SRQL: Sorted Relational Query Language

Raghu Ramakrishnan Donko Donjerkovic Arvind Ranganathan Kevin S. Beyer
Muralidhar Krishnaprasad

Department of Computer Sciences
University of Wisconsin-Madison

Abstract

A relation *is an unordered collection of records. Often, however, there is an underlying order (e.g., a sequence of stock prices), and users want to pose queries that reflect this order (e.g., find a weekly moving average). SQL provides no support for posing such queries. In this paper, we show how a rich class of queries reflecting sort order can be naturally expressed and efficiently executed with simple extensions to SQL.*

1. Introduction

Ordered data, or *sequences*, can be found in a wide range of commercial, statistical, and scientific applications. These applications require DBMS support to store, manipulate, and query sequences efficiently, and such support is missing in RDBMSs since the relational model provides *sets of tuples* as its only data structure. SQL [2], the most widely used query language for relational systems is incapable of answering some common queries posed by commercial and scientific applications, such as moving aggregates.

One approach that is being explored in many commercial systems is support for sequences as a new ADT. Users can store a sequence in a field of a tuple, and manipulate it using associated system-defined methods such as moving averages. In earlier work [8], we argued that the ADT approach was inadequate for supporting bulk data types (such as sequences) over which users might want to ask a rich class of queries. The approach limits both the ease with which queries can be formulated, and the degree of query optimization that can be achieved. We proposed the concept of an *enhanced ADT* (EADT) to handle such bulk data types, and demonstrated the benefits of this approach.

In this paper, we consider another approach to extending relational DBMSs with support for sequence data, based on treating sequences as sorted relations, with features in the query language that exploit the sort order. Our extensions to SQL allow a combination of unordered and (logically) ordered relations to be queried naturally, and permit efficient evaluation with minimal extensions to a traditional relational optimizer and evaluation engine. In comparison to the ADT approach, a richer class of queries can be naturally expressed, and efficiently evaluated. In comparison to the EADT approach, queries involving a mixture of relations and sequences are easier to express, and the full machinery of EADTs need not be implemented. (Although, the EADT approach offers more generality in that support for other data types with rich query capabilities can be added easily.)

We present an extension to SQL to illustrate our ideas; this is called *Sorted Relational Query Language*, or *SRQL* (pronounced "circle"). We emphasize how a few simple extensions allow sort order to be effectively exploited at both the language and optimization/evaluation levels. We have implemented SRQL as part of the data transformation engine of the DEVise system [5], which also supports powerful visualization features. (In this paper, we will not discuss any aspects of DEVise other than SRQL.)

The main contributions of this paper are:

- The approach to modelling sequences as sorted relations, rather than as ADTs, as is commonly being done in Object-Relational DBMSs or ORDBMSs (e.g., [3]), or as EADTs. This leads to easier querying of a combination of relations and sequences, and enables more integrated optimization and evaluation.
- An algebra over sequences that extends relational algebra to address sort order.
- The extension of SQL to query sorted relations in SRQL. We extend the earlier results of SEQUIN [8, 9]

significantly, and define SRQL semantics in terms of our algebra.

- An implementation demonstrating that efficient and scalable query evaluation is feasible for SRQL.

The rest of this paper is organized as follows. In Section 2, we provide the motivation for our work by considering queries that illustrate the deficiencies of SQL in querying sequences. We describe the extension of relational algebra to a sequence algebra in Section 3, and we introduce the SRQL language as an extension of SQL in Section 4. Performance results of our initial SRQL implementation are presented in Section 5. We talk of related work, especially with respect to ORDBMSs, TSQL [10] and RISQL [7] in Section 6. Section 7 concludes the paper.

2. Motivation

Sequence data is encountered in a wide variety of scientific and commercial applications, e.g., experimental traces, process evolution, satellite observations over time, stock market prices, and salary histories. There is also great interest in maintaining a history of a user's queries, or a log of the changes made to a database, and analyzing such trace data to identify interesting patterns of usage. Given these trends, the ability to analyze large sequences is becoming increasingly important and DBMS vendors are beginning to add such capabilities. The work reported here presents an attractive alternative to the two main existing approaches, which are based on ADTs and EADTs.

The following simple queries illustrate the usefulness of sequence query support:

1. Find the leading weekly moving average of the Dow Jones Industrial Average, given a table *DJIA(date, close)*.
2. Find the trailing weekly moving average (i.e., the average for the past week, for each date) for each stock, given a table *Stocks(date, symbol, close)*. Order the resulting sequences by stock name, date. (The result can be thought of as a relation with fields *symbol, date, average*, sorted on the composite key *(symbol, date)*.)
3. For each day in '97, find the ten cheapest stocks.
4. For each day in '97, find the ten most expensive stocks.
5. Compute the percent change of each stock during 1997, and then find stocks that were in the top 5%.
6. Find the of dates of stocks where the leading weekly average is greater than 1.2 times the trailing weekly average, given the Stocks table.
7. For each week, find the stock in which a given customer (say Joe) had the most invested. In addition to the Stocks table, a table *Trans(customer, symbol, date, shares)* records the change in shares for a given customer, symbol, and date.

These queries illustrate the rich class of queries one can ask over sequences. Of course, a wider variety of aggregate operations could be included, and various kinds of date arithmetic and calendar support could be added (e.g., understanding leap years, the difference between a sequence of week-days and a sequence of integers), but these will not be our focus. We focus on how *sequentiality* leads to novel query patterns, and how to express and optimize such queries in a relational setting.

