Sequence Query Processing

Praveen Seshadri * Miron Livny Raghu Ramakrishnan †
Computer Sciences Department,
University of Wisconsin-Madison, WI 53706, U.S.A.
{praveen,miron,raghu}@cs.wisc.edu.

Abstract

Many applications require the ability to manipulate sequences of data. We motivate the importance of sequence query processing, and present a framework for the optimization of sequence queries based on several novel techniques. These include query transformations, optimizations that utilize meta–data, and caching of intermediate results. We present a bottom-up algorithm that generates an efficient query evaluation plan based on cost estimates. This work also identifies a number of directions in which future research can be directed.

1 Introduction

Many real life applications manipulate data that is inherently sequential. Such data is logically viewed and queried in terms of a sequence abstraction and is often physically stored as a sequence. Databases should (a) allow sequences to be queried in a *declarative manner*, utilizing the ordered semantics of the data, and (b) take advantage of the opportunities available for query optimization. Relational databases are inadequate in this regard; data collections are treated as sets, not sequences. Consequently, expressing sequence queries is tedious, and evaluating them is inefficient. Sequence databases therefore require techniques that are distinct from established relational database techniques. This paper deals with issues of query optimization and evaluation for sequence queries. The techniques developed are applicable to a broad class of sequence data domains, including temporal databases, execution monitors, trigger mechanisms[GJS92], and list processing systems[Ric92].

Example 1.1 Consider the following motivating example. A weather monitoring system records information about various meteorological phenomena. There is a sequentiality in the occurrence of these phenomena; the various meteorological events are sequenced by the time at which they are recorded. A scientist asks the query: *"For which volcano eruptions was the strength of the most recent earthquake greater than 7.0 on the Richter scale?"*. It is a seemingly innocuous query, but it is difficult to *express* in a relational query language like SQL and inefficient to *evaluate*. One possible attempt uses a nested sub-query as follows:

```
SELECT V.name
FROM Volcanos V, Earthquakes E
WHERE E.strength > 7.0 AND
      E.time = (SELECT  max(E1.time)
                FROM  Earthquakes E1
                WHERE E1.time < V.time)
```

Other ways of representing the same query in SQL are not particularly simpler. A conventional relational query optimizer as described in [SMALP79] would probably generate the following query evaluation plan. For every Volcano tuple in the outer query, the sub-query would be invoked to find the time of the most recent earthquake. Each such access to the subquery involves an aggregate over the entire Earthquake relation. The time of the most recent earthquake is used as a join condition to probe the Earthquake relation in the outer query. Finally, the selection condition to check that the strength is greater than 7.0 would be applied. Even the knowledge that the Earthquakes and Volcano relations are sorted by time would not significantly alter the query plan. A more efficient evaluation strategy does however exist; the two sequences can be scanned in lock step (similar to a sort merge join). The most recent earthquake record scanned can be stored in a temporary buffer. Whenever a volcano record is processed, the value of the most recent earthquake record

*The work of Praveen Seshadri was supported by IBM Research Grant 93-F153900-000

†The work of Raghu Ramakrishnan was supported by a Packard Foundation Fellowship in Science and Engineering, a PYI Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563.

stored in the buffer is checked to see if its strength was greater than 7.0, possibly generating an answer. This query can therefore be processed with a single scan of the two sequences, and using very little memory. The key to such optimization is the sequentiality of the data and the query.

2 Sequence Model

In this paper, we use a simple model of sequences and a set of query operators. A more comprehensive model of sequences is under development[SLR]. A record schema R is defined as $R = < A_1{:}T_1, \ \ldots \ , A_N{:}T_N >$ for some finite N. Each of the T_is are indivisible atomic types of fixed size, and each A_i is a named attribute. The type domain T_R of R is $(T_1 \times T_2 \times \ldots \times T_N)$. A record of schema R is an element of T_R[1]. The domain of every record type T_R is associated with a special Null record $Null_R$. *A sequence S is a function from the integers to* $T_R \cup Null_R$. Every integer i is called a position, and the function is called the sequence ordering, denoted typically as S(i)[2]. The positions that map to $Null_R$ are called *empty positions.* Every record in $T_R \cup Null_R$ maps to zero or more positions.

We consider three basic categories of sequences:

- *Base sequences:* A base sequence is specified by an explicit materialized association of positions with records. All other records are assumed to map to no positions. All positions that do not explicitly map to a record are assumed to map to the Null record.

- *Constant sequences:* In a constant sequence, every position maps to the same unique record. We model constants as sequences so that the constructs of the model deal uniformly with sequences.

- *Derived sequences:* A derived sequence is defined by a *sequence operator.* Intuitively, this is similar to the definition of views in a relational database using relational operators. The next section describes operators that are used in the definition of such sequences.

2.1 Sequence Operators

All operators in our model are compositional; they operate on sequences producing a single derived sequence. The derived sequence defined by a sequence operator is called the *output sequence* of the operator, while the sequences used in this definition are called the *input sequences.* The *arity* of an operator is the number of input sequences. For an operator Op with output sequence S_{out} and input sequences S_1 through S_n, for all positions i, $S_{out}(i) = Op(S_1, S_2, ..., S_n, i)$. We consider the following specific operators in this paper:

Simple Unary Operators: The output sequence S_{out} of a unary operator Op is defined by $S_{out}(i) = Op(S_{in}, i)$.

- A *Selection* applies a selection predicate σ to the input sequence record r at each position i. If $(\sigma(S_{in}(i), i))$, $Op(S_{in}, i) = S_{in}(i)$, else $Op(S_{in}, i) = Null$. Note that it does not matter if a predicate evaluates to true or false on a Null input record.

- A *Projection* π projects a subset of the attributes of the input sequence record r at each position i. Any projection of a Null record is a Null record. $Op(S_{in}, i) = \pi(S_{in}(i))$.

- A *Positional Offset* takes as a parameter an integer specifying an offset l. For each record r at position i in the input sequence, $Op(S_{in}, i) = S_{in}(i + l)$. Intuitively, this operator "shifts" the input sequence by the number of positions specified by the offset.

