        TUP {AGGREGAND 3, AGGREGATION_SERIAL 3},
        TUP {AGGREGAND 4, AGGREGATION_SERIAL 4}
    } ORDER()
)

1.118033988749895
```

However, that's not particularly ergonomic for general use. Ideally, we'd like to be able to invoke it as:

```
STDEV(myvar, x)
```

Unlike SUM, AVG, and the other predefined aggregate operators, STDEV is not part of the **Tutorial D** specification. Therefore, the compiler can't know – by default, at least – that STDEV (or any other user-defined aggregate operator) should necessarily be treated as an aggregate operator invocation. Whilst aggregate operator invocations within SUMMARIZE can unambiguously be recognised by the compiler's implementation of SUMMARIZE – and thus automatically translated into an appropriate invocation of AGGREGATE_STDEV – direct invocation of "pseudo" aggregate operators like STDEV requires additional compiler and/or language support.

There are various means by which this could be achieved. In *Rel*, this is accomplished by requiring the user to prefix user-defined aggregate operator invocations with the keyword AGGREGATE. So, whilst the following is not a valid expression for obtaining the standard deviation of myvar's *x* attribute:

```
STDEV(myvar, x)
```

The following is valid:

```
AGGREGATE STDEV(myvar, x)
```

This style of AGGREGATE invocation supports an optional third argument, as follows:

```
AGGREGATE STDEV(myvar, x, 1)
```

It requires a corresponding operator definition with the signature:

```
OPERATOR AGGREGATE_STDEV(r ARRAY TUPLE {AGGREGAND INTEGER, AGGREGATION_SERIAL
INTEGER}, i <type>) RETURNS RATIONAL;
```

Where <type> may be any type. This is intended to provide an initial or default value, but may be used for any purpose appropriate to the user-defined aggregate operator.

User-defined aggregate operators may be invoked by SUMMARIZE, so the following is valid:

```
SUMMARIZE myvar BY {c}: {stdev := STDEV(x)}
```

Note that the AGGREGATE prefix is not required in SUMMARIZE, as it is when invoking an aggregate operator outside of SUMMARIZE.

An optional initial or default (or other purpose) value may be passed, assuming an appropriate user-defined aggregate operator has been defined. For example:

```
SUMMARIZE myvar BY {c}: {stdev := STDEV(x, 1)}
```

Tutorial D defines some built-in aggregate operators has having two versions, e.g., SUM and SUMD, or AVG and AVGD. The operators ending with 'D' may only be invoked inside SUMMARIZE, and represent an invocation that eliminates duplicate tuples from the attribute expression. To express the equivalent when invoking a user-defined aggregate operator, the keyword DISTINCT must prefix the attribute expression. For example:

```
SUMMARIZE myvar BY {c}: {stdev := STDEV(DISTINCT x)}
```

This may also be used with an initial or default (or other purpose) value. For example:

```
SUMMARIZE myvar BY {c}: {stdev := STDEV(DISTINCT x, 1)}
```

To support this, the aggregate operator must be defined with a corresponding additional parameter. For example:

