Relational Languages and Data Models for Continuous Queries on Sequences and Data Streams

YAN-NEI LAW, Bioinformatics Institute, Singapore
HAIXUN WANG, Microsoft Research Asia, China
CARLO ZANIOLO, Computer Science Department, University of California Los Angeles

Most data stream management systems are based on extensions of the relational data model and query languages, but rigorous analyses of the problems and limitations of this approach, and how to overcome them, are still wanting. In this paper, we elucidate the interaction between stream-oriented extensions of the relational model and continuous query language constructs, and show that the resulting expressive power problems are even more serious for data streams than for databases. In particular, we study the loss of expressive power caused by the loss of blocking query operators, and characterize nonblocking queries as monotonic functions on the database. Thus, we introduce the notion of $\mathcal{NB}$-completeness to assure that a query language is as suitable for continuous queries as it is for traditional database queries. We show that neither RA nor SQL are $\mathcal{NB}$-complete on unordered sets of tuples, and the problem is even more serious when the data model is extended to support order—a sine-qua-non in data stream applications. The new limitations of SQL, compounded with well-known problems in applications such as sequence queries and data mining, motivate our proposal of extending the language with user-defined aggregates (UDAs). These can be natively coded in SQL, according to simple syntactic rules that set nonblocking aggregates apart from blocking ones. We first prove that SQL with UDAs is Turing complete. We then prove that SQL with monotonic UDAs and union operators can express all monotonic set functions computable by a Turing machine ($\mathcal{NB}$-completeness) and finally extend this result to queries on sequences ordered by their timestamps. The proposed approach supports data stream models that are more sophisticated than append-only relations, along with data mining queries, and other complex applications.

Categories and Subject Descriptors: H.2.2 [**Languages**]: Data Stream Query Languages & Data Models

General Terms: Data Streams, Queries, Expressivity

1. INTRODUCTION

Data stream management systems represent a vibrant area of research [Babcock et al. 2002; Barbara 1999; Terry et al. 1992; Chandrasekaran and Franklin 2002; Cranor et al. 2002; Madden et al. 2002; Sullivan 1996; Liu et al. 1999; Chen et al. 2000; Carney et al. 2002; Golab and Özsu 2003]. The solution approach taken by most projects consists of extending database query languages and data models to support efficient continuous queries on stream data, and is based on the sound rationale that an unified programming environment will simplify the development of the many applications that span traditional databases and data streams. Nevertheless, database query lan-

This work was supported in part by the National Science Foundation, under grant IIS-0917333.
Author's addresses: Yan-Nei Law, Bioinformatics Institute, 30 Biopolis Street, Singapore 138671, lawyn@bii.a-star.edu.sg; Haixun Wang, Microsoft Research Asia, 49 Zhichun Road, Beijing 100080, China, haixunw@microsoft.com; Carlo Zaniolo, Computer Science Department, University of California, Los Angeles, CA 90095, USA, zaniolo@cs.ucla.edu.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

guages and data models were designed for persistent data residing on disks, rather than for transient data flowing through the wires. Therefore, their suitability to the new task must be reevaluated critically, and we must be prepared to address the limitations that impair their effectiveness in this new role. Indeed, such limitations are many and severe, and SQL's ineffectiveness in expressing queries on time series and sequences has been long recognized in the field and inspired much previous research [Seshadri et al. 1994; Seshadri 1998; Ramakrishnan et al. 1998; Perng and Parker 1999; Sadri et al. 2001a; 2001b]. Since data streams are basically unbounded sequences, this can be viewed as a limitation of SQL for continuous queries. Another well-known problem area for SQL is data mining [Han et al. 1996; Meo et al. 1996; Imielinski and Virmani 1999; Sarawagi et al. 1998].

While the SQL problems mentioned above hold for both databases and data streams, the problem that only nonblocking query operators can be used is specific to data streams [Babcock et al. 2002]. In this paper, we show that all and only monotonic queries can be expressed using nonblocking operators, and then show that neither relational algebra (RA) nor SQL are complete for nonblocking queries, since they can only express some monotonic queries by using blocking operators (which must be disallowed on data streams). This loss of expressive power is further aggravated by the fact that, in traditional databases, more complex applications are developed by embedding SQL queries in procedural language programs, whereby computations that cannot be easily expressed in SQL would therefore be written in the procedural program. As explained in Section 4.4, this solution loses much of its power in DSMS since these support a push-based computation model instead of the pull-based computation model of traditional DBMS[1].

The compounding of the traditional SQL problems with the new ones suggests that we need (i) theoretical models on how the blocking problem limit constructs and expressive power of continuous query languages, and (ii) practical language extensions to overcome these limitations. Thus in [Law et al. 2004], a formal model was proposed that characterizes the expressive power of a continuous query language in terms of its ability to compute monotonic functions, and allowed to derive (i) an incompleteness result for traditional relational algebra and (ii) a completeness result for SQL extended with user-defined aggregates (UDAs) natively defined using SQL itself. Thus the incompleteness result obtained in [Law et al. 2004] states that the set of monotonic operators of relational algebra cannot express all the monotonic queries expressible in relational algebra, while the completeness result in [Law et al. 2004] states that all the monotonic functions computable by a Turing machine can be expressed using the monotonic subset of SQL extended with native non-blocking UDAs (whereas, unrestricted SQL, extended with unrestricted native UDAs can express all computable queries).

In this paper, we provide a formal proof for the incompleteness result stated in [Law et al. 2004], and give the conditions under which the completeness result of [Law et al. 2004] can be generalized from sets of tuples to ordered sequences and streams of tuples. To achieve this generalization, we introduce the notion of monotonic approximation and show that monotonic SQL extended with non-blocking native UDAs can express all monotonic functions, modulo delays. In actual DSMS, these delays can be minimized by punctuation timestamps [Bai et al. 2008]. The original punctuation proposed in [P.Tucker et al. 2003; Tucker et al. 2003] focused on un-blocking single streams by exploiting the logical properties of the data; here instead we use punctuation timestamps to un-block operators idle-waiting on multiple inputs [Johnson et al.

[1]Of course, languages using data-flow-oriented execution models, callbacks, coroutines, etc., could be used instead of vanilla procedural languages.

2005; Bai et al. 2008]. In Section 8, we discuss the practical benefits of the expressiveness levels achieved by our operators along with their limitations. Thus, on the one hand, alternative data stream models and advanced applications can be supported by exploiting the superior expressiveness of the proposed SQL+UDAs framework; on the other hand, specialized extensions, such as the recently proposed Kleene-star constructs for searching patterns that we briefly discuss in Section 8, provide usability and optimizability benefits that are not available in our SQL+UDAs framework (but any other Turing-complete framework is likely to face similar problems).

1.1. Paper Organization

The next section presents an overview and discussion on related work. Then in Section 3, we define the notion of nonblocking query operators and show that a function can be expressed by nonblocking operators iff it is monotonic. We prove this result for a very general data stream model where tuples are not required to have timestamps and they are simply ordered by their arrivals. In Section 4, we explore the implications of the "monotonic-only" upon relational query languages, and introduce the formal notion of $\mathcal{N}B$-completeness to assure that these languages are as suitable on data streams as they are on stored data. The fact that RA and SQL are not $\mathcal{N}B$-complete further aggravates their severe expressive power limitations, and convince us that new constructs are needed to overcome this problem. This is accomplished in Section 5 with the introduction of user-defined aggregates, which can be easily classified as blocking and nonblocking on the basis of their syntactic structure. In fact, in Section 6, we show that SQL with natively defined UDAs become Turing complete on DB tables. However, completeness on data streams require the introduction of an operator that merges multiple data streams in a nonblocking way. Therefore, in Section 7, we introduce a data model whereby tuples are explicitly ordered by their timestamps, and we use this to define the notion of τ-union, that is equivalent to standard set union, modulo the delay due to the timestamp skew between the two streams. In Section 8, we discuss the power and generality of the proposed solution, and show that complex applications and even semantically richer data stream models can be emulated using append-only relations and UDAs. Section 9 concludes the paper.

2. RELATED WORK

Rather than trying to cover the large body of interesting previous work on data streams, including [Babcock et al. 2002; Barbara 1999; Terry et al. 1992; Chandrasekaran and Franklin 2002; Cranor et al. 2002; Madden et al. 2002; Sullivan 1996; Liu et al. 1999; Chen et al. 2000; Carney et al. 2002; Golab and Özsu 2003], we will refer our reader to two previous surveys [Babcock et al. 2002; Golab and Özsu 2003], and just focus on papers covering blocking operators, data model, query power, and other issues that are directly relevant to this paper's discussion.

The Tapestry project was the first to model data streams as append-only databases supporting continuous queries [Terry et al. 1992]. The problem of blocking operators was also identified in [Terry et al. 1992], where strategies were suggested for overcoming this problem for monotonic queries. Indeed the close relationship between monotonicity and nonblocking queries has been understood for a long time, however as far as we know, there has been no previous attempt to prove or formalize this relationship. For instance, two excellent survey papers [Babcock et al. 2002; Golab and Özsu 2003] clearly note the relationship, but make no statement to the fact that queries expressible by nonblocking operators are exactly the monotonic queries—more remarkably this property is not even mentioned as a 'folk theorem', or a formal conjecture. For instance, there is no formal characterization of blocking operators in [P.Tucker et al.

2003], where punctuated data streams are proposed to overcome the problems caused by such operators. In [P.Tucker et al. 2003], the data stream is modelled as an unbounded sequence of finite lists of elements. Then the punctuation marks proposed in [P.Tucker et al. 2003] can be viewed as predicates on stream elements that must evaluate to false for every element following the punctuation. Note that a punctuation is an ordered set of patterns which indicates what should be output and stored for future uses and when it should be output. Then a stream iterator is proposed that visits the input incrementally, outputting the results as another punctuated stream and storing the state, based on the punctuation of the input elements. To achieve this, a unary stream iterator is defined as five components `(inital_state, step, pass, prop, keep)`, where `inital_state` is the iterator state before any tuple arrives, `step` is a function that takes new tuples and a current state and output new tuples and a modified state and `pass, prop, keep` are three behavior functions that take punctuation marks and state as input and returns additional outputs tuples, output punctuation, and a modified state. Clearly, the structure of unary stream iterators is similar to that User-Defined Aggregates (UDAs), which are very versatile and can also deal with punctuation. UDAs defined in procedural languages, are also supported in the DSMS Aurora [Carney et al. 2002], where they can be used for more complex queries that are hard to express using Aurora's algebra which consists of the following operators delivered to users through an attractive UI: Filter, Map, Union, Bsort, Aggregate, Join, Resample [Carney et al. 2002]. In this paper, we model data streams as append-only bags of ordered tuples (i.e., unbounded sequences); this is the basic data model adopted by many successful DSMS [Cranor et al. 2002; Thakkar et al. 2011].

An assortment of semantically richer models have also been proposed in the literature starting with CQL [Arasu et al. 2003; Motwani et al. 2003], which engrafts windows onto this basic representation, and introduces concepts such as `Istream`, and `Dstream` to describe the tuples joining and leaving the window. Theoretical treatments of these richer data stream models have elucidated their temporal aspects [Krämer and Seeger 2005] and the relational algebra extensions needed to support them [Golab and Özsu 2005]. In particular, the model proposed in [Krämer and Seeger 2005; 2009] preserves the ability of expressing arbitrary relational algebra queries on multisets, including non-monotonic ones, by assuming that a validity lifespan is associated with each tuple. In fact, this assumption avoids a blocking behavior at the cost of (possibly severe) delays [Krämer and Seeger 2009]; similar observations can be made about the recently proposed Linq query language of StreamInsight [Krishnan and Goldstein 2010].

The approach presented in this paper instead retains the 'spartan simplicity' of append-only relations and, after a rigorous analysis of the problems encountered on data streams by relational query languages, proposes a powerful and general solution to these problems based on UDAs.

