Logical Foundations of Continuous Query Languages for Data Streams

Carlo Zaniolo

University of California at Los Angeles
zaniolo@cs.ucla.edu

Abstract. Data Stream Management Systems (DSMS) have attracted much interest from the database community, and extensions of relational database languages were proposed for expressing continuous queries on data streams. However, while relational databases were built on the solid bedrock of logic, the same cannot be said for DSMS. Thus, a logic-based reconstruction of DSMS languages and their unique computational model is long overdue. Indeed, the banning of blocking queries and the fact that stream data are ordered by their arrival timestamps represent major new aspects that have yet to be characterized by simple theories. In this paper, we show that these new requirements can be modeled using the familiar deductive database concepts of closed-world assumption and explicit local stratification. Besides its obvious theoretical interest, this approach leads to the design of a powerful version of Datalog for data streams. This language is called Streamlog and takes the query and application languages of DSMS to new levels of expressive power, by removing the unnecessary limitations that severely impair current commercial systems and research prototypes.

1 Introduction

Data stream management systems represent a vibrant area of new technology for which researchers have extended database query languages to support continuous queries on data streams [4,3,8,10,18,7,12,20]. These database-inspired approaches have produced remarkable systems and applications, but have yet to deliver solid theoretical foundations for their data models and query languages—particularly if we compare with the extraordinary ones on which the success of relational databases was built. Logic provided the theoretical bedrock for relational databases from the very time in which they were introduced by E.F. Codd, and this foundation was then refined, generalized and strengthened by the work on database and logic, and Datalog, which delivered concepts and models of great power and elegance [21,1,22].

Until now, DSMS researchers have made little use of logic-based concepts, although these provide a natural formalism and simple solutions for many of the difficult problems besetting this area, as we will show in this paper. In particular, we show that Reiter's Closed World assumption [19] provides a natural basis on which to study and formalize the blocking behavior of continuous query

P. Barceló and R. Pichler (Eds.): Datalog 2.0, LNCS 7494, pp. 177–189, 2012.

operators, whereby concepts such as local stratification can be used to achieve a natural and efficient expression of recursive rules with non-monotonic constructs.

The paper is organized as follows. In the next section, we present a short discussion of related previous work and then, in Section 3, we explore the problem of supporting order and recursion on single stream queries for both monotonic and non-monotonic constructs. Thus, in Section 4, we introduce Streamlog, which is basically Datalog with modified well-formedness rules for negation. These rules guarantee both simple declarative semantics and efficient execution (Section 5). Because of possible skews between their timestamps, multiple streams pose complex challenges at the logical and implementation levels. We study this problem in Section 6, where we propose a backtrack-oriented solution and show that its benefits extend well beyond union.

2 Continuous Queries on Relational Data Streams

As described in various surveys [4,12], data streams can be modeled as append-only relations on which the DSMS is asked to support standing queries (i.e., continuous queries). As soon as tuples arrive in the input stream, the DSMS is expected to decide, in real time or quasi real-time, which additional results belong to the query answer and promptly append them to the output stream. This is an incremental computation model, where no output can be taken back; therefore, the DSMS might have to delay returning an output tuple until it is sure that the tuple belongs to the final output—a certainty that for many queries is only reached after the DSMS has seen the whole input. The queries showing this behavior, and operators causing it, are called *blocking*, and have been characterized in [4] as follows: *A blocking query operator is one that is unable to produce the first tuple of the output until it has seen the entire input.* Clearly, blocking query operators are incompatible with the computation model of DSMS and should be disallowed, whereas all non-blocking queries should instead be allowed. However, many queries and operators, including essential ones such as union, fall in-between and are only partially blocking; currently, we lack simple rules to decide when, and to which extent, partially blocking operators should be allowed and how they should be treated. Therefore, better understanding and formal characterizations are badly needed.

The main previous results on blocking queries proved that non-monotonic query operators are blocking, whereas monotonic operators are non-blocking [17,13]. Given that negation and traditional aggregates are non-monotonic, most current DSMS simply disallow them in queries, although this exclusion causes major losses in expressive power [17]. However, a more sophisticated analysis suggests that these losses are avoidable, since (i) the monotonicity notion used in [17] is not the subset ordering used in databases and Horn clauses, and (ii) previous research on deductive databases made great strides in coping with non-monotonicity via concepts such as stratification and stable models [22].

