Query Languages and Data Models for Database Sequences and Data Streams

Yan-Nei Law Haixun Wang[1] Carlo Zaniolo

Computer Science Dept., UCLA
Los Angeles, CA 90095
{ynlaw, zaniolo}@cs.ucla.edu

IBM T. J. Watson Research[1]
Hawthorne, NY 10532
haixun@us.ibm.com

Abstract

We study the fundamental limitations of relational algebra (RA) and SQL in supporting sequence and stream queries, and present effective query language and data model enrichments to deal with them. We begin by observing the well-known limitations of SQL in application domains which are important for data streams, such as sequence queries and data mining. Then we present a formal proof that, for continuous queries on data streams, SQL suffers from additional expressive power problems. We begin by focusing on the notion of nonblocking ($\mathcal{NB}$) queries that are the only continuous queries that can be supported on data streams. We characterize the notion of nonblocking queries by showing that they are equivalent to monotonic queries. Therefore the notion of $\mathcal{NB}$-completeness for RA can be formalized as its ability to express all monotonic queries expressible in RA using only the monotonic operators of RA. We show that RA is not $\mathcal{NB}$-complete, and SQL is not more powerful than RA for monotonic queries.

To solve these problems, we propose extensions that allow SQL to support all the monotonic queries expressible by a Turing machine using only monotonic operators. We show that these extensions are (i) user-defined aggregates (UDAs) natively coded in SQL (rather than in an external language), and (ii) a generalization of the union operator to support the merging of multiple streams according to their timestamps. These query language extensions require matching extensions to basic relational data model to support sequences explicitly ordered by timestamps. Along with the formulation of very powerful queries, the proposed extensions entail more efficient expressions for many simple queries. In particular, we show that nonblocking queries are simple to characterize according to their syntactic structure.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

Data stream management systems represent a vibrant area of research [5, 6, 31, 10, 12, 19, 30, 17, 11, 8, 13]. The solution approach taken by most projects consists of extending database query languages and data models to support efficiently continuous queries on stream data, and is based on the sound rationale that, since many applications will span traditional databases and data streams, an unified programming environment will simplify their development. Nevertheless, database query languages were designed for persistent data residing on disks, rather than for transient data flowing through the wires: therefore their suitability to the new task need to be evaluated critically, and their limitations in this new role must be addressed. Indeed, the limitations of SQL in this new role are many and severe. For instance, the ineffectiveness of SQL to express queries on time series and sequences has been long recognized in the field and inspired much previous research [29, 27, 22, 2, 25, 24]. Since data streams are basically unbounded sequences, the inability of expressing sequence queries must be viewed as a serious limitation of SQL for continuous queries. Another well-known problem area for SQL is data mining [14, 20, 16, 26], since it is clear that SQL will be at least as ineffective at mining data streams as it is at mining persistent data. But in reality, the situation is significantly worse for data streams where additional issues arise to further impair the expressive power of

SQL. One is that queries involving traditional aggregates or constructs such as NOT IN, NOT EXISTS, ALL, EXCEPT cannot be allowed since they are blocking, i.e., they do not return their results until they have seen the whole input [5]. Only nonblocking query operators can be allowed on data streams [5], and we will prove that all monotonic queries, and only those, can be expressed using nonblocking computations—a result that was first claimed in [34].

This set the stage for one more problem (the fourth in our list) inasmuch as relational algebra (RA) and SQL are not complete for nonblocking queries, since they can only express some monotonic queries using blocking operators. The final problem follows from the fact that traditional database applications would normally be developed by embedding SQL queries in procedural languages using cursor-based interface mechanisms. Therefore, expressive-power limitations of SQL would be remedied by writing in the procedural language the part of the application that could not be readily expressed in the embedded SQL query. But the cursor-based model of embedded queries is one where the the procedural language program sees a static window onto the database and controls the movement of the cursor via get-next statements. But as data streams arrive furiously and continuously, the data stream manager cannot hold the current tuple, and all that have arrived after that, waiting for the application to issue a get-next statement. Indeed, most of current data stream management systems do not support cursor-based interfaces to programming languages.