This paper's treatment of expressive power issues benefits from a successful line of database research that led to important theoretical findings [Abiteboul et al. 1995] and major practical advances, such as the inclusion recursive queries and their enabling technology in DBMS [Zaniolo et al. 1997; ISO/IEC 2003]. While database-like query languages for DSMS have attracted much interest from database researchers, their focus has been mostly on implementation, and critical issues about expressive power have, in general, received surprisingly little attention. For instance, after proposing an abstract calculus for stream processing languages in [Soulé et al. 2010], the authors mainly focus on the efficient mapping of this calculus into alternative implementations. Two exceptions to this trend are the work by Law et al. mentioned in the introduction [Law et al. 2004], and the work by Gurevich et al. [Gurevich et al. 2007] discussed next. The work discussed in [Gurevich et al. 2007] uses bounded memory machines with the intent of providing a formal framework where the relationships between computability, continuity and monotonicity can be easily explored. Using this framework, the authors prove that a query Q is computable by a nonblocking operator iff Q is monotonic (i.e., the result first presented in [Law et al. 2004]). Then they discuss a monotonic version of set-intersection for sorted data streams, but they do not discuss the monotonicity issue for the union of multiple streams, that is central to this paper, where we also show that the issue of expressive power is critical for continuous query languages. Indeed, in the past, expressiveness proved to be a major challenge for database query languages [Abiteboul et al. 1995], and, as discussed in Section 4.4, it represents an even greater challenge for data stream query languages since (i) blocking query operators are disallowed, (ii) push-based queries can not be easily embedded in a pull-based procedural language, and (iii) non-blocking operators of relational algebra cannot express all monotonic queries expressible in relational algebra. In this paper, therefore, we focus on the expressive power problem for continuous query languages, and address and solve difficult issues that were ignored in [Law et al. 2004]. Paramount among these is the fact that non-unary operators, such as unions and joins, lose their monotonic properties when they are applied to ordered sets and display a partially blocking behavior [Johnson et al. 2005; Bai et al. 2008]. We deal with this issue in Section 7, by introducing the notion of monotonic approximation to compensate for skews between multiple streams. A second key issue addressed, and positively answered in Section 8, is whether semantically richer data stream models, that were recently proposed in the literature, are strictly required or their functionality can instead be attained using the basic append-only relation model and the richer expressiveness of the continuous query languages discussed here. We will use an amalgam of formal and informal arguments to make our points. In particular, in Sections 4.1–2 we provide the first formal proof of incompleteness of RA and SQL.

3. NONBLOCKING QUERY OPERATORS

We will next formalize the notion of *sequences*, which provides a natural data model for data streams and a simple generalization of database relations. Sequences consist of ordered tuples, whereas the order is immaterial in relational tables. Streams are sequences of unbounded length, where the tuples are ordered by, and possibly timestamped with, their arrival time. An open problem in this line of research is to find what generalizations of the relation data model, algebra, and query languages are needed to deal with sequences and streams [Babcock et al. 2002]. In this section, we will characterize:

— The blocking/nonblocking properties of operators independent of the language in which they are expressed, and
— The abstract properties of stream functions expressible by blocking/non-blocking operators.

According to [Babcock et al. 2002] *'A blocking query operator is a query operator that is unable to produce the first tuple of the output until it has seen the entire input.'* In an operational reading of this definition 'until it has seen the entire input' should be taken to mean 'until it has detected the end of the input'. For instance, the traditional aggregates in SQL never produce any tuple until they have seen the last input tuple: thus these are blocking operators. Since continuous queries must return answers without waiting for tuples that will arrive in the future, blocking operators are not suitable for stream processing [Babcock et al. 2002]. Nonblocking operators are instead suitable for stream processing. We can now define nonblocking operators, as follows (the opposite of the statement used to define blocking operators): *'A nonblocking query operator is* *one that produces all the tuples of the output before it has detected the end of the input.'* Here we have discussed operators that are either blocking or nonblocking; but the case of partially blocking operators is also possible, although less frequent in practice. For instance, an online average aggregate that returns results during the computation but also the final result at the end is partially blocking. To characterize the properties of stream operators we will first formalize the notion of sequences, and computation on sequences.

Definition 3.1. *Sequence:* Let $t_1, \ldots, t_n$ be tuples from a relation R. Then, the list $S = [t_1, \ldots, t_n]$ is called a sequence, of length n, of tuples from R. The empty sequence has length 0 and is denoted by [ ].

Observe that the tuples $t_1, \ldots, t_n$ in the sequence are not necessarily distinct. We will use the notation $t \in S$ to denote that, for some $1 \leq i \leq n$, $t_i = t$.

Definition 3.2. *Presequence:* Let $S = [t_1, \ldots, t_n]$ be a sequence and $0 < k \leq n$. Then, $t_1, \ldots, t_k$ is the *presequence* of S of length k, denoted by S^k. [ ] is the zero-length presequence of S.

Definition 3.3. *Partial Order:* Let S and L be two sequences. Then, if for some k, $L^k = S$ we say that S is a presequence of L and write $S \sqsubseteq L$. If $k < n$, we say that S is a proper presequence of L and write $S \sqsubset L$.

Given a relation R, $\sqsubseteq$ is a partial order (reflexive, transitive, and antisymmetric) on sequences of tuples from R. We can now consider operators that take sequences (streams) as input and return sequences (streams) as output. For instance consider an operator G that takes a sequence S as input and produces a sequence $G(S)$ as output:

$$S \longrightarrow \boxed{G} \longrightarrow G(S)$$

G operates as an incremental transducer, which for each new input tuple in S, adds zero, one, or several tuples to the output. At step j, G consumes the j^{th} input tuple and produces any number of tuples as output (as a function of only the tuples seen so far, since future tuples are still unknown). But rather than focusing on the new output produced at step j, we will concentrate on the *cumulative* output produced up to and including step j. Thus, let $G^j(S)$ be the cumulative output produced up to step j by our operator G presented with the input sequence S. $G^j(S)$ is a sequence whose content and length depend on G, j and S. Consider, for instance, a sequence of length n, i.e., $S = S^n$. If G is a traditional SQL aggregate, such as SUM or AVG, then $G^j(S)$ is the empty sequence for $j < n$, while, for $j = n$, $G^j(S)$ contains a single tuple. However, if G is the continuous count (continuous sum), defined as follows: for each new tuple, G returns the count of tuples (sum of a particular column) of the tuples seen so far—i.e., of S^j, then, by definition, $G^j(S) \sqsubseteq G^k(S)$, for $j \leq k$ — i.e., the output produced till step j is a presequence of that produced till step k. A null operator N is one where $N(S) = [\,]$ for every S. We now have the following definitions:

Definition 3.4. A non-null operator G is said to be

— nonblocking, when for every sequence S and every non-negative integer $j \leq |S|$: $G^j(S) = G(S^j)$;
— blocking, when for every sequence S and every positive integer $j < |S|$: $G^j(S) = [\,]$. (Whereas, $G^{|S|}(S) = G(S)$.)

Thus, a blocking operator is one that does not deliver any tuple in the output until the final input tuple. Instead, a nonblocking operator is one that performs the computation incrementally, i.e., the cumulative output at step $j < n$ (for an input sequence S of length n), can be computed by simply applying G to the presequence S^j. Partially blocking operators are those that do not satisfy either definition, i.e., those where, for some S and j:

$$[\,] \sqsubset G^j(S) \sqsubset G(S^j).$$

We would like now to elevate our abstraction level from that of operators and programs to that of mathematical functions. We ask the following question: what are the functions on streams that can be expressed by nonblocking operators? There is a surprisingly simple answer to this question:

PROPOSITION 3.5. *A function $F(S)$ on a sequence S can be computed using a non-blocking operator, iff F is monotonic with respect to the partial ordering $\sqsubseteq$.*

Proof. To show that 'nonblocking' implies 'monotonic', consider a sequence S, and its arbitrary presequence S^k (thus $k \leq n$). If G is nonblocking then $G(S^k) = G^k(S)$ and $G(S^n) = G^n(S)$; but since $k \leq n$, $G^k(S) \sqsubseteq G^n(S)$. Vice versa, say that we have a monotonic function $F(S)$ that can be computed by an operator $G(S)$. If G is nonblocking, the proof is complete. Otherwise, consider the operator $H(S)$ that at step $j+1$ returns all the tuples that are contained in $G^{j+1}(S^{j+1})$ but were not in $G^j(S^j)$. Obviously $H(S)$ is non-blocking. QED.

Streams are unbounded sequences; thus only nonblocking operators can be used to answer queries on streams. We have now discovered that a query Q on a stream S can be implemented by a nonblocking query operator iff $Q(S)$ is monotonic with respect to $\sqsubseteq$. The traditional aggregate operators (MAX, AVG, etc.) always return a sequence of length one and they are all nonmonotonic, and therefore blocking. Continuous count and sum are monotonic and nonblocking, and thus suitable for continuous queries. Other relational query operators that can be used on data streams are selection and projection. Selection is defined as the function that return the input tuple if this satisfies certain conditions. Projection (with or without duplicate elimination) is the function that return the input tuple with columns rearranged and removed (but for projection with duplicate elimination, only if this tuple was not returned previously). These functions are all monotonic with respect to the pre-sequence partial ordering, and thus can be freely used on data streams.

However, to use relational query languages on data streams we will also need to express RA operators such query operators, including, union, Cartesian product, joins, set difference, and relational division—at least to the extent in which these operators are non-blocking. Surprisingly enough, while the statement“only non-blocking query operators” has been repeated many times in topical papers and tutorial, we still do not have a clear definition of nonblocking/monotonicity for operators/function with multiple inputs.

3.1. Multiple Inputs

Although they were not considered in [Babcock et al. 2002], and discussed only briefly in [Law et al. 2004], operators with multiple inputs are very important and need to be characterized with respect to their blocking behavior. Here, we will limit our discussion to binary operators, which are necessary and sufficient for relation algebra queries.

Thus, our binary operator $G(S, W)$ is incrementally fed the two sequences S and W. $G^{ij}(S, W)$ denotes the cumulative sequence returned by G after it has seen the first i elements of S and the first j elements of W (i.e., after it has seen S^i and W^j). If $|S|$ (resp. $|W|$) denotes the length of S (resp. W), then we have that $G^{|S||W|}(S) = G(S, W)$ for both blocking and non-blocking functions: however the two behave quite differently on their presequences:

Definition 3.6. A binary non-null operator G is said to be

— nonblocking, when for every pair of sequences, S and W, and every pair of positive integers, $i \leq |S|$ and $j \leq |W|$: $G^{ij}(S, W) = G(S^i, W^j)$.
— blocking, when for every pair of sequences, S and W, and every pair of integers, $i \leq |S|$ and $j < |W|$: $G^{ij}(S, W) = [\,]$.

For instance, if S and W are two streams of elements, then we can define an operator $match$ that returns the maximal common pre-sequence of the two: $match$ is obviously a nonblocking operator. On the other hand, an operator that returns the sorted union of the two unsorted sequences is blocking (because of the blocking nature of sorting, and also other problems with union discussed below). We can also have operators (functions) that are blocking (nonmonotonic) on one argument and not on the other. For instance, consider a possible extension of set difference to sequences: we define a function that returns all the elements in S that are not in W is blocking on the latter but nonblocking on the former, i.e., nothing can be returned until the whole W is seen—but then after that, results can be returned incrementally as new elements of S arrive. When for every S and W $G(S, W) = [\,]$, then G is said to be a null operator.Thus:

Definition 3.7. A binary non-null operator G will be said to be nonblocking on its first argument and blocking on its second argument, when for every pair of sequences, S and W, and every pair of non-negative integers (i, j), $i \leq |S|$, $j < |W|$, the following two properties hold.

(i) $G^{ij}(S, W) = [\,]$, and

(ii) $G^{i|W|}(S, W) = G(S^i, W)$.

Proposition 3.5 and its proof can be easily extended to binary functions to yield:

PROPOSITION 3.8. *A binary function F can be computed by an operator that is nonblocking w.r.t. to one or both of its arguments iff F is monotonic w.r.t. the same argument(s).*

3.2. The NonMonotonic Curse of Order

Codd's relational algebra was designed for sets, i.e., collection of tuples where the order and duplicates are immaterial (as per the commutativity and idempotence property, respectively). Thus a function F on sequences is also valid on sets if presented with two sequences S_1 and S_2 that are equivalent modulo commutativity and idempotence, $F(S_1)$ and $F(S_2)$ are also equivalent modulo commutativity and idempotence: i.e., $\bar{S}_1 = \bar{S}_2 \Rightarrow \bar{F}(S_1) = \bar{F}(S_2)$ where $\bar{S}$ ($\bar{F}(S)$) denote the family of sequences equivalent to S ($F(S)$). We are now interested in functions that are valid on both sequences and sets. Then we have the following proposition:

PROPOSITION 3.9. *Let F be a function that is valid on both sequences and sets. Then, if F is a function on sequences that is monotonic w.r.t. the presequence partial order $\sqsubseteq$, then the set function F is monotonic w.r.t. set containment $\subseteq$.*

Proof. Let S_1 and S_2 be two sequences where $S_1 \sqsubseteq S_2$. Then if $\bar{S}_1$ and $\bar{S}_2$ denote the sets represented by S_1 and S_2 we have that $\bar{S}_1 \subseteq \bar{S}_2$. Since F is monotonic on sequences, $F(S_1) \sqsubseteq F(S_2)$; thus $\bar{F}(S_1) \sqsubseteq \bar{F}(S_2)$. The case of function with multiple arguments follows trivially in similar fashion.

Thus, every function that is monotonic with respect to the presequence order is also monotonic with respect to set containment. The opposite is not true. For instance, consider the function that presented with a sequence of numbers returns them in a sorted order: this function is nonmonotonic on sequences. However, if this is viewed as a set function, then it becomes an identity transformation, which is monotonic (and has a trivial nonblocking implementation, along with some blocking ones, such as the sorting one just described).