Therefore, in this paper, we provide a reasoned reconstruction of the basic non-monotonic theory of logic languages in the context of data streams, leading to the

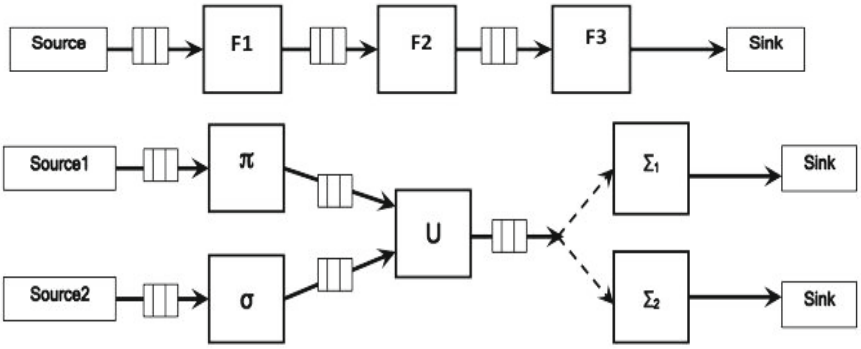

Fig. 1. Continuous Query Graphs

design of a concrete language called Streamlog. We will revisit the closed-world assumption and adapt well-known concepts such as stratification to discover, much to our surprise, that non-monotonic constructs dovetail with data stream languages, enabling Streamlog to achieve great expressive power.

Queries on data streams are commonly visualized using workflow models such as that of Figure 1, that show the pipelined execution used by the DSMS to implement continuous queries. The boxes labelled `Source` at the left of our graph, depict tuples coming from an external stream source or a database relation. For instance in the first query, the source feeds incoming tuples to a buffer; then query operator `F1` takes the tuples from this buffer and feeds them to its output buffer that supplies operator `F2`, and so on. As shown in Figure 1, some boxes might consist of very simple operators, e.g., the relational algebra operators of projection, selection and union. In general, however, the boxes can implement much more complex functions, including pattern search operators or data mining tasks [20]. Complex functions can been written as user-defined aggregates written natively in SQL [20], but Streamlog can also be quite effective in this role.

A key assumption is that operators are order-preserving. Thus, each operator takes tuples from the front of its input queue and add the tuple(s) it produces, if any to the tail of its output buffer. Thus, buffers might delay but not alter the functions computed by simply feeding the output of one operator directly into the input of the next. Thus, when the operators are defined by Streamlog rules, then the semantics of our continuous query is defined by the logic program consisting of (i) the goal defined by the `Sink` node (ii) the rules in the boxes feeding, directly or indirectly, into the sink node, and (iii) the facts streaming from the `source` nodes into said boxes and rules.

In this paper, we focus on data streams whose tuples are explicitly timestamped. More specifically, we will assume that the first column of our tuples contain a timestamp that either (i) was created by the external device that created the tuple (external timestamp) or (ii) it was added by the DSMS at the time it received the tuple (internal timestamp). In either case, tuples are arranged and processed by increasing values of their timestamps. Extending these results to data streams that are not timestamped will be discussed in future papers.

3 Single Stream Processing

The top query graph of Figure 1 shows the processing flow for a single stream, while the one below it shows the processing of multiple streams. In both cases we assume that timestamped data streams (i) enter each query operator in increasing timestamp order and (ii) leave the query operator in the same order. As we shall see, although (i) and (ii) represent two facets of the same problem, the technical issues and opportunities they bring about are quite different. For (i) consider the example of a stream of messages of the form `msg(Time, MsgCode)` and say that we are looking for repeated occurrences of code "red" messages. Then the following Datalog rule can be used to define multiple occurrences of the same alarm code "`X`":

Example 1. Repeated occurrences of the same alarm.

$$\texttt{repeated}(\texttt{T}, \texttt{X}) \leftarrow \texttt{msg}(\texttt{T}, \texttt{X}), \texttt{msg}(\texttt{T0}, \texttt{X}), \texttt{T} > \texttt{T0}.$$

Thus, the final query goal ?`repeated(T, red)` will detect repeated occurrences of code "red", whereby an application might sound an alarm, which is triggered for all but the first occurrence of code red.

The semantics of query Q on a stream, such as `msg`, is defined by the cumulative answer that Q has returned until time τ. This cumulative answer at time τ must be equal to the answer computed upon the database containing all the data stream tuples with timestamp $\leq \tau$. In a blocking query, this equality only holds at the end of the input, whereas for a continuous non-blocking query it must hold for every instant in time.

Massive data streams over long periods can exceed the system storage capacity. In DSMS, this problem is addressed with windows or other synopses [3]. Queries involving windows can be easily expressed using rules. For instance if `wsize(W)` defines the window within which we detect repetitions, Example 1 becomes:

Example 2. Multiple occurrences within a window

$$\begin{aligned}\texttt{windoweg}(\texttt{T2}, \texttt{red}) \leftarrow\ & \texttt{msg}(\texttt{T2}, \texttt{red}), \texttt{msg}(\texttt{T1}, \texttt{red}), \\ & \texttt{T1} < \texttt{T2}, \texttt{wsize}(\texttt{W}), \texttt{T2} \leq \texttt{T1} + \texttt{W}.\end{aligned}$$

But, unlike in other DSMS [3], windows do not play a key role in our semantics.