- A *Value Offset* takes as a parameter an integer specifying an offset l. For each record r at position i in the input sequence, $Op(S_{in}, i1) = S_{in}(i)$ iff the number of non-empty positions between i1 and i = l. If the input sequence has no empty positions, this is identical to the positional offset operator. Examples are the Previous operator that has an offset of -1, and the Next operator with an offset of +1.

Aggregate Unary Operators: The Aggregate Operators are unary operators that are defined by two functions. One function *agg_pos(i)* selects a set P of positions for each position i. The other function *agg_func* is an aggregate function over the records in $S_{in}(p)$ at the positions p in P. $Op(S_{in}, i) = agg\_func(\{S_{in}(p) \mid p \in agg\_pos(i)\})$. For example, the moving 3-position average of a sequence would have agg_func $\equiv$ Avg and $agg\_pos(i) \equiv \{p \mid i \geq p \geq i-2\}$. A special case is where the agg_pos function is always true (thereby selecting all positions). The aggregate functions allowed are Avg, Count, Min, Max and Sum. For all these aggregates, Null records in the inputs are ignored if there is at least one non-Null record; else the output is a Null record.

Compose Operator: *Compose* is a binary operator that composes the records r1 and r2 of the two input sequences at each position i. $Op(S_{in1}, S_{in2}, i) = S_{in1}(i).S_{in2}(i)$. If either of the records is a Null record, the output is a Null record. A Compose operator is also referred to as a *positional join* operator. Note that in an implementation, the Compose operator would probably allow the specification of additional "join" predicates.

[1] Notationally, a record is represented by a list of attributes between the symbols <>.

[2] For the sake of convenience, our model defines positions as integers. In general, positions could be elements of any totally ordered, countable domain. The Null records are introduced solely in order to model sequences cleanly. We stress that this does not mean that an implementation would actually need to materialize such records.

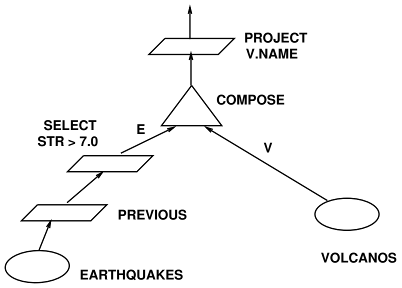

Figure 1: Motivating Example

2.2 Sequence Queries

A sequence query is an acyclic graph of operators such that the inputs of each operator are either the output of another operator or are base or constant sequences. The output of a query is the output sequence of the root operator of the query graph. As an example of the use of the operators described, we present a graphical representation of the query of Example 1.1 in Figure 1. The graphical representation is a declarative specification of the query. In this paper, we restrict the graph to be a tree; i.e. we do not allow the output of any operator to act as the input to more than one operator. We discuss the effects of relaxing this restriction in Section 5.

2.3 Operator Scope

We now introduce the concept of the scope of an operator. This is used in the optimization and evaluation of sequence queries. Consider a derived sequence S_{out} defined by an operator Op of arity n such that for all positions i, $S_{out}(i) = Op(S_1, .., S_n, i)$. The operator can be described by two functions: Scope that defines the positions of the input records to look at, and the operator function OpFunc that actually manipulates those input records to define the output sequence.

Definition 2.1 The scope of the operator Op is a function $Scope(k, i)$ that for each input sequence S_k, and for each position i, returns a minimal set of positions that satisfies the following property: whatever the actual data in the input sequences,

- $\forall k \in \{1..n\}, \forall j, S_{k'}(j) = S_k(j)$ if j $\in Scope(k, i)$, else $S_{k'}(j) = Null$, and
- $\forall i, S_{out'}(i) = O(S_{1'}, .., S_{n'}, i)$, and
- $\forall i, S_{out'}(i) = S_{out}(i)$

□

When the input sequence S_k is implicit from the context of the discussion, it is omitted from the notation. Further, when the operator in question is not implicit, it is explicitly specified as Op.Scope. Since $Scope(S_k, i)$ returns a set of positions, the notation is extended to specify the set of records at those positions as $S_k(Scope(i))$.

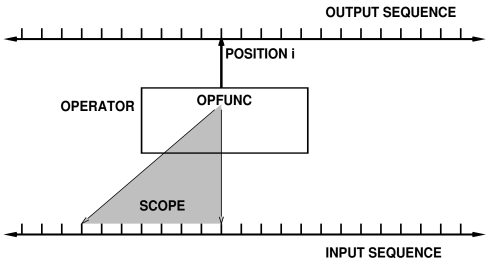

Figure 2: Operator Scope

Definition 2.2 The output record at position i is given by $S_{out}(i) = OpFunc(S_1(Scope(i), .., S_n(Scope(i)), i)$, □

In Figure 2, the operator function accesses the current input record and the last seven input records in order to derive the output record at a particular position i. Note that these positions are specified relative to the position i (in this case, i, i-1, i-2, ..., i-7). This defines Scope(i) as the position i and the previous seven positions. The value of Scope(j) for some other position j need not necessarily be similar; this depends on the nature of the operator. The following properties of the operator may be identified:

- The *scope size at position i* is the number of positions in Scope(i). The scope size could be fixed for all i or could vary with i. A special case is the unit scope (of size one). For example, a Selection operator has a fixed scope of size one, while a Previous operator has a variable scope size.
- The *scope sequentiality* indicates how the scopes at successive positions overlap. For all positions i, if $Scope(i) \subseteq Scope(i-1) \cup i$, then the scope is sequential. For example, the scope of an aggregate over the most recent three positions is sequential, while the scope of a positional offset operator is not.
- The *scope relativity at position i* indicates how the positions in Scope(i) are defined with respect to i. If the positions in the scope are defined as $\{K_1 + i, .., K_n + i\}$, where each value K_j is a constant independent of i, it is considered a relative scope. Otherwise, the scope is non-relative. All the operators that we have considered have relative scopes.

