The Design and Implementation of a Sequence Database System *

Praveen Seshadri **Miron Livny** **Raghu Ramakrishnan**
Computer Sciences Department
U.Wisconsin, Madison WI 53706
praveen,miron,raghu@cs.wisc.edu

Abstract

This paper discusses the design and implementation of SEQ, a database system with support for sequence data. SEQ models a sequence as an ordered collection of records, and supports a declarative sequence query language based on an algebra of query operators, thereby permitting algebraic query optimization and evaluation. SEQ has been built as a component of the PREDATOR database system that provides support for relational and other kinds of complex data as well.

There are three distinct contributions made in this paper. (1) We describe the specification of sequence queries using the $\mathcal{SEQUIN}$ query language. (2) We quantitatively demonstrate the importance of various storage and optimization techniques by studying their effect on performance. (3) We present a novel nested design paradigm used in PREDATOR to combine sequence and relational data.

1 Introduction

Much real-life information contains logical ordering relationships between data items. "Sequence data" refers to data that is ordered due to such a relationship. Traditional relational databases provide no abstraction of ordering in the data model, and do not support queries based on the logical sequentiality in the data. In earlier work, we had described a data model that could describe a wide variety of sequence data, and a query algebra that could be used to represent queries over sequences [SLR95]. We had also observed that sequence query evaluation could benefit greatly from algebraic optimizations that exploited the order information [SLR94]. This paper describes the issues that were addressed when building the SEQ sequence database system based on these ideas.

SEQ is a component of the PREDATOR[1] multi-threaded, client-server database system which supports sequences, as well as relations and other kinds of complex data. The system uses the SHORE storage manager library [CDF+94] for low-level database functionality like buffer management, concurrency control and recovery. A novel design paradigm provides query processing support for multiple data types, including both sequences and relations. The system implementation has been in progress for more than a year and is currently at approximately 35,000 lines of C++ code (excluding the SHORE libraries). In this paper, the focus is on the SEQ component which provides the $\mathcal{SEQUIN}$ language to specify declarative sequence queries, and an optimization and execution engine to process them. The PREDATOR system is described in detail in [Ses96], and only a high-level overview is presented here.

1.1 The State Of The Art

Financial management products like MIM [MIM94] provide special purpose systems for analyzing stock market data. Current general-purpose database systems provide limited support for sequence data. The Order-By clause in SQL only specifies the order in which answers are presented to the user. Most existing support deals with *temporal* data. While SQL-92 provides a timestamp data type, there are few constructs that can exploit sequentiality. Many temporal queries can be expressed in SQL-92 using features like correlated subqueries and aggregation, these are typically very inefficient to execute. Research in the temporal database community has focused on enhancing relational data models with temporal semantics [TCG+93], but there have been few implementations. Most commercial database systems will allow a sequence to be represented as a 'blob' which is managed by the system, but

* Praveen Seshadri was supported by IBM Research Grant 93-F153900-000 and an IBM Cooperative Fellowship. Miron Livny and Raghu Ramakrishnan were supported by NASA Research Grant NAGW-3921. Raghu Ramakrishnan was also 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.

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

[1] "PREDATOR" is (recursively) the PRedator Enhanced DAta Type Object-Relational DBMS.

interpreted solely by the application program. Some object-oriented systems like O2 [BDK92] provide array and list constructs that allow collections of data to be ordered. The object-relational database system Illustra [Ill94] provides database support for time-series data along with relational data. A time-series is an ADT(Abstract Data Type) value implemented as a large array on disk. A number of ADT methods are implemented to provide primitive query functionality on a time-series. The methods may be composed to form meaningful queries.

1.2 Desired Sequence Functionality

The abstract model of a data sequence is shown in Figure 1. An *ordering domain* is a data type which has a total order and a predecessor/successor relation defined over its elements (also referred to as 'positions'). Examples of ordering domains are the integers, days, seconds, etc. A *sequence* is a mapping between a collection of similarly structured records and the positions of an ordering domain. While every record must be mapped to at least one position, there is no requirement that there be a record mapped to *every* position. The 'empty' positions correspond intuitively to 'holes' in the sequence. The DBMS should efficiently process queries over large disk-based sequences. Further, in most applications, there is sequence data as well as relational and other kinds of data. Complex values like images, or even entire relations can be associated with a single position in a sequence [SLR96], and conversely, there can be a sequence associated with a single relational tuple.

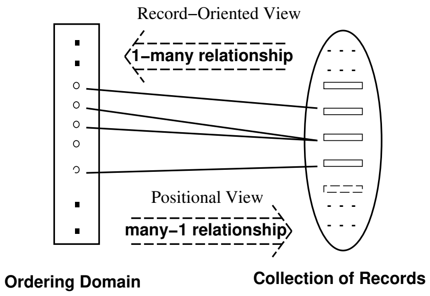

Figure 1: Data Sequence

In [SLR95], we proposed an algebra of Positional sequence query operators. In terms of Figure 1, these operators "view" the sequence mapping from the left (positions) to the right (records). While we do not describe the operators in detail in this paper, the $\mathcal{SEQUIN}$ query language is based on this algebra. The dual mapping from right (records) to the left (positions) leads to operators that are extensions of the relational algebra. Such operators have been extensively investigated in the temporal database community [TCG+93], and they are not considered here.

2 PREDATOR System Design

Object-relational systems like Illustra [Ill94], and Paradise [DKLPY94] allow an attribute of a relational record to belong to an Abstract Data Type (ADT). Each ADT defines methods that may be invoked on values of that type. An ADT can itself be a structured complex type like a sequence, with other ADTs nested inside it. Relations are the *top-level* type, and all queries are posed in the relational query language SQL. There has been much research related to ADT technology, beginning with [Sto86].

The PREDATOR design enhances the ADT notion by supporting "Enhanced Abstract Data Types"(E-ADTs). Both sequences and relations are modeled as E-ADTs . Each E-ADT supports one or more of the following:

Storage Management: Each E-ADT can provide multiple physical implementations of values of that type. The particular implementation used for a specific value may be specified by the user when the value is created, or determined automatically by the system.

Catalog Management: Each E-ADT can provide catalogs that maintain statistics and store schema information. Further, certain values may be named.

Query Language: An E-ADT can provide a query language with which expressions over values of that E-ADT can be specified (for example, the relation E-ADT may provide SQL as the query language, and the sequence E-ADT may provide $\mathcal{SEQUIN}$).

Query Operators and Optimization: If a declarative query language is specified, the E-ADT must provide optimization abilities that will translate a language expression into a query evaluation plan in some evaluation algebra.

Query Evaluation: If a query language is specified, the E-ADT must provide the ability to execute the optimized plan.

The E-ADT paradigm is a novel contribution that differentiates PREDATOR from the traditional ADT-method based approach to providing support for collection types in databases. The difference is crucial to the usability, functionality and performance of queries over complex data types like sequences. The ability to name objects belonging to different E-ADTs allows *any* E-ADT to be the top-level type. This allows users who are primarily interested in sequence data, for example, to directly query named sequences without having to embed the sequences inside relational tuples. While we believe that the E-ADT paradigm can and should be applied to provide database support for *any* complex data type, a detailed discussion of E-ADTs is beyond the scope of this paper. In this current paper, we only wish to place the support for sequence data in the context of the larger database system. The reader is referred to [Ses96] for further details on E-ADTs and the PREDATOR system.

