



Beyond Relational Operators: Programming with FlexStreams in the Aleri Streaming Platform

By: Jon Riecke, Lead Platform Architect

Almost 40 years ago, Codd proposed the relational model as a means of breaking free from low-level database programming. The idea is beautiful: instead of writing programs to query data stored in ISAM or VSAM structures, one writes queries against abstract tables. A whole industry of relational databases blossomed, with SQL as the core query language, because almost anyone can write a SQL query, and because, with indexing, the database can be as fast as any hand-coded database.

As inheritors of that tradition, many Complex Event Processing (CEP) systems, including the Aleri Streaming Platform, adopt the relational model at the core, where “streams” are analogous to relations (i.e. tables). As events arrive into a CEP system, they are passed to streams that compute new events from old based on familiar primitives from the relational model. For instance, joins and aggregations, as well as the simpler primitives of selection (filtering events by a boolean expression) and computations, can be used.

The relational model is beautiful for queries, but it's not always the best model of computation. In CEP systems, sticking to purely relational operations can lead to contorted code. Part of that is due to the relational model. It's stateless, meaning no memory is explicitly held, and has no control features, with looping done obliquely through joins.

Just a little bit of state and procedural programming can make CEP programs a lot simpler, and much more straightforward to read and maintain. It's for that reason that the Aleri Streaming Platform allows procedural programming with FlexStreams. Inside a FlexStream, events are processed by programs written in a language called SPLASH (Streaming Platform Language SHell, for those who like acronyms). You can think of it as an escape hatch: when the usual relational primitives don't quite match your intuition of how to compute new events from old, you can drop into a FlexStream.



Let's consider an example, one that's simple, slightly contrived, but indicative of a wide range of practical problems. Suppose there is a stream of Trades data, with fields Tradeld, Symbol (e.g., MSFT), and Price. Suppose we'd like to stamp each IBM trade—and only the IBM trades—with a sequence number. How might we do this? One obvious idea would be to use an aggregation to compute the sequence number, and a join to merge that sequence number back into the original table. In SQL, this might be expressed as

```
SequenceNumber =  
  select max(T.Tradeld) as Tradeld, count(T.Symbol) as SeqNo  
  from Trades T  
  where T.Symbol = 'IBM'  
  group by T.Symbol
```

```
TaggedIBMTrades =  
  select T.Tradeld as Tradeld, T.Symbol as Symbol, T.Price as Price,  
         (case when T.Symbol = 'IBM' then S.SeqNo else null end) as IBMNumber  
  from Trades T, SequenceNumber S  
  where T.Tradeld = S.Tradeld
```

Unfortunately, we need some guarantees from our CEP engine if these streams are calculated independently. More precisely, if a trade event arrives at the SequenceNumber stream and the TaggedIBMTrades stream simultaneously, the TaggedIBMTrades should wait for the updated count to pass through the SequenceNumber computation before joining with the Trades. Some CEP systems make that guarantee, but they pay a price in efficiency for doing so. But even if the CEP engine makes that guarantee, is it the best way to express the computation? The example cries out for a little bit of state. Wouldn't it be far simpler if we could remember, in the TaggedIBMTrades computation, the number of IBM trades we've seen thus far, and use it to generate the computed event?

This example thus shows one good use case for FlexStreams. Here's the bit of procedural code in SPLASH that we'd write to handle this example. First, we declare a variable for the FlexStream itself:

```
int32 seqNo := 0;
```



This variable holds information across events. Each event in a FlexStream is processed by a “method” that has access to these variables, with one method per input stream. We can make a FlexStream for the TaggedIBMTrades computation from one input stream, the Trades stream. Thus, the only method would have the code

```
if (Trades.Symbol = 'IBM') {
    output [Tradeld = Trades.Tradeld;
           Symbol = Trades.Symbol;
           Price = Trades.Price;
           IBMNumber = seqNo];
    seqNo := seqNo + 1;
} else {
    output [Tradeld = Trades.Tradeld;
           Symbol = Trades.Symbol;
           Price = Trades.Price;
           IBMNumber = null];
}
```

The syntax of SPLASH is quite close to C. It uses “:=” instead of “=” for assignments, and “=” instead of “==” for equality tests, to keep it in-line with the syntax of SQL. It also adds a few statements, like “output” for sending events to downstream streams and to subscribers, and square brackets for creating events (records with insert/update/delete operations). It's fairly easy to learn, though, for anyone with even the smallest bit of programming background.

[Note: the above example makes use of upcoming simplifications in Release 3.0 of the Aleri Streaming Platform, particularly in the freer syntax of records.]

The TaggedIBMTrades shows that variables can simplify programs. That's well-known in the programming language community. Languages like LISP/Scheme, ML/CAML, and F# have long included state, and even “pure” languages like Haskell have taken to adding it in a controlled manner. And languages made for dataflow computation—languages like Id and SISAL—have also incorporated state.

It's important that FlexStreams do not break the essential parallelism in the dataflow model. The state does not need to be synchronized across streams, with locks or other concurrency control mechanisms. It's local. That makes programs still easy to reason about, and not admit the bugs for which concurrent programs are famous.



FlexStreams, or rather SPLASH, has mechanisms for looping as well. Suppose, for instance, we'd like to match a trade whose price falls below 1 to see if the symbol is in a table of delisted stocks, and take some special action. We could do this with a join, but we could also do it with a loop in SPLASH:

```
if (Trades.Price < 1) {  
    for (rec in Delisted) {  
        if (rec.Symbol = Trades.symbol) ...  
    }  
}
```

There are times when the loop, rather than a join, is what the user wants, e.g., if the Delisted stream changes, we might not want to rematch the old trades against the changes. The “for” loops through all the records in the Trades input stream, calling each one “rec” in the body of the loop. There are also “while” statements in SPLASH, and “break” and “continue” as in C, and Release 3.0 will add a number of interesting data structures as well (dictionaries and vectors, for instance).

FlexStreams have been used at a number of installations of the Aleri Streaming Platform. Most streams in those production models aren't FlexStreams—and rightfully so, since the relational primitives do a decent job much of the time, are simpler to define, and have already been debugged and optimized. But FlexStreams are extremely helpful in certain situations (e.g., extended versions of the TaggedIBMTrades example), and allow programmers to represent computations in a natural and efficient manner. One might argue that it makes the system less “pure”, but I'd opt for pragmatism over purity any day. If one is interested in getting a job done without strange coding tricks, FlexStreams are a powerful feature.

UNITED STATES

Two Prudential Plaza, 41st Floor
Chicago, IL 60601
t +1 312 540 0100

430 Mountain Ave.
Murray Hill, NJ 07974
t +1 908 673 2400

UNITED KINGDOM

New Baltic House
65 Fenchurch Street
London EC3M 4BE
United Kingdom
t +44 (0) 20 7821 1110