In summary, the lack of expressive power and extensibility that were already serious problems for SQL (as per the sequence queries and data mining queries) are now made much more severe by data streams, where blocking query operators are disallowed and the remedy of embedding the SQL queries into a procedural language is also compromised. Therefore, an in-depth study of this problem and its possible solutions is sorrily needed, given that only limited studies have been proposed in the past (see next section). We will also show that the problem has interesting implications on the data model to be used for data streams: for instance, the presence of time stamps is required for query completeness.

The paper is organized as follows. In the next section, we survey several data models for sequences and streams. In Section 3, we study nonblocking query operators which we prove equivalent to monotonic operators; in Section 4 we show the incompleteness of relational query languages with respect to monotonic operators. In Section 5, we introduce a native extensibility mechanism for SQL which the data model is suitable for data stream and sequence queries. Also, this extension is Turing Complete—the result proven in Section 6. In Section 7, we prove completeness w.r.t. the functions computable by nonblocking computations. In section 8, we recap the benefits of the proposed extensions with sequence queries, data mining functions, and memory minimization.

2 Related Work

Significant projects on data streams include those described [5, 6, 31, 10, 12, 19, 30, 17, 11, 8, 13]. In this section we discuss issues such as blocking operators, data model, and query power that are most significant for this paper.

The Tapestry project was the first to model data streams as append-only databases supporting continuous queries [31]. The problem of blocking operators was also identified in [31] 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 [5, 13] 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. Even the work presented in [32], these focuses on overcoming the blocking operator problem has not pursued their formal characterization. The work described in [32] presents an interesting approach for overcoming the problems of blocking operators using punctuated data streams. The data stream is modelled as an infinite sequence of finite lists of elements. Then punctuation marks 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 accessing 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 we will show (i) can also deal with punctuation, (ii) are defined natively using SQL, and (iii) make the SQL's expressive power equivalent to that of a Turing machine. The use of UDAs for enhancing the power of query languages for data streams is also been advocated by the Aurora project [8], where non-SQL operators are however used to define UDAs.

While although the objective of overcoming the expressive power limitations caused by the exclusion of blocking operators provides the clear motivation for much previous work, at the best of our knowledge, there has been no attempt to characterize how much expressive power is lost without blocking query operators, or how much power is gained back with extensions such the unary stream operators [32], or the UDAs used in Aurora [8]. (In this paper, we will prove that the power loss due to blocking operators and the power gain due to UDAs are both very high.)

Although there has been no formal investigation of the limitations of SQL for data stream applications, the investigations for other application domains of interest are nearly too many to mention. Of particular significance are those focusing on sequence queries, including those presented in [29, 27, 22, 2, 25, 24]. In particular, the sequence model called $\mathcal{SEQ}$, introduced in [28], focuses on possible extensions to the relational data model and relational algebra. Therefore, many-to-many relations are defined between a set of records and a countable totally ordered domain (e.g., the integer set) to give positions for each record, along with two new classes of sequence operators, the *positional* operators and *record-oriented* operators. The expressive power entailed by these extensions, however, is not characterized.

Similar extensions to the relational model and relational algebra however have not been pursued in later studies of sequence queries [2, 25, 24] and stream queries and will not be considered in this paper. In this paper, we followed the generally accepted model of viewing data streams as bags of append-only of ordered tuples. In fact, we will show that (in Section 7) that time stamps must be added to achieve the completeness for non-blocking queries. After this necessary addition, our data stream can be modelled as an unbounded appended-only bags of elements <tuple, timestamp> as in CQL [4, 21], along the line of SQL (although CQLs *Istream*, *Dstream* and *Rstream* are not considered in this paper).

3 Nonblocking Query Operators

We can now formalize the notion of sequences as a bridge between database relations and streams. 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 time-stamped 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 [5]. 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/nonblocking operators.

