NESTREAM: Querying Nested Streams

Damianos Chatziantoniou
damianos@dmst.aueb.gr

Achilleas Anagnostopoulos
archie@istlab.dmst.aueb.gr

Department of Management Science and Technology
Athens University of Economics and Business

Abstract

This article identifies an interesting class of applications where stream sessions may be organized in a hierarchical fashion - i.e. sessions may contain sub-sessions. For example, log streams from call centers belong to different call sessions and call sessions consist of services' sub-sessions. We may want to monitor statistics and perform accounting at any level on this hierarchy, relative to any other higher level (e.g. monitoring the average service session per call vs. the average service session for the entire system.) We argue that data streams of this kind have rich procedural semantics - i.e. behavior - and therefore a semantically rich model should be used: a session may be defined by opening and closing conditions, may have data and methods and may consist of sub-sessions. We propose a simple conceptual model based on the notion of "session" - similar to a class in an object-oriented environment – having lifetime semantics. Queries on top of this schema can be formulated via HSA (hierarchical stream aggregate) expressions. We describe an algorithm dictating how stream data *flow down* session hierarchies and discuss potential evaluation and optimization techniques for HSAs. Finally we introduce NESTREAM, a prototype implementation for these ideas and give some preliminary experimental results.

1 Introduction

The technological explosion in the web, mobile communications, sensor/wireless technology, as well as the need for security, personalization, fraud detection, real-time billing, dynamic pricing, and others emphasize the necessity of real-time analysis and stream systems. The database research community has responded with an abundance of ideas, prototypes and architectures to address the new issues involved in data stream management systems ([1, 4, 5, 7]).

Although these systems go into detail on architectural and optimization issues, they do not address fully the topic of complex stream semantics. In this article we identify a class of common applications that require complex modeling of data streams. For example, stream sessions may have complex defining conditions (i.e. opening and closing conditions), depending not only on the incoming stream data, but also on running aggregates, state variables and external function applications (e.g. network flows, WAP sessions, stock exhange "bursts" etc.) Furthermore, sessions may *exist* only within other sessions creating thus a hierarchy (e.g. individual services within WAP sessions or stock performance during bursts.) Finally, within each session, "peculiar" (non-standard, that is) computations may take place (e.g. apply a correlation function against the category of a stock within the stock session during a burst.)

We propose a session-oriented framework (section 2) to model these applications. We also define the concept of hierarchical stream aggregate (HSA) - aggregates of session values at different levels, not necessarily successive – to express *queries* on top of session-oriented schemata. In section 3 we propose two algorithms to flow down stream tuples in session hierarchies and evaluate HSAs. NESTREAM's implementation is discussed in section 4.

1.1 Motivation

Assume a customer-related portal that provides useful information for one or more thematic areas. The user calls a specific number and chooses the service that she wants to access. For example, she may want to get information for AT&T. Within AT&T she can choose to listen to FAQ, find the nearest store, or find out details for her account. A portion of the services tree is shown in Figure 1.

The activity of all calls is typically recorded to a log that contains the following information: the channel number (the channel that handles the call, similar to IP in networks) and the type, date and time of the event. An event is something recorded by the system, for example the start or end of a call, the "category change" occasion, the caller's ID, etc. A typical log has the following schema (taken from a real computer-