Acyclic compositions of basic operators result in complex operators. Every complex operator can be modeled as an instance of the generalized operators discussed in the previous section. Consider two operators A and B composed as follows. Let $A(S_{A1}, .., S_{Am}, i) = S_A(i)$, and let $B(S_A, S_{B1}, .., S_{Bn}, i) = S_{out}(i)$. The composition forms a complex operator Op such that $Op(S_{A1}, .., S_{Am}, S_{B1}, .., S_{Bn}, i) = S_{out}(i)$. The scope $Op.Scope(S_{Ai}, i) = \{j \mid k \in B.Scope(S_A, i) \wedge j \in$

$A.Scope(S_{Ai}, k)\}$. The operator function

$$OpFunc(S_{A1}(Scope(i)), .., S_{Bn}(Scope(i)), i) =$$

$$B.OpF(S_A(Scope(i)), S_{B1}(Scope(i)), .., S_{Bn}(Scope(i))i)$$

The following properties of complex operator Op can be proved:

Proposition 2.1

(a) *If both A and B have fixed scope sizes on all their input sequences, then Op has a fixed scope size on all its input sequences.*

(b) *If both A and B have sequential scopes on all their input sequences, then Op has a sequential scope on all its input sequences.*

(c) *If both A and B have relative scopes on all their input sequences, then Op has relative scopes on all its input sequences.*

□

The properties of the scope of a complex operator can therefore be described in terms of the properties of its composite operators. A sequence query is a complex operator all of whose input sequences are base or constant.

3 Query Optimization Techniques

Sequence query processing offers a number of unique opportunities for query optimization that are not available in relational queries. In particular, the notion of *operator scope* plays an important role in query optimization. Sequence query optimizations fall into three categories:

- Those that transform the declarative query into an equivalent query. These optimizations are *independent* of the actual data in the input sequences, and are described in Section 3.1.
- Those that use meta-information in the input sequences to perform global query optimizations. One such optimization is described in Section 3.2.
- Those that use meta-information in the input sequences to perform local optimizations of each of the query operators. Such optimizations are described in Sections 3.3 and 3.5.

This section provides the motivation for individual optimizations, illustrating them with examples. Section 4 integrates the various optimizations into the framework of a single optimization algorithm. For base sequences, the following kinds of meta–information may be available in the underlying physical sequence representation[3]:

[3]These are similar to the properties associated with a Time Sequence in [SS87, SK86]

Sequence	Span	Density
IBM	200 .. 500	0.95
DEC	1 .. 350	0.7
HP	1 .. 750	1.0

Table 1: Example Sequence Data

- A start and end position that determine the *valid range* or *span* of the sequence. Any position outside of this range is mapped to a Null record.
- A *density* which specifies the fraction of the positions within the valid range that map to non-Null records.
- Other statistical information about the base sequences, including the distributions of values in the columns (used to determine the selectivity of predicates) and the correlations between sequences in the positions of Null records.
- Available access paths to base sequences, and the costs of access along these paths.

For the purposes of this section, we shall use the following three daily stock market sequences: the IBM sequence, the DEC sequence and the HP sequence displayed in Table 1.

3.1 Query Transformations

The first set of optimizations transform a query into an equivalent query that might be more efficient to evaluate.

Definition 3.1 Two sequence queries Q1 and Q2 are equivalent if they both have the same input sequences, the same scopes on the input sequences, and the same operator function. □

Note that this definition of query equivalence is independent of the actual data in the input sequences. Various transformations may be applied to a query graph to generate an equivalent query graph.

Proposition 3.1 *Consider a transformation that alters a sub-graph of a query, but does not affect the rest of the query graph. If the complex operator corresponding to the altered sub-graph is equivalent to the original sub-graph, then the transformed query is equivalent to the original query.* □

It is typically difficult to reason about the correctness of transformations that affect the entire query graph. However, an important class of transformations acts at the level of individual pairs of operators. The above proposition notes that such local transformations can be used to transform the entire query.

Depending on the particular set of basic operators chosen in a model, various equivalence transformations may be specified based on this proposition. Some results are well known for relational queries and carry

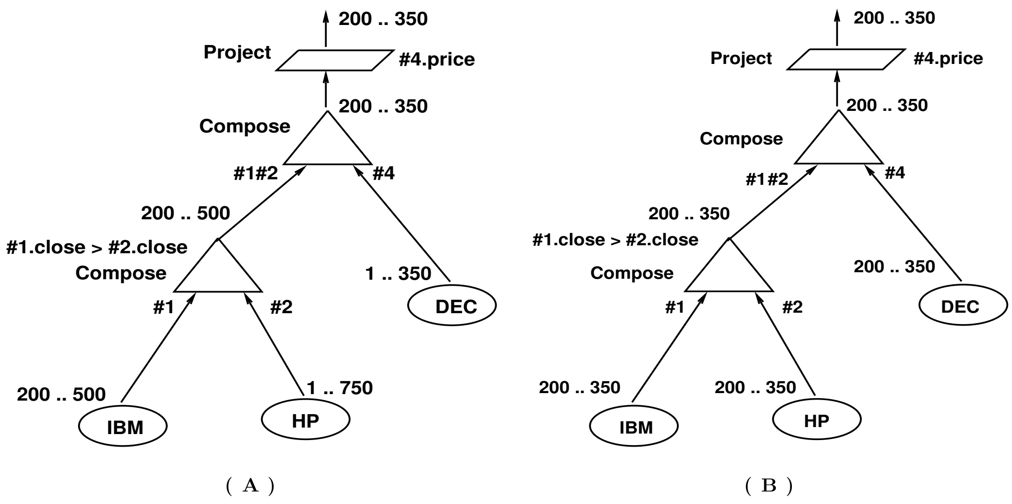

Figure 3: Using Span Information

over to sequence queries as well. For instance, two successive projections can be combined, as can two successive selections. A projection can be "pushed through" a relational join if the attributes projected out do not participate[4] in the join. A similar result holds for projections and compose operators in sequence queries. Relational joins are reflexive and associative; similarly the positional joins or compose operators are reflexive and associative. We informally state here a list of interesting additional transformations using the restricted set of operators of Section 2[5].

- A projection can be pushed through any sequence operator O iff all the attributes that participate in the projection are in the input sequences of O, and all attributes that participate in O are among the projected attributes.
- A positional offset can be pushed through any operator of relative scope on all its inputs.