The Importance of Order. Since query operators return sequences of tuples that are fed into the next query operator, assuring the correct order of their output sequences becomes critical. To illustrate this point, say that we modify Example 1, above, by keeping the body of the rule unchanged; but then we change the head of the rule so that the timestamp of the former occurrence is used, rather than the current one:

Example 3. Time-warped repetitions ?`wrepeated(Time, X)`

$$\texttt{wrepeated}(\texttt{T0}, \texttt{X}) \leftarrow \texttt{msg}(\texttt{T}, \texttt{X}), \texttt{msg}(\texttt{T0}, \texttt{X}), \texttt{T} > \texttt{T0}.$$

We immediately realize that there is a problem, since repetitions normally arrive in an order that is different from that of their previous occurrences. For instance, we might have that a message with code α arrives at time t_α, followed by a message with code β, which is then repeated immediately, while the first repetition of α arrives 10 minutes later. Then, to produce tuples by increasing timestamps, we will need to hold up the output for 10 minutes. Here too punctuation marks and windows are effective palliatives to control the problem, but in general the delay required can be unbound. The situation of *unbound wait* can be as bad as that of blocking queries. For instance say that at some point, a rare color shows up in our input stream, never to show up again. Then for any new color that has its first occurrence after our rare color, no output can ever be generated until the very end of the input. In a nutshell, rules such as that of Example 3 must be disallowed, although they contain no negation or other nonmonotonic operators.

Negated Goals: The addition of order-inducing constraints in the rules offers unexpected major benefits when dealing with negated goals. Say that we want to detect the first occurrence of "code red" warning. For that, we only need to make sure that once we receive such a message there is no identical other message preceding it:

Example 4. First occurrence of code red: ?$\texttt{first}(\texttt{T}, \texttt{red})$.

$$\begin{aligned} \texttt{first}(\texttt{T}, \texttt{X}) &\leftarrow \quad \texttt{msg}(\texttt{T}, \texttt{X}), \neg\texttt{previous}(\texttt{T}, \texttt{X}). \\ \texttt{previous}(\texttt{T}, \texttt{X}) &\leftarrow \texttt{msg}(\texttt{T0}, \texttt{X}), \texttt{T0} < \texttt{T}. \end{aligned}$$

To find the second occurrence of code red we will start by finding one that follows the first. Moreover there cannot be any other occurrence between this and the first one:

Example 5. Second occurrence of code red ?$\texttt{second}(\texttt{T}, \texttt{red})$.

$$\begin{aligned} \texttt{second}(\texttt{T2}, \texttt{Y}) &\leftarrow \texttt{first}(\texttt{T1}, \texttt{Y}), \texttt{msg}(\texttt{T2}, \texttt{Y}), \texttt{T2} > \texttt{T1}, \neg\texttt{befr}(\texttt{T2}, \texttt{Y}). \\ \texttt{befr}(\texttt{T2}, \texttt{Y}) &\leftarrow \quad \texttt{first}(\texttt{T1}, \texttt{Y}), \texttt{T1} < \texttt{T2}, \texttt{msg}(\texttt{Tb}, \texttt{Y}), \texttt{Tb} < \texttt{T2}, \texttt{T1} < \texttt{Tb}. \end{aligned}$$

These queries only use negation on events that, according to their timestamps, are past event. Thus the queries can be answered in the present: they are non-blocking. Therefore, they should be allowed by a DSMS compiler, which must therefore be able to set them apart from other queries with negation which are instead blocking.

For instance, a blocking query is the following one that finds the last occurrence of code-red alert:

Example 6. Last occurrence of code red: ?$\texttt{last}(\texttt{T}, \texttt{red})$.

$$\begin{aligned} \texttt{last}(\texttt{T}, \texttt{Z}) &\leftarrow \texttt{msg}(\texttt{T}, \texttt{Z}), \neg\texttt{next}(\texttt{T}, \texttt{Z}). \\ \texttt{next}(\texttt{T}, \texttt{Z}) &\leftarrow \texttt{msg}(\texttt{T1}, \texttt{Z}), \texttt{T1} > \texttt{T}. \end{aligned}$$

This is obviously a blocking query, inasmuch as we do not have the information needed to decide whether the current red-alert message is actually the final one,

while messages are still arriving. Only when the data stream ends, we can make such an inference: to answer this query correctly, we will have to wait till the input stream has completed its arrival, and then we can use the standard CWA to entail the negation that allows us to answer our query. But the standard CWA assumption will not help us to conclude that our query is non-blocking. We will instead exploit the timestamp ordering of the data streams to define a Progressive Closing World Assumption (PCWA) that can be used in the task. In our definition, we will also include traditional database facts and rules, since these might also be used in continuous queries.

