Abstract—This paper describes the design and implementation of an SQL-like language for performing complex queries on event streams. The Event Stream Query Language (eSQL) aims at providing a simple, intuitive and fully non-procedural syntax, while still preserving backwards compatibility with traditional SQL. More importantly, eSQL’s core syntax is designed to be extended by user-defined grammatical constructs. These new constructs can form domain-specific sub-languages, with eSQL being used as the “glue” to form very expressive queries. These concepts have been implemented in BeepBeep 3, an open source event stream query engine.

II. Desiderata for a Stream Query Language

In the realm of relational databases, desirable properties of a potential query language have been collated into a document called the Third Manifesto (3M) [8]. Event stream query engines, for a large part, provide interpreters for languages that borrow keywords from one of the most popular of these relational languages, SQL. However, as we mentioned above, there exists almost as many stream query languages as there are stream query engines. Moreover, the design of these query languages seems in many cases almost like an afterthought; their syntax is often quirky, and their semantics complicated to understand.

In the following, we list a number of observations and choices that should be taken into account when designing a query language for ESP. These design choices will be reflected in the implementation of our event query engine, BeepBeep, and its companion query language, eSQL.

A. Events Should Be Anything

Many ESP tools, which have a strong background in databases, assume all events to be tuples; in relational databases, the 3M (prescriptions 6–10) also enforces this rigid data model. Hence every tuple of a trace must have the same fixed set of attributes, and events must differ only in the values they define for each attribute. Moreover, these values must be scalar. A few engines allow events to be user-defined objects, but these objects are accessed through methods that return scalar values, which is tantamount.

A truly generic event processing system should not presuppose that any single type of events is appropriate for all problems. Rather, each type of event should come with its own set of event manipulation functions (EMF) to extract data, manipulate and create new events of that type. These functions should be distinct from trace manipulation functions (TMF), which, in contrast, should make very limited assumptions on the traces they manipulate. This clear separation of EMF and TMF should make it possible to easily mix events of different types into queries. It should also help avoid the “square peg in a round hole” problem, where one must write an overly complicated expression simply to work around the limitations of the single available event type.
B. Composition-Centered Processing

The database foundations of ESP have led many solutions to compute everything through a tentacular SELECT statement, with optional constructs attempting to account for every possible use case. A modular event processing framework should alleviate this problem by proposing a set of basic processing units that can be freely composed. Therefore, rather than proposing a single, all-encompassing query language, it should accommodate multiple query languages, along with lightweight syntactical “glue” to allow for their composition. Hence every step of the computation to be expressed in the notation most appropriate for it.

C. Behave Like SQL When Possible

However, when the context allows an event trace to be interpreted as an ordered relation whose events are tuples, then the SQL query computing some result over that relation should be a valid event stream query as well. This means that standard relational tables should be able to be used as drop-in replacements for event traces anywhere in an expression where tuples are expected.

This statement, in line with 3M’s “Very strong suggestion” #9, is in itself a distinguishing point to virtually every other ESP system around, whose input languages sometimes reuse keywords from the SQL language, but with a syntax that is incompatible with SQL. For example, in the Cayuga language [6], selecting all events where attribute cnt is greater than 10 is written as:

```
SELECT * FROM FILTER {cnt > 10}(webfeeds)
```

while extracting the same data from a database would be written as the following SQL query:

```
SELECT * FROM webfeeds WHERE cnt > 10
```

D. No Procedural Escapes

All event processing should be made through the combination of the language constructs, without resorting to procedural statements. As a counter-example, take this TelegraphCQ expression:

```
Select AVG(closingPrice)
From ClosingStockPrices
Where stockSymbol = 'MSFT'
for (t = ST; t < ST+50, t+= 5) {
    Windows(ClosingStockPrices, t - 4, t);
}
```

One can see that part of its processing is done through the use of a C-style for loop. There are many reasons why such an approach is undesirable. Besides being inelegant, it pollutes the declarative style of SQL with procedural statements which arguably should not occur in a query language. This, in turn, makes the semantics of the language very hard to define, and the actual meaning of a query difficult to grasp. Finally, one may argue that this hardly counts as a query language, but rather as a proxy to a procedural language interpreter, as one can write a Select * query where all processing is transferred into a block of Turing-complete procedural code. A similar remark applies to many other query languages.

We expect an event stream query language to be fully declarative. Syntactically, this entails that no procedural constructs (if-then blocks, loops, variables) should be offered to the user. This point of view is actually stricter than SQL, as most SQL engines extend the language with such constructs. This also contradicts 3M’s prescription 5, (which requires the presence of if-then blocks.