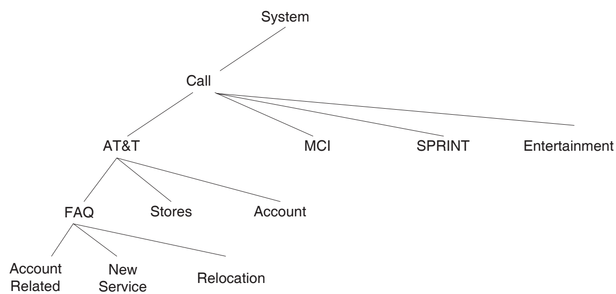

Figure 1: Part of a call center's services tree

Ch	Date	Time	Step	Event	Expression	Description
21	2001-12-06	12:25:47	0	107	2126199903	CID Received
21	2001-12-06	12:25:47	0	102	1075	3 %
21	2001-12-06	12:25:47	5	500	2126199903	Caller_ID
21	2001-12-06	12:25:55	7	501	QBRICK	ServerName
21	2001-12-06	12:26:03	41	500	ATT	RecognizedCateg
21	2001-12-06	12:26:03	378	500	Starting	ATTCateg
21	2001-12-06	12:26:11	342	501	FAQ	ATTCateg
21	2001-12-06	12:26:15	380	501	FAQ_Categ = 1	ATTCateg
21	2001-12-06	12:26:28	352	501	FAQ_Question = 2	ATTCateg
21	2001-12-06	12:27:07	352	501	FAQ_Question = #	ATTCateg
21	2001-12-06	12:27:12	380	501	FAQ_Categ = 2	ATTCateg
21	2001-12-06	12:27:16	354	501	FAQ_Question = 3	ATTCateg
21	2001-12-06	12:27:41	342	501	FAQ	ATTCateg
21	2001-12-06	12:27:43	380	501	FAQ_Categ = 3	ATTCateg
21	2001-12-06	12:27:46	356	501	FAQ_Question = 2	ATTCateg
21	2001-12-06	12:28:04	342	501	ACCOUNT	ATTCateg
21	2001-12-06	12:28:36	342	501	Location	ATTCateg
21	2001-12-06	12:28:46	348	501	New York	RecognisedATTState
21	2001-12-06	12:28:53	381	500	ATTCenter0076.wav	CUSTOMER CENTER IS
21	2001-12-06	12:29:14	348	501	New Jersey	RecognisedATTState
21	2001-12-06	12:29:38	348	501	#	RecognisedATTState
21	2001-12-06	12:29:42	342	501	#	ATTCateg
21	2001-12-06	12:29:42	0	104	2	Disconnect break

Figure 2: Part of a call center's log file

telephony platform [8]):

`Log(channel, date, time, step_num, event_type, expression, descr)`

where `channel` denotes the channel number of the system handling the call, `date` and `time` contains the date and time of the specific event, `event_type` is a number describing the type of the event, `expression` and `descr` contain values that either make sense for system events (e.g. caller's id) or values that we choose to print out during user-defined events. A portion of the log during a typical activity is shown in Figure 2 (only for a channel, in a real log records from multiple channels will be multiplexed.) Apparently such a log constitutes a data stream: data arrives in continuous, rapid, multiple, time-varying and unbounded streams.

One can ask several interesting queries for monitoring, billing or statistical purposes:

Q1. What is the running average call length? (measured from a specific starting point.) This could be used to monitor fluctuations on call lengths.

Q2. What is the running average of the calls' totals of AT&T sessions? (measured from a specific starting point.) This could be used to dispatch resources to different services (AT&T in this case) appropriately.

Q3. If a caller accesses more than four times the AT&T service, disconnect the call.

Q4. What is the running average of the total AT&T sessions within each call, for the last 1000 calls? AT&T pays for these sessions, so there must be a way to monitor usage.

Q5. What is the running average of FAQ sessions (for the entire system)?

All of the above-mentioned queries are examples of continuous queries. Query Q1 should compute the average call length. To do so, the length of each call has to be computed *when the call terminates* and averaged in at the system level. Query Q2 is similar to Q1 but requires some kind of nesting: sum all AT&T sessions for a call and report back this total to the system level to compute the average.

In principal, the processing of all these queries is more or less the same. Given a tree T representing nested session nodes as in figure 1, we "flow" down stream data s from the root to some leaf in a specific fashion: at each node we check whether a specific session instance is already "open" and either instantiate, update or destroy it. We then move recursively one level down. When a session instance completes, aggregate computations should propagate up.

One can find similar applications in finance, real-time billing, dynamic pricing, the web, and elsewhere. The (interesting) question is whether certain *common* tasks of these applications can be factored out so we can have a query language and an optimization framework.

1.2 Challenges

In a traditional programming environment, one would have to write individual programs for each one of the queries. For example, for query Q1, a program would utilize an array of timestamps, indexed by the channel number. On each log event, the program would check whether to "initialize a call" or "disconnect a call". In case of initialization, the timestamp has to be kept in the array, at `channel number` position. In case of disconnect, the duration of the call on this channel number is computed and used to update the average length call. This simple example shows some of the basic challenges in a stream environment:

Rich procedural semantics. Data streams can be seen not only as data elements but also as a sequence of operations. Our experience is that various stream applications require a wide range of ad hoc preprocessing of the streamed data, from simple string and date manipulation to complex function application. For

example, financial applications may apply some peculiar (and sometimes proprietary) mathematical function on a set of records to determine the correlation of stocks. We believe that data streams are about state *and* behavior and therefore richer semantics is required.

