$\mathcal{SEQ}$: A Model for Sequence Databases [*]

Praveen Seshadri   Miron Livny   Raghu Ramakrishnan

Computer Sciences Department
University of Wisconsin, Madison, WI 53706, USA
praveen,miron,raghu@cs.wisc.edu

Abstract

This paper presents the $\mathcal{SEQ}$ model which is the basis for a system to manage various kinds of sequence data. The model separates the data from the ordering information, and includes operators based on two distinct abstractions of a sequence. The main contributions of the $\mathcal{SEQ}$ model are: (a) it can deal with different types of sequence data, (b) it supports an expressive range of sequence queries, (c) it draws from many of the diverse existing approaches to modeling sequence data.

1 Introduction

Data over ordered domains such as time or linear position often reflect the underlying order, and can be naturally viewed as a sequence. Applications involving sequence data have typically led to a specialized model and a stylized language. In this paper, we present the $\mathcal{SEQ}$ model which is the basis for a system we are building to manage and manipulate diverse kinds of sequence data.

It is reasonable for a database user to ask, *"What kinds of queries can I ask on sequence data, and how can I express these queries ?"*. The motivation for this research is to answer this question, so that subsequent research can focus on the question, *"How can these queries be efficiently executed?"*. Given the large number of existing models for specific kinds of sequence data (see Section 4), a natural reaction is to wonder what good yet another model would do. The answer is twofold. Firstly, while each specialized data model may be as expressive as $\mathcal{SEQ}$ in its particular problem domain, the importance of a uniform model is that optimization and implementation techniques that are developed in one problem domain can be applied to other domains. Secondly, few other models have been carried through to a concrete implementation. Our research is part of a larger ongoing effort to build a sequence database system. In [14], we presented query optimization techniques based on a limited model of sequences. The constructs in our model have therefore been constrained by practical considerations of efficiency and ease of implementation.

[*] Praveen Seshadri was supported by IBM Research Grant 93-F153900-000 and an IBM Graduate Fellowship Award. Miron Livny and Raghu Ramakrishnan were supported by NASA Research Grant NAGW-3921. Raghu Ramakrishnan was supported by a Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563.

2 Model of Sequences

We begin with definitions of data records, and operations on them. We then associate a set of records with a totally ordered domain (like the integers) to define sequences as orderings of data records.

We assume the existence of a set T_{Basic} of atomic types. The type domain of a basic type is the (possibly infinite) set of values of that type. A *record schema* is defined as the set $R =< A_1{:}T_1, ... , A_N{:}T_N >$ for some finite N. Each of the T_is are types in T_{Basic}, and each A_i is a named *attribute*. The number N of attributes is called the *arity* of the record schema. The *type domain* T_R of R is $(T_1 \times T_2 \times ... \times T_N)$. A *record* of schema R is an element of T_R. Notationally, a record is represented by a set of attribute values between the symbols $<>$. Given two records $r_a :< a_1, ..., a_m >$ and $r_b :< b_1, ..., b_n >$, the concatenation $< a_1, ..., a_m, b_1, ..., b_n >$ is denoted as $r_a.r_b$ which is equal to $r_b.r_a$.

Definition 2.1 Let S be a set of records of schema R, and let O be a countable totally ordered domain. A many-to-many relationship O_S from O to T_R is an *ordering of S by O* iff:

- For every record $r_i \in$ S, there exists some p $\in$ O such that $O_S(p, r_i)$.
- $O_S(p, r_i) \Rightarrow$ p $\in$ O and $r_i \in S$.

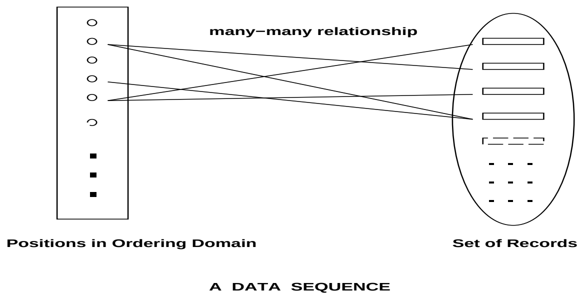

Figure 1: Mathematical Representation of a Sequence

O is called the *ordering domain*, and its elements are called *positions*. The first part of the definition specifies that every record in S is associated with some position. The second part specifies that no record outside of S is associated with any position.