According to [5] *'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' will 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 [5]. 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 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 is denoted by $[\,]$; $[\,]$ has length 0.*

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 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 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. 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 4 *A non-null operator G is said to be*

- blocking, *when for every sequence S of length n, $G^j(S) = [\ ]$ for every $j < n$, and $G^n(S) = G(S)$*
- nonblocking, *when for every sequence S of length n, $G^j(S) = G(S^j)$, for every $j \leq n$.*

Therefore, 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 1 *A function $F(S)$ on a sequence S can be computed using a nonblocking operator, iff F is monotonic with respect to the partial ordering $\sqsubseteq$.*

Proof: Say that $S^j \sqsubseteq S^k$, i.e., S^j is a presequence of S^k, and $j \leq k$. Let G be a nonblocking computation on S. Then $G(S^j) = G^j(S^j) = G^j(S^k)$, where $G^j(S^k) \sqsubseteq G^k(S^k) = G(S^k)$. Thus 'nonblocking' implies'monotonic'. 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 $H(S)$ defined as follows: $H^j(S^n) = G^j(S^j)$. We have that $H(S) = G(S)$ and H is nonblocking. QED.

Streams are infinite 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.

Order! In this section we have considered *physically ordered relations*, i.e., those where only the relative positions of tuples in sequence are of significance. In the next section, we will consider *unordered relations*, i.e., the traditional database relations, that we will call Codd's relations. Later, we will study *logically ordered relations*, i.e., sequences where the tuples are ordered by their timestamps or other logical keys. All three types of relations are important, since each type is needed in different applications and they have complementary properties.

For instance, the OLAP functions of SQL:1999 can compute the average of the last 100 tuples in the sequence (physical window). Besides OLAP functions, aggregates, such as continuous sum, and online average [15], are dependent on the physical order of relations. The physical order model is conducive to great expressive power, but cannot support binary operators as naturally as it does for unary ones. For instance, in SQL the union of two tables T1 and T2 is normally implemented by first returning all the tuples in T1 and then all the tuples in T2. The resulting operator, is not suitable for continuous queries, since it is partially blocking (and nonmonotonic) with respect to its first argument T1 (since tuples from T2 cannot be returned until we have seen the last tuple from T1). These issues can either be resolved by using Codd's relations (next section) or logically ordered relations, discussed in Section 7.

4 Unordered Relations, RA & SQL

Codd's relational model views relations as sets of tuples where the order is immaterial (commutativity property). In these relations duplicates are disallowed via candidate keys (or, duplicates can be simply disregarded as via the 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.) Therefore we have the following theorem:

Proposition 2 *A unary query operator on Codd's relations is nonblocking iff it is monotonic w.r.t.* $\subseteq$.

Since we are only interested in deterministic queries, the only operators that are legal on Codd's relations are those that deliver the same results for any order in which the tuples are arranged in the table—also independent of duplicates if these are present. (Of course, 'same results' here means results that are equal in terms of set equality.) For instance, the select and project operators of relational algebra, traditional aggregates and continuous count are legal operators on Codd's relations, since their results do not depend on the order of tuples. However, continuous sum, or continuous averages, is not a valid operator on a Codd's relation since it produces results that depend on the order in which the tuples are arranged (if they are not identical).

Union and Cartesian product are monotonic with respect to set containment and amenable to nonblocking implementations. Set difference $R - S$ is instead antimonotonic and blocking with respect to its second argument. In fact no result can be returned for $R - S$ until the last tuple of S is known. Therefore, query operators such as $R - S$ should be avoided in expressing continuous queries on a data streams S. We explore the crippling effects of this limitation in the next section.

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 $\mathcal{NB}$-RA, where $\mathcal{NB}$-RA $= \{\cup, \bowtie, \sigma, \Pi\}$.

The class of queries expressible by RA (and many equivalent query languages) is called *FO* queries [3]. Let $\mathcal{NB}$-*FO* denote the monotonic queries in *FO*. But some monotonic functions in *FO* are expressed using set difference, an operator not in $\mathcal{NB}$-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 $\mathcal{NB}$-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 We have a temporal domain, closed to the left and open to the right, which we will represent using nonnegative integers, originating at zero. (While examples are simpler with integers, any totally ordered temporal domain will do as well.) 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. Thus, we use intervals closed to the left and open to the right. 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` $\mathcal{U}$*ntil* `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 [7, 9, 23]. 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: `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., $\neg$`cep(K)`. This break excludes `(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 1 *Until* `(pUq)` *& Coalesce (*`coalscp)`

```
pUq(yes) ←          q(0,J).
pUq(yes) ←          coalscp(0,I),q(J,_),I ≥ J.
coalscp(I1,J2) ←    p(I1,J1),p(I2,J2),J1 < J2,
                    ¬broken(I1,J2).
broken(I1,J2) ←     p(I1,J1),p(I2,J2),p(_,K),
                    J1 ≤ K,K < I2,¬cep(K).
cep(K) ←            p(_,K),p(I,J),I ≤ K,K < J.
```

The safe non-recursive Datalog program of Example 1 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.