Further, it is possible to identify some transformations that are incorrect in general:

- A selection cannot be pushed through an aggregate operator or a value offset operator (i.e. an operator of non-unit scope).
- An aggregate operator or value offset operator cannot be pushed through a Compose operator.
- An aggregate operator cannot be pushed through a value offset operator and vice versa.

It is a good heuristic to propagate selections, projections and positional offsets as far down the query graph as possible. Selections cannot be pushed through operators of non-unit scope (like aggregates and previous/next), and such operators cannot commute with compose operators. The non-unit scope operators therefore break up the query into blocks, inside which the positional joins can be reordered. The blocks are similar to query blocks in SQL that need to be independently optimized. Each block is described by a set of input sequences that participate in positional joins. Selections and projections are applied to the result of the positional joins. The output sequence of a block can feed into the input of another block. The operators of non-unit scope form special blocks that contain a single operator.

3.2 Global Span Optimization

Consider the span of the sequences in the example of Figure 3. In Figure 3.A, the original query asks for the price of DEC stock when the close of the IBM stock was greater than the close of HP stock. In the figure, the composition of the IBM and HP sequences followed by a selection condition on their "close" values is condensed into a single operator for conciseness of representation. Note that the span of DEC is from position 1 to 350, IBM is from 200 to 500 and HP is from 1 to 750. The spans of derived sequences are computed by the operators that define them, based on the spans of the input sequences to the operator. This query is equivalent to the query in Figure 3.B where the spans of all the base sequences are restricted to the period from position 200 to position 350. This reduces query processing costs, since a smaller range of each sequence is accessed. The ability to restrict the span of a sequence based on the other sequences used in the query holds a tremendous potential for query processing efficiency. For every operator, given the span of the input sequences, the span of the output sequence can be determined. Similarly, if the span of the output sequence is known, the spans of the inputs may be modified, while retaining equivalence to the original

[4]An attribute of an input sequence record whose value is used by an operator function is said to *participate* in the definition of the output sequence. The term "pushed through" is well understood in relational queries and we do not bother with a formal definition here for reasons of conciseness.

[5]Formal statements and proofs are omitted for lack of space.

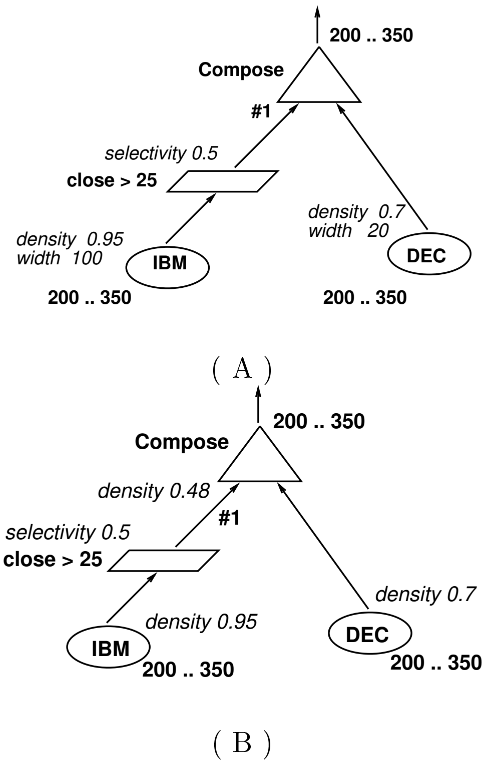

Figure 4: Using Statistical Information

query.

3.3 Access Modes

Consider a Compose operator with two input sequences as in Figure 4.A. In order to generate the output sequence, one possibility is to retrieve every non-Null record of the DEC sequence, and the corresponding record from the other input sequence (labeled #1 in the figure). Another possibility is to retrieve all the non-Null records in sequence #1 first, and to then look up the corresponding record from the DEC sequence. These represent two variants of *Join-Strategy-A* which streams one input sequence and probes the other[6]. Another strategy is to stream both sequences in lock step, performing the positional join at common positions. This is *Join-Strategy-B. Access mode* refers to whether the access to a sequence is "stream" or "probed". The per-record access cost of a stream access can be significantly different from that of a probed access. The basic operation for a stream access is to get the *next* non-Null record. This is as opposed to the basic operation for a probed access, which is to get the record at a specific position. The statistical meta-information that needs to be considered in choosing between the join strategies includes the density of the base sequences, the correlation in their densities, their access costs and the selectivity of the operator that generates the #1 sequence.

[6]This is analogous to the choice of two possible choices of outer and inner relations in a nested loop join

3.4 Stream–Access Evaluation

The goal of sequence query optimization is to choose a query evaluation plan of low estimated cost[7]. Our model of a sequence query evaluation associates a cache (a randomly accessible buffer) with each basic operator. Caches operate on a FIFO basis and can store records for efficient subsequent retrieval. Some mechanism is provided for accessing the cached records associatively by position. A query evaluation plan can specify the sizes of the caches used by each of its operators.

Definition 3.2 A query evaluation is *cache-finite* if the size of the cache at every operator is a constant determined independent of the actual data in the input sequences. A query evaluation possesses the stream–access property if it is cache-finite and performs a single scan of its input (base) sequences in positional order. □

It seems intuitively that the stream–access property would be the ideal property for a query evaluation to possess. The cost of a stream access query is limited to the cost of a single scan of the input sequences in positional order, the cache access costs and the computation costs. The caches are assumed to be small, and the cache access cost is typically negligible. It is however not the case that a stream access evaluation implies an optimal evaluation. For instance, the accesses to a particular sequence may be to a very few positions so that it may be preferable to directly access them using an indexing mechanism. This may also be the case if the data is not physically organized to favor stream access[8].

Theorem 3.1 *If every operator in a query graph has a sequential, fixed-size scope on all its inputs, and if caches of the size of the scopes are used, then the query has a stream–access evaluation.*

Lemma 3.1 *If the scope of a query on all its inputs is sequential and of fixed size, and if caches of the size of the scopes are used, then the query has a stream–access evaluation.*

If the scope of an operator does not satisfy these conditions, it might still be possible to execute the query in a stream-access fashion. This can be done by "broadening the scope" of the operator.