In terms of relational algebra, we find that operators such as set difference and division cannot be used in our continuous queries on data streams since they are nonmonotonic on sets, and thus by Proposition 3.9 also nonmonotonic on sequences. While this loss was expected, a much more devastating loss is that the union and Cartesian product operators that were monotonic on sets have now become nonmonotonic on sequences. In fact, a function $G([a],[b])$ to operate as a union, will have to either return $[a,b]$ or $[b,a]$. In the first case, we fail the monotonicity test with respect to $G([],[b]) = [b]$ (since this is not a presequence of $[a,b]$) and, in the second case, we fail with respect to $G([a],[]) = [a]$. Similar problems hold for Cartesian product.

Thus we are now faced with the realization that if we use sequences as our basic data model, nonmonotonic operators and queries become so dominant, that only basic project/select operations can be expressed as continuous queries. Thus, in the rest of the paper we will seek to correct this situation by introducing query operators and data models that are conducive to more expressive continuous query constructs. Our approach will consist of solving first the problem in the traditional framework of Codd's relations which, as we have just seen, is less prone to the nonmonotonicity curse. Then, after solving the problem for Codd's relations, in Section 7 we generalize our solution to data streams ordered by their timestamps.

4. RELATIONS, RA, AND SQL

Codd's relational model views relations as sets of tuples where the order is immaterial (commutativity property) [2]; moreover, duplicates are disallowed or considered immaterial (idempotence property).

Thus relations are sets ordered by set containment, $\subseteq$. For Codd's relations the notions $\subseteq$ and $\sqsubseteq$ coincide. (Indeed $\sqsubseteq$ always implies $\subseteq$; moreover, if $R_1 \subseteq R_2$, then R_2 can be arranged as a presequence identical to R_1 followed by the remaining tuples in $R_2 - R_1$, if any.)

Thus, for relations and functions on relations our monotonicity theorem specializes as follows:

PROPOSITION 4.1. *A function F on relations can be computed using a nonblocking operator iff F is monotonic w.r.t. the set containment ordering $\subseteq$.*

Proposition 4.1 also holds on binary relations since a function is monotonic with respect to certain arguments iff it can be implemented by an operator that is nonblocking on the same arguments.

Let us now discuss Codd's relational algebra, and explore the loss of expressive power caused by the fact that blocking operators are disallowed. A complete set of RA operators consists of the unary operators of projection and selection, and the binary operators of Cartesian product, union, and set difference.

Terminology. An important observation to be made here is that Codd's RA operators are actually mathematical *functions* defined as mappings from whole relations to whole relations. Thus they are defined by the fact that they are monotonic or not. Union and Cartesian product are monotonic with respect to set containment and amenable to nonblocking implementations. Set difference $R - S$ is instead antimonotonic with respect to its second argument. Only blocking implementations are possible for nonmonotonic functions: for instance, in an implementation of $R - S$ no tuple can be

[2]More limited uses of commutativity could take timestamps into account and these will be discussed in Section 7.

returned until the last tuple of S is known. Non-monotonic functions (and RA operators) should be avoided in continuous queries since they require blocking implementations. (We will explore the effects of this limitation in the next section.) For monotonic functions, both nonblocking and blocking implementations are possible, and only the first should be used on data streams. For instance, union is monotonic but a typical DB implementation of $R1 \cup R2$ consists in first fetching and returning all the tuples from $R1$ and, after this done, fetching and returning those in $R2$. This implementation makes the operator blocking w.r.t. $R1$ and thus cannot be used for continuous queries on data streams.

4.1. Relational Algebra

A complete set of operators for relational algebra consists of the following operators: $RA = \{\cup, \bowtie, \sigma, \Pi, -\}$. The monotonic (i.e., nonblocking) operators of relational algebra will be denoted $\mathcal{N}B$-RA, where $\mathcal{N}B$-RA = $\{\cup, \bowtie, \sigma, \Pi\}$. The class of queries expressible by RA (and many equivalent query languages) is called ***FO*** queries [Abiteboul et al. 1995]. Let $\mathcal{N}B$-***FO*** denote the monotonic queries in ***FO***. But some monotonic functions in ***FO*** are expressed using set difference, an operator not in $\mathcal{N}B$-RA. For instance, the intersection of two relations R_1 and R_2, a monotonic operation, can be expressed as: $R_1 \cap R2 = R_1 - (R_1 - R_2)$. On the other hand intersection is in $\mathcal{N}B$-RA, since it can also be expressed as the natural join of its operands. But the conclusion is different for the `coalesce` and `until` queries discussed next.

Coalesce and Until. Assume we have a temporal domain which, for simplicity, we represent using nonnegative integers originating at zero integers[3]. We use predicate $\mathtt{p(I,J)}$, with $\mathtt{I} < \mathtt{J}$, to denote that the property `p` holds from point `I`, included, till point `J`, excluded. In other words, we use an interval closed to the left and open to the right to represent the validity of a property. Our database consists of an arbitrary number of `p` facts, and of some `q` facts that use a similar interval-based representation. Then, the temporal-logic query `p` *Until* `q` is true when there exists a $\mathtt{q(I,J)}$ where `p` holds for every point before `I`. This query can be expressed in several ways [Bohlen 1994; Celko 1995; Rozenshtein et al. 1993]. Example 1 expresses it using non-recursive Datalog rules, that first coalesce the `p` intervals and then check if there is any interval that spans from 0 to the beginning of some `q` (second rule). The bottom rule in Example 1 defines `cep(K)` to hold for the 'covered end points' of intervals: i.e., when `K` is the endpoint of some interval that is contained in some other interval $\mathtt{p(I,J)}$. The next rule from the bottom defines broken intervals as follows: $\mathtt{broken(I1,J2)}$ holds true if (i) `I1` is the start-point of some interval, (ii) `J2` is the endpoint of an interval to its right, and (iii) there is a break point between the two in the form of the endpoint `K` that is not covered, i.e., $\neg\mathtt{cep(K)}$. This break excludes $\mathtt{(I1,J2)}$ from the coalesced intervals. Indeed, the third rule from the bottom defines coalesced intervals as those that satisfy conditions (i) and (ii), but are not broken.

Example 4.2. Until (`pUq`) & Coalesce (`coalscp`)

$$
\begin{array}{ll}
\mathtt{pUq(yes)} \leftarrow & \mathtt{q(0,J)}. \\
\mathtt{pUq(yes)} \leftarrow & \mathtt{coalscp(0,I), q(J,\_), I \geq J}. \\
\mathtt{coalscp(I1,J2)} \leftarrow & \mathtt{p(I1,J1), p(I2,J2), J1 < J2,} \\
 & \neg\mathtt{broken(I1,J2)}. \\
\mathtt{broken(I1,J2)} \leftarrow & \mathtt{p(I1,J1), p(I2,J2), p(\_,K),} \\
 & \mathtt{J1 \leq K, K < I2, \neg cep(K)}. \\
\mathtt{cep(K)} \leftarrow & \mathtt{p(\_,K), p(I,J), I \leq K, K < J}.
\end{array}
$$

[3]No relationship is assumed between these integers and the tuple timestamps.

The safe non-recursive Datalog program of Example 4.2 can be translated into an RA expression on the two relations P and Q, representing, respectively, the p facts and the q facts. The resulting RA expression uses set difference to implement negation. This program and its RA equivalent defines the two queries `pUq` and `coalscp`, the first on P and Q and the second on P only. We will refer to them as the coalesce query and the until query, and observe that they are monotonic. Indeed, as we add new intervals to P, we obtain all the old intervals in `coalscp` and possibly some new ones. For `pUq`, as we add new intervals to P and/or Q, the answer could change from an empty set to a singleton set containing 'yes' but never the other way around. Therefore, the coalesce query and the until query are in $\mathcal{N}B$-*FO*. However, we will now prove that they cannot be expressed in $\mathcal{N}B$-RA.

4.2. Incompleteness of $\mathcal{N}B$-RA

We use the notation T_α and T_ω to, respectively, denote the start-point and end-point of an interval T. Point x is contained in T iff $T_\alpha \leq x < T_\omega$. Given a non-empty set of intervals S, we will use the notation $\alpha(\mathbf{S})$ and $\omega(\mathbf{S})$ to, respectively, denote the least of the start points and the greatest of the end points of intervals in S. An interval is said to be *covered* by S when all its points are contained in some interval in S. Then,

Definition 4.3. A non-empty set of intervals S is said to be connected if the interval $[\,\alpha(\mathbf{S}), \omega(\mathbf{S})\,)$ is covered by S.

If $\{T_1, T_2\}$ is connected, then we say that T_1 is connected with T_2, and vice-versa. Let S be a set of intervals, and let $\mathbf{S}_{[a,b)}$ denote the intervals of S connected with $[a, b)$:

$$\mathbf{S}_{[a,b)} = \{T \in \mathbf{S} \mid T \text{ is connected with } [a, b)\}$$

Then we have the following properties:

LEMMA 4.4. *If a set of intervals* S *covers the interval* $[a, b)$*, then* $\mathbf{S}_{[a,b)}$ *is connected.*

PROOF. Given that S covers $[a, b)$, so does $\mathbf{S}_{[a,b)}$, and we have that: $\alpha(\mathbf{S}_{[a,b)}) \leq a < b \leq \omega(\mathbf{S}_{[a,b)})$. By construction, $\mathbf{S}_{[a,b)}$ covers the points in $[a, b)$. Thus, we only need to prove that $\mathbf{S}_{[a,b)}$ covers $[\,\alpha(\mathbf{S}_{[a,b)}), a\,)$, if $\alpha(\mathbf{S}_{[a,b)}) < a$, and $[\,b, \omega(\mathbf{S}_{[a,b)})\,)$, if $b < \omega(\mathbf{S}_{[a,b)})$. Indeed, if $\alpha(\mathbf{S}_{[a,b)}) < a$, then, for some $T \in \mathbf{S}_{[a,b)}$ where $T_\alpha = \alpha(\mathbf{S}_{[a,b)})$; but T is connected with $[a, b)$ and thus it contains every point in $[\,\alpha(\mathbf{S}_{[a,b)}), a\,)$. Symmetrically for $[\,b, \omega(\mathbf{S}_{[a,b)})\,)$, if $b < \omega(\mathbf{S}_{[a,b)})$. $\square$ $\square$

Let $sum(\mathbf{S}) = \sum_{T \in \mathbf{S}} T_\omega - T_\alpha$ denote the sum of lengths of the intervals in S:

LEMMA 4.5. *If* S *is a connected set of intervals, then:* $\omega(\mathbf{S}) - \alpha(\mathbf{S}) \leq sum(\mathbf{S})$.

PROOF. Let us first consider the case of $\mathbf{S} = \{T^1, \cdots, T^n\}$ where no interval contains another interval. We can assume, without any loss of generality, that the intervals are ordered by their start-points: $T^i_\alpha \leq T^{i+1}_\alpha$, $1 \leq i \leq n$, and since no interval contain another $T^i_\omega \leq T^{i+1}_\omega$, $1 \leq i \leq n$. Furthermore, $T^i_\omega - T^{i+1}_\alpha \geq 0$, because otherwise $[\,T^i_\omega, T^{i+1}_\alpha\,)$ is not covered by S. Thus:

$$\begin{aligned} T^n_\omega - T^1_\alpha &\leq T^n_\omega + (T^{n-1}_\omega - T^n_\alpha + \cdots + T^2_\omega - T^3_\alpha) - T^1_\alpha \\ &= T^n_\omega - T^n_\alpha + \cdots + T^1_\omega - T^1_\alpha = sum(\mathbf{S}) \end{aligned}$$

Since $\omega(\mathbf{S}) = T^n_\omega$ and $\alpha(\mathbf{S}) = T^1_\alpha$ this concludes our proof for the case where no interval of S contains another; otherwise let S′ be a maximal subset of S where no interval contains another. Then, $\omega(\mathbf{S}) - \alpha(\mathbf{S}) = \omega(\mathbf{S}') - \alpha(\mathbf{S}')$ whereas $sum(\mathbf{S}') \leq sum(\mathbf{S})$. $\square$ $\square$

PROPOSITION 4.6. *The Coalesce query and the Until query cannot be expressed in* $\mathcal{N}B$*-RA.*

Proof. Say that an expression of $\mathcal{NB} - \mathcal{RA}$ operators is applied to our binary relation $R(\alpha, \omega)$ containing the original intervals to derive another binary relation containing the coalesced intervals. We can assume, without loss of generality, that the operators are applied in this order: (1) cartesian products, (2) selections, (3) projections, and finally (4) union. Therefore, in phase 1, N cartesian products on $R(\alpha, \omega)$ will produce a relation $R'(\alpha_0, \omega_0, \ldots, \alpha_N, \omega_N)$, with $2(N + 1)$ columns; on this, we apply various selections to filter out some tuples in phase 2; then, in phase 3, we derive a two-column relation $R''(A, B)$ by projection operators. So, say that $(a, b) \in R''(A, B)$ was generated by the projection from a tuple $t \in R'$ (that survived the selection filter). Now, let S be the set of the following intervals $\{[t.\alpha_0, t.\omega_0), \ldots, [t.\alpha_N, t.\omega_N)\}$. Then, $[a, b)$ is a valid interval according to S only if $a < b$ and S covers $[a, b)$. But if S covers $[a, b)$, then $\mathbf{S}_{[a,b)}$ is connected, and $b - a \le sum(\mathbf{S}_{[a,b)}) \le sum(\mathbf{S})$. Now, if our relation R contains the intervals $[0, 1), \ldots, [M, M+1)$ then coalescing yields an interval of length $M+1$, whereas N cartesian products on R can only produce coalesced intervals of length $\le N + 1$, which is the sum of $N+1$ intervals. In general, $\mathcal{NB} - \mathcal{RA}$ expression with N cartesian products cannot perform the coalescing on a relation with more than $N + 1$ tuples. To prove the inexpressibility of $P\ Until\ Q$ let Q contain only interval $[M, M + 1)$ and let $Q = R$. $\square$

Thus, by disallowing non-monotonic RA operators, we also lose some of monotonic queries expressible in relational algebra. One possible solution to this problem consists in introducing a fixpoint operator to express recursive queries that are quite natural in conjunction with monotonic operators. For instance, the $Until$ and $Coalesce$ queries can be expressed in positive Datalog, as shown in Example 4.7. Indeed, recursive queries for data streams represent a topic of theoretical interest. However, this direction has not been pursued in current DSMS, since it is expected to bring about (i) major technical challenges, and (ii) limited practical benefits. Regarding (ii) in particular, recursive queries are not expected to solve pressing practical issues on how to use aggregates in data streams, which are discussed below.