However, while the coalesce query and the until queries are in $\mathcal{NB}$-*FO*, they cannot be expressed in $\mathcal{NB}$-RA:

Proposition 3 *The coalesce and until queries cannot be expressed in* $\mathcal{NB}$-*RA.*

Proof Sketch: Let P be the table containing the intervals to be coalesced. By selection and projection on the Cartesian product of P with itself $n-1$ times, we can express the coalescing of up to n intervals from P. But P can contain an arbitrary number of intervals. □

Meanwhile, we observe that this problem can be solved using $\mathcal{NB}$-RA with recursion. Here is a solution:

```
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.
```

SQL-$\mathcal{NB}$ We next consider $\mathcal{NB}$-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 $\mathcal{NB}$-SQL is the same as $\mathcal{NB}$-RA, although SQL can express more monotonic queries than RA. In fact, some queries expressed using aggregates are monotonic. For instance, Example 2, below, computes from empl(EmpNo, Sal, DeptNo) all the departments where the sum of employee salaries exceeds a given constant C.

Example 2 *Departments where the sum of employee salaries exceeds C. Assume Sal > 0.*

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

This is obviously a monotonic query, 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 [32] and windows [21] to address this problem. While these approaches deal effectively with important aspects of the problem, they do not solve the expressivity problems discussed so far. For instance, punctuation and windows cannot be used to implement queries of Example 1 or Example 2 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 1 or Example 2 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.

A better approach is to introduce new monotonic operators to 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 [3]. 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 [3]. 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 champion 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 [8, 18, 12]. ATLAS [33] and ESL [18] adopt from SQL-3 the idea of specifying a new UDA by an INITIALIZE, an ITERATE, and a TERMINATE computation; however, 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[1]. Example 3 defines an aggregate equivalent to the standard AVG aggregate in SQL. The second line in Example 3 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

[1]Although UDAs have been left out of SQL:1999 specifications, they were part of early SQL-3 proposals, and supported by some commercial DBMS.

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[2]. Thus, the TERMINATE statements are processed just after all the input tuples have been exhausted.

Example 3 *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 [8]. Then we can write the UDA of Example 4, 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) [1, 18].

Thus UDAs operate as general stream transformers. Observe that the UDA in Example 3 is blocking, while that of Example 4 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 [18]. 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 4 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 % 200 = 0;
         UPDATE state SET tsum=0, cnt=0
            WHERE cnt % 200 = 0
      }
      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 (as the cursor mechanism used by programming languages to interface with SQL) they dovetail with the physically-ordered sequence model, and can also express well the search for pattern in sequences. Say for instance that we want to find the situation where users, immediately after placing an order, ask for a rebate and then cancel the order. Finding this pattern in SQL requires two selfjoins to be computed on the incoming stream of webevents. In general recognizing the pattern of n events would require $n-1$ joins and queries involving the joins of many streams can be complex to express in SQL, and also inefficient to execute. Also the notion that a tuple must immediately follow another tuple is complex to formulate in SQL. UDAs can be used to solve these problems. For instance, say that we want to detect the pattern of an order, followed a rebate, and then, immediately after that a cancellation. Then the following nonblocking UDA 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 computation continues.

[2]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.