Progressive Closing World Assumption (PCWA): We consider a world consisting of one timestamped-ordered stream and database facts. Once a fact `stream(T, ...)` is observed in the input stream, the PCWA allows us to assume $\neg$`stream(T1, ...)`, provided that `T1` $<$ `T`, and `stream(T1, ...)` is not entailed by our fact base augmented with the `stream` facts having timestamp $\leq$ `T`.

Therefore, our PCWA for a single data stream revises the standard CWA of deductive databases with the provision that the world is in fact expanding according to its timestamps. Therefore, we will also allow the standard notions of entailment that guarantee consistency: besides the least models of Horn Clauses these also include the perfect models of (locally) stratified programs.

In the next section, we derive from the PCWA simple conditions that ensure syntactic well-formedness and efficient implementation for our programs.

4 Streamlog

In Streamlog, base predicates, derived predicates, and the query goal are all timestamped in their first arguments. These will be called temporal, to distinguish them from non-timestamped database facts and predicates that might also be used in the programs.

The same safety criteria used in Datalog can be used in Streamlog. Furthermore, we assume that timestamp variables are made safe by equality chains equating their values to the timestamps in the base stream predicates. Therefore, even if $T1$ is safe, expressions such as $T2 = f(T1)$ or $T2 = T1 + 1$ cannot be used to deduce the safety of $T2$. Only equality can be used for timestamp arguments.

We can now propose obvious syntactic rules that will avoid blocking behavior in the temporal rules of safe Streamlog programs.

- *Strictly Sequential:* A rule is said to be *Strictly sequential* when the timestamp of its head is $>$ than every timestamp in the body of the rule. A predicate is strictly sequential when all the rules defining it are strictly sequential.
- *Sequential:* A rule is said to be sequential when it satisfies the following three conditions:
  (i) the timestamp of its head is equal to the timestamp of some positve goal,
  (ii) the timestamp of its head is $>$ or $\geq$ than the timestamps of the remaining goals, and

(iii) its negated goals are strictly sequential or have a timestamp that is $<$ than the timestamp of the head.

– A program is said to be sequential when all its rules are sequential or strictly sequential.

The programs in Examples 4, and 5 are sequential, given that the predicates `previous` and `befr` in their negated goals are strictly sequential.

Next observe that the programs in Examples 4, and 5 are stratified with respect to negation with `previous` occupying a lower stratum than `first`, which is in a stratum not higher than `befr`, which is in a stratum lower than `second`.

Stratified Datalog programs have a syntactic structure that is easy for a compiler to recognize and turn into an efficient implementation [22]. In fact, the unique stable model of these programs, called the perfect model, can be computed efficiently using a stratified iterated fixpoint [22]. Unfortunately stratified programs do not allow negation or aggregates in recursive rules, and therefore, are not conducive to efficient expression of algorithms such as shortest path. Much previous research was devoted to overcoming this limitation. In particular, there is a class of programs called locally stratified programs that have a unique stable model, called perfect model. Unfortunately, the stratification for a locally stratified programs can only be verified against its instantiated version, whereby supporting perfect models is, in general, an Π^1_1-complete problem [9].

Overcoming this limitation and supporting negation or aggregates in recursion has thus provided a major focus of topical research where progress has been very slow. Therefore, we were pleasantly surprised to find out that the simple notion of sequential programs for Streamlog avoids the non-monotonicity problems that have hamstrung Datalog and frustrated generations of researchers. To illustrate this point, let us first use the stratified program of Example 7 to express the well-known shortest path problem. In this example, we use the paths computed for previous timestamps to discard longer arcs in the incoming stream. We also use a simple predicate `lgr(T1, T2, T)` whereby `T` is equal to the larger of the first two arguments:

Example 7. Continuous shortest paths in graphs defined by stream of arcs.

$$\begin{aligned}
&\texttt{path}(\texttt{T},\texttt{X},\texttt{Y},\texttt{D}) \leftarrow && \texttt{arc}(\texttt{T},\texttt{X},\texttt{Y},\texttt{D}), \neg\texttt{shorter}(\texttt{T},\texttt{X},\texttt{Y},\texttt{D}).\\
&\texttt{shorter}(\texttt{T},\texttt{X},\texttt{Y},\texttt{D}) \leftarrow && \texttt{path}(\texttt{T1},\texttt{X},\texttt{Y},\texttt{D1}), \texttt{T1} < \texttt{T}, \texttt{D1} \leq \texttt{D}.\\
&\texttt{path}(\texttt{T},\texttt{X},\texttt{Z},\texttt{D}) \leftarrow && \texttt{path}(\texttt{T1},\texttt{X},\texttt{Y},\texttt{D1}), \texttt{path}(\texttt{T2},\texttt{Y},\texttt{Z},\texttt{D2}),\\
& && \texttt{lgr}(\texttt{T1},\texttt{T2},\texttt{T}), \texttt{D} = \texttt{D1}+\texttt{D2}.
\end{aligned}$$

According to these rules, the arrival of one or more new arc will trigger addition of new edges in `path`. Then these new edges can trigger the addition of additional ones in recursive rule, where `path` appears twice. The differential fixpoint used in this computation [22] will result in at least one of these two `path` goals to have a timstamp equal to `T`—i.e., the larger of the two values is used to timestamp new fact generated in the head. This quadratic expression of transitive closure requires only the memorization of `path`; it is thus preferable to a linear rule that uses both `arc` and `path`, both of which then require memorization.