None of the above queries can be expressed in SQL-92. Some (e.g., the first) but not all can be expressed by some SQL extensions in current products such as Red Brick's RISQL. In particular, queries such as the second (which involves nested sequences) or the last (which involves combining sorted and unsorted tables) are not supported. In Appendix A we give SRQL versions of these queries.

Our approach is also relevant to temporal database query languages, but additional work is required to address temporal issues such as calendars and time intervals.

3. Extensions to Relational Algebra

To manipulate sorted relations, we extend relational algebra (plus *group-by* and *NULLs* as in SQL) with four new operators: Sequence (Ψ), Shift (δ), ShiftAll (Δ), and WindowAggregate (ω). Only Ψ is a necessary extension; the rest can then be defined using Ψ, relational algebra (plus group-by and NULLs). We define the semantics of SRQL queries in terms of this "sequence algebra" in Section 4. The algebra represents the logical operators; the implementation is free to define more efficient physical operators (like join).

For consistency with SQL, we consider a relation to be a *multiset of tuples* rather than a set of tuples. Like SQL, we use an operator "Distinct" to remove duplicates, and support operators that distinguish between groups of tuples in a relation that are partitioned (but not necessarily fully sorted) by value of the grouping attributes.

3.1. Sequences as Sorted Relations

We begin by defining our notion of sequences, which are essentially just sorted relations. A *simple sequence* is a relation that is (logically, though not necessarily physically) sorted on a key that is a concatenation of attributes, which we call the *sequencing attributes*.

A *composite sequence* is a relation that is first grouped on a set of attributes, called *grouping attributes*, and then sequenced by a concatenation of sequencing attributes within each partition defined by identical group values. The value of the grouping attributes in any tuple of the relation is called the *group value*; likewise, the value of the sequencing attributes is called the *sequence value*. In the remainder of the paper, we use the term *sequence* to refer to either a simple or composite sequence.

To be precise, a sequence is a relation plus a set of grouping attributes and a list of sequencing attributes. The grouping and sequencing attributes (which are allowed to be empty sets) induce an ordering over the tuples. For each tuple, we can therefore talk of the ordinal number of the tuple in the sequence (within the group of the tuple). Within each group, all integers between the first and last ordinal for the group are assigned to some tuple of the group (i.e., the ordinal numbering is *dense*). For convenience, we treat the ordinal number of a tuple as a special integer attribute, *ordinal*, but the reader should note that this attribute is not actually stored. [1]

3.2. Sequence Algebra

The algebra operates on sequences, of which unsorted relations are just a special case having empty sets of grouping and sequencing attributes. Thus, we begin with the basic relational operators (defined on multisets, as in the algebra underlying SQL, e.g., [6, 4], and extend them to work on sequences: select (σ), project (π), cross-product ($\times$), union ($\cup$), and set-difference ($-$). To extend them, we must define how the output is grouped and sequenced; we do this simply by defining the grouping set and sequencing lists to be empty for the result of each of the above operators. Observe that the *ordinal* "attribute" of the input relations is not propagated to the result! The most important point to note is that *we allow the select operator to specify selection conditions over the* ordinal *attribute of its input.*

In contrast to relational algebra, we need to treat join ($\bowtie$) as a primitive operator because we want to be able to refer to the *ordinal* attributes of the input tables in the join condition, and our version of cross-product does not propagate these special attributes. We also include left-outer join ($\blacktriangleright\!\!\triangleleft$), which is the same as join, except that it guarantees that all tuples from its left input appear in the output. Any left tuple that does not match a right tuple is matched with NULL values for the right tuple. The join operators in SRQL, like the other extended relational operators, are defined to have empty grouping sets and sequencing lists.

In the rest of this section, we present the operators that directly address sort order. The essential new operator, Sequence (Ψ), creates a sequence. No additional operators are necessary, but it is convenient to introduce shift operators that align tuples of a sequence based on the sequence order, and an operator for applying aggregate functions to sequences. These additional operators are defined in terms of the core SRQL algebra, which consists of the extended relational operators and the Ψ operator.

[1] We sometimes abbreviate the ordinal attribute as *ord*.

3.3. Creating a Sequence

Our fundamental extension of relational algebra consists of the Sequence operator (Ψ), which (re)sequences its input table by changing the grouping and sequencing attributes; this has the effect of appropriately changing the *ordinal* associated with each tuple. Let R be a table, g be the grouping attributes, and s be the sequencing attributes. $\Psi_{g,s}(R)$ partitions R based on g into a set of partitions, $P_1 \ldots P_n$, and then each P_i is sorted based on s. When grouping, all NULL values in a grouping attribute are considered equal (as in SQL), but if any sequencing attribute is NULL, the tuple is discarded.

We now describe how each tuple in a partition P_i is given an ordinal value; again, we emphasize that this is conceptual. The *ordinal* attribute can be used by selection operations, but is not preserved by other relational operators. All tuples with the first sequence value that appears in P_i are assigned ordinal value 1, the tuples with the next sequence value are assigned the ordinal 2, and so on. The last ordinal assigned to each partition P_i is called $LAST_i$. In this way, each distinct value of the sequencing attributes is assigned a unique ordinal value between 1 and $LAST_i$, and every ordinal in this range is assigned to some tuple.