Example 5 *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;
      }
}
```

Very often, the input order of sequence elements is the same as their production order — this fits the design of UDAs naturally. In [28], Seshadri et al. showed an example of query that asks for the 3-day average of the close of IBM stock values when the value of DEC is greater than that of HP. In the following example, the UDA only needs to store the last three-day values for IBM and compares the values of DEC and HP to see whether the average should be output. Note that it is easy to generalize the expression using UDA to compute n-day average using **state** to store last n-day values of IBM.

Example 6 *3-day average for IBM when DEC>HP*

```
AGGREGATE 3DayAve(ibm Real,dec Real,hp Real):Real
{     TABLE state(st Int, nd Int, rd Int,tcnt Int);
      INITIALIZE : {
         INSERT INTO state VALUES (0, 0, ibm, 1)}
         INSERT INTO RETURN
            SELECT third/tcnt FROM state
            WHERE dec>hp;}
      ITERATE: {
         UPDATE state
            SET st=nd, nd=rd, rd=ibm;
         UPDATE state
            SET tcnt=tcnt+1
            WHERE tcnt<3;
         INSERT INTO RETURN
            SELECT (st+nd+rd)/tcnt FROM state
            WHERE dec>hp;
      }
      TERMINATE : {  }
}
```

UDAs are also suitable for punctuated data streams [32]. When an input arrives, the UDA needs to compute the results, store the state and output based on punctuation. In Example 7, we want to output the average stock value of each company when we receive its closing value tuple which is a punctuation indicating that no more tuple of this company will arrive. We use the table **state** to store the summary (sum and count) of each company which is the minimal amount of information that we should store for further computations. Upon detection of a punctuation mark indicating the arrival of the closing-value tuple (with condition **close=1**), we return the average for this company.

Example 7 *Output average price for each company when closing price tuple enters*

```
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 : {  }
}
```

Therefore UDAs, unlike traditional SQL, are well-suited to supporting state-based reasoning and queries, as needed in sequence and data stream applications. The use of UDAs to support the mining of data streams is discussed in [18]. In the next section, we show that UDAs are able to express the ultimate state machine: a Turing machine. Readers who are primarily interested in the applications of this theoretical result to data streams can proceed directly to Section 7, where we discuss the $\mathcal{NB}$-completeness of monotonic UDAs and their benefits in data stream applications.

6 Completeness on DB Relations

Turing completeness is hard to achieve for database languages [3]. In particular, SQL is not Turing complete, and thus not capable of expressing all data-intensive applications. The power of a query language is defined as the class of functions it can express on (an input tape encoding) the database [3]. We will next show that UDAs can compute an arbitrary query function encoded as a Turing machine.

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 $Q \cap \Upsilon = \phi$, $! \subseteq \Upsilon - \Sigma$ 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, $\delta : Q \times \Upsilon \to Q \times \Upsilon \times \{1, 0, -1\}$ 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
          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 8 *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 1. 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.

	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	$w, 0, -1$		$w, o, -1$		$p, !, 1$

Table 1: Transition mapping δ for finding the maximum.

In the previous section, 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. But such an encoding of one or more tables into an input tape is a blocking computation. For continuous queries we seek nonblocking computations on one or more data streams. These are discussed next.

7 Completeness on Data Streams