Definition 3.3 The effective scope of an operator over an input sequence IS is a function $EffScope_{IS}$ such that for all positions i, $Scope_{IS}(i) \subseteq EffScope_{IS}(i)$. □

[7]In certain domains in which input data records arrive dynamically(eg. [GJS92]), it is important to optimize the cost of processing each arriving input record.

[8]The actual physical organization of a sequence can vary. A relation with an unclustered index on a position attribute does not particularly favor stream access.

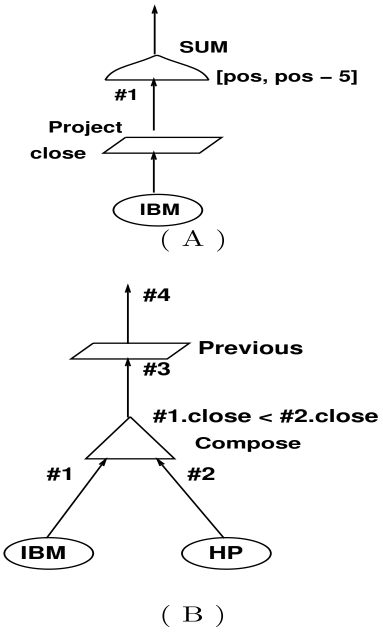

Figure 5: Caching of Derived Sequences

If the effective scope of every operator in a query is defined on all its inputs, then this defines the effective scope of the query on the input sequences. The properties of scope like size and sequentiality can be associated with effective scope as well.

Lemma 3.2 *If every operator in a query graph has a sequential, fixed-size effective scope on all its inputs, and if caches of the size of the effective scopes are used, then the query has a stream–access evaluation.*

Consider the positional offset operator with an offset of -5. This operator has a scope of size 1, but the scope is not sequential. By making the effective scope the current position and the five most recent positions, the effective scope becomes sequential of size six. A stream–access evaluation is now possible. Permitting a stream access therefore requires the identification of a sequential fixed-size effective scope for every operator in the sequence. Obviously, this can be trivially satisfied by including all positions in the valid range into the effective scope of an operator. As will be evident from the next section, it is important to find the minimal such effective scope.

3.5 Caching of Derived Sequences

Operators like aggregates and value offsets have a non–unit scope on their input sequences. *The purpose of caching is to ensure in conjunction with a stream mode of access that the records at positions in the operator scope are found in the cache.* Consider for example the query in Figure 5.A that computes for each position, the sum of the close of the IBM sequence over the previous six positions. The aggregate operator has a scope of size six. If the access to the Sum sequence is a stream access, the last six values of the derived sequence marked #1

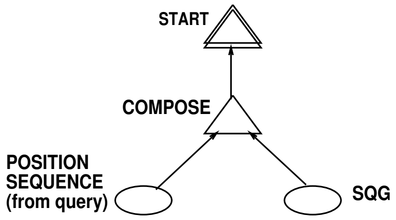

Figure 6: Sequence Query Template

can be cached. Since these values are repeatedly used in the computation of the aggregate, the Sum operator at every position needs to access the input sequence only at that position. This caching strategy is referred to as *Cache-Strategy-A*. When the effective scope is large or of variable size, it may not be feasible to cache the entire scope of the input sequence. However another caching strategy described below could be used.

Consider the query in Figure 5.B. The Previous operator generates as an output record, the most recent input record at an earlier position than the current position. Note that the scope of this operator is variable since it may have to go back an arbitrary number of positions in the input sequence. If the input sequence is a base sequence, this involves repeated retrievals, and for a derived sequence this involves recomputation as well. For instance, if the close of IBM is usually greater than the close of HP, a large number of IBM and HP records may need to be accessed to generate each record of the sequence marked #4. If instead of this naive algorithm, the value of the #4 sequence at the previous position were cached, then the record at a particular position in the #4 sequence is either the cached record at the previous position, or the record from the #3 sequence at the previous position if it is non-Null. We refer to this incremental cache strategy as *Cache-Strategy-B*. Such an optimization could be viewed as a graph transformation in which an operator is replaced by an equivalent operator that can be more efficiently evaluated. It could also be viewed as allowing a restricted form of cycles in the query graph (since the output sequence is being used to derive itself). However, it is most intuitive to present this as a caching optimization.

4 Query Plan Generation

Using the basic set of operators that have been described, we now consider an algorithm for the optimization and evaluation of sequence queries. Many minor details have been omitted for the sake of conciseness but the important details are mentioned either along with the algorithm or as part of the description of the optimizations in the preceding section.

A sequence may be queried by asking for its records at (a) specific positions, or (b) *all* positions in a range. In both cases, the query is associated with a Position Sequence that has non-Null records at those positions for which an answer is desired. Figure 6 shows a query specification. The sequence being queried is labeled SQG(Sequence Query Graph) to signify that this is not necessarily a base sequence, but could well be a derived sequence represented by a query graph. The *Start* operator is a special operator that initiates query evaluation by invoking a stream access on its input. The query optimizer needs to produce a query evaluation plan that will generate the output sequence at the lowest estimated cost.

We now examine the steps involved in the query optimization algorithm. The algorithm explores the space of possible query evaluation plans in a bottom-up fashion.

Step 1 - Query Specification: The query is specified as a Sequence Query Graph (SQG) composed with the query template. At this stage, the query is a hierarchical graph of operators, with base and constant sequences at the leaves of the graph and the Start operator at the root.

Step 2 - Meta–Information Propagation: In this stage, the meta–information associated with the base sequences is propagated so that the entire query graph is adorned with appropriate information.

Step 2.a - Bottom Up Annotation: The query graph is adorned with schema and meta–information in a bottom-up traversal. The purpose of this step is to perform type checking of the query and to define the meta–information associated with the derived sequences in the query. Every operator uses the span information from its input sequences to determine the span of its output sequence. Further, the densities of the input sequences can be used to determine the density of the output sequence. The semantics of each operator plays a part in the density propagation. With our definition of aggregate operators, a Null record is produced only if all the input records in the scope are Null. Similarly, for a positional join operator, the density of the output sequence would be a function of the input sequence densities, the selectivity of the join predicates and the correlation in the Null positions of the input sequences.