Definition 2.2 A *sequence* Seq is a 3-tuple $< S, O, O_S >$ where

- S is a set of records of schema R_S.
- O is an ordering domain.
- O_S is an ordering of S by O.

Figure 1 provides a visual interpretation of this definition of a sequence. The ordering is called a sequence ordering. Note that each position could be related to more than one record, and vice versa. While every record has to map to at least one position, the converse is not true.

2.1 Classes of Operators

Two equivalent definitions of sequence $< S, O, O_S >$ are:

- $O_S \equiv$ a function PF that defines for every position p $\in$ O a set of records PF(p) such that

  $\forall$ p $\in$ O $\forall$ r $\in T_R$, r $\in$ PF(p) $\Leftrightarrow O_S$(p,r).

- $O_S \equiv$ a function RF that defines for every record r in T_R, a set of positions RF(r) such that

  $\forall$ p $\in$ O $\forall$ r $\in T_R$, p $\in$ RF(r) $\Leftrightarrow O_S$(p,r) is true.

The first representation is called *Positional* while the second representation is called *Record-Oriented* . Each representation provides a distinct perspective of a sequence that influences the choice of query operators.

A sequence operator defines a new sequence using existing sequences. The existing sequences used are the *input sequences* of the operator, and the new sequence defined is the *output sequence*. The following are characteristics of all the operators in the data model:

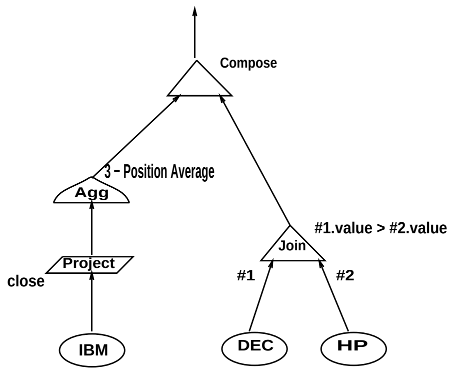

Figure 2: Query Using Positional Operators

1. Every operator defines an output *sequence*, possibly using input sequences. The number of input sequences defines the *arity* of the operator.
2. It is possible to compose multiple operators with the output of one operator becoming the input of another operator.
3. If the arity of an operator is greater than one, all its input sequences must have the same ordering domain.

A query is an acyclic composition of operators. We now define each type of operator, along with an informal explanation. An operator M that uses an auxiliary value V (for instance, a selection operator uses an auxiliary function that acts as a selection condition) is denoted as M[V].

2.2 Positional Operators

Positional operators are based on the view of the sequence as a set of records associated with each position in the ordering domain. The input and output sequences have the same ordering domains. The operators are defined by specifying the PF of the output sequence.

Transform Operators: M[Fn]: Fn takes a position and a set of input records at that position, and returns a set of output records. As examples, Fn could apply a selection condition or a projection operation to the input records. If the input sequence of the unary transform operator M is $< S_1, O, PF_1 >$, then the output sequence is $< S, O, PF >$, where for all positions p in O, $PF(p) = Fn(p, PF_1(p))$ and S is the union of the records in PF(p).

Binary Operators: M[Fn]: Fn takes a position and the two sets of records at that position, and returns a set of output records. Fn could be any of the relational operators: union, intersection, Cartesian product, set difference or join. As examples, the Positional Compose(Join) operator performs the Cartesian product(Join) of the sets of input records. If the input sequences of the binary operator M are $< S_1, O, PF_1 >$ and $< S_2, O, PF_2 >$, then the output sequence is $< S, O, PF >$, where for all positions p in O, $PF(p) = Fn(p, PF_1(p), PF_2(p))$ and S is the union of the records in PF(p).

Offset Operators: M[j]: There are two kinds of unary offset operators. The *position offset* operator returns an output sequence which is identical to the input sequence except that the entire ordering domain has been "shifted" by the specified number of positions. If the input sequence is $< S_1, O, PF_1 >$, then the output sequence is $< S, O, PF >$ where for all positions p in O, $PF(p) = PF_1(p1)$ and the number of positions between p and p1 is j. The *value offset* operators are similar, except that they ignore positions that map to no records. If the input sequence is $< S_1, O, PF_1 >$, then the output sequence is $< S, O, PF >$ where for all positions p in O, $PF(p) = PF_1(p1)$ and the number of non-null positions between p and p1 is j. The *Next* and *Previous* operators are special cases of value offset operators with offsets of 1 and -1 respectively. In both cases, S is the union of the records in PF(p) for all positions p.