Example 4.7. Until & Coalesce in Recursive Datalog

```
pUq(yes) ←          q(0, J).
pUq(yes) ←          coalscp(0, I), q(J, _), I ≥ J.
coalscp(I, J) ←     p(I, J).
coalscp(I1, J2) ←   coalscp(I1, J1), coalscp(I2, J2),
                    J1 ≥ I2.
```

4.3. Incompleteness of $\mathcal{N}B$-SQL

We next consider $\mathcal{N}B$-SQL, i.e., the nonblocking subset of SQL-2 that can be used for writing queries on data streams. We need to exclude nonmonotonic constructs, such as EXCEPT, NOT EXIST, NOT IN and ALL. Moreover all the standard SQL-2 aggregates, must be left out because they are blocking. The surprising conclusion is that expressive power of $\mathcal{N}B$-SQL is the same as $\mathcal{N}B$-RA, although SQL can express more monotonic queries than RA. In fact, some queries expressed using aggregates are monotonic. For instance, Example 4.8, below, computes from empl(EmpNo, Sal, DeptNo) all the departments where the sum of employee salaries exceeds a given constant C.

Example 4.8. Departments where the sum of employee salaries exceeds C. Assume $Sal > 0$.

```
SELECT DeptNo
FROM empl
GROUP BY DeptNo
  HAVING SUM(empl.Sal) > C
```

The above query uses aggregate SUM, which means it is not an $\mathcal{NB}$-SQL query. However, the computation it expresses is obviously monotonic, insofar as the introduction of a new `empl` can only expand the set of departments that satisfy this query; however this sum query cannot be expressed without the use of aggregates. The problem of the blocking SQL queries has long been recognized by data stream researchers, who have proposed the use of devices such as punctuation [P.Tucker et al. 2003] and windows [Motwani et al. 2003] to address this problem. While these approaches deal effectively with important aspects of the problem, they do not solve the expressiveness problems discussed so far. For instance, punctuation and windows cannot be used to implement queries of Example 4.2 or Example 4.8 unless some external constraints can be used to turn these blocking queries into nonblocking queries (such as, bounds on the maximum number of employees in a department). One approach to remedy these problems consists in allowing the programmer to use nonmonotonic constructs but exclusively to write monotonic queries. Then, the queries of Example 4.2 or Example 4.8 will be allowed and the loss of expressive power is avoided. Unfortunately, this approach is practically attractive only if the compiler/optimizer is capable of recognizing monotonic queries, and thus warning the user when a certain query is blocking and thus cannot be used as a continuous query. Unfortunately, deciding whether a query is monotonic can be computationally intractable and can also depend on information, such as **empl.Sal** $>$**0**, which is obvious to the user but not the optimizer.

4.4. The Expressive Power Issue

A practical solution approach must go beyond solving problems caused the exclusion of blocking operators since this is but one of the expressive power problems inflicted upon SQL by data streams. To illustrate this point, let us consider the old DB practice of embedding SQL into a procedural language program where the application logic that can not be expressed in SQL can then be easily implemented. In this approach cursors and get-next constructs, are used to express a pull-based computation model that loses much of its effectiveness in the push-based environment of data streams. In a DSMS, the answer tuples generated by the continuous queries must be pushed to the output buffers at once, and the DSMS cannot wait for get-next requests from an embedding procedural application. Datablades [Technologies 1994] (AKA database extenders and many other names) are library of external functions used in OR-DBMS to extend their power and ability to support new applications. These functions often use large objects (BLOBs and CLOBs) to exchange data with SQL: for instance, a whole sequence could be encoded as a BLOB and shared between the database and the datablade. This solution is less suitable for data streams, where the computation must proceed continuously in small increments—e.g., by processing each new tuple in the sequence, rather than having to wait for the sequence to be assembled into a BLOB. Indeed, unlike in OR DBMS, datablades have not played an important role as an extension mechanism for DSMS. Given the old problems that SQL experienced with sequence queries and mining queries, and the new ones introduced by data streams, the best solution is to introduce new operators that extend the $\mathcal{NB}$-power of the query language. For instance, a natural extensions could be to add least fixpoint (LFP) operators to relational algebra, or equivalently, recursion constructs could be used in SQL [Abiteboul et al. 1995]. LFP operators and recursive constructs are monotonic and they extend the power of RA or SQL to enable the expression of all DB-PTime queries [Abiteboul et al. 1995]. However, it is not clear whether $\mathcal{NB}$-RA+LFP, or $\mathcal{NB}$-SQL with recursion, are $\mathcal{NB}$-DB-PTime complete—i.e. capable of expressing all monotonic queries in DB-PTime. Although the coalesce and until query can be easily expressed in $\mathcal{NB}$-RA+LFP, we do not have a general answer for this interesting theoretical question. We will leave this question for later investigations, since it is not of urgent practical importance, given that, in the past, recursive SQL queries have not proven very useful for sequence queries and mining queries. In this paper, we instead pursue a very practical approach based of monotonic user-defined aggregates that deliver much higher levels of expressive power, not only in theory, but also in practice, as demonstrated in applications such as punctuated data streams, sequence queries, and mining queries.

5. USER-DEFINED AGGREGATES

User Defined Aggregates (UDAs) are important for decision support, stream queries and other advanced database applications [Carney et al. 2002; Luo et al. 2005; Cranor et al. 2002]. ATLaS [Wang and Zaniolo 2003] and ESL [Luo et al. 2005] adopt from SQL-3 the idea of specifying a new UDA by an INITIALIZE, an ITERATE, and a TERMINATE computation; however, both ATLaS and ESL let users express these three computations by a single procedure written in SQL—rather than by three procedures coded in procedural languages as prescribed by SQL-3[4]. Example 5.1 defines an aggregate equivalent to the standard AVG aggregate in SQL. The second line in Example 5.1 declares a local table, **state**, where the sum and count of the values processed so far are kept. Furthermore, while in this particular example, **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples. These SQL statements are grouped into the three blocks labeled, respectively, INITIALIZE, ITERATE, and TERMINATE. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1. The ITERATE statement updates the tuple in **state** by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the ratio between the sum and the count as the final result of the computation by the INSERT INTO RETURN statement[5]. Thus, the TERMINATE statements are processed just after all the input tuples have been exhausted.

Example 5.1. Defining the standard AVG

```
AGGREGATE myavg(Next Int) : Real
{    TABLE state(tsum Int, cnt Int);
     INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
     }
     ITERATE : {
        UPDATE state
           SET tsum=tsum+Next, cnt=cnt+1;
     }
     TERMINATE : {
        INSERT INTO RETURN
           SELECT tsum/cnt FROM state;
     }
}
```

Observe that the SQL statements in the INITIALIZE, ITERATE, and TERMINATE blocks play the same role as the external functions in SQL-3 aggregates. But here, we have assembled the three functions under one procedure, thus supporting the declaration of their shared tables (the **state** table in this example). This table is allocated just before the INITIALIZE statement is executed and deallocated just after the TERMINATE statement is completed. This approach to aggregate definition is very general. For instance, say that we want to support tumbling windows of 200 tuples [Carney et al. 2002].

[4]Although UDAs have been left out of SQL:1999 specifications, they were part of early SQL-3 proposals, and supported by some commercial DBMS.
[5]To conform to SQL syntax, RETURN is treated as a virtual table; however, it is not a stored table and cannot be used in any other role.

Then we can write the UDA of Example 5.2, where the RETURN statements appear in ITERATE instead of TERMINATE. The UDA **tumble_avg**, so obtained, takes a stream of values as input and returns a stream of values as output (one every 200 tuples). While each execution of the RETURN statement produces here only one tuple, in general, the UDA can return several tuples. Also observe that UDAs are allowed to declare local tables and apply arbitrary select and update actions on these tables, including the use of built-in and user-defined aggregates (possibly in a recursive fashion) [atl ; Luo et al. 2005]. Thus UDAs operate as general stream transformers. Observe that the UDA in Example 5.1 is blocking, while that of Example 5.2 is nonblocking. Thus, nonblocking UDAs are easily and clearly identified by the fact that *their* TERMINATE *clauses are either empty or absent*. The typical default implementation for SQL aggregates is that the data are first sorted according to the GROUP-BY attributes: thus the very first operation in the computation is a blocking operation. Instead, ESL uses a (nonblocking) hash-based implementation for the GROUP-BY (or PARTITION-BY) calls of the UDAs [Luo et al. 2005]. The semantics of UDAs therefore is based on sequential execution whereby the input sequence or stream is pipelined through the operations specified in the INITIALIZE and ITERATE clauses: the only blocking operations (if any) are those specified in TERMINATE, and these only take place at the end of the computation.

Example 5.2. AVG on a Tumble of 200 Tuples

```
AGGREGATE tumble_avg(Next Int) : Real
{    TABLE state(tsum Int, cnt Int);
     INITIALIZE : {
        INSERT INTO state VALUES (Next, 1)}
     ITERATE: {
        UPDATE state
           SET tsum=tsum+Next, cnt=cnt+1;
        INSERT INTO RETURN
           SELECT tsum/cnt FROM state
           WHERE cnt
        UPDATE state SET tsum=0, cnt=0
           WHERE cnt
     }
     TERMINATE : { }
}
```

UDAs can be called and used in the same way as any other built-in aggregate. For instance, say that we are given a stored sequence (or an incoming stream) of purchase actions:

webevents(CustomerID, Event, Amount, Time)

Since UDAs process tuples one-at-a-time (in a fashion similar to that of the cursors used to interface programming languages with SQL) they dovetail with the model of physically-ordered sequences, and can express well the search for patterns in such sequences. For instance, we want to find the situations where users, immediately after placing an order, ask for a rebate and then cancel the order. Finding this pattern in SQL requires two self-joins to be computed on the incoming stream of webevents. In general, recognizing a pattern of n events requires $n - 1$ joins, and queries involving many stream joins can be difficult to express in SQL, and can also be very inefficient to execute in a DSMS. Also, the notion that a tuple t_2 must immediately follow a given tuple t_1, without any tuple in between, is logically quite simple, but it requires a rather complex formulation in standard SQL. UDAs can be used to solve these problems. For instance, say that we want to detect the pattern of (i) an order, followed by (ii) a rebate, and then (iii) a cancellation of the original order. Then, the nonblocking UDA of Example 5.3 can be used to return the string 'pattern123' with the CustomerID whose events have just matched the pattern (the aggregate will be called with the group-by clause on CustomerID). This UDA models a finite state machine, where 0 denotes the failure state, which is set whenever the right combination of current-state and input is not observed. Otherwise, the state is first set to 1, and then advanced till 3, where 'pattern123' is returned, and the search for the pattern continues on the rest of the sequence.