```
OPERATOR AGGREGATE_STDEV(data ARRAY TUPLE {AGGREGAND RATIONAL, AGGREGATION_SERIAL INTEGER},
initial RATIONAL) RETURNS RATIONAL;
        RETURN WITH (
                mean :=
                        CAST_AS_RATIONAL(AGGREGATE(data, AGGREGAND, initial);
                                RETURN VALUE1 + VALUE2; END AGGREGATE) /
                        CAST_AS_RATIONAL(AGGREGATE(data, 1, 0);
                                RETURN VALUE1 + VALUE2; END AGGREGATE),
                squarediffs := EXTEND data UNORDER(): {
                        squaredifference := WITH (difference := AGGREGAND - mean): difference * difference
                }
        ): SQRT(AVG(squarediffs, squaredifference));
END OPERATOR;
```

Why ARRAY TUPLE {AGGREGAND INTEGER, AGGREGATION_SERIAL INTEGER}?

Using only existing **Tutorial D** constructs, how do we construct a user-defined aggregation operator – say, MYAGG – so that it's a generic, reusable operator?

In **Tutorial D**, we can easily define new operators that have specific parameters and work with specific operands, but generic parameters and operands are a challenge. If we wish to define an aggregate operator that can accept *any* relation – and any attribute expression valid for that relation – what would be appropriate parameter types for its operator definition, assuming we can only use existing **Tutorial D** types and operators or **Tutorial D** user-defined types and operators (if appropriate), and we can only use existing **Tutorial D** constructs, but we *can* use the compiler to transform **Tutorial D** code to other **Tutorial D** code?

Ideally, we'd like to define an aggregate operator such that the first argument can be *any* relation and the second argument can be *any* valid attribute expression for that relation. Unfortunately, there are no **Tutorial D** constructs for specifying (at least) generic relations or evaluate-it-later attribute expressions. Since – per the above – we can only use existing **Tutorial D** constructs, we have to rely on the compiler to translate an invocation of MYAGG into some as-generic-as-possible equivalent.

An obvious solution is to have the compiler translate any aggregate operator invocation like MYAGG(R, X * 10, -1) by doing the following:

1. EXTEND R with the expression X * 10. Every new attribute in an EXTEND must be given a name, so we'll call the result of evaluating X * 10, AGGREGAND. E.g., *temp* := EXTEND R: {AGGREGAND := X * 10}. Now *temp* is a relation with all the attributes of R, along with a new attribute AGGREGAND of the same type as X * 10.
2. Project away all the attributes of *temp* except AGGREGAND and convert to an ARRAY, which means we can invoke any operator that accepts an operand of *temp's* type, i.e., ARRAY TUPLE {AGGREGAND SAME_TYPE_AS(X * 10)}. That's certainly more generic than R!

Unfortunately, projecting away the attributes of R could remove duplicate values of X * 10 that need to be preserved for correctly computing the aggregate. Therefore, we need to do this instead:

1. EXTEND R with the expression X * 10. E.g., *temp* := EXTEND R: {AGGREGAND := X * 10, AGGREGATION_SERIAL := <unique number>}. Now *temp* is a relation with all the attributes of R, along with new attributes AGGREGAND of the same type as X * 10 and AGGREGATION_SERIAL of type INT.
2. Project away all the attributes of *temp* except AGGREGAND and AGGREGATION_SERIAL and convert to an ARRAY, which means we can invoke any operator that accepts *temp's* type, i.e., ARRAY TUPLE {AGGREGAND SAME_TYPE_AS(X * 10), AGGREGATION_SERIAL INT}. That's certainly more generic than R *and* the AGGREGATION_SERIAL attribute ensures that duplicate values of X * 10 are preserved.

Assuming the type of X * 10 is INT, a "generic" MYAGG operator may thus be defined as:

```
OPERATOR MYAGG(r ARRAY TUPLE {AGGREGAND INT, AGGREGATION_SERIAL INT}, initial INT) RETURNS INT;
        RETURN AGGREGATE(r, AGGREGAND, initial); RETURN VALUE1 + VALUE2; END AGGREGATE / 100;
END OPERATOR;
```

Using the latter steps above, we can translate *any* desired aggregate expression involving MYAGG — with any relation (or ARRAY) argument and any attribute expression of type INT, whether invoked as AGGREGATE MYAGG or inside SUMMARIZE — into a specific invocation of the MYAGG[vi] operator. That's as close to being generic as we can get without introducing new **Tutorial D** constructs, at the expense of having to define the operator with an ARRAY operand having the attributes AGGREGAND <type> and AGGREGATION_SERIAL INT.

The use of ARRAYs instead of RELATIONs ensures that tuple orderings can be defined or preserved in aggregate operators where tuple ordering is critical. In future work, this – along with explicit use of the AGGREGATION_SERIAL operand – will become the basis for providing explicit facilities for implementing LEAD, LAG, iTH, HEAD, TAIL, etc. operations in the context of aggregation.

END

Endnotes

[i] See http://www.thethirdmanifesto.com
[ii] The facilities described in this paper are available in *Rel* version 3.000, which as of August 2016, is not yet released.
[iii] See http://reldb.org
[iv] See http://reldb.org/docs/AnonymousAndFirstClassOperatorsInTutorialD.pdf
[v] There's a little bit of fiction here. The operator *actually* has the name AGGREGATE_SUM_INTEGER, for reasons irrelevant to this discourse. For the sake of understanding this paper, it is sufficient to regard the operator as being named AGGREGATE_SUM.
[vi] In *Rel,* the aggregate operator definition must be named AGGREGATE_MYAGG in order to be invoked by AGGREGATE MYAGG or used as MYAGG within a SUMMARIZE operator invocation.