Step 2.b - Top Down Annotation: This step propagates span information down the graph using a top-down traversal starting from the root. Each operator modifies the spans of its input sequences based on the span of its output sequence. The importance of this stage was illustrated in Section 3.2.

Step 3 - Query Transformations: Section 3.1 presented the rules for the various graph transformations. At this stage, the transformations discussed in Section 3.1 that are heuristically expected to prove beneficial are applied.

```
if (single operator of non-unit scope) {
  Find cheapest plan for each access mode;
} else { /* collection of positional joins */
  Curr_Set := Cheapest plans for each input
              sequence for each access mode;
  Repeat till all sequences have been joined {
   New_Set := Cheapest plans for joining one
              more input sequence to Curr_Set
              for each access mode;
   Curr_set := New_Set;} }
```

Figure 7: Block-wise Plan Generation

Step 4 - Identification of Query Blocks: As described in Section 3.1, the operators with non-unit scope divide the query into "blocks". At this stage, these query blocks are identified and ordered in a partial ordering as follows: if the output sequence of a query block A is an input for another block B, then $A < B$ in the block ordering.

Step 5 - Block-wise Plan Generation: For each block in increasing topological sort order, a set of possible query evaluation plans needs to be generated. This step is described in detail in the next subsection.

Step 6 - Plan Selection: Finally, when the Start operator at the root of the graph is reached, the most efficient stream access query evaluation plan for the entire query is selected.

4.1 Plan Generation Algorithm

The approach taken to generate a plan for each block is similar in spirit to the plan generation algorithm for SQL query blocks described in [SMALP79], which we will refer to as the Selinger algorithm. A brief explanation of the Selinger algorithm is in order. An SQL query block consists of a list of relations to join, a list of selection predicates on the relations, a list of join predicates and a list of projection attributes. The primary emphasis is on choosing an efficient order to join the relations involved in the query. While evaluating the cost of each join, different join methods are considered. At the first stage, the cheapest way of accessing each individual relation is determined for each "interesting" order. Interesting orders are defined as sort orderings on columns that might benefit Group-By or Order-By operators at higher levels, or that might help in a subsequent sort-merge join. At the next stage, one additional relation is joined to the existing relations to produce all joined pairs of relations. If there are multiple methods for computing a particular join, only the most efficient method is retained for each interesting order. This process is repeated until a join order has been chosen for all relations in the query.

There are many factors that make the problem of sequence query optimization different. As explained in Section 3.1, each query block is either an operator of non-unit scope, or consists of a set of positional joins on input sequences with possibly additional join predicates. The input sequences may be base sequences, constant sequences or the outputs from lower blocks. The order of evaluation of the positional joins and the method used to evaluate each of them remains important to determine. However, issues like access modes, caching and operator scopes need to be introduced into the algorithm. The plan generation phase for each block provides evaluation plans and cost estimates for the output sequence of the block accessed in both stream and probed modes. This information is then used to generate plans for the higher blocks. We now present the algorithm for processing a single block; the pseudo–code is presented in Figure 4.1. The algorithm deals separately with blocks of positional joins and blocks comprised of single operators of non-unit scope. We first discuss the access costs to base sequences, followed by each part of the algorithm.

4.1.1 Access Costs to Base Sequences

The stream mode access cost of a base sequence is determined by the size of the valid range of the sequence, the density of the sequence and the access paths available. The cost is measured as a product of the number of pages to be accessed and the cost of each access. The probed mode cost is determined by the average cost of accessing the record at a given position, multiplied by the number of positions in the valid range of the sequence. A constant sequence has no access cost and a density of one.

4.1.2 Blocks with Non-Unit Scope

Our model considered two types of operators with non-unit scope: aggregates and value offset operators. The scope of the aggregate operators is fixed but is not of unit size. The scope of value offset operators is of variable size. Both the naive algorithm and the incremental algorithm (if applicable) need to be considered. The probed access cost is the probed access cost of the input sequence multiplied by the size of the operator scope. For operators of variable scope, some reasonable estimate needs to be made of the number of input positions that will have to be accessed on average. This estimate can be made from the density of the input sequence, and multiplying this estimate by the probed access cost of the input sequence provides the probed access cost of the output sequence.

For stream access, if the naive algorithm is used, a possibility is to cache the entire effective scope as in Cache-Strategy-A. This is possible for fixed-size sequential scopes that are reasonably small (for instance, a scope of the last million records in the sequence would probably not be cached!). If the incremental algorithm is applied, Cache-Strategy-B can be used. For operators like Previous and Next, this is the case. The stream access interacts with caching to produce a cost that is the sum of the stream cost of the input sequence, the cost of storing each record in the cache, the cost of accessing the cache for each output record and the computational cost. The incremental algorithm is not usable in conjunction with a probed access.

4.1.3 Blocks with Positional Joins

Consider a positional join of two sequences S1(A1,a1,d1) and S2(A2,a2,d2), where A1(A2) corresponds to the stream access cost, a1(a2) to the probed access cost and d1(d2) to the density of sequence S1(S2). The costs for the output sequence are evaluated as follows:

- Stream Access Cost $= minimum(A1 + d1 * a2, A2 + d2 * a1, A1 + A2) + d1 * d2 * output\_span * K$, where the value K is the constant cost associated with a single application of the join predicates. This corresponds to the cheapest of the evaluation plans suggested by Join-Strategy-A and Join-Strategy-B. The first plan accesses the input sequence S1 in a stream access, and for every non-Null record, invokes a probed access on S2. The second plan is the converse of the first, while the third plan invokes a stream access on both sequences. The value $d1 * d2 * output\_span$ is the number of times the join predicates need to be applied.

- Probed Access Cost $= minimum(a1 + d1 * a2, a2 + d2 * a1) + d1 * d2 * output\_span * K$. This corresponds to the cheapest of two evaluation plans. The first accesses the sequence S1 in a probed access and invokes a probed access on S2 for each non-Null record, while the other plan does the converse.