Aggregate Operators: M[agg_func,agg_pos]: The *agg_pos* function is used to select a subset of the positions of the ordering domain, based on the current position. The function *agg_func* is an aggregate function over the union of records in PF(p) at the selected positions p. If the input sequence of the unary aggregate operator is $< S_1, O, PF_1 >$, then the output sequence is $< S, O, PF >$, where for all positions p in O, S is the union of the records in PF(p) and $PF(p) = agg\_func(\{r | p1 \in agg\_pos(p) \wedge r \in PF_1(p1)\})$. For example, an aggregate to compute the running 3-day average of a daily sequence would have agg_func $\equiv$ Avg and agg_pos(pos) $\equiv \{p | pos \geq p \geq pos - 2\}$.

While the operators are defined in general terms, a particular implementation might choose to provide a subset of these operators with predefined functions.

Example 2.1 Assume that daily sequences of stock data are available for IBM, DEC and HP. A query could ask for the 3-day average of the close of IBM stock values when the value of DEC is greater than that of HP. The query is visualized using a graphical notation in Figure 2. The Join operator on the right side of the query determines the positions (in this case, days) on which the value of DEC was greater than the value of HP. The left side of the query generates the sequence corresponding to the 3-day moving average of the close of IBM. Finally, the Compose operator ensures that the answer contains the value of the moving average for only the desired days. Such figures will be used throughout this paper to illustrate examples. As in this figure, obvious projections that clutter the presentation of the figure are omitted.

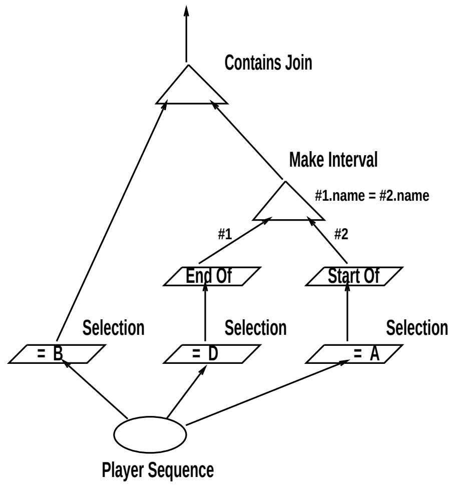

Figure 3: Query Using Record-Oriented Operators

2.3 Record-Oriented Operators

Record-Oriented operators are based on the view of a sequence as a set of positions associated with each record in the type domain. The operators are defined by specifying the RF of the output sequence. The input and output sequences have the same ordering domains.

Transform Operators: M[Fn_{rec},Fn_{pos}]: Both Fn_{rec} and Fn_{pos} take an input record and its associated set of positions as arguments; Fn_{rec} returns an output record and Fn_{pos} returns the set of positions to which that output record maps. If the input sequence of the unary transform operator M is $< S_1, O, RF_1 >$, then the output sequence is $< S, O, RF >$, where S is the union of the records r returned by $Fn_{rec}(r_1, RF_1(r_1))$ for all records r_1 in S_1, and RF is defined by $RF(r) = Fn_{pos}(r_1, RF_1(r_1))$. As examples, Start-Of (End-Of) is a unary operator that associates each output record with the first (last) of the set of positions that the corresponding input record maps to.

Binary Operators: M[Fn_{rec},Fn_{pos}]: If the inputs of the binary operator M are $< S_1, O, RF_1 >$ and $< S_2, O, RF_2 >$, then the output sequence is $< S, O, RF >$, where S is the union of the records r returned by $Fn_{rec}(r_1, RF_1(r_1), r_2, RF_2(r_2))$ for all records r_1 in S_1 and r_2 in S_2. The RF of the output sequence is defined by $RF(r) = Fn_{pos}(r_1, RF_1(r_1), r_2, RF_2(r_2))$. It should be noted that various kinds of "sequence joins" may be specified as binary Record-Oriented operators whose Fn_{rec} is a conditional concatenation. For instance, if Fn_{pos} is an intersection of the sets of positions of the two records, the resulting binary operator is an "Overlap Join". If Fn_{pos}

checks for containment of one of the position sets in the other, we get different variants of the "Contain-Join"[11]. Similarly, the Make-Interval operator is a sequence join where Fn_{pos} gives the minimal interval that includes all the positions of both the input records.