Complex definitions of sessions and sub-sessions. All queries Q1 to Q5 have a common characteristic: they require a particular kind of grouping and subgrouping. Each session is not based entirely on one or more attributes (the channel number), but also on certain conditions. Indeed, a call depends on channel number, but one also has to check for the opening event of the call (`event_type==102`.) Within a call, you may have subsessions that are similarly defined with opening and closing conditions and/or additional attributes. Extending group definitions with such conditions is useful in an optimization framework. We use the term *session* and not group, since we consider it more suitable in the stream world. One could argue that session – sub-session relationship is not necessary, since one could have different processing nodes for each (e.g. Aurora, [4]). However, by making the structure between session nodes explicit, conceptual modeling becomes easier, optimization possible, and hierarchical stream aggregates processing transparent.

Hierarchical stream aggregates (HSAs). One can compute aggregated values for a particular session (e.g. the duration of a specific call or the total length of a net flow.) Many applications however, require to "roll-up" these aggregated values to sessions at higher levels such as in queries Q1 to Q5. In other words, whenever a specific session closes out, it reports some of its values to an aggregate existing at an "ancestor" session not necessarily the "parent" session (e.g. query Q5.) We should have an intuitive syntax to formulate these nested aggregate expressions and a simple and optimizable algorithm to compute them.

2 Conceptual Framework

A session is a conceptual entity having state (data members), behavior (methods) and *lifetime* (i.e. a starting and an ending point). During its lifetime, its state may change. We assume that we can check the state of a session at any moment t. In that sense, a session is more similar to a function invocation within a program, rather than an object in an OO framework. Furthermore, some sessions cannot exist independently of other sessions.

Definition 2.1: (Events, Sources) An event e is an ordered list of n values $(v_1, v_2, \dots, v_n)$ (some n-tuple), where each value is either an element of a domain A_i or a special NULL value. A source of events S is any medium, able to generate a sequence of events $e_l, e_{l+1}, \dots$ where $l \geq 1$. □

In a stream application environment we assume that we have a set of sources $S_1, S_2, \dots, S_k$, each one generating a sequence of discrete, distinguishable events. We denote the source of event e, as $Source(e)$. We also assume that there is a total ordering relationship t between events, i.e. for all events e_i, e_j in the system, we have either $(e_i, e_j) \in t$ or $(e_j, e_i) \in t$ (we write $t(e_i) \leq t(e_j)$ or $t(e_j) \leq t(e_i)$ respectively.) There is a special system event denoted as e_∞, for which for all events e, $t(e) < t(e_\infty)$.

Definition 2.2: (Session object, instance) A *session* s (or session instance) is a conceptual entity described by a set of attributes (the *state* of the session), a set of methods (the *behavior* of the session) and a sequence of events $(e_1, e_2, \dots, e_k)$, where $e_1 \neq e_\infty$ and $t(e_1) \leq t(e_2) \leq \dots \leq t(e_k)$ (the *lifetime* of the session.) A *session object* (or session class) O is a set of sessions of the same type. □

We denote the first and last events of the lifetime of a session s as e_{init} and e_{term} respectively. We denote the lifetime of a session s as $lifetime(s)$.

Example 2.1: A `Call` session in figure 2 corresponds to the activity of channel 21 (`ch = 21`). Its lifetime consists of all rows (events), except the first one (the opening condition is `event_type==102`). The e_{init} is the second row and the e_{term} the last row of figure 2. The set of all `Call` sessions comprises the `Call` class. □

Definition 2.3: (Sub-session property, Hierarchies) Assume two session classes O_1 and O_2. We say that O_2 is a sub-session class of O_1 (denoted as $O_2 \sqsubset O_1$), iff, *for all* possible event sequences and $\forall s \in O_2, \exists s' \in O_1$ such that $lifetime(s) \subseteq lifetime(s')$. We call s' the parent session of s, denoted as $parent(s)$. □

Example 2.2: In our motivating example, a `Call` session can only exist within a `System` session. We can enforce this by passing to a `Call` session events that have already been processed only by the `System` session. Similarly, `AT&T` sessions can only exist within `Call` sessions. We can enforce this by passing to an `AT&T` session events that have already been processed by a `Call` session. □