Plan generation for other operators like projections and selections is trivial and we do not explain them here. The number of sequences joined is increased in steps of one until finally all the input sequences to the block have been joined and there are two access plans generated corresponding to the cheapest ways of accessing the output sequence in stream and in probed modes.

4.1.4 Algorithmic Analysis

The Selinger algorithm has been extensively studied and is known to generate the class of "left-deep tree" join plans. [GHK92] contains a detailed analysis of this algorithm. Our optimization algorithm too explores the class of left-deep query trees within each block. The entire query evaluation plan however is not restricted to be a left-deep tree because the graph may be bushy *across* query blocks. Given a block with N positional joins, we can make the following statements about the complexity of the optimization algorithm on the query block:

Property 4.1

a) The time complexity in terms of the number of join plans evaluated $= O\left(N * 2^{N-1}\right)$

b) The space complexity in terms of the maximum number of plans that need to be stored $= O\left(\begin{array}{c} N \\ \lceil N/2 \rceil \end{array}\right)$

Given a complete query plan, the actual query evaluation is straightforward. The Start operator at the root of the plan induces a stream access on its input sequence (i.e. it repeatedly asks for the next non-Null record). When asked to provide its next non-Null record, the operator immediately below the Start operator invokes the appropriate access on its inputs sequences. All the operators in the query graph operate as specified by the query plan.

5 Extensions

The sequence model presented in this paper is limited in the sequences it can represent and in the queries that can be posed. We are currently extending the model to capture a much broader class of sequence databases[SLR]. In this section, we briefly describe some of the directions in which the model is being extended, and the corresponding extensions needed to the optimization framework. Note that we do not consider query language issues which though important are beyond the scope of this paper.

5.1 Extensions to the Model

General Sequences: The model of a sequence can be generalized as a many-many mapping from positions to records. Each position is therefore associated with a *set* of records, and vice-versa. This extension allows the model to represent the kind of temporal data typically represented by temporal databases. A record could be associated with an interval of positions, and at any one position, more than one record might overlap. New operators are required that are based on the view of a sequence as a collection of records, each associated with a set of positions. The new operators include overlap-join, contain-join and precede-join[LM93], as well as typical relational operators like cross-product that operate on the records.

Ordering Domains: Instead of assuming that positions are integers, the existence of explicit ordering domains can be introduced. These ordering domains may be related in a well-known fashion (for instance, the domain of days and the domain of months are related). The knowledge of these relationships leads to operators that can "collapse" or "expand" a sequence from one ordering domain to another. For instance, this would allow a daily sequence to be treated as a weekly sequence so that a weekly average could be computed.

Multiple Orderings: In bitemporal databases[JCG+92], a set of records is typically associated with transaction time as well as valid time orderings. In general, it is useful to be able to associate multiple orderings with the same set of records.

Sequence Groupings: In some situations, it might be desirable to collectively query a group of sequences of similar record type. For instance, given a database of experimental result sequences, a query might ask for those sequences that satisfy some condition. This is a query that operates on "sequence groupings". The model needs to be extended so that the operators manipulate sequence groupings instead of sequences.

5.2 Extensions to the Queries

Generalized Query Graphs: So far, we have restricted the query graph to be hierarchical. One obvious extension is to allow the graph to be a DAG (i.e. to allow the output sequence of an operator to act as the input for more than one other operator). This sharing of sequences raises optimization issues in terms of caching strategies and choice of access modes. The placement of the caches becomes an important issue. For instance caches may be "pushed down" the operator graph to a shared operator, thus avoiding the duplication of cached values. The identification of query blocks may also need to be revisited. A further extension would allow a limited form of recursion into the query graph. This raises a whole set of issues regarding the semantics and correctness of such queries.

Correlated Queries: There is a class of queries that are difficult to express using the model presented in this paper. For instance, consider Example 1.1 dealing with volcanos and earthquakes. Let the query be slightly modified to ask: "For which volcano eruptions was the strength of the most recent earthquake *in the same region* greater than 7.0 on the Richter scale?". Suddenly, the query becomes difficult to express in our model. A relational language like SQL uses nested queries so that the region of each volcano is used as a "correlation" value that is used to determine the most recent earthquake of interest. Such a feature could be added to our model. However, it can no longer be evaluated with a stream access using the techniques described here. The problems raised by correlation are very similar to the issues raised by correlated SQL queries that have been extensively studied[SPL, Day87, GW87, Kim82]. Using the model of sequence groupings though, it is possible to declaratively represent such queries. Further it is possible to devise optimization strategies that can sometimes lead to a stream-access evaluation![SLR].

5.3 Extensions to the Framework

- In terms of the optimization framework, the criterion of minimizing the total execution cost of a sequence query may not always be the most appropriate. For instance, in applications where the data sequences are dynamic, and where the queries are acting as triggers, it may be important to optimize the incremental cost of processing each new arriving data item. This requires a different optimization algorithm, and different evaluation techniques.
- In estimating the costs of various access modes, one possibility that was not considered in this paper was materialization of derived sequences. This is definitely an option to consider, especially when stream access is not possible. Further, if the model were extended to allow for sequences with more than one ordering domain, it might be desirable to materialize and sort intermediate sequences. Finally, with regard to the base sequences, it might be efficient to first reorganize their physical representations before running the query (for example, sort them so that stream access is efficient).

6 Related Work

Since temporal sequences are among the most commonly occuring sequences in real life, there has been much research on temporal databases. We refer the reader to [Soo91] for a bibliography of recent work. Most of the research has concentrated on temporal models and languages[Gad86, CC87, SS88, NA89, WD92, Sno87]. Much of this research has focused on extending existing data models to support temporal data, or extending existing query languages to allow temporal queries to be expressed. Typically, the relational(OO) model is extended by associating a "timestamp" with each tuple(object). The "timestamp" represents the positions in time at which the tuple(object) was "true". The associated query language is extended with predicates that access the "timestamp" of the tuples(objects). There has been some work on query optimization based on such models[GS89a, LM93, NG93]. For instance [GS91, LM93] propose efficient stream access techniques of processing various types of temporal "joins", and [GS89b] proposes an optimization framework for temporal data based on such techniques. Our approach to sequences presented in this paper takes a strongly "positional" view of sequences, as opposed to the models mentioned above. In this aspect, it has been influenced by the model of Time Sequences in [SS87, SK86]

Research into sequence data in contexts other than temporal data include [GJS92], [Ric92], and [SP90]. [GJS92] presents techniques for expressing and evaluating pattern-match queries over a sequence of events. [Ric92] presents a model and operators for manipulating lists in a database. While our paper does not directly cover such issues, the broader model that we are currently developing[SLR] does cover some of these aspects of sequence query processing. [SP90] studies stream processing techniques using a logic language as the underlying data engine. While the emphasis was not on a cost-based query optimization for sequence queries in the database context, the emphasis on stream processing is similar.