This does not mean that the resulting system should not support user extensions. However, it should support them in a way that whatever procedural code that needs to be written can then be accessed through extensions to the query language’s syntax, thereby creating a Domain-Specific Language (DSL). While new processing units are made of (potentially Turing-complete) code, users should not have the possibility of writing procedural code inside their queries, thus preserving their declarative nature.

III. The BeepBeep Event Stream Query Engine

These observations motivated the design of BeepBeep 3, a new event stream processing engine. BeepBeep can be used either as a Java library embedded in another application’s source code, or as a stand-alone query interpreter running from the command-line. Releases of BeepBeep 3 are publicly available for download under an open source license. In this section, we briefly describe the basic principles underlying the architecture of BeepBeep. The reader is referred to a recent tutorial for more details [10].

A. Processors

Let \( T \) be an arbitrary set of elements. An event trace of type \( T \) is a sequence \( \tau = e_0 e_1 \ldots \) where \( e_i \in T \) for all \( i \). A function is an object that takes zero or more events as its input, and produces zero or more events as its output. In BeepBeep, functions are first-class objects; they all descend from an abstract ancestor named Function, which declares a method called evaluate() so that outputs can be produced for a given array of inputs. A processor is an object that takes zero or more event traces, and produces zero or more event traces as its output. While a function is stateless, and operates on individual events, a processor is a stateful device: for a given input, its output may depend on events received in the past. Processors in BeepBeep all descend from the abstract class Processor.

A processor produces its output in a streaming fashion: it does not wait to read its entire input trace before starting to produce output events. However, a processor can require more than one input event to create an output event, and hence may not always output something when given an input. Processors can then be composed (or “piped” together, by letting the output of one processor be the input of another. An important characteristic of BeepBeep is that this piping is possible as long as the type of the first processor’s output matches the second processor’s input type.

When a processor has an input arity of 2 or more, the processing of its input is done synchronously. This means that a computation step will be performed if and only if an event can be consumed from each input trace. Assuming synchronous

https://liflab.github.io/beepbeep-3
processing limits expressiveness, but greatly simplifies the
definition and implementation of processors in use cases where
it is appropriate. The output result is no longer sensitive to the
order in which events arrive at each input, or to the time it
takes for an upstream processor to compute an output.

This hypothesis entails that processors must implicitly
manage buffers to store input events until a result can be
computed. This buffering is implicit: it is absent from both the
formal definition of processors and any graphical representation
of their piping. Nevertheless, the concrete implementation of a
processor must take care of these buffers in order to produce
the correct output. In BeepBeep, this is done with the abstract
class SingleProcessor; descendents of this class simply need
to implement a method named compute(), which is called
only when an event is ready to be consumed at each input.

B. Built-In Processors

BeepBeep is organized along a modular architecture. The
main part of BeepBeep is called the engine, which provides
the basic classes for creating processors and functions, and contains
a handful of general-purpose processors for manipulating traces.
The rest of BeepBeep’s functionalities is dispersed across a
number of optional palettes, which will be discussed later.

A first way to create a processor is by lifting any $m : n$
function $f$ into a $m : n$ processor. This is done by applying $f$
successively to each input event, producing the output events.
A few processors can be used to alter the sequence of events
received. The CountDecimate processor returns every $n$-th
input event and discards the others. Another operation that can
be applied to a trace is trimming its output. Given a trace, the
Trim processor returns the trace starting at its $n$-th input event.

Events can also be discarded from a trace based on a
criterion. The Filter processor $f$ is a $n : n - 1$; the events are
let through on its $n - 1$ outputs, if the corresponding event of
input trace $n$ is $\top$; otherwise, no output is produced. This
filtering mechanism, although simple to define, turns out to
be very generic. The processor does not impose any particular
way to determine if the events should be kept or discarded, as
long as it is connected to a trace that produces Boolean values.
This should be again contrasted with ESP systems, that allow
filtering events only through the use of a WHERE clause inside
a SELECT statement, and whose syntax is limited to a few
simple functions.