For example, consider a table $R(g, t, x)$. The result of $\Psi_{g,t}(R)$ is:

g	t	x	ord
3	4	a	1
3	6	b	2
3	6	c	2
3	8	b	3
2	1	a	1
2	1	b	1
2	3	c	2
2	5	d	3
2	9	e	4
2	9	f	4

3.4. Shifting a Sequence

3.4.1. ShiftAll Operator

A basic sequence operation is to align the tuples of the sequence with other tuples at some relative or fixed offset in the sequence. For example, we can pair each tuple with

the next tuple in the sequence, or with the first tuple in the sequence.

The ShiftAll operator $\Delta_i(R)$ takes a sequence R and a relative ordinal value i and joins each tuple t of R with all the tuples of R in the same group as t that have an ordinal value = ordinal(t) + i. If no tuple with the appropriate ordinal is found in the group of t (i.e., shifting to an ordinal less than one or greater than *LAST*), then t is paired with NULL values. In other words:

$$\Delta_i(R) = \Psi_{R.g,R.s}(R \underset{\substack{R.g=R_2.g \wedge \\ R.ord+i=R_2.ord}}{\blacktriangleright\!\!\bowtie} R_2)$$

where g is the grouping attributes, s is the sequencing attributes, R_2 is a copy of R, and $\blacktriangleright\!\!\bowtie$ stands for left outer join.

We extend Δ so that it can take fixed as well as relative ordinals: when a fixed ordinal f is used (e.g., *FIRST*, to denote the first tuple), the expression $ord + i$ used in the outer join is replaced by f. This results in each tuple being paired with all the tuples at the given fixed ordinal position. We do not define the syntax here, but SRQL does include such constructs.

In general, we are interested in matching a tuple with tuples at several different ordinals, so we extend Δ to take a set of (fixed or relative) ordinals. Let $R_1, \ldots, R_k$ be copies of the sequence R. Now Δ is defined as:

$$\Delta_{\{i_1,\ldots,i_k\}}(R) = \Psi_{R.g,R.s}(\ldots \Psi_{R.g,R.s}(R \underset{C_1}{\blacktriangleright\!\!\bowtie} R_1) \ldots \underset{C_k}{\blacktriangleright\!\!\bowtie} R_k)$$

where each C_j is a predicate of the form:

$$(R.g = R_j.g) \wedge (R.ord + i_j = R_j.ord)$$

Taking $R(g, t, x, ord)$ from the example above, $\Delta_{\{-1,2\}}(R)$ is:

g	t	x	ord	t_{-1}	x_{-1}	t_{+2}	x_{+2}
3	4	a	1	NULL	NULL	8	b
3	6	b	2	4	a	NULL	NULL
3	6	c	2	4	a	NULL	NULL
3	8	b	3	6	b	NULL	NULL
3	8	b	3	6	c	NULL	NULL
2	1	a	1	NULL	NULL	5	d
2	1	b	1	NULL	NULL	5	d
2	3	c	2	1	a	9	e
2	3	c	2	1	b	9	e
2	3	c	2	1	a	9	f
2	3	c	2	1	b	9	f
2	5	d	3	3	c	NULL	NULL
2	9	e	4	5	d	NULL	NULL
2	9	f	4	5	d	NULL	NULL

3.4.2. Shift Operator

Sometimes we only want to match a tuple with the sequence value at some offset in the sequence, rather than the whole tuple. When this is the case, the Shift operator, δ, can be used to avoid the "cross-product effect" of Δ caused by duplicate sequence values (e.g., tuple (2,3,c,2) in the example above). δ is similar to Δ, except that it removes duplicate sequence values before joining. Let R be a sequence with grouping attributes g and sequencing attributes s. Let $T_1, \ldots, T_k$ be copies of the sequence $\Psi_{g,s}(\text{Distinct}(\pi_{g,s}(R)))$. We define δ as:

$$\delta_{\{i_1,\ldots,i_k\}}(R) = \Psi_{R.g,R.s}(\ldots \Psi_{R.g,R.s}(R \underset{C_1}{\blacktriangleright\!\!\bowtie} T_1) \ldots \underset{C_k}{\blacktriangleright\!\!\bowtie} T_k)$$

where each C_j is a predicate of the form:

$$R.g = T_j.g \wedge R.ord + i_j = T_j.ord$$

Like Δ, we can extend δ to take both fixed and relative ordinals.

Using the example above, $\delta_{\{-1,2\}}(R)$ =

g	t	x	ord	t_{-1}	t_{+2}
3	4	a	1	NULL	8
3	6	b	2	4	NULL
3	6	c	2	4	NULL
3	8	b	3	6	NULL
2	1	a	1	NULL	5
2	1	b	1	NULL	5
2	3	c	2	1	9
2	5	d	3	3	NULL
2	9	e	4	5	NULL
2	9	f	4	5	NULL

We note the following properties of ShiftAll and Shift. In what follows, R is a sequence with grouping attributes g and sequencing attributes s.