Observation 2.1: Definition 2.3 requires that a subsession instance should receive all of its input events from the *same* parent session. This guarantees for example that an `AT&T` session will only "inherit" events from the same `Call` session. It does not say anything however, on how the parent session "flows down" events (to one or multiple children sessions, in what way, etc.) In the context of NESTREAM we introduce a primary key constraint to control the "instantiation" and "flowing" of events. Note however that flowing of events can be controlled by more generic conditions. For example,

one can think of a schema where a `System` session passes events to multiple overlapping `Time-period` subsessions. □

Definition 2.4: (Layered Stream Schema - LSS) We create a graph in the following way. Each session object is a node o in this graph. If session object o_j is subsession of another session object o_i, then we add an edge $< o_i, o_j >$ to this graph. The resulting directed graph (a tree or a forest) is called the *layered stream schema* (LSS) of the application. By doing a topological sort on this graph, we can assign to each node a layer level, denoted as $layer(o)$. □

Definition 2.5: (Hierarchical Stream Aggregates) Assume a set of aggregate functions $\{f_1, f_2, ..., f_n\}$ and a specific tree T of a LSS. Also assume a non-empty sub-path $< o_{i_1}, o_{i_2}, ..., o_{i_k} >$ of some path from the root to a leaf of T. Any expression of the form $o_{i_1}.f_{j_1}(o_{i_2}.f_{j_2}(...(o_{i_k}.f_{j_k}(value))...))$, where *value* is either an attribute of the stream or an attribute or a method of the session object o_{i_k} and $f_{j_l} \in \{f_1, f_2, ..., f_n\}$, for each l=1,2,...,k, is called a *hierarchical stream aggregate.* □

There is a special aggregate function, called `any` that simply returns the value of its argument.

Example 2.3: We could define an HSA for query Q1 as: $System.average(Call.duration(time))$, where duration is a special built-in method of Call object.
We could define an HSA for query Q2 as: Call.average(ATT.duration(time)).□

Note that HSAs do not need to be defined at successive layers of session objects. For example, System.average(ATT.duration(time)) computes the average AT&T sessions for the System session without going through Calls sessions. The result is different than System.average(Calls.average(ATT.duration(time)).

We propose below a list of properties for a session class definition. We are currently implementing session class definitions in a system prototype called NESTREAM.

Definition 2.6: (Session class description) A session class O is described by the following characteristics:
Name, denoted as N - A unique name to distinguish it from other session classes.
Source of Events, denoted as S - A primary source of events or the name of another session class from which it receives an event.
Attributes and Methods, denoted as two sets, A and M - Data and function members, similar to an object-oriented framework. These can be used for ad hoc computations, to keep temporary values, etc.
Primary Key, denoted as PK - A subset of attributes that can be used as primary key for the session class, to uniquely identify session instances. There are cases where there is no primary key, i.e. the class has only one session instance (e.g. `System` and `AT&T` sessions in our example). In these cases, the system adds a unique attribute with a single value.
Flowing Constraint, denoted as Θ_{FC} - There must be a way to "control" the flowing of stream data from parent to children sessions (i.e. to which children session instance(s) the parent should flow the event.) In its simplest form (and currently in NESTREAM system), this is a foreign key constraint correlating event attribute(s) and the primary key of the session class. This means that a session may flow the event to only one child session. However, there are applications that stream data can flow to multiple children.
Opening Condition and Constructor, denoted as Θ_o and f_o - Θ_o is a condition that has to be met in order for a session to be instantiated (in addition to primary key constraint checking). A constructor is a special method, executed when a session instance is created.
Closing Condition and Destructor, denoted as Θ_c and f_c - A condition that has to be met in order for a session to be destroyed. A destructor is a special method executed when a session instance is destroyed.