Example 5.3. First the order, then the rebate and finally the cancellation

```
AGGREGATE pattern(Next Char) : Char
{    TABLE state(sno Int);
       INITIALIZE : {
       INSERT INTO state VALUES(0);
       UPDATE state SET sno = 1
          WHEN Next='order';}
     ITERATE: {
       UPDATE state SET sno = 0
          WHERE NOT(sno = 1 AND
                    Next = 'rebate')
          AND NOT(sno = 2 AND Next = 'cancel')
          AND Next <> 'order'
       UPDATE state SET sno = 1
          WHERE Next='order';
       UPDATE state SET sno = sno+1
          WHERE (sno = 1 AND Next = 'rebate')
             OR(sno = 2 AND Next = 'cancel')
       INSERT INTO RETURN
          SELECT 'pattern123' FROM state
          WHERE sno = 3;
     }
}
```

UDAs can also be used effectively to handle punctuated data streams which have been proposed in [P.Tucker et al. 2003] to overcome blocking problems. In addition to performing their normal computations, the UDAs on punctuated data stream must recognize the punctuation marks, and produce their results when these arrive. For instance, in Example 5.4, we want to output the average stock value of each company, when we receive its closing value tuple denoted by a special punctuation mark indicating that no more tuple of this company will arrive. In this case, a value of 1 in the **close** column serves as the special punctuation mark; once this value is detected, we return the average for this company. We use the table **state** to store the summary (sum and count) of each company.

Example 5.4. The average price for each company is returned when its closing price is received.

```
AGGREGATE CoSum(cid Int, price Real, close Int) : Real
{    TABLE state(tcid Int, tsum Int, tcnt Int);
     INITIALIZE : {
       INSERT INTO state VALUES (cid, price, 1);}
     ITERATE: {
       UPDATE state
          SET tsum=tsum+price, tcnt=tcnt+1;
          WHERE tcid=cid;
       INSERT INTO state
          SELECT cid, price, 1 FROM state
          WHERE cid NOT IN (
                  SELECT tcid FROM state);
         INSERT INTO RETURN
            SELECT tsum/tcnt FROM state
            WHERE tcid=cid AND close=1;
      }
      TERMINATE : { }
}
```

A native extension of SQL, called SQL-TS, was proposed in [Sadri et al. 2001a], to find patterns in sequences and data streams. While SQL-TS represents a powerful special-purpose language for querying sequential patterns, we found that the same queries can also be supported efficiently using UDAs. In fact both SQL-TS and its powerful extensions called K*SQL [Mozafari et al. 2010a] have been implemented efficiently in our DSMS [Thakkar et al. 2011] by mapping them into UDAs.

Indeed, UDAs provide a powerful vehicle for state-based reasoning, and pattern-searching. In general, concepts and queries that can be expressed using an FSM can be easily and automatically translated into UDAs. In fact, in Appendix A we prove that the following property holds:

PROPOSITION 5.5. *Assume query Q searches for a pattern defined by a FSM M. Let n be the number of links in M. Then there exists a UDA with $n + 2$ statements which can implement Q.*

The strong synergy between UDAs and finite-state automata is confirmed by the fact that UDAs are able to express the ultimate state machine: the Turing machine.

Observe that most of the UDAs in our examples have an empty terminate state. Since the terminate state is the only one that respond to the end of the input we have the following property:

PROPOSITION 5.6. $\mathcal{NB}$ **UDAs:** *UDAs* TERMINATE *state is empty or missing are non-blocking.*

UDAs represent a very powerful computational device and combined with union they deliver Turing completeness; likewise, $\mathcal{NB}$-UDAs combined with union will deliver $\mathcal{NB}$-completeness under the assumptions described in the coming sections.

6. COMPLETENESS ON RELATIONS & DATA STREAMS

The power of a query language is defined as the class of functions it can express on (an input tape encoding) the database [Abiteboul et al. 1995]. Achieving a high level of expressive power is difficult for database query languages, and attempts to characterize and improve their expressive power have been the focus of much topical research [Abiteboul et al. 1995]. Furthermore, in Section 4, we investigated the power of RA and SQL in expressing continuous queries on sets of tuples, and found that this is further impaired by the fact that (i) only monotonic functions can be expressed (to avoid blocking operators) and (ii) RA and SQL are not $\mathcal{NB}$-complete. In the rest of the paper, we focus on improving this dismal state of affairs. In this section, we show that the UDAs introduced in the previous section change the situation completely, inasmuch as their addition makes SQL (a) Turing complete, i.e., capable of expressing all computable set functions, and (b) $\mathcal{NB}$-complete, i.e., capable of computing all monotonic set functions using only non-blocking constructs. Then in Section 8, we extend these results to data streams explicitly ordered by their timestamps.

The proof that UDAs make SQL Turing-complete is given in Appendix B where we show that UDAs can express any function (and therefore any set function) by encoding one or more tables into an input tape. However such an encoding is a blocking computation which is not allowed for continuous queries on data streams. Therefore, in order to achieve $\mathcal{NB}$-completeness, i.e., the ability of expressing every monotonic set function, we proceed in two steps A, and B as follows:

A: We combine multiple streams into one stream using the set union operator. Before the union is actually computed we (i) make the various data streams union-compatible by adding columns filled with suitable values to the various streams, and (ii) add an additional column to remember from which data stream each tuple came from (both these operations can be performed by a monotonic UDA). Our given monotonic set function on multiple streams can now be re-expressed as a monotonic set function F on the union stream generated by this step.

B: Our monotonic set function F on a single data stream can be computed by a UDA that uses three local tables, called IN, $TAPE$, and OUT, and performs the following operations for each new arriving tuple:

   Delta Computation:
   (a) Append the encoded new tuple to IN,
   (b) Copy IN to $TAPE$, then
   compute $F(IN) - OUT$ as described in Section 5[6].
   (c) Return the result obtained in (b) and append it to OUT.

Since these operations are executed on each arriving new tuple, they are performed in the ITERATE state of the UDA, which is therefore nonblocking. Since both the UDAs used in [A] and [B] are non-blocking and set union is monotonic, we can conclude that non-blocking UDAs and set union are $\mathcal{NB}$-complete for set functions, insofar as:

PROPOSITION 6.1. *A query language that supports non-blocking UDAs and set union can express all monotonic set functions on data streams.*

In summary, after observing the devastating effect that the nonblocking assumption has on relational query languages, we moved to boost their expressive power by UDAs, and following [Law et al. 2004], showed that (i) SQL with UDAs is Turing-complete (i.e., can express all computable functions) and (ii) also $\mathcal{NB}$-complete (i.e., it can compute all monotonic functions). Now, observe that (i) is practically significant, since it provides a solution to the expressive power problem that has long eluded database researchers. However, (ii) is only of modest practical interest since unordered set of tuples and query languages that only express set functions are not sufficient for data streams applications where temporal ordering plays a pivotal role. Therefore, in the next section we formalize the notion of timestamp-ordered sequences and then endeavor at extending (ii) to this framework.

7. TIME-STAMPED DATA STREAMS

We consider explicitly timestamped sequences and data streams, where tuples are ordered by the increasing values of their timestamps. Thus, two sequences are considered equivalent if they are obtained from each other by commuting tuples having the same timestamps. However, if we commute two tuples with different timestamps we obtain two different sequences. Therefore, let us reiterate that the data model under which we are now operating, is one where (i) the order of tuples sharing the same timestamp is immaterial (commutativity for contemporary tuples) but (ii) if $t_1 < t_2$,

[6]Since F is monotonic, its corresponding UDA is nonblocking, and thus the delta computation is performed in its ITERATE state where the difference $F(IN) - OUT$ is actually computed and its result is returned. For specific aggregates, rather than computing the difference of the materialized $F(IN)$ and OUT, we can derive and use in ITERATE the symbolic delta-expression of this difference. Examples 5.2–5.4 illustrate this more efficient alternative.

every tuple with timestamp t_1 precedes every tuple with timestamp t_2. We will now define a partial ordering between timestamped sequences and investigate monotonic functions with respect to that order.

Let S be a timestamped sequence, and τ be an arbitrary timestamp. The τ-presequence of S (or more precisely, the pre-τ presequence of S) is obtained from S by eliminating all tuples with timestamp following τ:

Definition 7.1. *τ-presequence:* Let S and R be two sequences ordered by their timestamp. R^τ is defined as the set of tuples of R with timestamp less than or equal to $\tau > 0$. If $S = R^\tau$ for some τ, then S is said to be a τ-presequence of R, denoted $S \sqsubseteq^t R$. In general, let $S_1, ..., S_n$ and $R_1, ..., R_n$ be timestamped sequences. $(S_1, ..., S_n) \sqsubseteq^t (R_1, ..., R_n)$ when $(S_1, ..., S_n) = (R_1^\tau, ..., R_n^\tau)$ for some τ.

Thus, two time-stamped sequences are considered equal if they are τ-presequences of each other.

Valid query operators on timestamped sequences take as their input(s) and generate as their output sequences ordered by their timestamps. We are only interested in query operators and functions that accept as input(s) and produce as outputs sequences by increasing timestamp order. Examples of these include relational selection and projection (assuming that the later does not eliminate the timestamp column). Also, all operators must compute valid functions on timestamped sequences: i.e., functions which produce the same result irrespective of the order in which contemporary tuples are arranged. Thus, for instance, if the input is the pair `(timestamp, value)`, a function that returns the sum of `value` for each `timestamp` is a valid function[7]. However, the function that returns a continuous sum for the values having the same timestamp would return different sequences for different orders of contemporary tuples and thus does not define a valid function, even if the input and output are properly ordered[8].

Order is also essential when working with timestamped sequences. For instance take an operator that returns all the elements in the input sequence except for those with maximal value. Let us first consider an operator $\texttt{allbutmax}_1$, which returns the values as soon as it can conclude that this cannot be a maximal element: thus on input $[(t_1, 6), (t_2, 5), (t_3, 4), (t_4, 8)]$, $allbutmax_1$ (will return $(t_1, 6)$ as soon as it sees $(t_4, 8)$, thus producing $[(t_2, 5), (t_3, 4), (t_1, 6)]$. If we assume that $t_1 < t_2 < t_3 < t_4$, the sequence so produced violates the required timestamp order. Thus $\texttt{allbutmax}_1$ is neither a valid operator, nor it defines a valid function on timestamped sequences. Of course, we could use instead a blocking operator $\texttt{allbutmax}_2$ that presented with the same sequence delays the output (possibly till the end of the sequence) and then returns $[(t_1, 6), (t_2, 5), (t_3, 4)]$. Now, $\texttt{allbutmax}_2$ is a valid operator on timestamped sequences, defining a valid function on the same. Unfortunately, $allbutmax_2$ is blocking and the corresponding function is nonmonotonic with respect to the τ-presequence ordering. However, this function is obviously monotonic on sets, and in that domain, $allbutmax_1$and $allbutmax_2$, provide two alternative implementations: the first is nonblocking and the second is blocking.

The notion of monotonicity can be defined quite naturally for τ-presequences:

Definition 7.2. A unary operator G is monotonic if $L \sqsubseteq^t S$ implies $G(L) \sqsubseteq^t G(S)$. A binary operator H is monotonic when $(L_1, L_2) \sqsubseteq^t (S_1, S_2)$ implies $H(L_1, L_2) \sqsubseteq^t H(S_1, S_2)$.

[7]In SQL, this set function can be expressed as:
```
select timestamp, sum(value) from myinput groupby timestamp.
```
[8]In SQL:2003, this OLAP function can be expressed as:
```
select timestamp, sum(value)over(partition by timestamp) from myinput.
```

In operational terms, $S \sqsubseteq^t R$ can be viewed as a statement that R was obtained from $S = R^\tau$ by appending some additional tuples with timestamps larger than those in S: for instance, S might be the stream received up to time τ, and R the stream received after waiting a little longer i.e., up to time $\tau' > \tau$.

Then, the notion of nonblocking operators on logical sequences will be defined as follows ($G^\tau(S)$ and $H^\tau(L, S)$ denote the τ-presequence of $G(S)$ and $H(L, S)$, respectively):

Definition 7.3. Nonblocking Query Operators:

—A nonnull unary operator G is said to be nonblocking, when $G^\tau(S) = G(S^\tau)$, for every τ.
—A nonnull binary operator H is said to be nonblocking, when, $H^\tau(L, S) = H(L^\tau, S^\tau)$, for every τ.

Then we have the following proposition that can be easily proved along the lines of Propositions 3.9 and 6.1

PROPOSITION 7.4. *Functions on sequences ordered by their timestamps can be implemented by nonblocking operators iff they are monotonic w.r.t.* $\sqsubseteq^t$.

For instance, selection and projection are clearly monotonic single-input functions and implementable by nonblocking operators. Now UDAs that express monotonic functions on timestamped sequences have the following simple characteristics: (i) their TERMINATE state is either missing or return no value, and (ii) they only produce tuples whose timestamped value is equal or greater than that of the previous one[9]

Thus, there is no serious issue with unary operators, and we can use the argument used in the previous section to show that UDA can express all functions that are monotonic to respect to the τ-presequence ordering. Unfortunately, we will see next that the situation for binary operators is more difficult.

