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

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

Version 1.00 August 2016

As defined by Date & Darwen in *The Third Manifesto*[[1]](#endnote-1) 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. It is perhaps better thought of as a “pseudo” operator invocation that the compiler translates into some conventional – but otherwise less egonomic – expression and/or operator invocation.

Like SUM, aggregate operators AVG, MIN, MAX, UNION, XUNION, COUNT – and others – are predefined.

## User-defined Aggregation

Ideally, we would like to be able to create user-defined aggregate operators.

As a first step, we propose a generic aggregate operator invocation that permits a user-defined body. For example:

AGGREGATE(myvar, x);

RETURNS VALUE1 + VALUE2;

END AGGREGATE

This example 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, the accumulator is set to the accumulator + *x.* When there are no more tuples, the accumulator is returned.

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

Similarly, the following…

AGGREGATE(myvar, x);

RETURNS VALUE1 + VALUE2;

END AGGREGATE

…specifies that an accumulator is set to the evaluation of *x* for the first tuple. For each subsequent tuple, 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 parameter names VALUE1 and VALUE2 would probably benefit from being changed to something more intuitive.)

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);

RETURNS VALUE1 + VALUE2;

END AGGREGATE

…will return 0 if myvar is empty. This…

AGGREGATE(myvar, x, 10);

RETURNS VALUE1 + VALUE2;

END AGGREGATE

...will return 10 if myvar is empty, and return 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*[[2]](#endnote-2), the implementation of **Tutorial D** used to prototype AGGREGATE for this paper[[3]](#endnote-3), every aggregate operator *<op>* is defined as a corresponding operator with the signature:

AGGREGATE\_*<op>*(RELATION {AGGREGAND <type>, AGGREGATION\_SERIAL}) RETURNS <type2>

For example, the operator invoked by SUM(myvar, x) is defined with the signature[[4]](#endnote-4):

*AGGREGATE\_SUM(RELATION {AGGREGAND <type>, AGGREGATION\_SERIAL}) RETURNS INT*

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

*AGGREGATE\_AVG(RELATION {AGGREGAND <type>, AGGREGATION\_SERIAL}) RETURNS RATIONAL*

The *Rel* compiler converts an expression like SUM(myvar, x) to an appropriate invocation of AGGREGATE\_SUM(RELATION {AGGREGAND <type>, AGGREGATION\_SERIAL INT}) RETURNS INT by extending myvar with two new attributes:

1. A new attribute named AGGREGAND whose value is the result of evaluating x; and
2. A new attribute named AGGREGATION\_SERIAL that has a unique number for each tuple. This attribute ensures that duplicate values of AGGREGAND are retained.

Then all other attributes are projected away and AGGREGATE\_SUM is invoked with the result as the sole operand.

Of course, AGGREGATE\_SUM is 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 RELATION {AGGREGAND CHARACTER, AGGREGATION\_SERIAL INTEGER}) RETURNS CHARACTER;

RETURN

AGGREGATE (r RENAME {AGGREGAND AS p, AGGREGATION\_SERIAL AS q}, p);

RETURN VALUE1 || VALUE2;

END AGGREGATE;

END OPERATOR;

We’ve used AGGREGATE, described above, to implement this new aggregation operator. Because AGGREGATE is defined to work the same way as any other aggregate operator – i.e., it expects a single operand consisting of a RELATION {AGGREGAND <type>, AGGREGATION\_SERIAL INT} – the RENAME is necessary to internally prevent AGGREGATE\_SUM’s operand’s AGGREGAND and AGGREGATION\_SERIAL attributes from colliding with AGGREGATE’s operand’s (otherwise invisible) AGGREGAND and AGGREGATION\_SERIAL attributes.[[5]](#endnote-5)

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

Noting again that an aggregate operator invocation like SUM(myvar, x) or AGGREGATE(myvar, x) can have an optional final operand for specifying an initial value, like SUM(myvar, x, 1) or AGGREGATE(myvar, x, 0), the initial value can be accepted as an additional parameter on the user-defined aggregate operator. For example, this is a definition for AGGREGATE\_SUM that accepts the initial value:

OPERATOR AGGREGATE\_SUM(r RELATION {AGGREGAND CHARACTER, AGGREGATION\_SERIAL INTEGER}, initial CHARACTER) RETURNS CHARACTER;

RETURN

AGGREGATE (r RENAME {AGGREGAND AS p, AGGREGATION\_SERIAL AS q}, p, initial);

RETURN VALUE1 || VALUE2;

END AGGREGATE;

END OPERATOR;

## 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(r RELATION {AGGREGAND INTEGER, AGGREGATION\_SERIAL INTEGER}) RETURNS RATIONAL;

RETURN WITH (

data := r RENAME {AGGREGAND AS X, AGGREGATION\_SERIAL AS serial},

mean := AVG(data, X),

squarediffs := EXTEND data: {

squaredifference := WITH (difference := CAST\_AS\_RATIONAL(X) - mean):

difference \* difference

}

): SQRT(AVG(squarediffs, squaredifference));

END OPERATOR;

We can invoke it 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}**

**}**

**)**

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)

However, 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)

1. See <http://www.thethirdmanifesto.com> [↑](#endnote-ref-1)
2. See <http://reldb.org> [↑](#endnote-ref-2)
3. The facilities described in this paper are available in *Rel* version 3.000, which as of August 2016, is not yet released. [↑](#endnote-ref-3)
4. 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. [↑](#endnote-ref-4)
5. This is quirky. A future *Rel* update will almost certainly remove the need for RENAME. [↑](#endnote-ref-5)