Some of the optimization techniques used in this paper are similar to other proposals in the literature. For instance, our concept of operator caches is similar to the notion of "working–memory" in [LM93, SP90]. [GS89b] argues like we do that the kind of statistical information that needs to be maintained in a temporal database is significantly different from that maintained by relational systems. Transformations like the bidirectional propagation of span information are not common in relational systems. The magic rewriting optimization[BR91] and predicate pushdown which propagates selection predicates into sub-queries come close to the spirit of this kind of optimization. Finally, we have not addressed physical storage and access structure issues, and there have been a number of specialized access structures proposed for temporal data that are of relevance(eg. [EWK90, LS89, RS87]).

7 Conclusion

We have presented a framework and an algorithm for optimizing sequence queries. Many of our optimization techniques rely upon the sequentiality of the data and query, and have no counterparts in the domain of relational databases. We have studied these optimizations using a simple model of sequences and a restricted set of query operators. The concept of operator scope and the importance of access modes and caching strategies have been introduced and emphasized. The query plan generation algorithm is the first such concrete algorithm presented for sequence queries to the best of our knowledge. This paper raises a number of fresh research issues that are challenging and that have significant practical benefits.

Acknowledgements

The authors would like to thank Joe Hellerstein, Navin Kabra and Jignesh Patel for useful discussions.

References

[BR91] Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. *Journal of Logic Programming*, 10(3&4):255–300, 1991.

[CC87] James Clifford and Albert Croker. The historical data model and algebra based on lifespans. In

Proceedings of the International Conference on Data Engineering, pages 528–537, 1987.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates and quantifiers. In *Proceedings of ACM SIGMOD '87 International Conference on Management of Data, San Francisco, CA*, pages 23–33, 1987.

[EWK90] Ramez Elmasri, Gene Wuu, and Yeung-Joon Kim. The time index : An access structure for temporal data. In *Proceedings of the International Conference on Very Large Databases*($VLDB$), pages 1–12, 1990.

[Gad86] S.K. Gadia. Towards a multihomogenous model for a temporal database. In *Proceedings of the International Conference on Data Engineering*, pages 390–397, 1986.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *Proceedings of ACM SIGMOD '92 International Conference on Management of Data, San Diego, CA*, pages 9–18, 1992.

[GJS92] 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.

[GS89a] Himawan Gunadhi and Arie Segev. Event-join optimization in temporal relational databases. In *Proceedings of the Fifteenth International Conference on Very Large Databases (VLDB), Amsterdam, Netherlands*, 1989.

[GS89b] Himawan Gunadhi and Arie Segev. A framework for query optimization in temporal databases. In *Fifth International Conference on Statistical and Scientific Database Management Systems*, 1989.

[GS91] Himawan Gunadhi and Arie Segev. Query processing algorithms for temporal intersection joins. In *Proceedings of the International Conference on Data Engineering*, 1991.

[GW87] Richard A. Ganski and Harry K.T. Wong. Optimization of nested sql queries revisited. In *Proceedings of ACM SIGMOD '87 International Conference on Management of Data, San Francisco, CA*, pages 23–33, 1987.

[JCG+92] C.S. Jensen, J. Clifford, S.K. Gadia, A. Segev, and R.T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Record*, 21(3), sep 1992.

[Kim82] W. Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems*, 7, September 1982.

[LM93] Cliff T.Y. Leung and Richard R. Muntz. *Temporal Databases, Theory, Design and Implementation*, chapter 14. Benjamin/Cummings, 1993.

[LS89] David Lomet and Betty Salzberg. Access methods for multiversion data. In *Proceedings of ACM SIGMOD International COnference on Management of Data*, pages 315–324, 1989.

[NA89] S.B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49:147–175, 1989.

[NG93] Sunil S. Nair and Shashi K. Gadia. Algebraic optimization in a relational model for temporal databases. In Richard Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages 390–397, Arlington, Texas, 1993.

[Ric92] Joel Richardson. Supporting lists in a data model. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 127–138, 1992.

[RS87] Doron Rotem and Arie Segev. Physical organization of temporal data. In *Proceedings of the International Conference on Data Engineering*, pages 547–553, 1987.

[SK86] Arie Shoshani and Kyoji Kawagoe. Temporal data management. In *Proceedings of the Twelfth International Conference on Very Large Databases (VLDB), Kyoto, Japan*, pages 79–88, 1986.

[SLR] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Seq: A framework for sequence databases. Submitted for publication.

[SMALP79] Patricia G. Selinger, D. Chamberlin M. Astrahan, Raymond Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of ACM SIGMOD '79 International Conference on Management of Data*, pages 23–34, 1979.

[Sno87] Richard Snodgrass. The temporal query language tquel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Soo91] Michael D. Soo. Bibliography on temporal databases. *ACM SIGMOD Record*, 20(1):14–23, March 1991.

[SP90] D. Stott Parker. *Stream Data Analysis in Prolog*, chapter 8. MIT Press, 1990.

[SPL] Praveen Seshadri, Hamid Pirahesh, and Cliff Leung. Decorrelating complex queries. Submitted for Publication.

[SS87] 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.

[SS88] Arie Segev and Arie Shoshani. The representation of a temporal data model in the relational environment. In *Proceedings of the 4th Conference on Statistical and Scientific Database Management*, pages 39–61, June 1988.

[WD92] Gene Wuu and Umeshwar Dayal. A uniform model for temporal object-oriented databases. In *Proceedings of the International Conference on Data Engineering*, 1992.