For binary functions, we can define the standard set-based intersection operator that takes the set of tuples that are present in both streams, ordering them by increasing timestamps. This is a function that is obviously monotonic w.r.t. τ-preordering, and thus trivially implementable by nonblocking operators. Unfortunately the standard set union which lists the tuples in either stream ordered by their timestamps is neither monotonic w.r.t. τ-preordering nor it can be implemented by a non-blocking operator! Indeed, consider the following two sequences of $(time, value)$ pairs:

$S = [(1, a), (4, b)]$ and

$L = [(2, c)]$. The set-union of these two sequences is

$W = [(1, a), (2, c), (4, b)]$.

But if

$S = [(1, a), (4, b)]$ and

$L' = [(2, c), (3, d)]$. The set-union of the two becomes

$W' = [(1, a), (2, c), (3, d), (4, b)]$.

Now, W is not a τ-presequence of W' whereas L is a τ-presequence of L'.

Therefore, union is nonmonotonic and cannot be used. But without union we cannot express computations on multiple streams, and other binary operators, such as

[9]For instance, this condition holds if the timestamp value of each output tuple is taken from the current input tuple, or from a clock.

joins, suffer from similar problems. In order to achieve, or at least approach, $\mathcal{N}B$-completeness on multiple streams we will next introduce a *monotonic approximation*.

7.1. Monotonic Approximation

. The generalizations of functions of relational algebra for timestamped data streams is trivial for selection and projection, and very difficult for union, cartesian product, joins and other binary operators (from these, the case of operators with several arguments can be easily derived). In fact, these operators are no longer monotonic, and thus we will not be able to perform step A: in Section 6 to merge streams if we use the standard union.

To address this problem we will introduce monotonic approximations of union and cartesian product that we will call τ-union. (Similar generalizations can be defined for Cartesian products and joins, as we will discuss later.) Therefore, let S and L be timestamped sequences, and $\omega(L)$, $\omega(S)$ denote the largest timestamp in L and S, respectively then:

Definition 7.5. Let L and S be timestamped sequences and $t = \min(\omega(L), \omega(S))$. Then, the τ-Union of L and S, denoted $\tau\text{-}union(L, S)$, is defined as follows:
$\tau\text{-}union(L, S) = L^t \cup\ S^t$.

Thus, τ-union is computed by only considering the two sequences or streams only up to the min of the last timestamps in the two streams. Thus, for instance consider sequences of pairs, such as $(2, c)$ where 2 is the pair timestamp in seconds and c is the actual value. Then:

$$\begin{aligned} \tau\text{-}union([(1,a)],\ [\ ]) &= [\ ]; \\ \tau\text{-}union([(1,a)],\ [(2,c)]) &= [(1,a)] \end{aligned}$$

Observe that while the result of $\tau\text{-}union$ and set union are normally different, the output of $\tau\text{-}union$ tends to catch-up with the result of set union after the arrival of additional tuples. Thus, the result of the set union of $[(1, a)]$ and $[\,]$ is actually produced by $\tau\text{-}union$ after $(2, c)$ is added to its second input—i.e., the same result is produced by τ-union one second later than the set union. Likewise, the set-union result of $[(1, a)]$ and $[(2, c)]$ is only returned by τ-union after a new tuple with timestamp greater or equal to 2 is added to the first input—i.e., after some non-negative delay. Thus, τ-union is a *delayed approximation* of set union. We will now formalize this behavior by the notion of *delayed approximation* for both binary and unary functions (where an example of the latter is given below):

Definition 7.6. Let F and G be two valid unary [resp. binary] functions on timestamped sequences. F is said to be an approximation of G with delay D for input S [resp. inputs S and W] when the following two conditions hold:

(1) If t is a timestamp of some tuple in $F(S)$, then $F^t(S) = G^t(S)$ [resp. If t is a timestamp of some tuple in $F(S, W)$, then $F^t(S, W) = G^t(S, W)$].
(2) For every $t \geq \omega(S) - D$, there exists a $\delta \leq D$ such that $F(S^{t+\delta}) = G(S^t)$ [resp. For every timestamp $t \geq\ \max(\omega(S), \omega(W)) - D$, there exists a $\delta \leq D$ such that $F(S^{t+\delta}, W^{t+\delta}) = G(S^t, W^t)$].

The first condition is basically a correctness condition that states that the sequence of tuples produced by F up to and including the tuples with timestamp t is the same sequence as that produced by G up to and including the tuples with timestamp t. The second condition instead specifies that, once we see certain tuples returned by G, we will only have to wait a time $\delta \leq D$ to see the same tuples returned by F, but this property only holds until we reach a time that precedes by D the end of the input stream(s). Let us now illustrate these definitions with examples. An interesting example for unary functions is provided by the computation of window with panes (a.k.a., slides) aggregates. Consider for instance the function `count@3` that returns the cumulative count of `value` every 3 seconds (i.e., we use a slide construct [Carney et al. 2002; Thakkar et al. 2011] to modify the behavior of the count aggregate, which now returns results every 3-second slide, instead of every incoming tuple as in basic window aggregates). Therefore results will only be produced after 3 seconds, 6 seconds, 9 seconds, etc. For instance consider the following sequence:

$$S = [(1,a),(2,d),(4,b),(4,c),(5,c),(8,e)]$$

On this sequence, our `count@3` aggregate should return the following result:

$$\mathit{count}@3(S) = [(3,2),(6,5),(9,6)]$$

However, `count@3` is nonmonotonic (e.g., if we add a pair $(8,f)$, then the last pair in $\mathit{count}@3(S)$ above should be $(9,7)$ instead of $(9,6)$). Thus, $\mathit{count}@3(S)$ cannot be supported in real time, and will have to settle for its monotonic approximation $\mathit{mcount}@3(S)$, described next. Our $\mathit{mcount}@3(S)$ cannot produce any result until it has seen the third tuple with timestamp 4. Only at that point, we can conclude that all the tuples in the first 3 second slide have been counted, and return the result for that slide: thus $\mathit{mcount}@3(S^4) = [(3,2)] = \mathit{count}@3(S^2)$.

Now, for S^5, $\mathit{count}@3(S^5)$ returns a total count of 5 (this is the running count at the completion of the second slide), but $\mathit{mcount}@3$ cannot produce this result until if has seen the next tuple that has timestamp 8. Thus in our case $\mathit{mcount}@3(S^8) = [(3,2),(6,5)] = \mathit{count}@3(S^5)$. Moreover, since $\mathit{mcount}@3(S^8) = \mathit{count}@3(S^5)$, the last three seconds of the input are simply ignored by $\mathit{mcount}@3$, and the final pair $(6,5)$ is never returned. Thus, for this example $D = 3$ using the notation of Definition 7.6.

The problem is even more serious for binary operators, such as τ-union. For instance, say that the tuples in the two streams keep flowing in every d chronons. While this is the case, τ-$\mathit{union}(S,W)$ will catch up with $S \cup W$ with a delay of d. But then say, that S terminates before W does. In this case all the tuples in W with timestamps falling in the interval $\omega(W) - \omega(S)$ will never appear in τ-$\mathit{union}(S,W)$ although they rightfully belong to $S \cup W$. Thus according to Definition 7.6, the worst case delay is $d + \omega(W) - \omega(S)$.

Thus, τ-union can turn into a very coarse approximation of union when infrequent arrivals in one streams delay the output of the tuples from the other stream, and when the arrivals stop in one input (e.g., because the source of the data or the transmission line fails). Similar problems occur in other binary operators, such as the joins of different data streams, which are not discussed here because, although very useful, they are not needed in terms of expressiveness.

Although it was not treated in the formal framework of monotonicity and expressiveness we address here, the practical problem that the computation must wait idly for the arrival of more tuples before it can predict accurately the next output tuple has been recognized in the past, and simple but effective solutions to this problem have been proposed in [Johnson et al. 2005; Bai et al. 2008] using punctuation tuples containing only timestamp information. In the simplest version, timestamps are injected into the streams at a regular time intervals [Johnson et al. 2005]. Punctuation tuples can also be generated on demand as described in [Bai et al. 2008]. In this case, when computing the union on data stream A and B, when only the tuple on A is available the DSMS takes care of generating a punctuation tuple for B with timestamp value greater or equal than that of A (and symmetrically when B has a tuple and A has none). This will not occur in an instantaneous manner in typical DSMS implementations using chains of cascading operators and buffers [Cranor et al. 2002; Madden et al. 2002; Johnson et al. 2005; Bai et al. 2008]. In fact, for internally timestamped data streams, there will be a small delay to check the upstream buffers and, if these are all empty to consult the system clock. An even more significant delay might be needed for externally timestamped data streams. However, we will move past these delays and define the notion of *perfectly harmonized* data streams. We say that data streams A and B are perfectly harmonized when for each tuple in A there exists a tuple in B with greater or equal timestamp, and vice-versa. Observe that, since we work with unbounded but finite data streams, the two tuples ending two perfectly harmonized data streams have identical timestamps (e.g., the end-of-time timestamp often used for 'now' in temporal databases, is quite suitable in this role)[10].

Our interest in perfectly harmonized data streams follows from the fact that τ-union and set union behave identically on these data streams, thus we can apply the reasoning used on sets to conclude that:

PROPOSITION 7.7. *A query language that supports non-blocking UDAs and τ-union can express all monotonic set functions on perfectly harmonized data streams.*

This important theoretical result about the $\mathcal{NB}$-completeness of our τ-union + UDAs framework, is however limited with the realization that this holds only for the idealized situation. In reality, some delay will be needed for the system to punctate data streams to perfectly harmonize them, i.e., to assure that when there is a data tuple in one stream there always is a tuple on the other stream with greater or equal timestamp. In practice moreover, we have the more common situation where the continuous query function F, that would deliver the intended result on perfectly harmonized data stream, is instead applied to the two data streams that are affected by a skew of δ. Then, the current τ-union result approximates the correct result with a delay of δ. Moreover, since the UDA introduces no further delay, we can conclude that this delayed approximation also holds for the final query result.

This conclusion also holds for cartesian products and joins, for which monotonic τ-approximation versions have been proposed and discussed in the literature, along with punctuation techniques to minimize their idled-waiting [Johnson et al. 2005; Bai et al. 2008]. In terms of expressive power, which is the focus of this paper, we see that products and joins are not necessary once we have UDAs and unions. In fact, joins can be computed by first computing the τ-union of the two streams, and then feeding it to a UDA that computes the join.

8. MORE COMPLEX DATA STREAM MODELS

Windows, synopses, concrete views and related concepts have been proposed to supplement the basic append-only-table model of data streams. In this section, we show that the operators previously presented can be extended naturally to realize these approaches, thus achieving a unified treatment and understanding.

Synopses. The past history of a data stream quickly grows in time to sizes that exceed what can be effectively stored and searched; therefore, synopses are used to summarize recent data and derive approximate answers for queries. For example, the join of a stream A with stream B can be approximated as the join of $W(A)$ with $W(B)$, where $W(A)$ and $W(B)$ respectively denote a window on A and a window on B. Then, the window join of A and B is computed by matching every new tuple arriving in W(A) with the tuples in W(B) and viceversa. Many variations of this basic scheme have been

[10]For periodically punctuated data streams the perfect model can be also be realized when the time granularity is coarse enough that punctuation marks can be generated every chronon.

studied, and various window joins algorithms have been proposed to support their efficient implementation and minimize the number of output tuples that were dropped because only the synopsis is used in lieu of the complete data stream [Kang et al. 2003; Golab and Özsu 2003]. Window join operators are highly desirable builtin operators in DSMS, because of performance considerations. However, they are not indispensable in terms of expressive power since they can be computed by (i) making A and B union-compatible with the addition of new columns, (ii) taking the union of the two stream so derived, and (iii) computing their join using a UDA that stores the two windows $W(A)$ and $W(B)$ and computes their result as described above. In special situation where application-specific constraints can be used to customize the UDA used in (iii), this approach might also yield superior performance.