We denote this session object as $O = \{N, S, A, M, PK, \Theta_{FC}, E, \Theta_o, f_o, \Theta_c, f_c\}$. □

3 Evaluation Algorithms

In this section we describe the algorithms to flow down stream data in a layered stream schema and evaluate HSAs. In NESTREAM implementation we designed a top level object, the Data Manager Interface (called DMI), which acts as an interface between the data stream sources (database tables, flat files, network sockets, etc.) and the session objects of the LSS. It may handle several data sources. There is one such session instance for each tree in the LSS, which is always open. We use this instance as the starting point for a recursive algorithm (called the Flow-Downward algorithm) to flow down a stream tuple. The driver routine for a specific tree T of an LSS is given below. The specific DMI instance is denoted as DMI_si.

```
DMI.InitDataSources(T);
DMI_si = CreateDMIInstance(T);
while (! DMI.EndOfData(T)) {
      e = DMI.ReadNext(T);
      FDA(DMI_si, e);
}
DMI.CleanUp(T);
```

3.1 The Flow-Downward Algorithm

This algorithm "flows down" an event (i.e. a stream tuple) in a specific LSS and updates appropriately the session instances. It starts at a session instance s with an input event e. The function `ObjectOf(s)` returns the session class of session s.

Algorithm 3.1: The Flow-Downward Algorithm:

```
FDA (session s, event e) {
   Update(s,e);
   foreach subsession obj O of ObjectOf(s) {
      s′ = FindMatchingInstance(s,O.Θ_FC,e);
      if  ((s′==NULL) and (s′.Θ_o(e)) {
          s′.constructor();
          make s′ child of s;
      }
      if (s′!=NULL) FDA(s′,e);
   }
   if (s.Θ_c(e)) Close(s);
}
Close (session s) {
     foreach subsession obj O in ObjectOf(s)
        for  each child s′ of s in O
             Close(s′);
     s.destructor();
}□
```

The algorithm proceeds as follows: Given a session instance s and a read event e, it first updates s's attributes by executing its methods and updating its attributes. Then, for each subsession class of s's class, it finds (if exists) the child session instance s' of s that matches the event e by invoking the `FindMatchingInstance()` system function.[1] If there is no matching child session s' of s (`FindMatchingInstance()` returns NULL), it creates one, if the opening condition for s' with respect to e is satisfied. It then calls recursively itself on session s' with event e. Once all the recursive calls to s's children have been completed, it checks whether s should close out. If yes, it calls the system function `Close()`, which closes recursively all descendents of s in bottom-up fashion.

3.2 Evaluating Hierarchical Stream Aggregates

Once we have a collection $H = \{h_1, h_2, \ldots, h_m\}$ of HSAs declared in our system, we must have a consistent and efficient way to evaluate them. We propose a basic algorithm that propagates changes up to the root of a LSS. There is a number of fundamental questions that greatly affects performance, such as how often we propagate changes up, up to what level, etc. These questions are closely related to optimization, approximation, data shedding and QOS issues in data stream management systems [2].

[1] `FindMatchingInstance()` in current NESTREAM implementation returns one or none matched instances of each subsession class because Θ_{FC} searches for a match between the primary key of the subclass and the corresponding event attribute(s). In future versions however, when Θ_{FC} will allow more general expressions, it will return a *list* of matched instances of the subsession object.

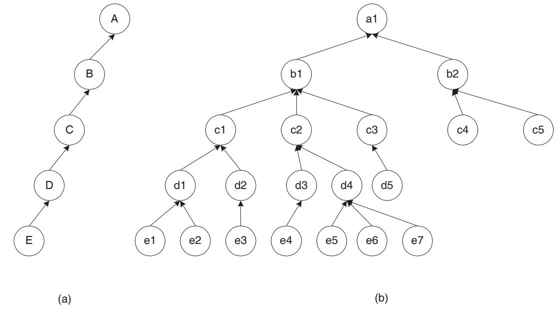

Figure 3: A LSS and an instance tree at a specific moment

Definition 3.1: Assume a well-defined HSA h, $h = o_{i_1}.f_{j_1}(o_{i_2}.f_{j_2}(...(o_{i_k}.f_{j_k}(value))...))$. The ordered list $(o_{i_1}, o_{i_2}, \ldots, o_{i_k})$ is called the *session aggregation path* of h and is denoted as $SAP(h)$. The first o_{i_1} and last o_{i_k} session objects in this list are denoted as $first(h)$ and $last(h)$ respectively. Given any session object o in $SAP(h)$, $next(o)$ denotes the object following o in the SAP list and $previous(o)$ the object preceding o in $SAP(h)$. Note that any suffix of h beginning with an object name O_{j_l} is also an HSA. This is called the *partial stream aggregate* of h with respect to O_{j_l} and is denoted as $HSA(h, O_{j_l})$. The partial stream aggregate $HSA(h, last(h))$ (i.e. $o_{i_k}.f_{j_k}(value)$) is called the *stream handler* of h and is denoted as $StreamHandler(h)$. Each object's O definition in $SAP(h)$ is extended with an attribute of the appropriate type, called the O's value with respect to h, to keep the intermediate result of the aggregation for h. This attribute is named as $O. < HSA_name > _value$ (e.g. $O_{i_2}.h_value$). □