The design philosophy of E-ADTs is carried directly over into the system architecture. PREDATOR is a client-server database in which the server is a loosely-coupled system of E-ADTs . The high-level picture of the system is shown in Figure 2. An underlying theme in the implementation of most components of the system is to allow for extensibility by specifying uniform interfaces. The server is built on top of a layer of common database utilities that all E-ADTs can use. Code

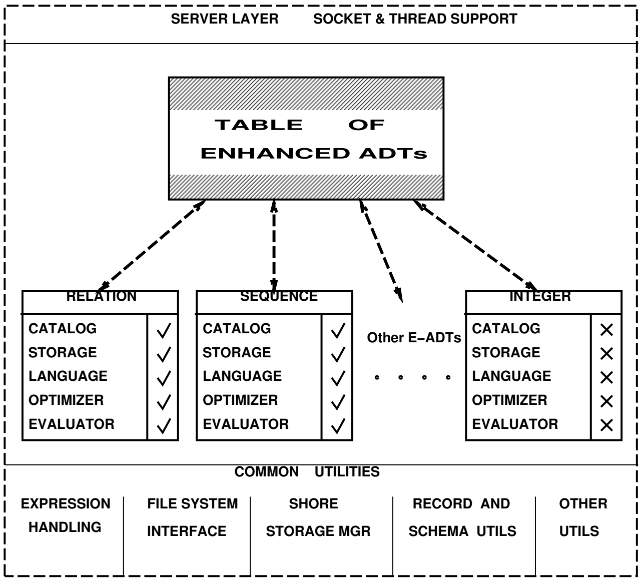

Figure 2: System Architecture

to handle arithmetic and boolean expressions, constant values and functions is part of this layer. An important component of the utility layer is the SHORE Storage Manager [CDF+94]. SHORE provides facilities for concurrency control, recovery and buffer management for large volumes of data. It also provides a threads package that interacts with the rest of the storage management layers; PREDATOR uses this package to build a multi-threaded server.

The core of the system is an extensible table in which E-ADTs are registered. Each E-ADT may (but does not have to) support and provide code for the enhancements described. As shown in the figure, some of the basic types like integers do not support any enhancements. Two E-ADTs that do support enhancements are sequences and relations. The important question to ask is: how does the interaction between sequences and relations occur? The answer is difficult to explain with meaningful examples at this stage because the sequence E-ADT has not yet been described. Instead, we first provide an isolated discussion of the sequence E-ADT . We then return to the issue of how sequences and relations interact in Section 4.

3 The Sequence E-ADT

An important component of the model of a sequence is the ordering domain. Each ordering domain is modeled as a data type with some additional methods that make it an *ordered* type. *LessThan(Pos1, Pos2), Equal(Pos1, Pos2)* and *GreaterThan(Pos1, Pos2)* allow comparisons to be made among positions. *NumPositions(Pos1, Pos2)* counts the number of positions between the two specified end points. *Next(StartPos, N)* and *Prev(StartPos, N)* compute the Nth successor and predecessor of the starting position. All ordering domains are registered in an extensible table maintained by the sequence E-ADT . Additionally, we need to capture the hierarchical relationship between various ordering domains. For instance, Figure 3 shows one set of hierarchical relationships between common temporal ordering domains. A table of *Collapses* is maintained by the sequence E-ADT . Each *Collapse* represents an edge in the hierarchy and provides methods that map a position in one ordering domain to a position or set of positions in the other domain. For example, a Collapse involving 'days' and 'weeks' maps each day to the week it belongs in, and each week to the set of days of that week.

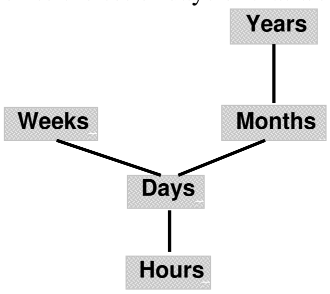

Figure 3: Sample Ordering Hierarchy

As shown in Figure 1, a sequence models a many-to-one mapping between positions in the ordering domain and a set of records. As a simplification, we restrict each record to be mapped to a single position (the one-to-many abstraction is modeled by making copies of the record). In SEQ, the position mapping is maintained as an explicit attribute of each record. Although there are different storage implementations of sequences in the system, they all provide certain common interface methods:

OpenScan(Cursor), GetNext(Cursor), CloseScan(Cursor). These methods provide a scan of the sequence in the forward order of the ordering domain. Any positions in the domain which are not mapped to a record are ignored in the scan.

GetElem(Pos). This finds the record at the specified position in the sequence (or fails if none exists at that position).

3.1 Experimental Database

We wish to quantitatively demonstrate (a) some possible choices of storage techniques for sequences, and (b) the importance of various optimization techniques. The sequences used in the experimental database were generated synthetically. While we could have used a real-life data set instead, it would have been more difficult to control various properties of each sequence. The properties of interest in each sequence are: (1) the *cardinality*, i.e., the number of records in the sequence, (2) the *record width*, i.e., the number of bytes in each record, (3) the *density*, i.e. the percentage of the positions in the underlying ordering domain that are non-empty. All the sequences have an *hourly* ordering domain and start at midnight on 0100/01/01 (i.e. January 1st in the year 100 AD). We considered sequences with two different densities: 100% and 20%. The cardinality of each sequence was either 1000 (1K), 10000(10K) or 100000(100K) records. For sequences of each density, the final time-points are shown in Table 1[2].

[2] The entries in the table are approximate since they only show the last day, not the last hour.

	Cardinality		
Density	1K	10K	100K
100%	0100/02/15	0101/04/02	0112/07/16
20%	0100/08/16	0106/05/09	0162/11/15

Table 1: Synthetic Data Upper Bounds

Notice that because of empty positions, the 20% density sequences span about 5 times as many positions as the 100% density sequences. The empty positions were chosen randomly so that the overall density was 20%. The first field of every sequence record is an SQL time-stamp value. Different sequences were generated with 1, 5, 10 and 20 fields in addition to the timestamp. The values in the fields were 4-byte integers generated randomly between 0 and 1000. All experiments were performed on a SUN-Sparc 10 workstation equipped with 24MB of physical memory. The data was loaded into a SHORE storage volume implemented on top of the Unix file system. The SHORE storage manager buffer pool was set at 200 8K pages, which is smaller than the available physical memory, but is realistic for this small sample database. Logging and recovery was turned off to mimic a query-only environment. In all the experiments, the queries used contain a final aggregate over the entire sequence, thereby minimizing any time spent in printing answers. Each query was executed four times in succession, the maximum and minimum execution times were excluded, and the average of the other two times was used as the performance metric.

3.2 Storage Implementation

SEQ supports two repositories for sequence data, the Unix file system and the SHORE storage manager. The default repository is built using the SHORE storage manager library. Data volumes maintained by SHORE can reside either directly on raw disk, or on the file system; our experiments used the latter approach. A sequence can also be stored as an ascii file on the Unix file system. Much real-world sequence data currently exists in this format. It may be more expedient to directly run queries off this data, instead of first loading it into the database. Of course, this repository does not provide any of the database properties of concurrency control, recovery, etc. We studied three alternative implementations of a sequence using SHORE:

File: SHORE provides the abstraction of a 'file' into which records can be inserted. A scan of the file returns the records in the order of insertion; this enabled us to implement a sequence as a SHORE file. One advantage of this implementation was that we could code it with minimal effort. The major drawback is that the storage manager imposes several bytes (at least 24) of space overhead for every record, in addition to a large space overhead for creating a file. While concurrency control is available at the record level, inserts in the middle of a sequence are difficult to implement without an index.