Concrete Views and Expiring Tuples. Concrete views can be used to create windows on data streams [Jagadish et al. 1995], and often call for explicit events to occur when tuples leave the windows (whereas, as discussed above, no explicit action is required when tuples leave the windows used to compute joins). A technique shown to be effective in many applications [Golab and Özsu 2005] consists in modelling tuples leaving a window by means of 'negative tuples' (whereas those arriving in the window are modelled as positive tuples). For instance, let us consider a situation where the user looking at the screen of a client workstation should see a list of transportation stocks whose trading during the last hour exceeded a certain level. Then, the DSMS must perform the following operations continuously: (i) add up the activities on each stock for the last hour (using a window aggregate that is further discussed below), (ii) send a positive tuple for each new stock in the list, and (iii) send a negative tuple for each stock that must be removed from the list because its insufficient trading during the last hour. The positive tuples and negative tuples could be in two different data streams or could be combined into a single stream by the addition tags to identify whether the tuple is positive or negative. The use of negative tuples in implementing relational operators, including nonmonotonic operators on concrete views is discussed in [Golab and Özsu 2005]: all these operators are easily expressed using UDAs. Therefore, we can retain the basic data model of append-only relations, and use the power of the UDAs to deal with the positive/negative tags, which now provide an application-specific representation rather than a model enhancement. In fact, in many applications it is highly desirable to go beyond this simple positive/negative tuple scheme. For instance, to show the current value of those stocks in the concrete view, we could use an update stream. Then, we might use one data stream, where each tuple is tagged as either append, remove, and replace, or use three separate streams as in CQL [Arasu et al. 2003]. The best choice between these alternatives is bound to be application-dependent. Another common situation is when concrete views have unique keys. In this situation we only need positive tuples since these can be assigned the following meaning: insert this new value, and eliminate the old tuple with the same key value if one was present. To signify that a tuple with given key must be eliminated, rather than replaced, we can use null values, or values that are not allowed (e.g., negative prices for stocks). Yet another possible solution, that offers unique advantages in certain applications, is that of time stamping tuples with their validity period [Krämer and Seeger 2005]—this is the 'direct' representation studied in [Golab and Özsu 2005]. In summary, there are many ways to represent deleted and updated tuples, and an interesting spectrum of data stream models and extended relational algebra operators have been defined for that [Arasu et al. 2003; Krämer and Seeger 2005; Golab and Özsu 2005]. On the other hand, each of these extensions adds complexity to the basic data model and it is desirable in some applications but not in others. In this paper, we have instead proposed an approach where different representations of expiring or updated tuples are supported very effectively via application-specific UDAs—thus retaining the simplicity of append-only relations as the basic data stream model.

Windows and Aggregates. Aggregates on logical or physical windows are invaluable in data stream applications since they turn blocking aggregates into nonblocking ones that return results continuously. A logical window is specified as the set of stream tuples that have arrived since time $T - \tau$, where T is the current time and τ is the size of the window. A physical window instead contains the last n tuples, where n is the size of the window. SQL:2003 window aggregates (e.g., its OLAP functions) return results only when a new tuple arrives . This is the only semantics of interest for physical windows where tuples expire only when a new tuple arrives. But for logical windows, there have been proposals [et al. 2004; Golab and Özsu 2005] that advocate returning the value for the aggregate both when a new tuple arrives (α-event) and when an old one expires (ω-event). We will refer to this approach where both the α and the ω events are recognized as $\alpha + \omega$ semantics.

Both these semantics can be expressed using UDAs. The realization of the α semantics requires a UDA that memorizes the tuples in the window. Then, when a new tuple arrives, the expired tuples are first removed from the window, then the new tuple is added, and finally the aggregate is recomputed on the updated window, and its value returned to the output. Performance-oriented improvements to this basic scheme have been proposed for simple aggregates [Arasu and Widom 2004; Li et al. 2005], and the case of general UDAs was discussed in [Thakkar et al. 2011]. Data structures similar to SteMs [Madden et al. 2002] are effective in maintaining the state of windows. These performance-oriented improvements are quite beneficial but fall outside the scope of this paper that concentrates on semantic issues. Therefore, we will only discuss the $\alpha + \omega$ semantics since this has been considered a more desirable alternative than the α-only semantics inherited from SQL:2003 [Arasu and Widom 2004; Li et al. 2005]. We will next show that the $\alpha + \omega$ semantics can also be achieved quite naturally in our framework. In fact, say that we have a data stream S of time-stamped tuples, and a logical windows of size τ on such stream, and an aggregate $\mathbf{A}$ (built-in or user defined). Then the computation of $\mathbf{A}$ on τ under the $\alpha + \omega$ semantics can be computed using the algorithm shown below, where we use the first three steps to generate the positive/negative tuples corresponding to the logical window, and the last step computes the actual aggregate:

Supporting the $\alpha + \omega$ Semantics:

(1) Let Sp (resp. Sn) be the stream obtained from S by adding a positive (resp. negative) tag to each tuple in S.
(2) For each tuple in Sn, update its timestamp by adding the size of the window—i.e., the time interval τ. Let us denote the data stream so obtained by $S'n$.
(3) Let S' denote the union $Sp \cup S'n$.
(4) Process S' using an UDA that operates as follows: if the new tuple in S' is positive, then (i) add it to the list of memorized tuples, (ii) recompute the aggregate $\mathbf{A}$ on such list, and (iii) return the result. Likewise, if the new tuple in S' is negative, then (i) delete it from the list of memorized tuples, (ii) recompute the aggregate $\mathbf{A}$ on the resulting list, and (iii) return the result.

Therefore, we use the union operator to sort-merge the positive tuples and the negative tuples into the correct order, thus implementing the $\alpha + \omega$ semantics for logical windows. This implementation completely captures the declarative semantics of our query operators viewed as mappings from the input streams to the output streams. But it also reveals that basic quality-of-service issues of $\alpha + \omega$ windows can be reduced to those of union, and are thus amenable to similar solution. The practical usefulness of $\alpha + \omega$ windows depends on the promptness with which ω tuples are generated once the tuples have logically expired. Here we have discovered that prompt responses in $\alpha + \omega$ windows can be achieved by eliminating idle-waits in the execution of union operators, which is the problem discussed in Section 7.1. Idle-waits can compromise the promptness of response upon negative tuples produced by expirations. Indeed, a negative tuple with time $T + \tau$ might wait indefinitely until a tuple with time stamp value $\geq T + \tau$ appears in Sp, incurring into an idle wait problem. Heartbeat techniques have been shown to be effective to reduce the idle wait problem for union and related operators, and can be used for the case at hand [Johnson et al. 2005]. Idle-waits are not a problem for the positive tuples in Sp, since for each tuple with timestamp T there is a tuple with timestamp $T + \tau$ in $S'n$.

Impact of Query Languages upon Stream Data Models. Therefore, by extending our query language into a very powerful one, we can retain the 'Spartan simplicity' of the basic data model (i.e., append only tables), and yet support in a flexible and general way the vast assortment of extensions that have been advocated for data streams, each of which is most beneficial in different applications. Our approach is conducive to a a unified treatment, and unifies disparate semantic issues. For instance, we showed that new semantics proposed for aggregates such as the $\alpha + \omega$ semantics can be reduced to the α-only semantics of SQL:2003, and related issues of responsiveness can be addressed using the heartbeat techniques used to minimize idle waits in the implementation of union operators [Srivastava and Widom 2004].

Greater expressiveness for continuous query languages has been the main focus of this paper, which showed that this goal can be achieved using UDAs defined in SQL. Ease of use and query optimizability represent other important goals in the design of effective query languages. Thus, inasmuch as joins lead to simpler queries more amenable to optimization, they should be supported in a practical query language, although they are not strictly needed in terms of expressive power. Likewise, the UDA extensions proposed in [Bai et al. 2006], are not essential in terms of expressive power, but greatly simplify the specification of window aggregates and the efficient computation of the aggregate values on windows and panes [Li et al. 2005]. In a similar vein, several pattern languages have been recently proposed using Kleene-closure constructs. Languages, such as SQL-TS [Sadri et al. 2004], SASE+ [Wu et al. 2006; Gyllstrom et al. 2008], the newly proposed SQL standards for match-recognize [Zemke et al. 2007], and finally K*SQL [Mozafari et al. 2010c; 2010b], illustrate the desirability of these extensions because of the great usability and specialized optimization techniques [Mozafari et al. 2010b] they entail. These exciting developments do not diminish the practical and theoretical significance of the results presented in this paper. On the practical side, it should be noted that the implementations of both SQL-TS and K*SQL were based on UDAs [Mozafari et al. 2010c; 2010b]. On the theoretical side, we observe that these extensions have yet to explore the blocking and monotonic approximation issues that limit their expressive power—particularly for languages such as SASE+ where patterns are specified over multiple data streams and thus rely on implicit union operators.

9. CONCLUSIONS

There is much recent interest on how relational query languages and data models can be extended to support effectively continuous queries on data streams. This approach is promising insofar as it allows DSMS researchers and designers to build on theory and experience developed with relational query language; also for users this can simplify programming for the very many applications that span both data streams and database tables. However, the data stream computing environment is so different from the database one, that extending to the latter theories and constructs developed for the former requires an in-depth analysis and critical evaluation of the interrelated issues that arise at (i) the data model level due to the need to support order in data streams, and (ii) the query language level where the exclusion of blocking query operators compromises expressive power. There is a strong relationship between the two, since e.g., a nonblocking operator in the standard relational model might no longer be so in an ordered data model. This paper is the first to elucidate this important difference and how it impacts key operators such as union.

We began by extending the definition of blocking operators from [Babcock et al. 2002] to binary operators, and showing that functions can be realized by nonblocking query operators iff they are monotonic. This allows us to introduce the notion of $\mathcal{N}B$-completeness and use it as a criterion to determine the suitability of query languages for data streams, and as a tool for establishing the expressive power hierarchy of such languages. A language is $\mathcal{N}B$-complete if all the monotonic queries expressible in the language can be expressed via its nonblocking query operators. We proved that RA and SQL are not $\mathcal{N}B$-complete: thus by using them on relational streams, besides loosing the ability to express non-monotonic queries, we also lose the ability of writing some monotonic queries. In order to make up for this severe loss of expressive power, we suggested the support of natively defined UDAs in SQL: we show that, with this extension, SQL becomes Turing complete on database tables and $\mathcal{N}B$-complete (i.e., can express via nonblocking computations all set functions that are monotonic w.r.t. to set containment). Then the paper showed how these results can be extended to data streams where ordering becomes an integral part of the data model. Thus we proposed the notions of presequence and τ-presequence that generalize the notion of of set-containment for tuples that are, respectively, ordered implicity by their arrival or explicitly by their timestamps. We showed that union and other monotonic operators in relational algebra lose their monotonicity properties when they are applied to ordered sets. Thus we introduce the notion of monotonic approximations for these operators, that emulate their relational counterparts modulo a time delay. We were thus able to extend our results on Turing-completeness through UDAs to data streams explicitly ordered by their timestamps. Therefore the paper lays the foundation for a powerful class of relational data models and query languages that are Turing-complete on stored tables, and nb-complete on data streams. The power and practicality of this framework was further demonstrated by several application examples, and the fact that it can be used to emulate richer data stream models (e.g., those using negative tuples), which have been proposed because of their usefulness in specific applications.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for many suggested improvements. Yijian Bai, Richard Luo Chang, Hetal Thakkar and Nikolay Laptev deserve credit for their efficient implementation of UDAs in the Stream Mill DSMS.

REFERENCES

ATLaS user manual. http://wis.cs.ucla.edu/atlas.

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley.

ARASU, A., BABU, S., AND WIDOM, J. 2003. CQL: A language for continuous queries over streams and relations. In *DBPL*. 1–19.

ARASU, A. AND WIDOM, J. 2004. Resource sharing in continuous sliding-window aggregates. In *VLDB*. 336–347.

BABCOCK, B., BABU, S., DATAR, M., MOTAWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *PODS*.

BAI, Y., THAKKAR, H., WANG, H., LUO, C., AND ZANIOLO, C. 2006. A data stream language and system designed for power and extensibility. In *CIKM*. 337–346.

BAI, Y., THAKKAR, H., WANG, H., AND ZANIOLO, C. 2008. Timestamp management and query execution models in data stream management systems. *IEEE Internet Computing 12,* 6, 13–21.

BARBARA, D. 1999. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems 8,* 4, 295–323.

BOHLEN, M. H. 1994. The temporal deductive database system chronolog. Ph.D. thesis, Department Informatick, ETH Zurich.

CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2002. Monitoring streams - a new class of data management applications. In *VLDB*. Hong Kong, China.

CELKO, J. 1995. *SQL for Smarties*. Morgan Kaufmann, Chapter Advanced SQL Programming.

CHANDRASEKARAN, S. AND FRANKLIN, M. 2002. Streaming queries over streaming data. In *VLDB*.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*. 379–390.

CRANOR, C., GAO, Y., JOHNSON, T., SHKAPENYUK, V., AND SPATSCHECK, O. 2002. Gigascope: High performance network monitoring with an SQL interface. In *SIGMOD*. ACM Press, 623.

ET AL., M. A. H. 2004. Nile: A query processing engine for data streams. In *ICDE*. 851.

GOLAB, L. AND ÖZSU, M. T. 2003. Issues in data stream management. *ACM SIGMOD Record 32,* 2, 5–14.

GOLAB, L. AND ÖZSU, M. T. 2003. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*. 500–511.

GOLAB, L. AND ÖZSU, M. T. 2005. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD*. 658–669.

GUREVICH, Y., LEINDERS, D., AND DEN BUSSCHE, J. V. 2007. A theory of stream queries. In *Database Programming Languages, 11th International Symposium, DBPL 2007*, M. Arenas and M. I. Schwartzbach, Eds. Springer, 153–168.