Another important functionality of ESP is the application
of some computation over a window of events. If $\varphi$ is any 1:1
processor, the Window processor of $\varphi$ with width $n$ sends the first
$n$ events (i.e. events numbered 0 to $n - 1$) to an instance of $\varphi$,
which is then queried for its $n$-th output event. The processor
also sends events 1 to $n$ to a second instance of $\varphi$, which is then
also queried for its $n$-th output event, and so on. The resulting
trace is indeed the evaluation of $\varphi$ on a sliding window of $n$
successive events. In existing ESP engines, window processors
can be used in a restricted way, generally within a SELECT
statement, and only a few simple functions (such as sum or
average) can be applied to the window. In contrast, in BeepBeep,
any processor can be encased in a sliding window, provided it
outputs at least $n$ events when given $n$ inputs.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Definition} & \textbf{Example} \\
\hline
\texttt{combine} & \texttt{combine(\$P, \$Q)} \\
\hline
\texttt{decimate} & \texttt{decimate(\$P, \$Q)} \\
\hline
\texttt{freeze} & \texttt{freeze(\$P)} \\
\hline
\texttt{function-call} & \texttt{\$P(\$Q)} \\
\hline
\end{tabular}
\caption{Built-in processors}
\end{table}

\section{IV. eSQL: The Event Stream Query Language}

BeepBeep provides multiple ways to create processor pipes
and to fetch their results. The first way is programmatic,
using BeepBeep as a library and Java as the glue code for
creating the processors and connecting them. Another powerful
way of creating queries is by using BeepBeep’s query language,
the Event Stream Query Language (eSQL). The basic syntax of
eSQL is shown in Table I. It provides the syntactical definitions
for instantiating and composing the built-in TMF processors
described in the previous section.

A. Creating Definitions

One notable feature of eSQL is the capability for a user to
extend the grammar dynamically through expressions, using the
WHEN keyword. The corresponding syntactical rules are
described in Table II. For example, the following code snippet shows how a new processor counting the elements of a trace
can be defined by the user:

\begin{verbatim}
WHEN \$P IS A PROCESSOR:
THE COUNT OF \$P IS THE PROCESSOR
COMBINE
SELECT 1 FROM \$P
WITH SUM.
\end{verbatim}

The second line of the expression declares a new rule
for the non-terminal symbol (processor) in the grammar of
Table I. It gives the syntax for that new rule; in that case, it
is the expression THE COUNT OF, followed by the symbol
"\$P". The first line declares that "\$P" must be a grammatical
construct matching the non-terminal (processor). Finally, the
remainder of the expression describes what THE COUNT OF \$P
should be replaced with when evaluating an expression; in this case, it is a SELECT statement taken from one of BeepBeep’s extensions for manipulating tuples. From that point on, THE COUNT OF @P can be used anywhere in an expression where a grammatical construct of type (processor) is required, and this expression itself can accept for @P any processor expression.

This mechanism proves to be much more flexible than user-defined functions provided by other languages, as any element of the original grammar can be extended with new definitions. For example, one can easily define a numerical constant with an expression like PI IS THE NUMBER 3.1416.

B. Creating Custom Processors

Sometimes, creating a new processor cannot easily be done by combining existing ones using the WHEN construct. BeepBeep also allows users to define their own processors directly as Java objects, using no more than a few lines of boilerplate code. The simplest way to do so is to extend the SingleProcessor class, which takes care of most of the “plumbing” related to event management: connecting inputs and outputs, looking after event queues, etc. We illustrate this process on a small example.

Consider a processor that takes as input two traces. The events of each trace are instances of a user-defined class Point, which contains member fields x and y. We will write a processor that takes one event (i.e. one Point) from each input trace, and return the Euclidean distance between these two points. The input arity of this processor is therefore 2 (it receives two points at a time), and its output arity is 1 (it outputs a number). Specifying the input and output arity is done through the call to super() in the processor’s constructor: the first argument is the input arity, and the second argument is the output arity.

The actual functionality of our processor will be written in the body of method compute(). This method is called whenever an input event is available, and a new output event is required. Its argument is an array of Java Objects; the size of that array is that of the input arity that was declared for this processor (in our case: 2).

C. Grammar Extensions

By creating a custom processor, it is possible to pipe it to any other existing processor, provided that its input and output events are of compatible types. We have seen in Section ?? how a combination of existing processors can be defined directly within eSQL; it is also possible to extend the grammar of the eSQL language for a custom Processor object, so that it can be used directly in eSQL queries.

As an example, let us consider the following processor, which repeats every input event n times, where n is a parameter decided when the processor is instantiated. Its implementation is as follows:

```java
public class Repeater extends SingleProcessor {
    private final int numReps;

    public Repeater(int n) {
        super(1, 1);
        this.numReps = n;
    }

    public Queue<Object[]> compute(Object[] inputs) {
        Queue<Object[]> queue = new LinkedList<Object[]>();
        for (int i = 0; i < this.numReps; i++) {
            queue.add(inputs);
        }
        return queue;
    }
}
```

The first step is to decide what syntax one shall use to invoke the processor. A possibility could be: "REPEAT (p) n TIMES". In this syntax, p refers to any other eSQL expression that builds a processor, and n is a number. The result of this expression is itself another object of type Processor.