Aggregate Operators: M[agg_func,agg_rec]: The Groupby Aggregate operators are unary operators defined by two functions. One function *agg_rec* is used to partition the input records into disjoint sets. The other function *agg_func* is an aggregate function over each partition set of input records. There is one output record corresponding to each partition, whose value is determined by applying agg_func to the partition. This output record maps to all positions that mapped to input records in the partition (i.e. RF(r) $= \cup$ RF(r_i) , $r_i \in$ partition(r)).

While the operators are defined in general terms, a particular implementation might provide a subset of these operators with predefined functions. The following example demonstrates a simple sequence query that can be expressed using Record-Oriented operators.

Example 2.2 Assume that a temporal database has a sequence of records that place a player in a particular league of teams for each month. Consider a query that asks for those players who were promoted from a D team to an A team, while some other player remained on a B team. The query is visualized in the graphical notation in Figure 3. The left side of the query finds the sequence of players on B teams. The right side of the query constructs interval sequences corresponding to the transition of the same player from a D team to an A team. Finally, the Contains Join returns the desired answer. Note that this query formulation assumes that players are never demoted! This query was adapted from [16].

2.4 Duality of Sequences

Some queries involving Positional operators are difficult to express using Record-Oriented operators, and vice versa. For instance, it would be difficult to express the Previous operator in the Record-Oriented model (and vice-versa with the Precede and Contains Joins)[1]. For this reason, we include both the Record-Oriented and the Positional operators in the data model. The $\mathcal{SEQ}$ model therefore does not merely provide the Positional abstraction or the Record-Oriented abstraction, but allows *both* to coexist simultaneously in specifying a query. This is a recognition of the fact that a sequence is at once "Positional " as well as "Record-Oriented ". This duality has also been recognized in the context of temporal databases by [20], who define a temporal calculus based on these two abstractions.

[1]In fact, in the absence of aggregate operators, this is not merely difficult, but impossible. Since our definitions of aggregates are very broad, it is possible in either kind of aggregate operator to treat the entire set of records as the aggregate set, and introduce an arbitrary degree of complexity into the function that actually computes the aggregate. However, we view this as unrealistic, since an implementation would probably provide a limited collection of aggregate functions.

3 Extensions to the Data Model

We now present two extensions to the basic model of sequences that result in the $\mathcal{SEQ}$ data model.

3.1 Sequence Zooming

The first extension involves the Collapse and Expand operators. The motivation for this extension is as follows. Consider a sequence of stock data records, with values recorded every trading day. The ordering domain of this data sequence is trading days. However, someone querying this sequence might want to view the data at a weekly granularity, thus effectively modifying the ordering domain to weeks instead. Since the mapping from days to weeks is well defined, it should be possible to allow such "zoom" operations.

Definition 3.1 Consider two totally ordered domains O_1 and O_2. A *collapse* γ is a many-to-one onto function from O_1 to O_2. The inverse mapping γ^{-1} is called an *expansion* from O_2 to O_1.

Examples of collapses include days to months, months to years, feet to yards, etc.

Definition 3.2 Let $Seq_1 =< S, O_1, O_{S_1} >$. Let O_{S_1} be represented by the positional function PF_1. Consider another totally ordered domain O_2. Let γ be a collapse from O_1 to O_2. A sequence $Seq_2 =< S, O_2, O_{S_2} >$ is a γ-*collapse* of Seq_1 iff O_{S_2} is represented by positional function PF_2 defined as follows:

$\forall p_2 \in O_2$, $PF_2(p_2)$ = {r | $p_1 \in O_1 \wedge p_2 = \gamma(p_1) \wedge r \in PF(p_1)$ }

Definition 3.3 Let $Seq_2 =< S, O_2, O_{S_2} >$. Let O_{S_2} be represented by the positional function PF_2. Let γ^{-1} be an expansion from O_2 to another totally ordered domain O_1. A sequence $Seq_1 =< S, O_1, O_{S_1} >$ is a γ-*expansion* of Seq_2 iff O_{S_1} is represented by positional function PF_1 defined as follows:

$\forall p_1 \in O_1$, $PF_1(p_1)$ = $PF_2(p_2)$ where $p_2 = \gamma(p_1)$

Note that γ-expansion(γ-collapse(Seq)) is not equivalent to Seq. For example, if a daily sequence is collapsed to a weekly sequence, the information of which record mapped to which day is not retained.