According to [13], 'queries over streams run continuously over a period of time and incrementally return new results as new data arrive.' In the following, we will show how to compute a query over streams. We will focus on monotonic functions as they are the only continuous queries supported on data streams.

Every monotonic function F on an input 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:

1. Append the encoded new tuple to IN,
2. Copy IN to $TAPE$, and compute $F(IN) - OUT$ as described in Section 5,
3. Return the result obtained in 2 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. Thus, every monotonic function on a single data stream can be computed by a nonblocking UDA.

However, the situation is more complex for multiple data streams, since these need to be merged into a single stream before UDAs can be applied. For instance, the operator used in SQL:1999 for computing the union, $R_1 \cup R_2$ of the ordered relations R_1 and R_2 while preserving duplicates cannot be used. In fact, this operator will list all the tuples in R_1 before the tuples in R_2. Thus this operator is blocking with respect to its first argument. We instead need operators that merge the two streams by assuring not only fairness, but also minimizing the delay across streams. To achieve this timestamps are needed and then the union operator can be defined that union-merges these multiple streams into one by their timestamps.

Therefore we now consider explicitly timestamped data streams and time-series sequences, where tuples are explicitly ordered by increasing values of their timestamps. [3] We begin with notion of τ-presequence defined as the sequence of tuples up to a given timestamp τ:

Definition 5 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 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.*

[3]Similar considerations can be made to arbitrary logically ordered sequences, where tuples are arranged and visited sequentially according to an ordering key consisting of one or more attributes.

Then the notion of monotonicity can also be defined naturally. A unary operator G is monotonic if $L_1 \sqsubseteq^t S_1$ implies $G(L_1) \sqsubseteq^t G(S_1)$. A binary operator H is monotonic when $(L_1, L_2) \sqsubseteq^t (S_1, S_2)$ implies $H(L_1, L_2) \sqsubseteq^t H(S_1, S_2)$.

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 $\tau' > \tau$.

For $\tau = 0$, $S^\tau = \emptyset$ is an empty sequence. Let $\Omega(S)$ denote the largest timestamp in S (0 if S is empty). A query operator is said to be null when it returns the empty sequence for every possible value of its argument(s).

Then, the notion of nonblocking operators on logical sequences can be defined as follows:

Definition 6 *Nonblocking.*

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

We can then show that functions on logically ordered sequences can be implemented by nonblocking operators iff they are monotonic w.r.t. $\sqsubseteq^t$. It also follows that only blocking implementations are possible for an operator that computes the difference of two streams, since difference is antimonotonic on its second argument.

The previous notions lead to natural generalizations for selection, projection and union; suitable generalizations of Cartesian product and join are also available [5] but they are outside the scope of this paper (since they are not needed for the completeness of our language). For union we have:

Union. Let $\cup^\tau$ denote the stream transducer implementing union. $\cup^\tau$ returns, at any given time τ, the union of the τ-presequences of its inputs:

$$L \cup^\tau S = L^\tau \cup S^\tau$$

In the following example, we demonstrate how to express a query using Union and UDA. Consider two streams of phone-call records:

StartCall(callID, time);
Endcall(callID, time);

The stream **StartCall** is used to record a starting time of each call with its ID, while the stream **EndCall** is used to record a finishing time of each call with its ID. Given the above two streams, we are interested in finding the length of each call. Instead of joining two streams, we first union them together: **CallRecord**, which is sorted by the arrival timestamp. Moreover, we use a tag to indicate which stream does each tuple come from. Tuples are grouped by different callID group.

Example 9 *Compute the length of each call.*