5 Declarative Semantics and Operational Semantics

Example 8 above, shows how to improve our rules by *pushing negation into recursion*. The program so obtained is sequential, and therefore it has a formal semantics and efficient implementation that are discussed after the example.

Example 8. Negation can be pushed inside recursion.

$$\begin{aligned}
&\texttt{minpath}(\texttt{T},\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{arc}(\texttt{T},\texttt{X},\texttt{Y},\texttt{D}), \neg\texttt{shorter}(\texttt{T},\texttt{X},\texttt{Y},\texttt{D}).\\
&\texttt{minpath}(\texttt{T},\texttt{X},\texttt{Z},\texttt{D}) \leftarrow \texttt{minpath}(\texttt{T1},\texttt{X},\texttt{Y},\texttt{D1}), \texttt{minpath}(\texttt{T2},\texttt{Y},\texttt{Z},\texttt{D1}), \texttt{lgr}(\texttt{T1},\texttt{T2},\texttt{T}),\\
&\qquad\qquad\qquad \neg\texttt{shorter}(\texttt{T},\texttt{X},\texttt{Z},\texttt{D}), \texttt{D} = \texttt{D1} + \texttt{D2}.\\
&\texttt{shorter}(\texttt{T},\texttt{X},\texttt{Z},\texttt{D}) \leftarrow \texttt{minpath}(\texttt{T1},\texttt{X},\texttt{Z},\texttt{D1}), \texttt{D} \leq \texttt{D1}, \texttt{T1} < \texttt{T}.
\end{aligned}$$

The timestamps in our data stream form a sequence that is unbound but finite. We can denote them by their position in the sequence, and talk about the n^{th} timestamp, without fear of ambiguity. Then, sequential programs are locally stratified by their timestamp values as discussed next. To prove this we will construct the *bistate* equivalent of our program. The first step is a temporal expansion, where a rule with a condition $\leq$ ($\geq$) between temporal arguments, is replaced by two rules: in the first rule, the temporal arguments are set to equal, and in the other they are set to $<$ ($> S$). Likewise, a rule with `lgr` is replaced by three rules, resp. for $=$, $<$ and$>$. Then we have the following rewriting:

1. In each rule, rename with the suffix `new_` the head predicate and the body predicates that have a timestamp equal to the that of the head,
2. Rename all the predicates in the body whose temporal argument is less than that of the head by the suffix `_old`
3. Remove the temporal arguments from the rules.

Thus, for Example 8 we obtain:

Example 9. Bistate represention for the program of Eample 8

$$\begin{aligned}
&\texttt{minpath_new}(\texttt{X},\texttt{Y},\texttt{D}) \leftarrow \texttt{arc_new}(\texttt{X},\texttt{Y},\texttt{D}), \neg\texttt{shorter_new}(\texttt{X},\texttt{Y},\texttt{D}).\\
&\texttt{minpath_new}(\texttt{X},\texttt{Z},\texttt{D}) \leftarrow \texttt{minpath_new}(\texttt{X},\texttt{Y},\texttt{D1}), \texttt{minpath_new}(\texttt{Y},\texttt{Z},\texttt{D1}),\\
&\qquad\qquad\qquad \neg\texttt{shorter_new}(\texttt{X},\texttt{Z},\texttt{D}), \texttt{D} = \texttt{D1} + \texttt{D2}.\\
&\texttt{minpath_new}(\texttt{X},\texttt{Z},\texttt{D}) \leftarrow \texttt{minpath_old}(\texttt{X},\texttt{Y},\texttt{D1}), \texttt{minpath_new}(\texttt{Y},\texttt{Z},\texttt{D1}),\\
&\qquad\qquad\qquad \neg\texttt{shorter_new}(\texttt{X},\texttt{Z},\texttt{D}), \texttt{D} = \texttt{D1} + \texttt{D2}.\\
&\texttt{minpath_new}(\texttt{X},\texttt{Z},\texttt{D}) \leftarrow \texttt{minpath_new}(\texttt{X},\texttt{Y},\texttt{D1}), \texttt{minpath_old}(\texttt{Y},\texttt{Z},\texttt{D1}),\\
&\qquad\qquad\qquad \neg\texttt{shorter_new}(\texttt{X},\texttt{Z},\texttt{D}), \texttt{D} = \texttt{D1} + \texttt{D2}.\\
&\texttt{shorter_new}(\texttt{X},\texttt{Z},\texttt{D}) \leftarrow \texttt{minpath_old}(\texttt{X},\texttt{Z},\texttt{D1}), \texttt{D} \leq \texttt{D1}.
\end{aligned}$$