1. Shift with multiple offsets can be expressed as the composition of multiple Shift operators:
$$\delta_{\{i,j\}}(R) = \delta_i(\delta_j(R)) = \delta_j(\delta_i(R))$$

2. Shift with two offsets is not the same as the join (or outer join) of the two shifted sequences.
$$\delta_{\{i,j\}}(R) \neq \delta_i(R) \underset{\substack{g=g \wedge \\ ord=ord}}{\bowtie} \delta_j(R)$$

3. ShiftAll with multiple offsets is not the same as the composition of multiple ShiftAll operators:
$$\Delta_{\{i,j\}}(R) \neq \Delta_j(\Delta_i(R))$$

4. ShiftAll with two offsets is not the same as the join (or outer join) of the two shifted sequences:

$$\Delta_{\{i,j\}}(R) \neq \Delta_i(R) \underset{\substack{g=g \wedge \\ ord=ord}}{\bowtie} \Delta_j(R)$$

5. Shift and ShiftAll can be interchanged:

$$\Delta_I(\delta_J(R)) = \delta_J(\Delta_I(R))$$

6. Shift simply adds attributes to a sequence:

$$\Psi_{g,s}(\pi_R(\delta_i(R))) = R$$

7. ShiftAll not only adds attributes to a sequence, but can also duplicate tuples in the sequence:

$$\text{Distinct}(\pi_R(\Delta_i(R))) = \text{Distinct}(R)$$

3.5. Aggregate Operations

When applying an aggregate function on a sequence, a different aggregate value can result from each ordinal value of the sequence. For each ordinal, a section of the sequence, called a *window*, is created, and the aggregate function is applied to the window. For example, when computing a trailing weekly moving average of daily stock prices, the window is the current day plus the previous six days.

Because each ordinal value of the sequence can have its own window, we do not collapse groups when aggregating. Instead, the aggregate result is simply appended to the tuple. Aggregating in this manner allows us to use a different window for each aggregate function that is applied to the sequence.

The WindowAggregate operator, ω, allows a different window to be defined for each tuple t of a sequence R. It uses a selection predicate $p(t)$, which can refer to the value of the *ordinal* attribute in the current tuple t to select a subset of t's group in R. The result of the WindowAggregate operator has the same grouping set and sequencing list as its input, and contains one tuple per input tuple. For each input tuple t, the output contains the tuple:

$$< t, Agg(\pi_x(\sigma_{t.g=R.g \wedge p(t)}(R))) >$$

where Agg is an aggregate operator (e.g., *MIN, MAX, SUM, COUNT, AVG*), x is an attribute of R, and g is the grouping attributes of R. [2] [3]

Using our running example, $\omega(R, true, \textit{MAX}, x)$ is:

g	t	x	MAX(x)
3	4	a	c
3	6	b	c
3	6	c	c
3	8	b	c
2	1	a	f
2	1	b	f
2	3	c	f
2	5	d	f
2	9	e	f
2	9	f	f

If we let T be this result, notice that the SQL query:

```
SELECT g,MAX(x)
FROM R
GROUP BY g
```

is equal to Distinct$(\pi_{g,MAX(x)}(T))$.

ω is similar to, yet distinct from, the Φ operator defined by Chatziantoniou and Ross in [1]. The Φ operator was also introduced to define aggregation windows, although for different motivating problems. We defined ω because it allows a more natural treatment of SRQL.

As described in Section 4.3, SRQL currently restricts the use of ω to ensure efficient evaluation. We are investigating ways to efficiently incorporate additional functionality of ω.

4. The SRQL Language

The SRQL language enhances SQL to support queries that reflect sort order. Using the operators defined in the previous section, we can now define the syntax and semantics of SRQL. An implementation of SRQL is allowed to execute the query in any manner, as long as the semantics is preserved.

4.1. Shift Operators

SRQL allows tables to be ordered in the *FROM* clause so that we can match tuples of the sequence with other tuples at some relative or fixed offset in the sequence. The *SHIFT* function takes a (sequence) tuple variable and an offset and provides access to the sequencing attributes at that offset. Similarly, the *SHIFTALL* function provides access to all of the attributes, not just the sequencing attributes. For example:

```
SELECT S.t, S.x, SHIFTALL(S,-1).x,
       SHIFT(S,1).t
FROM R GROUP BY g SEQUENCE BY t as S
```

returns each tuple of S matched with all of the x values immediately before the tuple in S, and the single t value that occurs immediately after the tuple. If no x or t value is

[2] We note that the WindowAggregate operator can be defined using the other operators, but will not discuss this further.

[3] There is also a value-based version of WindowAggregate where $p(t)$ is allowed to reference the sequencing attributes.

found, a NULL value is used. If S does not have duplicate (g, t) values, then only one x value will occur immediately before the tuple.

To evaluate queries with *SHIFT* and *SHIFTALL*, the offsets from all of the *SHIFT* calls for a sequence are gathered into a set I, and the offsets from the *SHIFTALL* calls are gathered into a set J. The definition of the sequence is then replaced with calls to δ and Δ on that definition; the above query is equivalent to the expression:

$$\pi_{t,x,x_{-1},t_{+1}}(\Delta_{\{-1\}}(\delta_{\{+1\}}(\Psi_{g,t}(R))))$$

SRQL has two special values, *FIRST* and *LAST*, that can be used with *SHIFT* and *SHIFTALL* to get fixed ordinals. For example, *SHIFTALL*(S,*LAST*-3).x gets all x values associated the fourth to last ordinal. The only types of offsets allowed are $\pm i$, *FIRST*$+i$, and *LAST*$-i$, where i is an integer constant.

4.2. Joining Sequences

To illustrate the power of shifting, consider the following query: For each volcano eruption where the the most recent earthquake that was greater than 7.0 on the Richter scale, what was the name of the earthquake?