Example 3.1: Consider the LSS given in figure 3(a) and a HSA `h = A.avg(C.min(E.sum(v)))`, where `v` some attribute of E's description or the event's schema. According to definition 3.1 we have: SAP(h)=(A,B,C), first(h)=A, last(h)=C, HSA(h,C)=C.min(E.sum(v)), StreamHandler(h)=E.sum(v). We also have that A's, B's and C's descriptions are extended to have an attribute named `h_value`.
□

Observation 3.1: Stream handlers have special status during evaluation of HSAs because they can be implemented as object methods and not as HSAs. As a result, they can be evaluated along with other methods

in a session, without propagating results up as the Back Propagation algorithm (described below) dictates. This is a frequent case in many applications. □

Another issue has to do with aggregate semantics. Some aggregate functions have to be updated only when a session instance closes out while others require continuous update. Consider for example a HSA `h1 = C.avg(E.sum(v))` defined on figure 3(a). E's value with respect to `h1` (`E.h1_value`) has to propagate up to the corresponding C instance, only when E's instances close out, otherwise the result is erroneous. For example, in figure 3(b), `c2.h1_value` changes only when one of the `e4, e5, e6, e7` E instances closes out. On the other hand, if we have an HSA `h2 = C.min(E.sum(v))`, we may want to update C's value with respect to `h2` as soon as `E.h1_value` changes. So both semantics could be desirable. To handle this in a uniform way in our algorithm, we mark each object's value with respect to an HSA h (i.e. `O.h_value`) with a tag that states when this value becomes available to $previous(O)$ in the SAP list of the HSA (possible values: `on_open, continuous, on_close`). This tag depends on the semantics of the aggregate function of $previous(O)$ session object. If the tag of `O.h_value` is `on_open`, then `O.h_value` is reported once to higher levels and then it reports NULL. This could be used for `count` aggregates. If the tag is `on_close`, then `O.h_value` reports its real value when the instance closes out. All the other times it reports NULL. This could be used for `average` aggregates. Since NULL values do not participate in aggregate computations, the algorithm presented below computes HSAs correctly.

The following algorithm, given a HSA h, a session instance s and a value v, propagates changes up from $ObjectOf(s)$ to $first(h)$.

Algorithm 3.2: The Back-Propagation Algorithm:

```
BPA (HSA h, session s, value v) {
     if (v != NULL) {
         O = ObjectOf(s);
         if (O ∈ SAP(h)) {
             UpdateHSA(s,h,v);
             if (O != first(h))
                 BPA(h, parent(s),
                         reported(s.h_value));
         }
         else if ((O != first(h))
                   BPA(h, parent(s), v);
     }
}□
```

We assume that BPA has access to a data dictionary containing information about declared HSAs. The algorithm uses as parameters a HSA h, a session instance s and a value v. The algorithm computes only part of the HSA h, namely from $ObjectOf(s)$ to $first(h)$ - it considers that the partial stream aggregate $HSA(h, next(ObjectOf(s))$ has already been computed and its value is v. This assumption is useful in most of the cases (e.g. stream handlers compute their values as object methods and not implementing the above algorithm) and allows for greater flexibility in using the algorithm.

If the class of s is part of the SAP of h, then it uses value v to compute s' value with respect to h (s.h_value) and propagates the *reported* result to the parent of s[2]. In case of $ObjectOf(s)$ is not part of $SAP(h)$, then BPA propagates up v with no changes.

This algorithm can be modified to propagate changes up to a specific session object O in $SAP(h)$, not necessarily to $first(h)$. This is useful in some cases to optimize performance in the presence of extra information or simply to compute an approximate value.