IdList: In order to eliminate the space overhead per file, a sequence is stored as an array of record-ids. Each such array is a SHORE large object, which can grow arbitrarily large. Each record-id occupies 4 bytes, and identifies the appropriate record. All records are created in a single "super" file. While the space overhead for each file is eliminated, the other drawbacks still remain (primarily, the storage overhead per record). Further, since the record-id is a logical identifier in SHORE, this needs to be mapped to an internal physical identifier when the record needs to be retrieved. This problem could be avoided by using the less portable solution of actually storing the list of physical identifiers instead. Concurrency control is now at the level of the entire sequence, but inserts are easier to code because SHORE allows new data to be inserted into the middle of a large object.

Array: In this implementation, a sequence is an array of records. The array is implemented using a single SHORE large object which contains all the records. Since we expect many sequences to be irregular (i.e., have empty positions), we chose a compressed array representation in which no space is wasted for an empty position. This can dramatically reduce space utilization for data sets of very low density. However, this makes some operations within a sequence (like positional lookup, insert and delete) more expensive to implement. Variable length records require additional complex code. However, there are two important benefits to this implementation: the per-record space overhead is minimal and there is physical sequentiality for the records of a sequence. With fixed-size records in a mostly-query environment, this should be the implementation of choice.

Experiment 1: We measured the time taken to scan each of the example sequences stored using each of the implementation techniques just described. A scan is the most basic sequence operation that is used in almost every query. Consequently, the time taken to scan a sequence is a suitable indicator of the efficiency of the storage implementation. The results for the sequences with density 100% are shown (there was no significant difference with the 20% density sequences, hence they have been omitted). The actual $\mathcal{SEQUIN}$ query run was:

```
PROJECT count(*)
FROM    <data_sequence>
ZOOM  ALL;
```

Figures 4, 5, and 6 show the results for the sequences of cardinality 100K, 10K and 1K respectively. In all the graphs, the number of fields in each record varies along the X-axis, while the runtime is plotted on the Y-axis. For all the implementations, the scan cost grows with the width of the records. Note that the SHORE Array implementation is the most efficient whatever the cardinality or width of the sequence. Therefore, in all the remaining experiments, this was the storage implementation used. The SHORE File implementation is worse than SHORE Array because of the file handling overhead per record. IdList is the worst SHORE implementation primarily because of the added cost of converting from logical to physical identifiers. The Unix ascii file implementation is the most sensitive to the width of the data records because each attribute needs to be parsed at run-time to convert it from ascii to binary format.

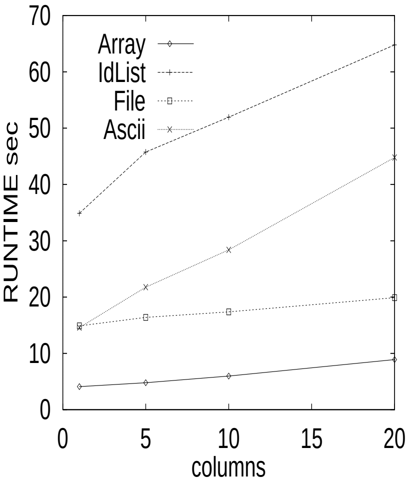

Figure 4: Expmt.1 : Card 100K

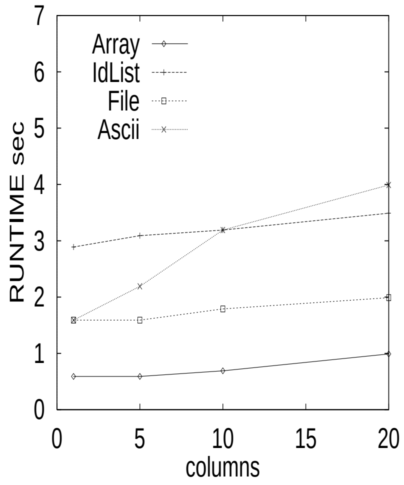

Figure 5: Expmt.1 : Card 10K

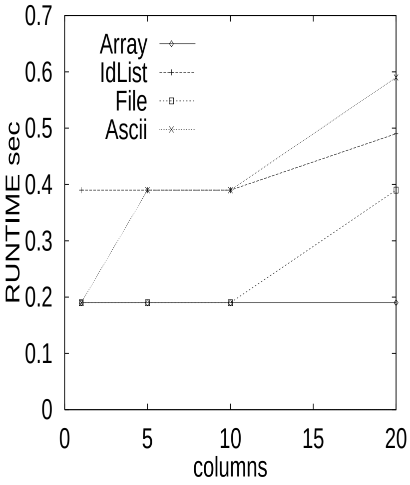

Figure 6: Expmt.1 : Card 1K

3.3 $\mathcal{SEQUIN}$ query language

$\mathcal{SEQUIN}$ [3] is a declarative language for sequence queries, similar in flavor to SQL. The result of a $\mathcal{SEQUIN}$ query is always a sequence. The overall structure of a $\mathcal{SEQUIN}$ query is:

```
PROJECT  <project-list>
FROM     <sequences-to-be-merged-on-position>
[WHERE   <selection-conditions>]
[OVER    <start-window> TO <end-window>]
[ZOOM    <zoom-info>];
```

We now explain the various constructs using simple examples based on the following sample database. Consider the sequences Stock1 and Stock2 representing the hourly price information on two stocks. Both sequences have the same schema: {time:Hour, high:Double, low:Double, volume:Integer}, where the 'time' field is underlined to show that it defines the order.

The first query estimates the monetary value of Stock1 traded in each hour when the low price fell below 50. The answer is a sequence with the monetary value computed for each such hour.

```
PROJECT ((A.high+A.low)/2)*A.volume
FROM    Stock1 A
WHERE   A.low < 50;
```

The query demonstrates the use of the PROJECT and WHERE clauses. The PROJECT clause with a target list of expressions is similar to the SELECT clause of SQL.There is no output record for positions at which the WHERE clause condition fails; these are empty positions in the output sequence. Since the result is a sequence of the desired values, it should have an ordering attribute; however none exists in the PROJECT list. In such cases, the ordering attribute from the input sequence is automatically added to the output schema.

We now consider finding the 24-hour moving average of the difference between the high prices of the two stocks.

```
PROJECT avg(A.high - B.high)
FROM    Stock1 A, Stock2 B
OVER    $P-23 TO $P
```

This query demonstrates the use of the FROM clause, and the OVER clause for moving window aggregates. When there is more than one sequence specified in the FROM clause, there is an implicit join between them on the position attribute (in this case, on 'time'). Since this is a declarative query, the textual order of the sequences in the FROM clause does not matter. Note that the PROJECT clause uses the *avg* aggregate function. The set of records over which the aggregate is computed is defined by the moving window of the OVER clause. In this case, the window spans the last 24 hours, but in general, the bounds of the window can use any arithmetic expression involving addition, subtraction, constant integers and the special $P symbol representing the 'current' position for which the record is being generated. Empty positions in the input sequence are ignored as long as there is at least one valid input record in the aggregation window.