The program so obtained is stratifiable in several ways, including the following one: we assign to stratum 0 all and only the predicates with suffix `old`, and the predicates with suffix `new` are all in higher strata. For instance, we will assign `minpath_old` to level 0, and then `shorter_new`, and `arc_new` to level 1, and `minpath_new` to level 2. Thus here we have one stratum with `_old` predicates, and $S = 2$ strata for `_new` predicates.

Now we can generate a local stratification based on the distinct temporal values of the timestamps which form a finite sequence $\tau_1, \ldots, \tau_n$. In our case $\texttt{minpath}(0, \ldots)$ will be asigned to stratum 0, $\texttt{shorter}(\tau_1, \ldots)$ and $\texttt{arc}(\tau_1, \ldots)$ are assigned to stratum 1, and $\texttt{minpath}(\tau_1, \ldots)$ are assigned to stratum 2. Then the process repeats with with temporal argument T2 being assigned as follows: $\texttt{shorter}(\tau_2, \ldots)$ and $\texttt{arc}(\tau_2, \ldots)$ to stratum 3, and $\texttt{minpath}(\tau_2, \ldots)$ to stratum 4. Thus, for our sequence $\tau_1, \ldots, \tau_n$ we have $1 + n \times S$ strata, where a $\texttt{p\_new}(\tau_j, \ldots)$ that belonged to state k in the bistate version will now be assigned to stratum $j \times S + k$ in the local stratification.

The computation of perfect model for the locally stratified program now becomes straightforward. Basically, we iterate over the following two steps *for each set of tuples arriving with a new timestamp*: (i) the stratified bistate version of the program is used to derive additional new values for the predicates, and (ii) the old version is incremented with this newly derived atoms.

6 Multiple Streams

A much studied DSMS problem is how to best ensure that binary query operators, such as unions or joins, generate outputs sorted by increasing timestamp values [14,5,6]. To derive a logic-based characterization of this problem, assume that our msg stream is in fact built by combining the two message streams sensr1 and sensr2. For stored data, this operation requires a simple disjunction as follows:

Example 10. Disjunction expressing the union of two streams.

$$\texttt{msg}(\texttt{T1}, \texttt{S1}) \leftarrow \texttt{sensr1}(\texttt{T1}, \texttt{S1}).$$
$$\texttt{msg}(\texttt{T2}, \texttt{S2}) \leftarrow \texttt{sensr2}(\texttt{T2}, \texttt{S2}).$$

However even if sensr1 and sensr2 are ordered by their timestamps, this disjunction says nothing about the fact that the output should be ordered. Indeed, assuring such an order represents a serious problem for a DSMS, due to the time-skews that normally occur between different data streams. Thus, for the union in Figure 1, when one of the two input buffers is empty, we cannot take the first item from the other buffer, until we know what its timestamp value will be. This problem has been extensively studied, but only at the implementation level [14,5,6]. At the logical level the problem can be solved as follows:

Example 11. Union of synchronized streams.

$$\texttt{msg}(\texttt{T1}, \texttt{S1}) \leftarrow \texttt{sensr1}(\texttt{T1}, \texttt{S1}), \neg\texttt{missing2}(\texttt{T1}).$$
$$\texttt{msg}(\texttt{T2}, \texttt{S2}) \leftarrow \texttt{sensr2}(\texttt{T2}, \texttt{S2}), \neg\texttt{missing1}(\texttt{T2}).$$

Now we check that all the stream2 tuples (resp. the stream1 tuples) with timestamp less than T1 (resp. less than T2) added to msg:

$$\texttt{missing2}(\texttt{T1}) \leftarrow \texttt{sensr2}(\texttt{T2}, \texttt{S2}), \texttt{T2} < \texttt{T1}, \neg\texttt{msg}(\texttt{T2}, \texttt{S2}).$$
$$\texttt{missing1}(\texttt{T2}) \leftarrow \texttt{sensr1}(\texttt{T1}, \texttt{S}), \texttt{T1} < \texttt{T2}, \neg\texttt{msg}(\texttt{T1}, \texttt{S1}).$$

The expression given in Example 11 is clearly better than the sort-merge approach proposed in the literature that can be described as follows:

Example 12. Union of unsynchronized streams by sort merging.

$$\texttt{msg(T1, S1)} \leftarrow \texttt{sensr1(T1, S1)}, \texttt{sensr2(T2, \_)}, \texttt{T2} \geq \texttt{T1}.$$
$$\texttt{msg(T2, S2)} \leftarrow \texttt{sensr2(T2, S2)}, \texttt{sensr1(T1, \_)}, \texttt{T1} \geq \texttt{T2}.$$

This expression is correct but not complete[1]. As a result, this operator might have to enter an idle-waiting state that is akin to temporary blocking [5].