Volcano		Earthquake		
time	name	time	name	magnitude
3	v1	1	e1	8
4	v2	2	e2	2
5	v3	5	e3	8
8	v4	6	e4	9
9	v5	7	e5	8

```
SELECT V.name, E.name
FROM Volcano AS V,
     Earthquake SEQUENCE BY time AS E
WHERE E.time <= V.time
  AND (SHIFT(E,1).time > V.time
      OR SHIFT(E,1).time IS NULL)
  AND E.magnitude > 7
```

The result is:

V.name	E.name
v3	e3
v4	e5
v5	e5

Now consider a similar query: For each earthquake that was greater than 7.0 on the Richter scale, what was the next non-concurrent volcano eruption?

```
SELECT V.name, E.name
FROM Volcano SEQUENCE BY time AS V,
     Earthquake AS E
WHERE V.time > E.time
  AND (SHIFT(V,-1).time <= E.time
      OR SHIFT(V,-1).time IS NULL)
  AND E.magnitude > 7
```

The result is:

V.name	E.name
v1	e1
v4	e3
v4	e4
v4	e5

Since we expect this type of sequence join to be common, we created four predicate operators that express it more succinctly. $A.s$ SUCCEEDS $B.x$ means join a B tuple with the A tuple (or tuples if A has duplicate sequence values) having the *minimum* $A.s$ value that is greater than $B.x$. A must be sequenced by s. Similarly, $A.s$ PRECEDES $B.x$ says to join a tuple of B with the tuple(s) of A that have the *maximum* value of $A.s$ which is less than $B.x$. SUCCEEDS= and PRECEDES= allow $A.s$ to be equal to $B.x$. Using these predicates, the above queries become:

```
SELECT E.name, V.time
FROM Volcano AS V
     Earthquake SEQUENCE BY time AS E
WHERE E.time PRECEDES= V.time
  AND E.magnitude > 7

SELECT E.name, V.time
FROM Volcano SEQUENCE BY time AS V,
     Earthquake AS E
WHERE V.time SUCCEEDS E.time
  AND E.magnitude > 7
```

4.3. Aggregation

Moving aggregation operators are used for calculating an aggregate function (e.g., *average*) repeatedly on subsections of a sequence (called windows). For example, a 2-day moving average of expenses over a week will produce a list of values, starting with the average of expenses on Mon and Tue, followed by the average of expenses on Tue and Wed, and so on. The aggregate is calculated over the window as the window slides down the sequence.

Moving aggregates can be categorized as position-based or value-based. The position based operators ignore the actual distance between the sequencing values. For example, a positional window of (-3,0) includes the records corresponding to the three previous sequence values in the sequence plus the current sequence value. The value-based moving aggregates take into account the actual value of the position when calculating the window. Thus a

value window of (-3,0) would include only those records whose sequencing attribute value falls within the range (current_value−3, current_value).

Each of the normal aggregate functions (*MAX, MIN, SUM, COUNT, AVG*) can be used as a window aggregate function. The window of aggregation is specified in the *OVER* clause. The offsets to the *OVER* clause follow the same rules as the *SHIFT* operator, and allow one or both of the ends of the window to be fixed by using *FIRST* or *LAST*. The corresponding value based windows use the *OVER VALUES* clause. A moving aggregation with no *OVER* clause is the same as the corresponding normal aggregation over the entire sequence (i.e., the default window is defined to be all tuples in the same group).

Here is an example that illustrates the difference between the two classes of moving aggregate operators:

Example	
num	vol
1	100
2	200
3	90
5	120
7	50
8	120

The positional moving average is given by the following query:

```
SELECT num, AVG(vol) OVER 0 TO 1
FROM Example
SEQUENCE BY num
```

The equivalent algebra expression is:

$$\omega(\Psi_{\{\},num}(Example),\\ ord >= t.ord + 0 \wedge ord <= t.ord + 1,\\ AVG, vol)$$

and results in:

num	*AVG*(vol)
1	150
2	145
3	105
5	85
7	85
8	120

Similarly, the value based moving average is given by the following query:

```
SELECT num,
   AVG(vol) OVER VALUES 0 TO 1
FROM Example
SEQUENCE BY num
```

The equivalent algebra expression is:

$$\omega(\Psi_{\{\},num}(Example),\\ num >= t.num + 0 \wedge num <= t.num + 1,\\ AVG, vol)$$

and results in:

num	*AVG*(vol)
1	150
2	145
3	90
5	120
7	85
8	120

Consider the record at position 3 of the input sequence. The position based moving average looks ahead one step in the *given sequence* (at the record at position 5) to calculate the average of 90 and 120. However, the value based moving average looks ahead one position in the *domain* of the sequencing attribute to calculate the average of the values corresponding to positions 3 and 4 which is just 90, because records corresponding to non-existent positions are not considered in the aggregation.

Note that we have defined aggregation on a sequence to always produce the same number of tuples as the original sequence. In other words, sequence aggregates only add attributes to the sequence, and the entire original tuple can be placed in the *SELECT* clause.

4.3.1. Aggregates on Composite Sequences

GROUP BY and *SEQUENCE BY* may be combined to create a composite sequence which may be thought of as a set of sequences. Consider the following example relation *Sales(product, batch, volume)* and suppose we need to find the 2-batch moving average of volumes sold for each product. This may be expressed in SRQL as:

```
SELECT product, batch,
       AVG(volume) OVER 0 TO 1
FROM Sales
GROUP BY product
SEQUENCE BY batch
```

The equivalent algebra expression is:

$$\omega(\Psi_{product,batch}(Sales),\\ (ord >= t.ord + 0) \wedge (ord <= t.ord + 1),\\ AVG, volume)$$

and results in:

product	batch	volume	*AVG*(volume)
bolts	1	10	15
bolts	2	20	55
bolts	5	90	90
nails	43	50	75
nails	44	100	80
nails	45	60	60
tacks	24	100	90
tacks	25	80	70
tacks	26	60	50
tacks	27	40	40

Clearly the bolts have done well with each successive batch while tacks have lost their ground, and the picture with nails is not so clear!