```
SELECT callID, length(time, tag) AS CallLength,
FROM
      ( SELECT callID, time, 'start'
      FROM StartCall
      UNION ALL
      SELECT callID, time, 'end'
      FROM EndCall) AS
   CallRecord (callID, time, tag)
GROUP BY callID;
```

The UDA **length** is used to compute the difference between the starting time and finishing time. We design this UDA to handle all the arrival ordering. This UDA is shown below:

```
AGGREGATE length(time, tag) : (CallLength)
{     TABLE state(ttime);
      INITIALIZE: ITERATE :{
         INSERT INTO state VALUES(time);
         INSERT INTO RETURN
            SELECT time-ttime FROM state
            WHERE tag='end';
         INSERT INTO RETURN
            SELECT ttime-time FROM state
            WHERE tag='start';}}
```

We can now show the completeness of languages supporting union operators and nonblocking UDAs on data streams, in the sense that they can express every monotonic function on their input.

$\mathcal{NB}$ UDAs: $\mathcal{NB}$-UDAs are those where the TERMINATE state is empty or missing.

Proposition 4 *$\mathcal{NB}$-Completeness. Every computable monotonic function on timestamped data streams can be expressed using $\mathcal{NB}$-UDAs and union.*

From a formal viewpoint, these results can be extended to physically ordered data streams by simply viewing sequence numbers as time stamps (then, $\sqsubseteq$ becomes a special case of $\sqsubseteq^t$). The problem is that tuples from two streams that have the same sequence number could have arrived at very different times. Therefore most systems and users prefer the solution of merging the tuples of two streams according to the order in which they are actually processed. This is equivalent to viewing them as logically ordered streams where the time stamp is the current time at the point in which the tuples are processed for union.

8 Conclusions

Data streams require significant changes in traditional DB technology. This paper is the first to propose a formal analysis of how query languages and also data models are impacted by these changes, to propose practical solutions for the resulting problems. We studied how traditional models, where data is viewed as unordered sets of tuples, can be enriched with (physical and logical) ordering, and the notion of set containment can be generalized to the new framework. While data streams bring enrichments to data models, they bring restrictions to the query languages since they require nonblocking queries.

We characterized nonblocking queries as monotonic queries, and introduce the notion of $\mathcal{NB}$-completeness that characterize the expressive power hierarchies for continuous query languages on data streams. We thus proved that RA and SQL are no longer complete for nonblocking queries (exacerbating limitations which had already surfaced with data mining and sequence queries). To solve this problem, we proposed the use of UDAs, a native extensibility mechanism that makes SQL Turing-complete on stored data. For data streams, we introduced the notion of $\mathcal{NB}$-completeness for query languages capable of expressing all functions computable via nonblocking procedures; then we showed that any query language supporting UDAs and nonblocking union is $\mathcal{NB}$-complete. Of course, this is not to suggest that practical continuous query languages, such as the ESL language we are implementing [18], only need these two constructs. Practical data-streams languages should support SQL, and additional constructs needed for data streams, such as logical and physical windows [4]. (For instance, ESL also supports windows on UDAs [18]). But there are situations where the additional power of UDAs becomes critical: these situations include data mining functions [33], sequence queries [25], and special situations where more control is needed to minimize the use of memory [18]. The fact that the notion of UDAs is fully compatible with the syntax and semantics of existing prototypes, and they already part of some systems [8, 18], enhances the practical import of the theoretical findings summarized in this extended abstract.

Acknowledgements

This work was supported in part by a gift from Teradata, and by the National Science Foundation grant NSF-IIS 0339259.

References

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

[2] *SQL/LPP: A Time Series Extension of SQL Based on Limited Patience Patterns*, volume 1677 of *Lecture Notes in Computer Science*. Springer, 1999.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[4] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University, 2002.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[6] D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, 1999.

[7] M. H. Bohlen. *The Temporal Deductive Database System ChronoLog.* PhD thesis, Department Informatick, ETH Zurich, 1994.

[8] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, Hong Kong, China, 2002.

[9] J. Celko. *SQL for Smarties*, chapter Advanced SQL Programming. Morgan Kaufmann, 1995.

[10] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[11] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, May 2000.

[12] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *SIGMOD Conference*, pages 647–651. ACM Press, 2003.

[13] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[14] J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A data mining query language for relational databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 27–33, Montreal, Canada, June 1996.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[16] T. Imielinski and A. Virmani. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery*, 3:373–408, 1999.

[17] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4):583–590, August 1999.

[18] Chang R. Luo, Haixun Wang, and Carlo Zaniolo. ESL: a data stream query language and system designed for power and extensibility. In *submitted for publication*, 2004.

[19] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–61, 2002.

[20] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *VLDB*, pages 122–133, Bombay, India, 1996.

[21] R. Motwani, J. Widom, A. Arasu, B. Babcock, M. Datar S. Babu, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *First CIDR 2003 Conference*, Asilomar, CA, 2003.

[22] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. Beyer, and M. Krishnaprasad. SRQL: Sorted relational query language, 1998.

[23] D. Rozenshtein, A. Abramovich, and E. Birger. Loop-free SQL solutions for finding continuous regions. In *SQL Forum 2(6)*, 1993.

[24] Reza Sadri, Carlo Zaniolo, and Amir M. Zarkesh and-Jafar Adibi. A sequential pattern query language for supporting instant data minining for e-services. In *VLDB*, pages 653–656, 2001.

[25] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *PODS*, Santa Barbara, CA, May 2001.

[26] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, 1998.

[27] P. Seshadri. Predator: A resource for database research. *SIGMOD Record*, 27(1):16–20, 1998.

[28] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *ICDE*, pages 232–239, Taipei, Taiwan, March 1995.

[29] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *ACM SIGMOD 1994*, pages 430–441. ACM Press, 1994.

[30] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 1996.

[31] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 6 1992.

[32] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng*, 15(3):555–568, 2003.

[33] Haixun Wang and Carlo Zaniolo. ATLaS: a native extension of SQL for data minining. In *Proceedings of Third SIAM Int. Conference on Data Mining*, pages 130–141, 2003.

[34] Carlo Zaniolo, Chang Richard Luo, Y. Law, and Haixun Wang. Incompleteness of database languages for data streams and data mining: the problem and the cure. In *Eleventh Italian Symposium on Advanced Database Systems: SEBD 2003*, June 2003.