Next, we show a rather complex query that demonstrates the possible variations in the FROM clause. The desired answer is a sequence containing for every hour, the difference between the 24-hour moving average of the high price of Stock1, and the high price of Stock2 at the most recent hour when the volume of Stock2 traded was greater than 25,000. The answer sequence is only of interest to the user after hour 2000.

```
// first define the moving average as a view
CREATE VIEW MovAvgStock1 AS (
       PROJECT avg(C.high) as avghigh
       FROM    Stock1 C
       OVER    $P-23 TO $P);
// then use the view in the query
PROJECT A.avghigh - B.high
FROM MovAvgStock1 A,
     Previous(PROJECT D.high
              FROM    Stock2 D
              WHERE   D.volume > 25,000) B
WHERE $P > 2000;
```

Note that the sequences in the FROM clause may themselves be defined using another $\mathcal{SEQUIN}$ query block. This

[3] SEquence QUery INterface.

may be effected using a view (as is the MovAvgStock1 sequence A), or a nested query block defining a sequence expression (as is the sequence B). Three special modifiers with functional syntax are allowed in the FROM clause: *Next, Previous and Offset*. *Previous* (as in this example) defines a sequence which associates with every position the record at the most recent non-empty position in the input sequence. Remember that sequences need not be regular, and consequently there can be positions which are not associated with any records. The Previous modifier fills these 'holes' with the most recent record. Similarly, *Next* defines a sequence in which the holes are filled with the most imminent record. Both these modifiers can take a second optional argument which specifies how many such steps to take (which is 1 by default); for example, Previous(S, 2) defines a sequence of the second-most recent input record at each position. The *Offset* modifier defines a sequence in which the position-to-record mapping of the input sequence is shifted by a specified number of positions. Finally, note that the WHERE clause can also use the $P notation to access the 'current' position attribute.

The next query demonstrates the use of the ZOOM clause to exploit the hierarchical relationship between ordering domains[4]. Here is the $\mathcal{SEQUIN}$ query to compute the daily minimum of the volume of Stock1 traded every hour.

```
PROJECT min(A.volume)
FROM    Stock1 A
ZOOM    days
```

We assume that 'days' is the name of an ordering domain known to the system, and that there is a Collapse registered with the system from 'hours' (the ordering domain of the input) to 'days'. The answer sequence has an implicit attribute of type 'days' that provides the ordering. If the resulting ordering domain is at a coarser granularity in the hierarchy than the source ordering domain, as in this example, then the PROJECT clause must be composed of aggregate expressions.

Our final example shows how the ZOOM clause can perform conditional collapses. Suppose that just as in the previous query, we want to compute the minimum volume of Stock1 traded over consecutive periods of time. However, these periods are not well-defined like 'days' or 'weeks'. Instead, they depend on the data. Specifically, the periods may be bounded by those times when the high and low values were very close (implying an hour of stability for the stock). This can be expressed as follows:

```
PROJECT min(A.volume)
FROM    Stock1 A
ZOOM    BEFORE (A.high - A.low < 0.1);
```

The query states that the aggregation window includes records upto but not including the record which satisfies the stability condition. If the last record is also to be included in the aggregation window, the word BEFORE is replaced by AFTER. As a final variant, the ZOOM clause could simply be 'ZOOM ALL', specifying that the aggregation is to be performed on the entire sequence. These versions of the ZOOM operator generate sequences that are ordered by an implicit integer field that starts at value 1 and increases in increments of 1 (since this is the only meaningful sequence ordering for the result).

In this paper, we have omitted discussion of some other features of $\mathcal{SEQUIN}$ including a construct to re-define the ordering field of a sequence, update constructs and DDL features. A $\mathcal{SEQUIN}$ query is parsed into a directed acyclic graph of algebraic operators, which is then optimized by the query optimizer. We have described the algebra operators and some optimization techniques in [SLR95, SLR94].

3.4 Query Optimizations

This section describes the effects of four categories of implemented optimizations. Each optimization is first explained in principle, and then demonstrated by means of a performance experiment. We have tried to keep the queries in the experiments as simple as possible, in order to isolate the effects of each optimization.

3.4.1 Propagating Ranges of Interest

This class of optimizations deals with the use of information that limits the range of positions of interest in the query answer. There are two sources of such information: one is from selection predicates in the query that use the position attribute. Experiment 2 demonstrates the benefits of propagating such selections into the sequence scans. The other source is from statistics on the valid ranges of positions in each sequence. These valid ranges can be propagated through the entire query as described in [SLR94]. Experiment 3 demonstrates how the valid-range can be used for optimization.

Experiment 2:

```
PROJECT count(*)
FROM 100K_10flds_100dens S
WHERE S.time > "<timestamp1>"
ZOOM ALL;
```

This query is a variant of the query used earlier to measure the performance of a sequence scan. In this case, the scan is over only a portion of the sequence. SEQ can optimize the query by pushing the selection predicate into the scan of the sequence. Since the default implementation of sequences in SEQ expects irregular sequences and uses a compressed Array implementation, there is no simple way to directly access a specific position. If the selection range is from Pos1 to Pos2, the first record within the range (at Pos1) is difficult to locate exactly. Based on the density of the sequence, the valid range of the sequence, and the desired selection range, SEQ performs a weighted binary search to get close to the correct starting position. However, if the query is modified so that the $>$ is replaced by a $<$ (i.e. the desired range is at the beginning of the sequence), then the binary positioning is not needed.

We studied the effect of varying the predicate selectivity from 1% to 100%. We ran the experiment twice, once with the

[4] The word "zoom" is used because the action of moving down or up through the ordering hierarchy is similar to zooming in and out with a lens.