4.3.2. Variants of Aggregation

Another variant of the aggregate operations is cumulative functions. These calculate "running" sums, averages etc. over the entire sequence. Cumulative functions are specified using the *CUMULATIVE* keyword ahead of the appropriate aggregation function. When *CUMULATIVE* is specified with *GROUP BY*, the function only accumulates within each partition.

Cumulative aggregates introduce a number of interesting syntactic short-cuts:

1. *CUMULATIVE* is an just an abbreviation for *OVER FIRST TO* 0, and traditional (non-moving) aggregates are equivalent to *OVER FIRST TO LAST*.
2. We define *RANK()* as *CUMULATIVE COUNT(*)*.
3. We define *PERCENTILE()* as *RANK()* / *COUNT(*)* * 100.
4. We define *QUARTILE()* as $\lceil PERCENTILE()/25 \rceil$.

4.3.3. A Note on Duplicates

Since a sequence is just a sorted relation, duplicates still need to be handled. Each position in the sequence is treated as a set of one or more records. All operations described above hold in the presence of duplicates. Hence a window of (0,0) is well defined for all moving aggregates; it is essentially a window on the set of values for each position. For example, "*COUNT*(*) *OVER* 0 *TO* 0" gives a count of the number of duplicates for each position in the sequence.

4.3.4. *WITH*, *WHERE*, and *HAVING*

We introduce a new clause, the *WITH* clause, and extend the *HAVING* clause to deal with sequences. The *WITH* clause specifies the selection condition to be applied to each record *within a window* before the aggregation is done. When using position based aggregates, this is different from specifying a selection condition in the *WHERE* clause: If the selection condition is specified in the *WHERE* clause, then certain records may be eliminated and hence the window for the neighboring records would change. When we need the window to cover the original set of records and the selection to be applied within this window the *WITH* clause is required.

The following example illustrates the difference between *WITH* and *WHERE*. Consider the sequence *Sales(day, volume, profits)* and suppose we need to find the 2-day moving average of the profits made such that only the sales of volume greater than 100 are significant (the rest should be omitted from the aggregation):

```
SELECT day, AVG(profits) OVER 0 TO 1
FROM Sales
SEQUENCE BY day
WITH volume > 100
```

The equivalent algebra expression is:

$$\omega(\Psi_{\{\},day}(Sales), \\ (ord >= t.ord + 0) \wedge (ord <= t.ord + 1) \\ \wedge (volume > 100), AVG, profits)$$

day	volume	profits	*AVG*(profits)
Mon1	100	10	20
Tue1	200	20	20
Wed1	90	30	40
Fri1	110	40	40
Mon2	50	10	20
Tue2	120	20	20

Suppose we replace the *WITH* clause in the above query with a similar *WHERE* clause:

```
SELECT day, AVG(profits) OVER 0 TO 1
FROM Sales
WHERE volume > 100
SEQUENCE BY day
```

This changes the window of aggregation and hence the result, shown below, corresponds to the query: For all days with volume greater than 100, find the 2-day moving average of the profits made.

day	volume	profits	*AVG*(profits)
Tue1	200	20	30
Fri1	110	40	30
Tue2	120	20	20

Next, we consider the *HAVING* clause. Just as in SQL, predicates in the *HAVING* clause are evaluated after aggregation is complete, and are applied to the table computed

using the grouping, sequencing and aggregation steps. Of course, unlike SQL, there may be several tuples per group if the *SEQUENCE BY* clause is used. Let us replace the *WITH* clause from above with a similar *HAVING* clause:

```
SELECT day, AVG(profits) OVER 0 TO 1
FROM Sales
SEQUENCE BY day
HAVING volume > 100
```

Now the window of aggregation is the same as in the first query, but all tuples are included in the moving average, and those tuples that have volume < 100 are removed after aggregation. This corresponds to the query: Find the 2-day moving average of the profits made and report those that had volume > 100. Note that just like in SQL, aggregate functions, whether moving or not, can be placed in the *HAVING* clause but not the *WHERE* clause.

day	volume	profits	*AVG*(profits)
Tue1	200	20	25
Fri1	110	40	25
Tue2	120	20	20

4.4. Summary of Evaluation

In general, a SRQL query block looks like this:

```
SELECT <expr list>
FROM <table/sequence list>
WHERE <predicate>
GROUP BY <expr list>
SEQUENCE BY <expr list>
WITH <predicate>
HAVING <predicate>
ORDER BY <expr list>
```

and the order of evaluation is:

1. *SHIFT* and *SHIFTALL* are moved to the *FROM* clause.
2. All expressions in the *FROM* clause are evaluated, including sequencing and shifting.
3. If there is no *SEQUENCE BY* clause, then proceed as in SQL; otherwise continue.
4. Take the cross-product of all the tables in the *FROM* clause.
5. The *WHERE* clause is evaluated.
6. The *GROUP BY* and *SEQUENCE BY* clauses are used to define a sequence.
7. For each tuple (and each aggregate), a window is formed using the *GROUP BY*, *SEQUENCE BY*, *OVER*, *OVER VALUES*, and *WITH* clauses. Then the window is aggregated.
8. The *HAVING* clause is evaluated.

5. Performance of SRQL

In this section, we present the results of our implementation of SRQL, which is still an ongoing effort. The guiding principle behind the design of SRQL was that the language must be implementable with minimal extensions to a standard relational DBMS at both the query optimization and query evaluation levels.