The second step is to tell the BeepBeep interpreter to add to its grammar a new case for the parsing of the existing (processor) rule. This rule should correspond to the parsing of the newly-defined Repeater processor. This is done as follows:

```java
Interpreter my_int = new Interpreter();
my_int.addCaseToRule("<processor>", "<repeater>");
my_int.addRule("<repeater>", "REPEAT ( <processor> ) <number> TIMES");
my_int.addAssociation("<repeater>", "my.package.Repeater");
```

The second instruction tells the interpreter that (processor) can be parsed as a (repeater). The parsing pattern for this non-terminal is then added with the call to addRule(). This allows the interpreter to know that REPEAT xxx n TIMES corresponds to a processor. The last call tells the interpreter that encountering the <repeater> rule will result in the instantiation of a Java object of the class Repeater.

Upon parsing the <repeater> rule, the interpreter will look for a method called build(Stack<Object>) in the corresponding class. The task of the build() method is to consume elements of the parse stack to build a new instance of the object to create, and to put that new object back on the stack so that other objects can consume it during their own construction. Creating a new instance of Repeater is therefore straightforward. One simply has to pop() the stack to fetch the value of n and the Processor object to use as input, and discard all “useless” keywords. One can then instantiate a new Repeater, pipe the input into it (using

```java
public class EuclideanDistance extends SingleProcessor {
    public EuclideanDistance() { super(2, 1); }

    public float compute(Object[] inputs) {
        Point p1 = (Point) inputs[0];
        Point p2 = (Point) inputs[1];
        float distance = Math.sqrt(Math.pow(p2.x - p1.x, 2)
            + Math.pow(p2.y - p1.y, 2));
        return Processor.wrapObject(distance);
    }
}
```
with BeepBeep's main program to extend its functionalities in a modular organization that allows an output tuple to be created by picking and combining attributes of one or more input tuples. The grammar extension for the SELECT statement is given in Table III. For the sake of simplicity, we only show a few arithmetical functions that manipulate numerical values; the actual syntax of SELECT can easily be made to accommodate functions manipulating other types of scalar values.

One can see how this syntax precisely mirrors the basic form of SQL's command of the same name. In contrast to the SELECT statement found in other ESP tools, eSQL's only manipulates tuples, and not traces. Operations such as filtering or windowing are obtained by composing this statement with other constructs from BeepBeep's grammar. For example, selecting tuples that match some condition is done by piping the output of SELECT into BeepBeep's Filter processor, which is invoked syntactically through the WHERE keyword, as the grammar of Table II has already shown. This, as it turns out, results in an expression that reads exactly like SQL's SELECT …WHERE, ensuring the backward compatibility that was one of the design goals stated in Section II.

### E. Some Examples

We shall now illustrate by means of a few examples the extensible syntax of eSQL by comparing it with other ESP query languages. As a first example, we compare BeepBeep with Cayuga, using examples taken from its original demo paper [6]. The paper describes a scenario where events are web feeds containing various attributes, such as a URL, a summary, etc.

Using the tuple palette, the eSQL query producing the same output is written as follows:

\[
\begin{align*}
\text{(select) := } & \text{SELECT } (\text{tuple}) \text{ FROM } (\text{processor-list}) \\
\text{(tuple) := } & * (\text{tuple-element}) | (\text{tuple-element}) . (\text{tuple}) \\
\text{(tuple-element) := } & (\text{tuple-name}) (\text{rename}) | (\text{function-call}) (\text{rename}) \\
\text{(tuple-name) := } & (\text{word}) . (\text{word}) | (\text{word}) \\
\text{(processor-list) := } & (\text{processor}) (\text{rename}) | (\text{processor}) (\text{rename}) . (\text{processor-list}) \\
\text{(rename) := } & e | \text{AS} (\text{eml-attribute}) \\
\text{(function-call) := } & (\text{constant}) | (\text{tuple-name}) | (\text{add}) | (\text{sub}) | (\text{mul}) | (\text{div}) \\
\text{(constant) := } & (\text{string}) | (\text{number}) \\
\text{(add) := } & (\text{function-call}) + (\text{function-call}) \\
\text{(sub) := } & (\text{function-call}) - (\text{function-call}) \\
\text{(mul) := } & (\text{function-call}) \times (\text{function-call}) \\
\text{(div) := } & (\text{function-call}) \div (\text{function-call}) \\
\end{align*}
\]