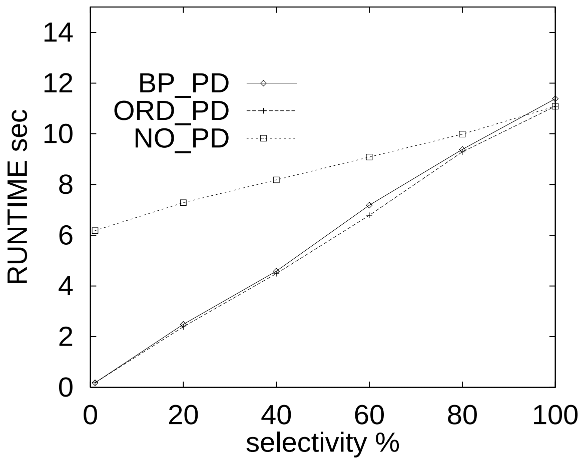

Figure 7: Range Selections At Start: Expmt. 2

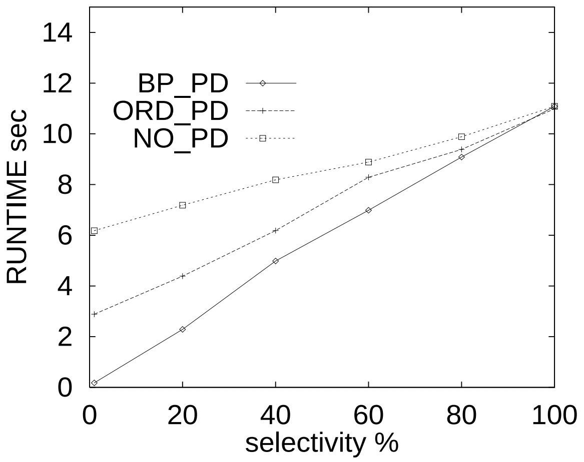

Figure 8: Range Selections At End: Expmt. 2

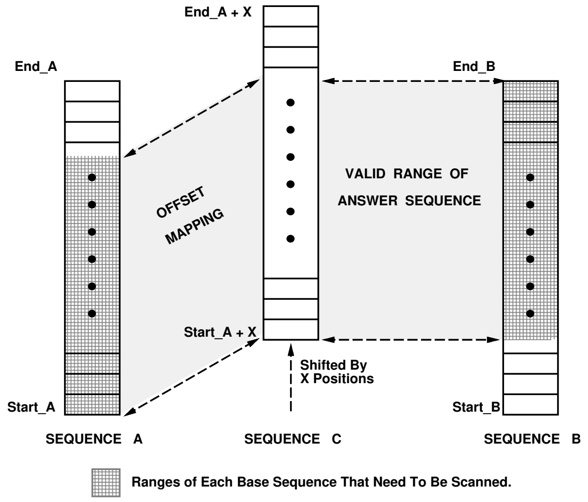

Figure 9: Range Propagation: Intuition

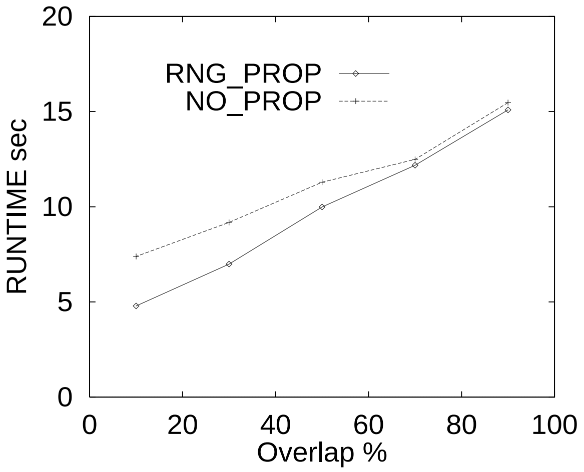

Figure 10: Range Propagation: Expmt. 3

selection windows at the start of the valid-range (**at_start**), and once with the selection windows at the end(**at_end**). Three algorithms were considered: no selection push-down (NO_PD), simple push-down with no binary-positioning (ORD_PD), and selection push-down with binary positioning (BP_PD).

The results for **at_start** are shown in Figure 7. The predicate selectivity is shown on the X-axis, and the query execution time is on the Y-axis. While there is no difference between BP_PD and ORD_PD (since the predicate is at the start of the window), NO_PD performs much worse because the entire sequence is scanned. As the selectivity increases, all the algorithms become more expensive because there is additional work being done in the final count aggregate.

The results for **at_end** are shown in Figure 8. The performance of NO_PD is the same as in the **at_start** experiment. The performance of BP_PD is almost the same as in the **at_start** experiment, because it is able to use the selection information to position the scan at the appropriate start position. On the other hand, ORD_PD cannot do this, and therefore scans the entire sequence. However, ORD_PD can apply the selection predicate at a lower level in the system and therefore performs better than NO_PD. Note that the BP_PD algorithm, which performs best, can only be applied if the valid range and density statistics are maintained for the sequences.

Experiment 3:

```
// View applies selection to the base sequence
CREATE VIEW ViewSeq AS (
        PROJECT A.fld2
        FROM a100K_10_100 A
        WHERE A.fld1 > 900);
// Merge B with offset ViewSeq
PROJECT count(*)
FROM 100K_5flds_100dens B,
     Offset(ViewSeq, <offset_distance>) C;
```

This query joins two sequences on position; however, one of the sequences is first shifted by some specified number of positions. Each of the base sequences in this query has 100K records spanning an identical range (see Table 1). However, since one of them is shifted, neither of the sequences needs to be scanned in its entirety; only the mutually overlapping region needs to be scanned. This is shown intuitively in Figure 9. The valid-range propagation optimization is able to recognize

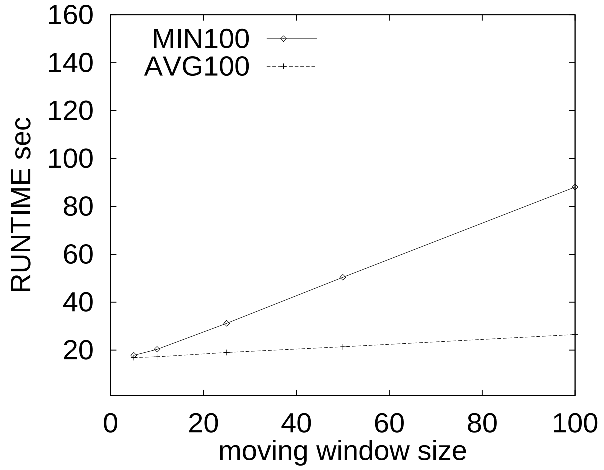

Figure 11: Expmt.4: 100% Density

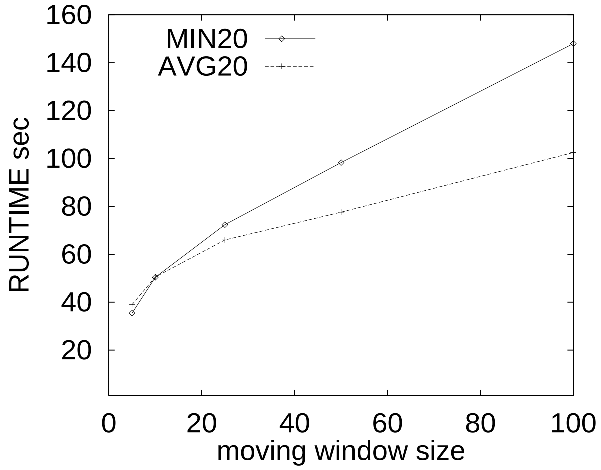

Figure 12: Expmt.4: 20% Density

such optimization opportunities in all SEQ queries.

We varied the overlap from 90% of the valid-range to 10%, and executed the query with (RNG_PROP) and without (NO_PROP) the valid-range optimization. The results are shown in Figure 10. The smaller the overlap between the two sequences, the better is the relative performance of RNG_PROP. The difference between the two lines is due to the work saved in scanning the sequence.

3.4.2 Moving Window Aggregates

All aggregate functions in SEQ (used in both relational and sequence query processing) are implemented in an extensible manner. Each aggregate function provides three methods: *Initialize(), Accumulate(record), Terminate()*. This abstract interface allows the aggregation operator to compute its result incrementally, independent of the specific aggregate function computed. The presence of moving window aggregates in SEQ creates new opportunities for optimization. Note that in relational aggregates, the input data is partitioned into *disjoint* portions over which the aggregation is performed. Contrast this with the moving window sequence aggregates in which there is an *overlap* between successive aggregation windows. For example, consider the 3-position moving average of a sequence 1,2,3,4,5. Once the sum $1+2+3$ has been computed as 6, this computation can be used to reduce the work done for the next aggregate. Instead of adding $2+3+4$, one could instead compute $6-1+4$. Due to the small aggregation window in this example, there is little benefit. However, when the windows become larger and the operations are more expensive, there can be significant improvements due to this approach. Importantly, the time required for aggregation is independent of the size of the window.

While some aggregates like Count, Sum, Avg and Product are directly amenable to this optimization, others like Min, Max, Median and Mode are not. We call this the *symmetry* property of an aggregate function. In order to exploit the symmetry property in an extensible manner, we require each aggregate function to provide two more methods: *IsSymmetric()* and *Drop(record)*. Experiment 4 demonstrates the importance of exploiting symmetric aggregates.