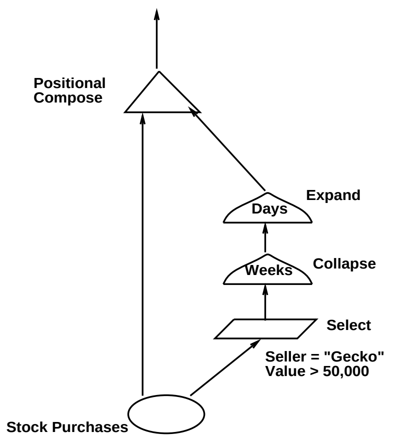

Figure 4: Query Using Collapse and Expand

The Collapse and Expand operators are unary operators based on the definitions of sequence collapse and expansion. These operators take an argument specifying a collapse (expansion) and derive an output sequence by appropriately collapsing (expanding) the input sequence. It is expected that certain "well-known" collapses and expansions on the ordering domain are pre-defined, and can be used by the query operators. Here, we present an example of a query that uses them.

Example 3.1 Assume that a sequence of stock market transactions is available, recorded on a daily basis. The following query needs to be posed: *select all daily stock purchases during all weeks in which Gordon Gecko sold more than $50,000 of stock in a single transaction*. Figure 4 shows the representation of the query. A selection is applied to the sequence of stock purchases to select the transactions of Gecko of value greater than 50,000. This sequence is then collapsed to a weekly sequence, in which a week that maps to no records implies Gecko had no major transaction that week. This is then expanded back to a daily sequence, in which all days belonging to such weeks map to no records. In essence, this provides a selection of those days that are of interest to the query. Finally this is composed with the original stock sequence, thereby producing the desired answer. Project operators have been omitted in order to keep the figure simple.

3.2 Sequence Group

So far, all operators have manipulated input sequences and produced output sequences. We now introduce the concept of a *sequence group* and operators that manipulate sequence groups. *Sequence groups along with the set of operators on them constitute the $\mathcal{SEQ}$ data model*. The motivation for this enhancement is that a model for sequence databases has to deal with the concept of collections of sequences. This also expands the scope of $\mathcal{SEQ}$ to include an important class of sequence applications: those involving temporal data that are currently modelled using temporal relational databases. Other important consequences of this extension are that relations fall out as a special case of sequence groups, and that nested queries can be cleanly modeled.

Definition 3.4 A *labelled sequence* is a pair L::S where S is a sequence and L is a record of zero or more fields. L is called a *sequence label*. A *sequence group* is a set S_L of labelled sequences in which all the sequences in S_L have the same schema(label and record). The sequence label acts as the key of each labelled sequence (therefore, no two labelled sequences in a sequence group can have the same label).

The label represents the attributes of a sequence that do not vary with position. These can also be used to uniquely identify the sequence. A *sequence group operator* manipulates input sequence groups to define an output sequence group. There are two categories of operators; one category is derived from the operators already defined on sequences, while there are five new operators in the other category.

Definition 3.5 Every operator Op_S defined on sequences has a counterpart Op_{SG} of the same arity defined on sequence groups, based on the following rules:

- If Op_{SG} is unary, for every sequence $L_i :: S_i$ in the input sequence group, there is a sequence $L_i :: Op_S(S_i)$ in the output sequence group if $Op_S(S_i)$ is non-empty. Else, there is no sequence in the output with label L_i.

- If Op_{SG} is binary, for every sequence $L_i :: S_i$ in the first input sequence group and sequence $L_j :: S_j$ in the second input sequence group, there is a sequence $L_i.L_j :: Op_S(S_i, S_j)$ in the output sequence if $Op_S(S_i, S_j)$ is non-empty. Else, there is no sequence in the output with label $L_i.L_j$.

The sequence group operator Op_{SG} therefore applies the sequence operator Op_S individually to each combination of sequences in the input sequence groups. Four of the new operators on sequence groups are presented here; the fifth is presented in the section on nested queries and is used solely in that context.

- The *Select_Label* operator allows selection conditions to reference the label attributes as well.
- The *Project_Label* operator allows a projection to removes some attributes of the input sequence label as well. Each output sequence is the positional union of the input sequences that project to the same label.