In NESTREAM system, stream handlers are implemented as object methods so they are updated whenever the session attributes/methods are updated/called (with the `update()` method of algorithm 3.1.) The remaining part of an HSA is computed whenever a session instance closes out, by calling `BPA()` on that session [3]. The `Close()` function given in algorithm 3.1 is modified in NESTREAM to the following:

```
Close (session s) {
     foreach subsession obj O in ObjectOf(s)
         foreach child s′ of s in O
             Close(s′);
     foreach h in HSAs
         if (ObjectOf(s) ∈ SPA(h))
             BPA(h,parent(s),s.h_value);
     s.destructor();
}
```

Example 3.2: Consider example 3.1 and an instance tree, as depicted in figure 3. When session $e4$ closes out, it calls `BPA(h,d3,e4.h_value)`, since `d3=parent(e4)` and `ObjectOf(e4)=E and E ∈ SAP(h)`. Since `ObjectOf(d3)=D and D ∉ SAP(h)` there is a recursive call to `BPA(h,c2,e4.h_value)`. At that level, if `e4.h_value` is less than `c2.h_value`, the later changes to the former and there is a new invocation of `BPA` with parameters `(h,b1,reported(c2.h_value))`. Since `reported(c2.h_value)` is NULL (because A's aggregate function is `avg`) the call to BPA is `BPA(h,b1,NULL)`. At that level, since v == NULL, recursion stops there. □

---

[2] `reported` is a function that checks the tag of `s.h_value` and returns either the actual value of `s.h_value` or NULL.

[3] Note that this implies that computation of `count` becomes available when a session instance closes and not when it opens.

4 NESTREAM Implementation

NESTREAM is a stream prototype that incorporates many of the ideas described above. It has been developed in C/C++ and all preliminary testing was performed on a Pentium 4 2GHz machine running Linux OS. The front end is a textual interface, where the user can define session objects in C++-like notation or XML, specify data sources and configure the DMI. Session object specifications follow definition 2.6. Once the LSS has been defined, the user can compile the schema to a C++ program that links to the FDA engine. All object functions are implemented using callbacks coded in C. All object instances are maintained by the FDA engine inside linked lists. Hashing is used for speeding-up all searches using primary keys and Θ_{FC} constraints.

Currently, registration of HSAs require recompilation of the schema and restart of the engine. Another alternative was to introduce a simple programming language for defining both the LSS and the HSAs. The stream language would then be compiled into some form of bytecode and executed inside a virtual machine. The obvious advantage to this approach is the ability to dynamically modify the HSAs during runtime, which is a natural approach when dealing with continuous streams compared to a stop-recompile-run process. Unfortunately, such an approach would consume a great deal of development time for a functioning prototype and it would narrow the gap for successful optimizations.

We are in the process of building a GUI on top of the textual front-end. Users will be able to define all objects, their relationships and HSAs graphically, using drag and drop operations. A GUI will also provide the user with an intuitive interface to view the HSAs results in real-time by clicking on each session object.

Figure 4 presents some preliminary performance results. We used a LSS of four levels (similar to figure 3 with levels A, B, C, and D). The first level has just one instance (e.g. `System`), while the other levels may have up to a hundred instances per node (i.e. level B may have 10^2 instances, level C 10^4 instances and level D 10^6). We experimented with three HSAs. H1 involves aggregates of all levels (`A.avg(B.avg(C.avg(D.value)))`), H2 involves aggregates of two levels (`A.avg(B.avg(C.value))`) and H3 involves a single-level aggregate. The tuple size was 40 bytes.

The graph in figure 4 shows the stream rate when the average lifetime size of session instances at level two ranges from 10 to 1000 tuples with a step of 10. Recall that when a high-level instance closes-out, all descendants also close out, calling BPA recursively. Longer session lifetimes at level two means larger number of sub-sessions at lower levels and thus, greater number of BPA invocations when a second-level session closes out. This explains the degradation in performance as

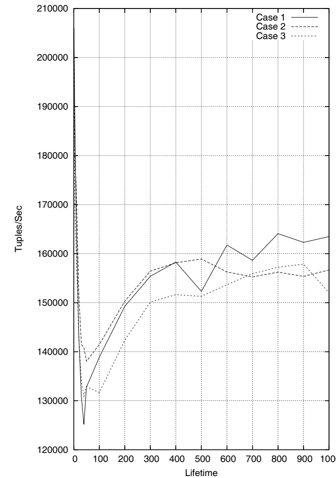

Figure 4: Stream rate vs. lifetime size of sessions

the average lifetime size increases up to 100. When lifetime reaches 100 performance improves somehow, since on average most instances at level three are open, so additional tuples do not translate to more instances.