Experiment 4: We considered queries of the form

```
// Define the moving aggregate
CREATE VIEW MovAggr AS
    (PROJECT <aggr_function>(S.fld1)
     FROM    <data_sequence> S
     OVER    $P-<window_size> TO $P);

// Count records to minimize printing
PROJECT count(*)
FROM    MovAggr
ZOOM  ALL;
```

Moving window aggregates are among the most important sequence queries posed in stock market analysis applications. Our example query is the simplest form of a moving aggregate (with a final count operator thrown in as usual to eliminate the time for printing answers). This experiment was restricted to only the 100K_10cols_100dens and 100K_10cols_20dens sequences. The window size was varied from 5 to 100, while the aggregate functions tried were MIN (non-symmetric) and AVG(symmetric).

The results for the 100% density sequence are shown in Figure 11. Notice that the performance of MIN100 grows linearly with the size of the aggregation window. This is because the entire aggregation window has to be processed for each MIN aggregate computed. In comparison, the performance of AVG100 is almost independent of the size of the aggregation window. The slight dependence of AVG100 on the window size has an interesting reason. Given a particular timestamp, it is more expensive to compute the 100th previous timestamp, than the 10th previous timestamp. Simple arithmetic cannot be applied to temporal ordering domains because the variable number of days in a month has to be accounted for.

The results for the 20% density sequence are shown in Figure 12. Note that a moving aggregate over a sequence with holes generates many more records than exist in the input sequence. Assume that there is an input record at hour 100 and the next record is at hour 102. A 3-hour moving aggregate sequence has a value at hour 101 as well, because there is at least

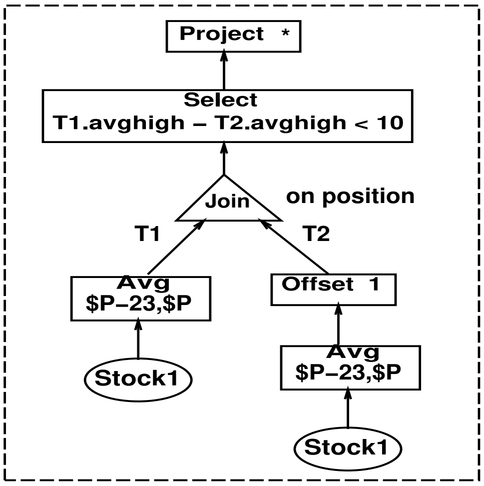

Figure 13: Graph 1: Repeated Computation

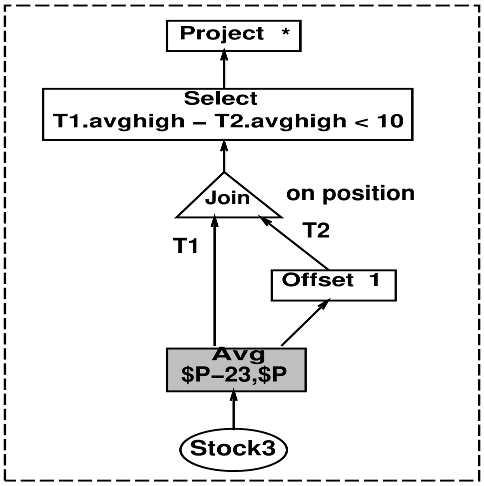

Figure 14: Graph 2: Common Sub-Expression

one record in its aggregation window from hour 99 to hour 101. This explains why the cost of both aggregates increases with window size. Since the density is low (20%), there are also fewer records in each aggregation window, and the relative difference between the AVG20 and MIN20 grows more slowly with the size of the aggregation window. The relative difference between AVG20 and MIN20 at window size 100 is about the same as the relative difference between AVG100 and MIN100 at window size 20. This is to be expected, because the ratio of the densities of the two sequences is also 100:20.

3.4.3 Common Sub-Expressions

The same sequence may be accessed repeatedly in different parts of a query. For example, the following query compares the values of a moving average at successive positions looking for stability in the stock prices.

```
// View: moving average over last 24 hours
CREATE VIEW MovAvgStock1 AS
     (PROJECT avg(S.high) as avghigh
      FROM    Stock1 S
      OVER    $P-23 to $P);

// Check change in moving average
PROJECT *
FROM MovAvgStock1 T1, Offset(MovAvgStock1, 1) T2
WHERE T1.avghigh - T2.avghigh < 10.
```

Figures 13 and 14 show two possible algebraic query graphs that can be constructed from this query. The meaning of each query graph is obvious. The difference between the two query graphs is that one uses a common sub-expression, while the other does not. Common sub-expressions occur frequently in sequence queries, so this is an important issue. When a query graph with a common sub-expression is constructed for a *relational* query, the query optimizer chooses one of two options. One option is to repeatedly evaluate the common sub-expression; this is equivalent to using the version of the query graph without a common sub-expression(Figure 13). The other option is to compute the sub-expression once, store the result, and repeatedly access the stored result. For sequence queries, we will show that materializing intermediate results is not a desirable option.

By an analysis of the query graph and the scopes of the various operators involved, SEQ can determine exactly how much of the common sub-expression result should be cached, so that the entire query can be evaluated with a single stream access of the common sub-expression. In other words, neither is the common-subexpression evaluated multiple times, nor is it materialized [Ses96].

Experiment 5: We ran the very same query shown above (except that the Stock1 sequence was replaced by 100K_10flds_100dens). We varied the size of the aggregation window from 10 to 100; as the window size increases, so does the cost of the common sub-expression. The query execution time was measured with the SEQ optimization (Common-Subexp) and with repeated evaluation (Re-Computed). The results are shown in Figure 15. The common sub-expression optimization used by SEQ obviously performs much better than repeated evaluation. As the cost of the common sub-expression increases (i.e., as the window size grows), this optimization becomes extremely important. While we have ignored the possibility of materializing the intermediate result,