From the viewpoint of users, neither the solution in Example 11 nor that in Example 12 are satisfactory. What users instead want is to write the simple rules shown in Example 10 and let the system take care of time-skews. Therefore in Streamlog, we will allow users to work under the *Perfect Syncronization Assumption (PSA)*, whereby the data streams of interest are perfectly synchronized. Under PSA, we ca now extend the PCWA we had previously defined for a single data stream to a collection of N data streams $\texttt{stream}_\texttt{j}(\texttt{T}, \ldots)$, $j = 1, \ldots, N$ as follows: We can assume $\neg\texttt{stream}_\texttt{j}(\texttt{T}_\texttt{j}, \ldots)$ iff for some i, $1 \leq i \leq N$ $\texttt{stream}_\texttt{i}(\texttt{T}_\texttt{i}, \ldots)$ with $\texttt{T}_\texttt{i} > \texttt{T}_\texttt{j}$, and $\texttt{stream}(\texttt{T}_\texttt{j}, \ldots)$ is not entailed by our fact base augmented with all the stream facts with timestamp $\leq \texttt{T}_\texttt{i}$ Since in reality PSA does not hold, the DSMS is given the responsibility to enforce efficient policies and conditions needed to ensure that queries return the same answers as those produced under PSA. For instance, for the query at hand, the DSMS system might in fact enforces conditions such as $\neg\texttt{missing2(T1)}$ in the first rule of our union. Efficient support of these PSA-emulating conditions requires the use of sophisticated techniques, such as *intelligent backtracking* [5]. For instance in Figure 1, say that the lower buffer feeding the union has a tuple with timestamp t_1, while the other buffer is empty. Rather than waiting idly for the arrival of some tuples in the empty buffer, we can backtrack to the previous operators feeding the buffer. If a tuple with timestamp $< t_1$ is found, it must be moved quickly through the operators since this is the one that must pass through to the union next. But if only tuples with timestamps $> t_1$ are found, then the union operator will be signalled (e.g., via punctuation marks) to let the tuple with timestamp t_1 to go through. Finally, if the buffer is empty, an additional backtracking step will be performed to visit the buffer supplied by `Source1`, and so on. As discussed in [5], this backtracking approach can lead to significant improvements in the response time of our DSMS. Although space limitations prevent us from discussing this approach further, we observe that (i) while Streamlog is clearly inspired by Datalog, its execution exploits Prolog's backtracking mechanism, and (ii) the techniques used to support PSA are also very useful to control and expedite execution of single-stream queries.

To illustrate (ii), say that $\texttt{tempr(Time, Locat, Celcius)}$ is the stream containing the temperature readings of our sensors, from various locations. Then the following rules could be used to continuously return each new temperature maximum:

[1] For instance, the `sensr2` stream might have ended and the current clock is past `T1`.

$$\texttt{max}(\texttt{T}, \texttt{Loc}, \texttt{Cel}) \leftarrow \texttt{tempr}(\texttt{T}, \texttt{Loc}, \texttt{Cel}), \neg\texttt{hotter}(\texttt{T}, \texttt{Cel}).$$
$$\texttt{hotter}(\texttt{T1}, \texttt{C1}) \leftarrow \texttt{tempr}(\texttt{T2}, , \texttt{C2}), \texttt{C2} \geq \texttt{C1}, \texttt{T2} \leq \texttt{T1}.$$

Here the backtracking technique used to support NSA for union can be used to detect that all the the `tempr` tuples with timestamp $\leq$ `T` have already arrived and thus the query ?`max`(`T`, `Loc`, `Celcius`) can be answered at once without awaiting for tuples with timestamps larger than `T`. Also, if we construct the bistate equivalent of our rules we see that we obtain a stratified program, whereby the original program is locally stratified and the efficient execution techniques previously discussed remain valid. Therefore we can relax the definition of Strictly Sequential rules as follows:
Strictly Sequential: A rule is said to be *Strictly sequential* when the timestamp of the head of the rule is $>$ than the timestamp of each recursive goal and $\geq$ the timestamps of the non-recursive goals.

This extension does not compromise the key properties of our sequential programs, for which the following properties hold[2]:

Theorem 1. *If P is a Sequential Program then: (i) P is locally stratified, and (ii) the unique stable model of P can be computed by repeating the iterated fixpoint of its bistate version for each timestamp value.*

For an additional example illustrating the uses of this generalization, let us return to our shortest-path program in Example 8. When several arcs arrive with the same timestamp, they might result in the addition of multiple paths between the same node pair. Thus we could add an additional rule (and stratum) to select the shortest among such paths that share the same timestamp.