The results we present here are illustrative of SRQL's capabilities, rather than a thorough performance evaluation. In particular, we do not provide a comparison of our implementation with other systems such as RISQL, but merely aim to illustrate that our proposed language extensions in SRQL may be efficiently implemented. In most cases, the cost of the query is dominated by the cost of scanning or sorting the relation. The optimizer uses catalog information (or other information from its execution plan) to determine whether the relation is already sorted, whether there is an index on the sequencing attribute, or whether the relation needs to be sorted.

Many moving aggregates can be computed incrementally using a small cache of tuples. Consider the following query with a window of size three:

```
SELECT age, AVG(salary) OVER 0 TO 2
FROM EMPLOYEE
SEQUENCE BY age
```

Initially, tuples covering three distinct values in the sequencing field are scanned in and added to the cache (which is essentially a queue). The average of the tuples in the cache is calculated and output along with the first value of the sequencing attribute. Next, the window is slid down by one value: all tuples of the next value are read into the cache, and all tuples of the first value in the cache are thrown out.

To demonstrate that moving aggregates can be efficiently computed using this caching technique, we timed the execution of a number of queries. Each of the queries are posed on the relation "Sales that has three relevant fields *(batch: Integer, price: Integer, date: Date)* and seven integer fields that are not referenced in the queries. We consider the following queries:

Scan: `SELECT * FROM Sales`

Projection: `SELECT price FROM Sales`

Selection 1:

```
SELECT * FROM Sales
WHERE price > 100
```

Sort:

```
SELECT * FROM Sales
ORDER BY price
```

Aggregate:

```
SELECT AVG(price) FROM Sales
```

Multiple Aggregate:

```
SELECT AVG(price), SUM(cost)
FROM Sales
```

Grouping:

```
SELECT batch, SUM(price)
FROM Sales GROUP BY batch
```

Moving Aggregate:

```
SELECT batch,
 SUM(price) OVER 0 TO 2
FROM Sales SEQUENCE BY batch
```

Multiple Moving Aggregates:

```
SELECT batch,
       SUM(price) OVER 0 TO 2,
       AVG(cost)  OVER 0 TO 2
FROM Sales SEQUENCE BY batch
```

Composite Sequence:

```
SELECT batch, date,
       AVG(price) OVER 0 TO 2
FROM Sales
GROUP BY batch SEQUENCE BY date
```

Duplicate Count:

```
SELECT batch,
       COUNT(cost) OVER 0 TO 0
FROM Sales SEQUENCE BY batch
```

Selection 2:

```
SELECT batch, SUM(price) OVER 0 TO 2
FROM Sales WHERE price > 100
SEQUENCE BY batch
```

These queries were run against three tables of sizes 1MB, 10MB and 100MB (40,000 tuples, 400,000 tuples, and 4,000,000 tuples respectively). We allocated memory for 100,000 tuples during the in-memory sorting phase of the external sort. Selectivity of the predicate $price > 100$ is 1/20. All the fields of the relation Sales are uniformly distributed with 20 duplicate values for each attribute. The results of our experiments are shown in the following table (all values are in seconds).

Query	1MB	10MB	100MB
Scan	1.05	8.86	79.58
Projection	0.81	6.65	68.95
Selection 1	0.94	7.31	69.05
Sort	3.70	31.98	318.65
Aggregate	0.93	6.83	71.16
Multiple Aggregate	0.74	6.98	71.60
Grouping	1.65	6.99	72.32
Moving Aggregate	0.84	8.70	82.22
Mult. Moving Aggregate	0.94	9.18	92.87
Composite Sequence	1.25	11.02	110.26
Duplicate Count	0.99	7.41	74.30
Selection 2	0.87	6.84	70.58

From these results, the following key observations may be made:

- The smaller output size of a projection makes it cost less than a scan. For the same reason, grouping takes less time than a sort.
- The cost of moving aggregates is just the cost of sorting and scanning without the overhead of writing the entire relation out. If the table is stored in sorted order, the sorting phase may be omitted, which is the case in our experiments.
- Calculating multiple (moving) aggregates in the same query does not have a significant effect on the performance since they are evaluated during the same pass over the sequence.
- Composite sequences may be evaluated at a slightly higher cost than grouping. Once the relation is grouped, the moving aggregate for each group is computed in a single pass over the relation.
- Our implementation exhibits scalable performance across a large range of data set sizes.
- As one might expect, pushing down selections is a useful strategy for sequence queries too.

These numbers are not intended to be a comprehensive performance evaluation of SRQL. Rather, they demonstrate that SRQL is implemented in an efficient and scalable manner, and show that query cost is comparable to the costs of scanning and sorting for the (broad class of) sequence queries where we would expect this to be the case. We expect the behavior of the remaining SRQL operators that we have not presented results for (in particular, queries involving joins) to follow the same pattern. For more detailed experimental results we refer you to the related work on SEQ presented in [9].

6. Related Work

Object-relational (O-R) systems handle sequences by considering them as another ADT. Sequences are stored as objects in the database and methods are provided for performing operations on sequences. While this approach is attractive for providing extensibility, the treatment of sequences in this fashion may not be optimal. In particular, since each invocation of a method for the sequence ADT is typically executed independently of other methods, certain queries requiring interaction of these methods may not be optimized. This point was experimentally demonstrated in the SEQ system [9].

SEQ extended the ADT approach of O-R systems by treating the sequence type as an enhanced ADT (EADT) with its own query optimizer and evaluator. This allowed the SEQ optimizer to consider interactions of different methods (and properties such as associativity and commutativity of sequence operators) when finding an optimal plan.