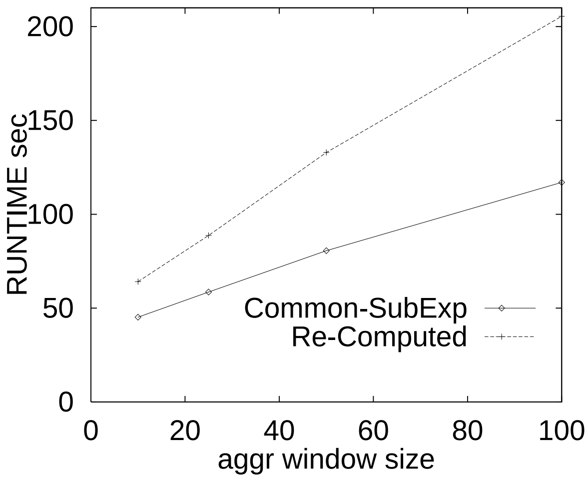

Figure 15: Common Subexpressions: Expmt. 5

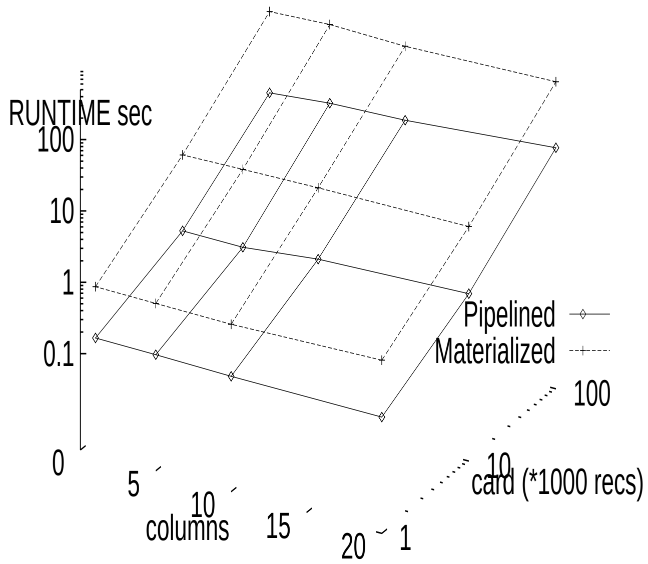

Figure 16: Pipelining: Expmt. 6

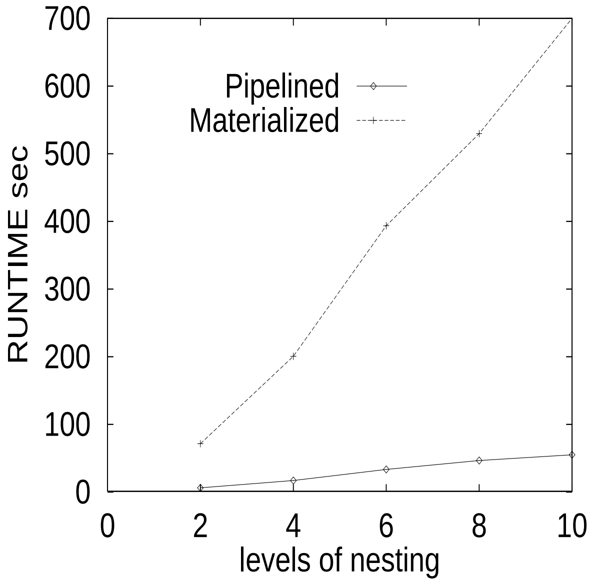

Figure 17: Pipelining: Expmt. 7

the next experiment will show that materialization is very inefficient in general.

3.4.4 Operator Pipelining

An important optimization principle in SEQ is to try and ensure *stream access* [SLR94] to the stored sequence data as well as to intermediate data; i.e., each sequence is read in a single continuous pipelined stream without materializing it. This is accomplished by associating buffers with each operator, to cache some relevant portion of the most recent data from its inputs. In our example of the hourly sequences, a 24-hour moving aggregate would need a buffer of no more than the 24 most recent input records. This 'window' of recent data is called the *scope* of the operator. All the operators in the algebra have fixed size scopes in a particular query. Consider the simple query below that scans a sequence and computes an aggregate over the entire data:

```
PROJECT count(*)
FROM    <data_sequence>
ZOOM  ALL;
```

Experiment 6 will show that there is a tremendous penalty to pay for failing to pipeline even such a simple query between the Scan operator and the Count operator. Experiment 7 shows that when the query becomes complex, with several nested operators, the relative importance of pipelining becomes even more clearly defined.

Experiment 6: We ran the query shown above over all the sequences in the sample database. The results with the pipelining optimization (Pipelined) and without it (Materialized) are shown in the 3-D graph of Figure 16. The number of columns in each record varies along the X-axis, while the sequence cardinality varies on a logarithmic scale on the Y-axis. The Z-axis shows the query execution time on a logarithmic scale. Once again, we only show the results for the 100% density sequences (the 20% density results are similar). Notice that materialization increases the cost by almost an order of magnitude!

Experiment 7: In this experiment, we want to show the effects of increased query complexity on materialized execution. Section 3.3 had several examples of non-trivial queries. By using the view mechanism, many complex queries can be generated. It is difficult to choose a single representative for all complex queries. Instead, since the purpose of this experiment is to isolate and study the performance of pipelining and materialization, we use a query that, though not intuitively meaningful, can be varied in a controlled manner. We consider one particular data sequence (100K_10flds_100dens) and vary the number of levels of operators in the query from 2 to 10. For instance, with 4 levels, the corresponding $\mathcal{SEQUIN}$ query is

```
PROJECT count(*)
FROM    (PROJECT *
         FROM (PROJECT *
               FROM 100K_10flds_100dens)) S;
ZOOM  ALL;
```

We disabled the SEQ optimization that merges consecutive scans which would otherwise reduce all these queries to a common form. The results with and without the pipelining optimization are shown in Figure 17. The X-axis shows the number of levels of nesting in each query, while the Y-axis shows the query execution time. Notice that while the cost of the default SEQ execution with pipelining grows moderately (due to the presence of more operators on the query execution path), the cost of the materialized execution grows dramatically with the complexity of the query expression.

4 Combining Sequences and Relations

We now return to the issue of how sequences and relations interact in PREDATOR. The important questions are: how does a query access both relational and sequence data, how does optimization of this query occur, and how is the query evaluated? In order to discuss these questions, we slightly extend the example that we used to explain the $\mathcal{SEQUIN}$ language in Section 3.3. Consider a relation *Stocks* of securities that are traded on a stock exchange, with the schema *(name:String, stock_history:Sequence)* . The *stock_history* is a sequence of hourly information on the high and low prices, and the volume of the stock traded in each hour.

4.1 Nested Language Expressions

In this example, since the sequence data is nested within the relational data, it is appropriate for the user to think of the relational E-ADT as the top-level type. A query will therefore be posed in the relational query language (SQL) with nested query expressions in the sequence query language ($\mathcal{SEQUIN}$).

Let us consider the SQL query to find for each stock, the number of hours after hour 3500 when the 24-hour moving average of the high price was greater than 100.

```
SELECT S.name, SEQUIN(
  "PROJECT count(*)
   FROM (PROJECT avg(H.high) as avghigh
         FROM   $1  H
         OVER   $P-23 TO $P ) A
         WHERE  A.avghigh > 100 AND $P > 3500
         ZOOM ALL",
   S.stock_history)
FROM   Stocks S;
```

The SQL query has the usual SELECT clause target list of expressions. One of these expressions is a $\mathcal{SEQUIN}$ query, whose syntax is functional. There is one such implicit function for every E-ADT language registered in the system. The first argument to the $\mathcal{SEQUIN}$ function is a query string in that language. Any parameters to be passed from SQL (the calling language) to the embedded query in $\mathcal{SEQUIN}$ are provided as additional arguments. These parameters are referenced inside the embedded query using the positional notation \$1, \$2, etc. In this particular query, the passed parameter (S.stock_history) is a sequence. Note that the SQL language parser does not know about the grammar of the embedded language, and merely treats the $\mathcal{SEQUIN}$ subquery as a function call whose first argument is some string. For the SQL parser, this query is treated in the same manner as the following query would be:

```
SELECT S.name, Foo("hello", S.stock_history)
FROM   Stocks S;
```

As part of the type-check of the SQL query, the type of the $\mathcal{SEQUIN}$ function is also checked. This causes the embedded $\mathcal{SEQUIN}$ query to be parsed by the parser of the sequence E-ADT . It is no longer sufficient to identify the type of every parameter passed. In this example, the parameter is of a sequence type, but this is not sufficient to type-check the embedded query. The schema information for the sequence must also be specified along with the type. This implies that throughout the system code that handles values and expressions, meta-information like the schema must be maintained as part of the type information. The return type of the $\mathcal{SEQUIN}$ query expression is a sequence as usual. Expressions of a particular type may be cast to another type using cast functions that are registered with the system. The cast mechanism is also used to convert sequences into relations. The cast from relations to sequences additionally requires the specification of the order attribute.