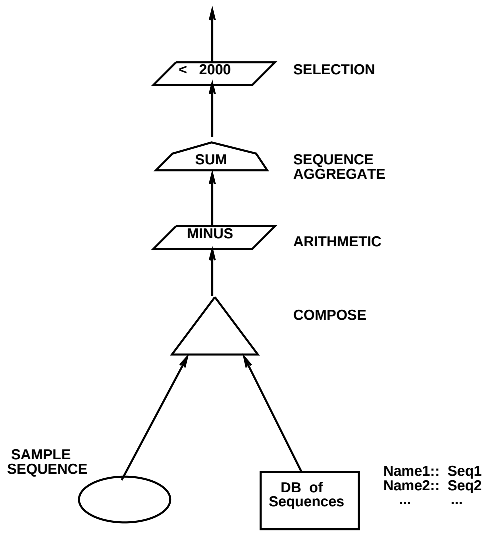

Figure 5: Example of Collective Operations

- The *Group_Union*(*Group_Difference*) operator generates the set union(difference) of two input sequence groups, based on their labels. The sequence groups must have the same record and label schemas. If two sequences have identical labels, their union(difference) appears in the output.

As an example of a query requiring sequence groups, consider an example from [1] which performs a similarity search on a database of sequences with identical schemas. If the records in the sequences have a numeric attribute on which the similarity comparison is being made, the “distance” between two sequences at a position is the absolute difference in the values of the attribute in the corresponding records of the sequences at that position. The total distance between the two is the sum over all positions of the per-position distance. Given a sample sequence, the query needs to find all sequences in the database that are within a certain “distance” (say 2000) of the sample sequence.

With the basic sequence model, it is possible to express such a comparison between two individual sequences. It requires a positional composition, an arithmetic operator to determine the position-wise difference, followed by an aggregate operation that sums the total distance between the two sequences. However, it is desirable to express such a query over a collection of sequences. Figure 5 demonstrates how the $\mathcal{SEQ}$ model can represent this query. Let the database of sequences be represented by a sequence group where each individual sequence is labelled by its name. Recall that all the sequences have the same schema, since they belong to the same sequence group. The result is a sequence group which contains those sequences in the database that satisfy the similarity query.

3.3 Extending Relational Databases

Much of existing temporal database research has extended the relational model to add temporal semantics. Typically, a timestamp (which is a collection of time points) is associated either with each tuple, or with each attribute of a tuple. An observation made by [10] was that temporal relations should be modeled not as sets of tuples, but as sets of Time Sequences, with basic operations like selections and projections on the Time Sequences. This is the natural way to represent temporal relations in $\mathcal{SEQ}$, which we shall call $\mathcal{SEQ}$ -Rel. The time varying values of each real-world entity are represented as a sequence, labelled by the non-varying attributes of the entity. A sequence group containing a set of such labelled sequences models a temporal relation. We study the properties of $\mathcal{SEQ}$ -Rel in [15], and show that by extending a sequence group to be a multiset, it is a “temporally grouped” model[3]. We also present some results on the expressiveness of the model.

One important result is that relations can be modeled as a special case of sequence groups. In the degenerate case where the sequence record schema has no attributes, each element of the sequence group is essentially a tuple. The sequence group operators defined earlier give $\mathcal{SEQ}$ -Rel the operators of the relational algebra, making it relationally complete. This indicates that the integration of sequence data with existing relational databases may be relatively simple. Since the $\mathcal{SEQ}$ model is relationally complete, existing systems need only be extended with the additional functionality needed to manipulate sequence data. We are currently building a complete database system based on $\mathcal{SEQ}$ that will also handle relations cleanly under this common framework (while $\mathcal{SEQ}$ currently does not have features like duplicates and aggregates that relational database systems provide, adding these should not be difficult). We are also in the process of extending SQL to allow $\mathcal{SEQ}$ queries to be expressed. It is important to realize the degree of flexibility that $\mathcal{SEQ}$ provides such a system. $\mathcal{SEQ}$ does not specify any particular data representation format; therefore, the underlying system may “flatten” sequences into 1NF tuples, or implement them as complex tuples. Further, there is the ability to ask queries over sets of sequences (like many of the queries in the consensus test-suite of temporal queries[9]), or complex queries over individual sequences (like the queries that are described in [14]).