GYLLSTROM, D., AGRAWAL, J., DIAO, Y., AND IMMERMAN, N. 2008. On supporting kleene closure over event streams. In *ICDE*. 1391–1393.

HAN, J., FU, Y., WANG, W., KOPERSKI, K., AND ZAIANE, O. R. 1996. DMQL: A data mining query language for relational databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*. Montreal, Canada, 27–33.

IMIELINSKI, T. AND VIRMANI, A. 1999. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery 3*, 373–408.

ISO/IEC. 2003. Database languages—sql, iso/iec 9075-*:2003.

JAGADISH, H., MUMICK, I., AND SILBERSCHATZ, A. 1995. View maintenance issues for the chronicle data model. In *PODS*. 113–124.

JOHNSON, T., MUTHUKRISHNAN, S., SHKAPENYUK, V., AND SPATSCHECK, O. 2005. A heartbeat mechanism and its application in gigascope. In *VLDB*. 1079–1088.

KANG, J., NAUGHTON, J. F., AND VIGLAS, S. 2003. Evaluating window joins over unbounded streams. In *ICDE*. 341–352.

KRÄMER, J. AND SEEGER, B. 2005. A temporal foundation for continuous queries over data streams. In *COMAD*. 70–82.

KRÄMER, J. AND SEEGER, B. 2009. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst. 34,* 1.

KRISHNAN, R. AND GOLDSTEIN, J. 2010. Microsoft streaminsight: A hitchhiker's guide to microsoft streaminsight queries.

LAW, Y.-N., WANG, H., AND ZANIOLO, C. 2004. Data models and query language for data streams. In *VLDB*. 492–503.

LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKE, P. A. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*. 311–322.

LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. A. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record 34,* 1, 39–44.

LIU, L., PU, C., AND TANG, W. 1999. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering (TKDE) 11,* 4, 583–590.

LUO, C., THAKKAR, H., WANG, H., AND ZANIOLO, C. 2005. A native extension of SQL for mining data streams. In *SIGMOD*. 873–875.

MADDEN, S., SHAH, M. A., HELLERSTEIN, J. M., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *SIGMOD*. 49–61.

MEO, R., PSAILA, G., AND CERI, S. 1996. A new SQL-like operator for mining association rules. In *VLDB*. Bombay, India, 122–133.

MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., S. BABU, M. D., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. 2003. Query processing, approximation, and resource management in a data stream management system. In *First CIDR 2003 Conference*. Asilomar, CA.

MOZAFARI, B., ZENG, K., AND ZANIOLO, C. 2010a. From regular expressions to nested words: Unifying languages and query execution for relational and xml sequences. *PVLDB 3,* 1, 150–161.

MOZAFARI, B., ZENG, K., AND ZANIOLO, C. 2010b. From regular expressions to nested words: Unifying languages and query execution forrelational and xml sequences. In *VLDB 2010, Singapore*.

MOZAFARI, B., ZENG, K., AND ZANIOLO, C. 2010c. K*sql: a unifying engine for sequence patterns and xml. In *SIGMOD Conference*. 1143–1146.

PERNG, C.-S. AND PARKER, D. S. 1999. SQL/LPP: A time series extension of SQL based on limited patience patterns. In *DEXA*. Lecture Notes in Computer Science Series, vol. 1677. Springer.

P.TUCKER, MAIER, D., AND T.SHEARD. 2003. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Engineering Bulletin 26,* 1, 33–40.

RAMAKRISHNAN, R., DONJERKOVIC, D., RANGANATHAN, A., BEYER, K., AND KRISHNAPRASAD, M. 1998. SRQL: Sorted relational query language.

ROZENSHTEIN, D., ABRAMOVICH, A., AND BIRGER, E. 1993. Loop-free SQL solutions for finding continuous regions. In *SQL Forum 2(6)*.

SADRI, R., ZANIOLO, C., ZARKESH, A., AND ADIBI, J. 2001a. Optimization of sequence queries in database systems. In *PODS*. Santa Barbara, CA.

SADRI, R., ZANIOLO, C., ZARKESH, A. M., AND ADIBI, J. 2001b. A sequential pattern query language for supporting instant data minining for e-services. In *VLDB*. 653–656.

SADRI, R., ZANIOLO, C., ZARKESH, A. M., AND ADIBI, J. 2004. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst. 29,* 2, 282–318.

SARAWAGI, S., THOMAS, S., AND AGRAWAL, R. 1998. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*.

SESHADRI, P. 1998. Predator: A resource for database research. *SIGMOD Record 27,* 1, 16–20.

SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. 1994. Sequence query processing. In *SIGMOD*, R. T. Snodgrass and M. Winslett, Eds. ACM Press, 430–441.

SOULÉ, R., HIRZEL, M., GRIMM, R., GEDIK, B., ANDRADE, H., KUMAR, V., AND WU, K.-L. 2010. A universal calculus for stream processing languages. In *Programming Languages and Systems, ESOP 2010*, A. D. Gordon, Ed. Springer, 507–528.

SRIVASTAVA, U. AND WIDOM, J. 2004. Flexible time management in data stream systems. In *PODS*. 263–274.

SULLIVAN, M. 1996. Tribeca: A stream database manager for network traffic analysis. In *VLDB*.

TECHNOLOGIES, I. I. 1994. Illustra user guide. 1111 Broadway, Suite 2000, Oakland, CA.

TERRY, D., GOLDBERG, D., NICHOLS, D., AND OKI, B. 1992. Continuous queries over append-only databases. In *SIGMOD*. 321–330.

THAKKAR, H., LAPTEV, N., MOUSAVI, H., MOZAFARI, B., RUSSO, V., AND ZANIOLO, C. 2011. Smm: a data stream management system for knowledge discovery. *ICDE 2011: International Conference on Data Engineering 27,* 1.

TUCKER, P. A., MAIER, D., SHEARD, T., AND FEGARAS, L. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng. 15,* 3, 555–568.

WANG, H. AND ZANIOLO, C. 2003. ATLaS: a native extension of SQL for data minining. In *Proceedings of Third SIAM Int. Conference on Data Mining*. 130–141.

WU, E., DIAO, Y., AND RIZVI, S. 2006. High-performance complex event processing over streams. In *SIGMOD Conference*. 407–418.

ZANIOLO, C., CERI, S., FALOUTSOS, C., SNODGRASS, R., SUBRAHMANIAN, V. S., AND ZICARI, R. 1997. *Advanced Database Systems* ISBN 1-55860-443-X Ed. Morgan Kaufmann.

ZEMKE, F., WITKOWSKI, A., CHERNIAK, M., AND COLBY, L. 2007. Pattern matching in sequences of rows. In *[sql change proposal, march 2007], http://asktom.oracle.com/tkyte/row-patternrecogniton-11-public.pdf http://www.sqlsnippets.com/en/topic-12162.html*.

A. FINITE STATE MACHINES

Observe that UDAs can be used to implement efficiently finite states machines (FSM). Let n be the number of arcs in the FSM. Then we can construct a UDA as follows: we use a local table to store the current state, the next state and the information for comparison. Then we use one statement for initialization. Then for each of the n arcs in the FSM we use an update statement changing the current state where the conditions on the arc are satisfied. Finally, we use one statement for transitioning to the final state, and the result is returned once we reach this final state. Thus as stated in Proposition 5.5, a FSM with n arcs can be expressed using a UDA with $n+2$ statements. This ability to support FSM implies that the search for patterns expressed using Kleene-closure expressions can also be implemented using UDAs [Mozafari et al. 2010b].

B. IMPLEMENTING TURING MACHINES USING UDAS

A Turing Machine is defined by a tuple $M = (Q, \Sigma, \Upsilon, \delta, q_0, !, F)$, where Q is a finite set of states, $\Sigma \subseteq \Upsilon$ is a finite set of input symbols, Υ is a finite set of tape symbols with $Q \cap \Upsilon = \phi$, $! \in \Upsilon - \Sigma$ is a reserved symbol representing the blank symbol, $q_0 \subseteq Q$ is an initial state, $F \subseteq Q$ is a set of accepting or final states, $\delta : Q \times \Upsilon \rightarrow Q \times \Upsilon \times \{1, 0, -1\}$ is a transition mapping where 1,0,-1 denote motion directions. In our implementation, a user may define a Turing Machine by giving four elements: a transition map(**E1**), accepting states(**E2**), a tape containing the input(**E3**) and an initial state(**E4**). With UDA, we put **E1** into a table called **transition**. **E2** is put into table **accept**. **E3** is put into table **tape**, which uses an attribute called **pos** to memorize the position of each symbol in the tape. Also, there is a table called **current**, which stores the current state, the current symbol and its position on the tape during each iteration. At the first iteration, the initial state (**E4**) and the leftmost symbol on the tape (**pos**=0) are put into **current**. For each iteration, a tuple of current is passed to a UDA called **turing**. If the transition function is defined for the (state, symbol) pair, we obtain the next state, the new symbol and the motion direction for the tape head. Then, the symbol pointed by the tape head is replaced by the new symbol. We move the head to the next position, which is given by **pos + move**. If it is a non-existing position on the tape, a new blank symbol is inserted at that position. Then, the updated tuple is inserted into **current** which is then passed to the UDA **turing** for the next iteration. The above procedures are repeated until the transition function δ is not defined for some (state, symbol) pair. In this case, the machine halts and checks whether the current state is an accepting state or not, based on the list of accepting states in table **accept**. The following is the implementation of a Turing Machine using UDAs.

```
TABLE current(stat Char(1), symbol Char(1), pos Int);
TABLE tape(symbol Char(1), pos Int);
TABLE transition(curstate Char(1), cursymbol Char(1),
    move int, nextstate Char(1), nextsymbol Char(1));
TABLE accept(accept Char(1));
AGGREGATE turing(stat Char(1), symbol Char(1),
                    curpos Int) : Int
{   INITIALIZE: ITERATE: {
      /*If TM halts, return 1/0(accept/reject)*/
      INSERT INTO RETURN
        SELECT R.C
        FROM (SELECT count(accept) C
                FROM accept A
                WHERE A.accept = stat) R
        WHERE NOT EXISTS (
          SELECT * FROM transition T
```

Table I. Transition mapping δ for finding the maximum.

	0	1	2	3	!
p	$q, 2, 1$	$u, !, 1$			$p, !, 1$
q	$q, 0, 1$	$r, 1, 1$			$q, !, 1$
r	$s, 3, -1$	$t, 1, -1$		$r, 3, 1$	$t, !, -1$
s	$s, 0, -1$	$s, 1, -1$	$p, 2, 1$	$s, 3, -1$	$s, !, -1$
t	$w, 0, -1$	$t, !, -1$	$t, 0, -1$	$t, !, -1$	$t, !, 1$
u	$u, 0, 1$	$v, 1, -1$		$u, 0, 1$	
v	$v, 0, -1$				$p, !, 1$
w	$w, 0, -1$		$w, o, -1$		$p, !, 1$

```
          WHERE stat = T.curstate AND symbol = T.cursymbol);
      /* write tape */
      DELETE FROM  tape
        WHERE pos = curpos;
      INSERT INTO tape
        SELECT T.nextsymbol, curpos
        FROM transition T
        WHERE T.curstate = stat AND T.cursymbol = symbol;
      /* add blank symbol if necessary */
      INSERT INTO tape
        SELECT '!', curpos + T.move
        FROM transition T
        WHERE T.curstate = stat AND T.cursymbol = symbol
          AND NOT EXISTS (
           SELECT * FROM tape
           WHERE pos = curpos + T.move);
      /* move head to the next position */
      INSERT INTO current
        SELECT T.nextstate, A.symbol, A.pos
        FROM tape A, transition T
        WHERE T.curstate = stat AND T.cursymbol = symbol)
          AND A.pos=curpos+T.move;}}
INSERT INTO current
    SELECT 'p', A.symbol, 0
    FROM tape A WHERE A.pos = 0;
SELECT turing(stat, symbol, pos) FROM current;
```

In the following, we implement a Turing Machine to find the maximum among the input numbers. The maximum will be stored back into the tape.

Example B.1. Turing Machine for finding the maximum

Let $M = (Q, \{0, 1\}, \{0, 1, 2, 3, !\}, \delta, p, !, \{\})$ be a Turing Machine for finding the maximum where δ is given by Table I. For simplicity, we assume that each number is an integer. Then we represent them in unary, i.e. $i \geq 0$ is represented by the string 0^i. These integers are placed on the input tape separated by 1's. The idea of this machine is to repeatedly compare the two left most integers in the input tape and to store the largest one back into the input tape. When the machine halts, we eliminate all symbols but 0's to extract the integer(in unary) in the input tape as the output of the query, which is the maximum number.

We have shown that UDA can express any function encoded in arbitrary input tape. A simple UDA can be used to encode a given table and then, on its TERMINATE state call the UDA that performs the actual computations. For several tables we can let the various UDAs write into the same input tape, with the last UDA calling the actual computation.