When optimizing a nested query, each E-ADT is responsible for optimizing its own query blocks. Since the nested languages are introduced in the guise of functions, each optimizer must be sure to 'plan' any function invoked. Planning a function like $\mathcal{SEQUIN}$ causes the optimization of the embedded query to be performed. In this example, the SQL optimizer is called on the outer query block, and the $\mathcal{SEQUIN}$ optimizer operates on the nested query block. There is currently no optimization performed across query blocks belonging to different E-ADTs . During execution of the SQL query, the nested $\mathcal{SEQUIN}$ expression is evaluated just as any other function would be. There are several other implementation details that are described in [Ses96].

4.2 Comparison with Existing Systems

Some current systems like Illustra [Ill94] support sequences (more specifically, time-series) as ADTs with a collection of methods providing query primitives. A query is a composition of the primitive functions or methods. While queries in $\mathcal{SEQUIN}$ are declarative, queries based on the functional composition of methods have a procedural semantics. When a query expression involves the composition of more than one of these methods, little or no inter-function optimization is performed, and each individual method is evaluated separately. While we did perform a performance comparison with Illustra [Ill94], we are not permitted to discuss those results. Instead, we provide a qualitative comparison.

Experiments 6 and 7 showed that materialization can perform an order of magnitude worse than pipelining with stream-access. In the ADT-method approach, pipelining is not possible without inter-function optimization. The simple query of Experiment 6 is expressed in a form similar to *Count(Scan(S))*. Since methods are independently evaluated, the result of the scan is materialized, and then the count of this materialized result is computed. The optimizations that propagate valid ranges and selection predicates (Experiments 2 and 3) once again require the ability to push range selections from one function to another. Consequently, ADT-method based systems do not exploit these optimizations. Experiment 5 showed that the common sub-expression optimization could reduce query execution time by almost a factor of two. An ADT-method approach cannot identify common sub-expressions without inter-function optimization, let alone take advantage of them to optimize query execution. Putting these together, the ADT-method approach is unable to apply optimization techniques that could result in overall performance improvements of approximately two orders of magnitude! In order for these systems to derive some of the efficiencies of the SEQ approach, they will have to adopt some or all of the system design used by PREDATOR [SLR96, Ses96].

5 Related Research

Research work directed at modeling time-series data [SS87, CS92, Sto90] provided initial direction to our efforts. The model of a time-series in [SS87] is similar to ours, and an SQL-like language was also proposed; implementation issues were discussed in the context of how the model could be mapped to a relational data model.The Tangram Stream Processor [Sto90] uses transducers and stream processing to query sequences in a logic programming context; there are many similarities between the stream processing ideas in this work and in SEQ. The dual nature of sequences (Positional versus Record-Oriented) is recognized by the temporal query language of [WJS93]. The extensive work on temporal database modeling, query languages, and query processing [TCG+93] is mostly complementary to our work, because it involves changes to relations and to SQL [TSQL94]. However, it would be interesting to study how time-ordered sequences can be efficiently converted into relations with time-stamps, and vice-versa.

While most object-oriented database proposals include constructors for complex types like lists and arrays [VD91, BDK92], they can either be treated as collections, or manipulated using a primitive set of methods; no facilities for sequence queries are provided. The work described in [Ric92] is an exception, and proposes an algebra based on temporal logic to ask complex queries over lists. There have also been languages proposed and systems built to match regular expression patterns over event sequences [GJS92]. This work is complementary to ours, since SEQ is oriented to more traditional database queries, and currently does not have powerful pattern-matching capabilities.

6 Conclusions

We have described the design and implementation of the support for sequences in SEQ. The primary contribution of this research is to underscore the importance of algebraic optimization for sequence queries along with a declarative language in which to express them. We have demonstrated the effects of query optimization by means of performance experiments. The PREDATOR system (of which SEQ is a component) supports relational data as well as sequence data, using a novel design paradigm of enhanced abstract data types (E-ADTs). The system implementation based on this paradigm allows sequence and relational queries to interact in a clean and extensible fashion.

We have compared the merits of our approach with the alternative ADT-method approach used by some current systems. If issues like usability and performance are important, our conclusion is that it is inadequate to rely upon procedural methods of a sequence ADT to express queries.

Acknowledgements

Mike Zwilling and Nancy Hall helped with details of SHORE. Illustra Information Technologies, Inc. give us a free version of their database and time-series datablade. David DeWitt and Mike Carey gave helpful advice during the performance evaluation of SEQ. Kurt Brown, Mike Carey, Joey Hellerstein, Navin Kabra, Jignesh Patel, Kristin Tufte, and Scott Vandenberg provided useful discussions on the subject of E-ADTs .

References

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis (eds). Building an Object-Oriented Database System: The Story of O2.. Morgan Kaufmann Publishers, 1992.

[CDF+94] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O. Tsatalos, S. White and M.J. Zwilling. Shoring Up Persistent Objects. In *Proceeding of the ACM SIGMOD Conference on Management of Data*, May 1994.

[CS92] 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.

[DKLPY94] D.J. DeWitt, N. Kabra, J. Luo, J.M. Patel and J. Yu. Client-Server Paradise. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Santiago, Chile, September 1994.

[MIM94] Logical Information Machines. MIM User Manual. 6869 Marshall Road, Dexter, MI 48130.

[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.

[Ill94] Illustra Information Technologies, Inc. Illustra User's Guide, June 1994. 1111 Broadway, Suite 2000, Oakland, CA 94607.

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

[SLR96] Praveen Seshadri, Miron Livny and Raghu Ramakrishnan. Design and Implementation of a Sequence Database. Technical Report. *University of Wisconsin, CS-Dept*, 1996.

[Ses96] Praveen Seshadri. Management of Sequence Data. Ph.D. Thesis. *University of Wisconsin, CS-Dept*, 1996.

[SLR95] Praveen Seshadri, Miron Livny and Raghu Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proceedings of the IEEE Conference on Data Engineering*, March 1995.

[SLR94] Praveen Seshadri, Miron Livny and Raghu Ramakrishnan. Sequence Query Processing. In *Proceeding of the ACM SIGMOD Conference on Management of Data*, pages 430-441, May 1994.

[SS87] Arie Segev and Arie Shoshani. Logical Modeling of Temporal Data. In *Proceedings of ACM SIGMOD '87 International Conference on Management of Data, San Francisco, CA*, pages 454–466, 1987.

[Sto86] Michael Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Proceedings of the IEEE Conference on Data Engineering*, pages 262-269, 1986.

[Sto90] D. Stott Parker. Stream Data Analysis in Prolog. In *The Practice of Prolog*, Chapter 8, MIT Press, 1990.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass (eds). Temporal Databases, Theory, Design and Implementation. Benjamin/Cummings Publishing Company, 1993.

[TSQL94] TSQL2 Language Design Committee. TSQL2 Language Specification. In *ACM SIGMOD Record*, 23, No.1, pages 65–86, March 1994.

[VD91] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *Proceedings of ACM SIGMOD '91 International Conference on Management of Data*, pages 158–167, 1991.

[WJS93] Sean X. Wang, Sushil Jajodia, and V.S. Subrahmanian. Temporal Modules: An Approach Toward Federated Temporal Databases. In *Proceedings of ACM SIGMOD '93 International Conference on Management of Data, Washington, DC*, pages 227–237, 1993.