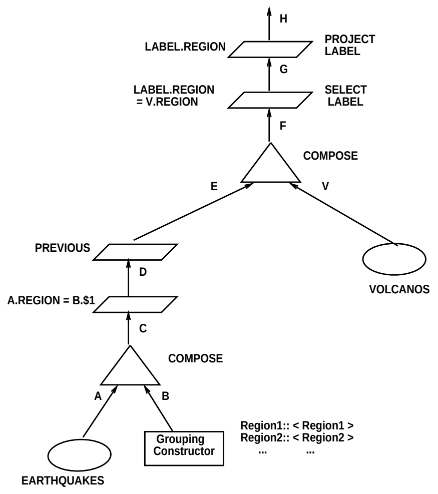

Figure 6: Nested Query Using Sequence Groups

3.4 Nested Queries

Consider an example of a nested query(from [14]): *Find each Volcano eruption and the most recent Earthquake in the same region*. For each Volcano record X at position p, the previous earthquake record in the same region as X needs to be determined and composed with X. We need to find an algebraic representation of this operational notion. We note that the same issue arises in the context of nested aggregate queries in SQL, and that our solution can be applied almost identically in that domain as well. We require the addition of one new operator; the *Group_Construct(T)* operator is defined for every type T $\in T_{Basic}$ and has no inputs. For every value $v \in$ type-domain(T), there is a sequence $< v >:: Const(< v >)$ in the output sequence group.

Figure 6 shows the desired $\mathcal{SEQ}$ query. Each sequence group in the query graph is given a name for ease of explanation. A is a singleton sequence group with empty label containing the Earthquake sequence. B is the result of a Group_Construct on the domain of regions. The composition of A and B results in C which has one labelled sequence for each region (i.e. one for each label in B). The Select operator applied to the attributes of the C records results in the D sequence group, in which each sequence is labeled by a particular region and contains records of all earthquakes in that region. The Previous operator now produces sequence group E. For each region label, E contains a sequence that for every position specifies the most recent earthquake(s) in the same region. This is composed with V which is the Volcano sequence group of empty label, resulting in the sequence group F. F is now the input to a Select_Label operator which selects in each labelled sequence, only those records for which the volcano's region matches the region in the label. At this stage, sequence group G contains one sequence for each of the region labels. Any records within each sequence are a concatenation of a volcano record and the previous earthquake record in the same region. Finally, the Project_Label operator eliminates the sequence label, thus merging all the sequences in the group into the desired answer sequence.

While it may appear tedious to express a correlated query using the sequence group operators, in practice, a user would express the query using some easier notation (like the use of a scoped variable). The algebraic form of the query would be automatically generated from the user's query to then be optimized and evaluated.

4 Related Work

$\mathcal{SEQ}$ draws features from many diverse existing approaches to modeling sequence data, and appears to be suited to most sequence database application domains. In this sense, $\mathcal{SEQ}$ is a unifying sequence data model. We now provide brief pointers to interesting related work. There has been considerable research on temporal data models[8], algebras and semantics[18]. We merely cite references to articles that survey the extensive work in these areas. TSQL2[5] represents the ongoing efforts of a portion of the temporal database research community to specify an extension to the SQL-92 language standard. TSQL2 is based on a temporal relational model, and we have discussed the relationship of $\mathcal{SEQ}$ with such models earlier in the paper. A more specific comparison with TSQL2 is presented in the expanded version of this paper[15]. [12] describes operators based on temporal logic for supporting lists in a data model, while [19] provides a model based on logic programing for stream oriented data and processing. Much of this work is similar to our Positional operators on individual sequences. The Time Sequence model[13] views a database as being composed of collections of time-varying objects, with query operators over these collections. We show in [15] that the Positional operators of the $\mathcal{SEQ}$ model with minor extensions to the Expand operator can represent all Time Sequence queries. [6] presents an object-oriented approach to providing lower level operators on temporal objects. Our approach instead uses high level algebraic operators to query sequences. The concepts of collapse and expansion of ordering domains are similar to the intervallic partitions of [4] and the time units of [20]. There has also been closely related work on calendric systems by [2] and [17]. We discuss the similarities and differences in greater detail in [15]. An expression recognition system on event sequences has been implemented as part of the

ODE project[7]. In [15] we show that some of the same functionality can be provided by $\mathcal{SEQ}$, but with potentially increased efficiency of evaluation for large sequences. [1] describes an efficient technique for similarity searches over sequence databases; the problem was used as an example query in Section 3.2. Much of this work appears disjoint or at best remotely related, and it is encouraging that $\mathcal{SEQ}$ is able to relate these diverse applications involving sequence data.