This graph indicates that the main performance bottleneck is the number of BPA invocations (which relates to the opening and closing rate of session instances) and not the back propagation process (which relates to the number of levels). We are in the process of conducting a full set of experiments, varying the complexity of session methods, number of levels and number of instances.

5 Future Work

This research work is just at the beginning. There are several interesting research questions we have not discussed either due to lack of space or simply because they are not mature enough.

A Complete Session-Oriented Data Model. In this paper we have given the basic definitions for session objects and we have identified a common and useful relationship among them, hierarchies. However, in an object-oriented framework (e.g. UML class diagrams [3]) there are other interesting relationships, such as associations and aggregations (whole-part associations). What do these mean (semantically) for a session-oriented framework? What multiple inheritance would mean?

Query Language Issues. Although we have introduced the concept of hierarchical stream aggregates as means of a query language, we have not touched upon other issues, such as selections or projections, creating thus a complete query language. We are aware of applications that require some kind of conditional selection of session instances. In example 3.1, we may want only

E sessions that have `sum(v) > 30` to participate in the computation of the minimum for C instances. Opening and closing conditions may involve HSAs allowing thus a greater flexibility in defining when a session instance closes out.

Optimization. There is plenty of room for optimizing the two basic algorithms 3.1 and 3.2. For example, FDA can avoid flowing down stream data if it can *deduce* that it will not affect session instances lower in the tree. Similarly, BPA does not need to propagate up computed aggregates if it can deduce that these will not change aggregates further up in the HSA SAP. Multiple HSAs can be computed in the same invocation of BPA.

6 Conclusions & Related Work

In this paper we introduced a class of applications dealing with nested stream processing and argue that this class includes many interesting and common real-life problems. Our experience shows that these applications require high level constructs to describe complex relationships between stream processing nodes (e.g. hierarchies) and ad hoc computations (e.g. peculiar manipulation of incoming stream data). We have implemented many of these concepts in a system prototype called NESTREAM. Our goal is to make NESTREAM a useful tool in managing data streams in practical ways.

Fundamental properties and assumptions of data stream management systems can be found in [2]. The need for defining substreams in an infinite stream is identified in [13] by Tucker et.al. This is achieved by a punctuation mechanism. Punctuations denote the end of a subset of data and can be seen as predicates over data in a stream. This is something to consider for our opening and closing conditions. Hierarchical stream aggregates are similar in spirit to correlated aggregates, either in traditional databases [6, 11] or continual data streams [9]. The difference between the two worlds is the requirement of one pass. In [10], Lerner and Shasha present a framework to query ordered data and discuss the problem of inter-tuple communication. Many issues on sequence databases ([12]) are similar to those presented here. Wang and Zaniolo argue for user-defined aggregates (UDAs) [14]. Our approach for a stream system is similar to theirs, identifying the rich procedural semantics and the individual needs of stream applications. But UDAs lack coupling with structural relationships, such as hierarchies. Finally, there are similarities with well-known stream projects ([4, 7, 1, 5]), particularly Aurora, where tuples also flow through a loop-free directed graph of processing boxes. The key difference is in state and semantics (e.g. instances and hierarchies).

References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[4] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *28th International Conference on Very Large Databases (VLDB)*, pages 215–226, 2002.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and S. M. A. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Conference on Innovative Data Systems Research*, 2003.

[6] D. Chatziantoniou and K. Ross. Querying Multiple Features of Groups in Relational Databases. In *22nd International Conference on Very Large Databases (VLDB)*, pages 295–306, 1996.

[7] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD, Conference on Management of Data*, pages 647–651, 2003.

[8] Envox AB. *Envox Communications Development Platform*. Envox Group, 2002.

[9] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates Over Continual Data Streams. In *ACM SIGMOD, Conference on Management of Data*, 2001.

[10] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *29th International Conference on Very Large Databases (VLDB)*, pages 345–356, 2003.

[11] K. Ross, D. Srivastava, and D. Chatziantoniou. Complex Aggregation at Multiple Granularities. In *Extending Database Technology (EDBT), Valencia*, pages 263–277, 1998.

[12] P. Seshadri, M. Livny, and R. Raghu. The design and implementation of a sequence database system. In *Proceedings of the 22nd VLDB Conference*, 1996.

[13] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[14] H. Wang and C. Zaniolo. User Defined Aggregates in Object-Relational Systems. In *International Conference on Data Engineering (ICDE)*, pages 135–144, 2000.