7 Conclusion

While the results presented here are still preliminary, they show that logic can bring sound theoretical foundations and superior expressive power to DSMS languages which, currently, are dreadfully lacking in both. In terms of syntax, Streamlog is just standard Datalog over timestamed predicates; however Streamlog obtains the greater level of expressive power that negation (and aggregates) in recursive rules entail by guaranteeing that simple sequentiality conditions holds between the timestamped predicates in the rules. The use of standard Datalog implies that the implementation techniques developed for XY-stratification [23] can be used for Streamlog, and similar results are at hand for the many DSMS that use continuous versions of SQL. This also sets our approach apart from that proposed in [2] that relies on an explicit sequencing operator SEQ and an operational semantics that is realized through a Prolog-based implementation.

[2] The outline of the proof of this property is similar to that outlined in previous section and it is based on a similar proof for XY-stratification presented in [22]. In fact, the temporal arguments define an explicit stratification that is similar to that of XY-stratified programs [23] and Statelog programs [15].

Acknowledgements. This work was supported in part by NSF (Grant No. IIS 1118107). Thanks are due to the reviewers for many useful comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: A Rule-Based Language for Complex Event Processing and Reasoning. In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 42–57. Springer, Heidelberg (2010)
3. Arasu, A., Babu, S., Widom, J.: CQL: A Language for Continuous Queries over Streams and Relations. In: Lausen, G., Suciu, D. (eds.) DBPL 2003. LNCS, vol. 2921, pp. 1–19. Springer, Heidelberg (2004)
4. Babcock, B., et al.: Models and issues in data stream systems. In: PODS (2002)
5. Bai, Y., Thakkar, H., Wang, H., Zaniolo, C.: Optimizing timestamp management in data stream management systems. In: ICDE (2007)
6. Bai, Y., Thakkar, H., Wang, H., Zaniolo, C.: Time-stamp management and query execution in data stream management systems. IEEE Internet Computing 12(6), 13–21 (2008)
7. Carney, D., et al.: Monitoring streams - a new class of data management applications. In: VLDB, Hong Kong, China (2002)
8. Chandrasekaran, S., Franklin, M.: Streaming queries over streaming data. In: VLDB (2002)
9. Cholak, P., Blair, H.A.: The complexity of local stratification. Fundam. Inform. 21(4), 333–344 (1994)
10. Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., Spatscheck, O.: Gigascope: High performance network monitoring with an sql interface. In: SIGMOD, p. 623. ACM Press (2002)
11. Gallaire, H., Nicolas, J.-M., Minker, J. (eds.): Advances in Data Base Theory, vol. 1. Plemum Press (1981)
12. Golab, L., Tamer Özsu, M.: Issues in data stream management. ACM SIGMOD Record 32(2), 5–14 (2003)
13. Gurevich, Y., Leinders, D., Van den Bussche, J.: A Theory of Stream Queries. In: Arenas, M. (ed.) DBPL 2007. LNCS, vol. 4797, pp. 153–168. Springer, Heidelberg (2007)
14. Johnson, T., Muthukrishnan, S., Shkapenyuk, V., Spatscheck, O.: A heartbeat mechanism and its application in gigascope. In: VLDB, pp. 1079–1088 (2005)
15. Lausen, G., Ludäscher, B., May, W.: On Active Deductive Databases: The Statelog Approach. In: Freitag, B., Decker, H., Kifer, M., Voronkov, A. (eds.) DYNAMICS 1997, and ILPS-WS 1997. LNCS, vol. 1472, pp. 69–106. Springer, Heidelberg (1998)
16. Law, Y.-N., Wang, H., Zaniolo, C.: Data models and query language for data streams. In: VLDB, pp. 492–503 (2004)
17. Law, Y.-N., Wang, H., Zaniolo, C.: Relational languages and data models for continuous queries on sequences and data streams. ACM Trans. Database Syst. 36, 8:1–8:32 (2011)
18. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously Adaptive Continuous Queries over Streams. In: SIGMOD, pp. 49–61 (2002)
19. Reiter, R.: On closed world data bases. In: Logic and Data Bases, pp. 55–76 (1977)

20. Thakkar, H., Laptev, N., Mousavi, H., Mozafari, B., Russo, V., Zaniolo, C.: Smm: A data stream management system for knowledge discovery. In: ICDE, pp. 757–768 (2011)
21. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. I. Computer Science Press (1988)
22. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann (1997)
23. Zaniolo, C., Arni, N., Ong, K.: Negation and Aggregates in Recursive Rules: The LDL++ Approach. In: Ceri, S., Tsur, S., Tanaka, K. (eds.) DOOD 1993. LNCS, vol. 760, pp. 204–221. Springer, Heidelberg (1993)