5 Conclusions and Future Work

We have presented the $\mathcal{SEQ}$ model of sequence data, along with algebraic operators to query sequence databases. The algebraic representation of nested queries presented in the context of $\mathcal{SEQ}$ is independently an important contribution of this paper. The importance of $\mathcal{SEQ}$ lies in its ability to model sequence data from a variety of domains and applications, and the possibilities for efficient query processing that arise from a relatively simple and uniform data model. Our future work in this area focuses on three major areas: (a) extending the $\mathcal{SEQ}$ model with new constructs, specifically a limited form of cyclic queries, (b) devising further optimization mechanisms and evaluation strategies, (c) Implementing a sequence database system based on the $\mathcal{SEQ}$ model; this is currently in progress.

Acknowledgements

The authors thank Joey Hellerstein and H.V. Jagadish for many useful comments on the presentation of the paper and lively discussion of the subject matter. Navin Kabra also provided comments on a draft of the paper.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search In Sequence Databases. In *Proceedings of the Fourth International Conference on Foundations of Data Organization and Algorithms*, pages 69–84. Springer-Verlag, Berlin, October 1993. Lecture Notes in Computer Science, V303.

[2] Rakesh Chandra and Arie Segev. Managing Temporal Financial Data in an Extensible Database. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 238–249, 1992.

[3] James Clifford, Albert Croker, and Alexander Tuzhilin. Grouped and Ungrouped Historical Data Models: Expressive Power and Completeness. In Richard Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages H1–H18, Arlington, Texas, 1993.

[4] James Clifford and Ahobala Rao. A Simple, General Structure for Temporal Domains. CRIS 154, Center for Research on Information Systems, Stern School of Business, New York University, October 1986.

[5] TSQL2 Language Design Committee. TSQL2 Language Specification. Draft in preparation, December 1993.

[6] Umeshwar Dayal and Gene Wuu. A Uniform Approach to Processing Temporal Queries. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 407–418, 1992.

[7] N.H Gehani, H.V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 327–338, 1992.

[8] C.S. Jensen, M.D. Soo, and R.T. Snodgrass. Unification of Temporal Relations. Technical Report 92-15, Computer Sciences Department, University of Arizona, July 1992.

[9] C.S. Jensen(editor). A Consensus Test Suite of Temporal Queries. Consensus document collectively drafted by temporal database researchers, September 1993.

[10] Wolfgang Käfer. Temporal Selection, Temporal Projection and Temporal Join Revised. In Richard Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages U1–U20, Arlington, Texas, 1993.

[11] Cliff T.Y. Leung and Richard R. Muntz. Query Processing for Temporal Databases. In *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, California, February 1990.

[12] Joel Richardson. Supporting Lists in a Data Model. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 127–138, 1992.

[13] Arie Segev and Arie Shoshani. Logical Modelling of Temporal Data. In *Proceedings of ACM SIGMOD '87 International Conference on Management of Data, San Francisco, CA*, pages 454–466, 1987.

[14] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence Query Processing. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, May 1994.

[15] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A Model for Sequence Databases. Technical report, University of Wisconsin-Madison, 1995. Extended version of paper in ICDE 95.

[16] Richard Snodgrass. *An Overview of TQuel*, chapter 1. Benjamin/Cummings, 1993.

[17] M. Soo, R. Snodgrass, C. Dyreson, C. S. Jensen, and N. Kline. Architectural extensions to support multiple calendars. TempIS Technical Report 32, uazcsd, Revised May 1992.

[18] Michael D. Soo. Bibliography on Temporal Databases. *ACM SIGMOD Record*, 20(1):14–23, March 1991.

[19] D. Stott Parker, Eric Simon, and Patrick Valduriez. SVP - a Model Capturing Sets, Streams, and Parallelism. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), Vancouver, Canada*, pages 115–126, 1992.

[20] Sean X. Wang, Sushil Jajodia, and V.S. Subrahmanian. Temporal Modules: An Approach Toward Federated Tempora Databases. In *Proceedings of ACM SIGMOD '93 International Conference on Management of Data, Washington, DC*, pages 227–237, 1993.