SRQL is based on the SEQ system's query language (SEQUIN). However, unlike SEQ, which deals with arbitrary EADTs, SRQL considers only relations and sequences. The motivation for the design of SRQL is to identify simple language extensions to SQL that can support queries on mixtures of sequences and relations, and that can be implemented with minimal extensions to conventional RDBMSs.

Red Brick System's RISQL (Red Brick Intelligent SQL) has an approach similar to our own in extending SQL to handle sequence data. However, based on the limited information available to us [4], SRQL is overall a richer language and our approach to optimization is more comprehensive. On the other hand, RISQL contains a large number of statistical functions for business applications, which are presently lacking in SRQL. In particular, RISQL seems to focus on extending SQL to handle more easily certain queries arising in business databases whereas SRQL does not make any such assumption about the domain of the application.

TSQL [10] is an extension of SQL to handle temporal databases. While our work on sequences is applicable to temporal sequences, we have not directly addressed issues of temporality.

7. Conclusion

We have presented an approach to handling sequence queries by considering a sorted relation as a sequence and extending SQL with features to exploit the sort order. We have shown that it is possible to express a large class of sequence queries very naturally using SRQL. Moreover, we have demonstrated that it is possible to evaluate these queries very efficiently with a few simple extensions to the standard relational query optimizer and evaluation engine. The performance numbers presented in this paper are based upon an implementation of SRQL in the DEVise system being developed at UW-Madison.

[4] There is no published description of RISQL, and our knowledge of the language is based on the white paper on Red Brick's home pages [7], and on discussions with Donovan Schneider at Red Brick.

References

[1] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *Proceedings of the 22nd VLDB Conference*, Mumbai, India, 1996.

[2] M. Gruber. *SQL Instance Reference*. Sybex, Alameda, CA, 1993.

[3] Illustra Information Technolgies, Inc., 1111 Broadway, Suite 2000, Oakland, CA 94607. *Illustra's User Guide*, June 1994.

[4] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *TODS*, 20(3):288–324, Sept. 1995.

[5] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: Integrated querying and visual exploration of large datasets. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1997.

[6] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *TODS*, 16(3):513–534, Sept. 1991.

[7] Red Brick Systems, Inc. *Decision-Makers, Business Data and RISQL*, August 1996. White paper. http://www.redbrick.com/rbs-g/whitepapers/risql_wp.html.

[8] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 430–441, May 1994.

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

[10] Snodgrass et al. TSQL2 language specification. *ACM SIGMOD Record*, 23(1):65–86, March 1994.

A. Motivation Queries Expressed in SRQL

1. Find the leading weekly moving average of the Dow Jones Industrial Average, given a table *DJIA(date, close)*.

```
SELECT date, AVG(close) OVER 0 TO 6
FROM DJIA
SEQUENCE BY date
```

2. Find the trailing weekly moving average (i.e., the average for the past week, for each date) for each stock, given a table *Stocks(date, symbol, close)*. Order the resulting sequences by stock name, date. (The result can be thought of as a relation with fields *symbol, date, average*, sorted on the composite key *(symbol, date)*.)

```
SELECT symbol, date,
       AVG(close) OVER -6 TO 0
FROM Stocks
GROUP BY symbol
SEQUENCE BY date
ORDER BY symbol, date
```

3. For each day in '97, find the ten cheapest stocks.

```
SELECT symbol, date, RANK()
FROM Stocks
GROUP BY date SEQUENCE BY close
HAVING RANK() <= 10
```

4. For each day in '97, find the ten most expensive stocks.

```
SELECT symbol, date, RANK()
FROM Stocks
GROUP BY date SEQUENCE BY close
HAVING RANK() > LAST() - 10
```

5. Compute the percent change of each stock during 1997, and then find stocks that were in the top 5%.

```
CREATE VIEW Change AS
SELECT S.symbol,
  SHIFTALL(S,LAST).change /
  SHIFTALL(S,FIRST).change - 1
    as pct_change
FROM (SELECT * FROM Stocks
      WHERE year(date) = 1997)
  GROUP BY symbol
  SEQUENCE BY date AS S
WHERE ORDINAL(S) = FIRST(S);

SELECT symbol, PERCENTILE() as pct
FROM Change
SEQUENCE BY pct_change
HAVING pct < 5;
```

6. Find the of dates of stocks where the leading weekly average is greater than 1.2 times the trailing weekly average, given the Stocks table.

```
SELECT symbol, date
FROM Stocks
GROUP BY symbol SEQUENCE BY date
HAVING AVG(close) OVER 0 TO 6 >
   1.2 * AVG(close) OVER -6 TO 0
```

7. For each week, find the stock in which a given customer (say Joe) had the most invested. In addition to the Stocks table, a table *Trans(customer, symbol, date, shares)* records the change in shares for a given customer, symbol, and date.

```
CREATE VIEW Shares AS
SELECT customer, symbol, date
  CUMULATIVE SUM(shares) AS shares
FROM Trans
GROUP BY customer, symbol
SEQUENCE BY date;

CREATE VIEW Invested AS
SELECT C.customer, C.symbol, C.date
 C.shares * S.close as value
FROM Shares SEQUENCE BY date AS C,
     Stocks AS S
WHERE C.date PRECEDES= S.date;

SELECT WEEK(date), symbol, value
FROM Invested
WHERE customer = "Joe"
GROUP BY WEEK(date) SEQUENCE BY NULL
HAVING value = MAX(value);
```