Table III: Grammar for an event manipulation function (EMF), when events are associative maps of scalar values (i.e. tuples)

### D. Palettes

BeepBeep was designed from the start to be easily extensible. Any functionality beyond the few built-in processors presented in Section II is implemented through custom processors and grammar extensions, grouped in packages called palettes. Concretely, a palette is implemented as a JAR file that is loaded with BeepBeep's main program to extend its functionalities in a particular way, through the mechanisms described above. This modular organization is a flexible and generic means to extend the engine to various application domains, in ways unforeseen by its original designers.

BeepBeep ships with a number of pre-defined palettes for various use cases. For example, one palette provides functions and processors to manipulate XML events, while another contains processors to create finite-state machines and temporal logic expressions. Of particular interest to this paper is the palette manipulating events that are associative maps of scalar values—in other words, tuples in the relational sense of the term. This palette defines a new grammatical construct, called SELECT, that allows an output tuple to be created by picking and combining attributes of one or more input tuples. The grammar for the SELECT statement is given in Table III. For the sake of simplicity, we only show a few arithmetical functions that manipulate numerical values; the actual syntax of SELECT can easily be made to accommodate functions manipulating other types of scalar values.

Apart from subjective readability, other elements vouch in favour of the eSQL expression; for example, in Cayuga, the FILTER sub-expression that selects the iPod-related events needs to be written twice. Similarly the threshold of 10 posts per day appears twice: in the global FILTER expression, and as a stopping condition inside FOLD. It also needs an auxiliary variable cnt whose incrementing is explicitly taken care of in a loop-like statement inside FOLD.

We now consider the following query, taken from the documentation of TelegraphCQ: on every fifth trading day starting today, calculate the average closing price of MSFT for the five most recent trading days, and keep the query standing for fifty trading days. The corresponding TelegraphCQ expression was shown in Section [II]. This query presents several issues with respect to language design. In addition to the for-loop construct, it also requires the special global variable

```java
public static void build(Stack<Object> stack) {
    stack.pop(); // TIMES
    Number n = (Number) stack.pop();
    stack.pop(); // )
    Processor p = (Processor) stack.pop();
    stack.pop(); // ) REPEAT
    Repeater r = new Repeater(n.intValue());
    Connector.connect(p, r);
    stack.push(r);
}
```
ST, designating the start time of the query. There is also a coupling between the value 5 in “t += 5” and the value 4 in “t – 4” in the following line. Finally, AVG must be a language primitive; it cannot be defined using other constructs of the language.

This is one opportunity where, in eSQL, the query can be made more readable by extracting the computation of the average into its own processor, and extending the grammar with a new parsing rule.

WHEN @T IS A PROCESSOR:
THE AVERAGE OF @T IS THE PROCESSOR
SELECT X ÷ Y FROM
COMBINE @T WITH SUM AS X,
THE COUNT OF @T AS Y.

The previous query then becomes:

SELECT EVERY 5TH OF
THE AVERAGE OF
SELECT closingPrice FROM ClosingStockPrices
WHERE stockSymbol = "MSFT"
ON A WINDOW OF 5

V. Conclusion

This paper presented BeepBeep, an event stream processing engine based on the concept of basic units of computation called processors, which can be freely composed to evaluate a wide range of expressions. Given an appropriate toolbox of processors, properties involving extended finite-state machines, temporal logic, aggregation and various other concepts can be evaluated. Moreover, through the modular mechanism of palettes, end users can easily create their own processors, thereby extending the expressiveness of the tool.

BeepBeep proposes its own declarative input language, eSQL, which provides an alternative to creating processor chains through “glue” code. The language eSQL was designed with a focus on modularity and extensibility: simple syntactical constructs can be composed to form complex expressions, and user-defined palettes can dynamically add new rules to the grammar. As one particular example of this mechanism, we have seen how a palette for manipulating tuples allows eSQL queries to be written in a syntax compatible with SQL—a feature notably absent from most existing ESP software.

BeepBeep has already been used in concrete use cases, which we can only mention in passing: these include the evaluation of stateful properties in web service interactions [13], the recognition of home appliances based on the analysis of their electrical signal [11] and the detection of bugs in video games [18]. In time, it is hoped that BeepBeep will be adopted as a modular framework under which multiple event processing techniques can be developed and coexist.

References
