Paper Number 531

SEQ : Design and Implementation of a Sequence Database System

Praveen Seshadri **Miron Livny** **Raghu Ramakrishnan**

Computer Sciences Department

U.Wisconsin, Madison WI

praveen,miron,raghu@cs.wisc.edu

Abstract

This paper discusses the design and implementation of SEQ, a database system with support for persistent sequence data as well as relational data. Sequence data is common in a variety of application domains, and complex queries over such data arise frequently. SEQ models a sequence as an ordered collection of records. The system supports a declarative sequence query language based on an algebra of query operators, thereby permitting algebraic query optimization and evaluation. This is a fundamental aspect of the SEQ system design and implementation, and is similar in spirit to the support for relational queries in a RDBMS. An alternative approach implemented in some current database systems is to provide a sequence Abstract Data Type (ADT), with a collection of methods that can be composed to express queries over sequences. We show that this approach can lead to queries that are difficult to express and to optimize, and consequently inefficient to execute.

There are four distinct contributions made in this paper. (1) We compare the algebraic and ADT-method approaches to sequence queries using qualitative as well as experimental comparisons. (2) We describe the specification of sequence queries using the $\mathcal{SEQUIN}$ query language, and their execution in the SEQ system. (3) We quantitatively demonstrate the importance of various optimization techniques by studying their effect on performance. (4) We present a novel nested design paradigm used in SEQ to combine sequence and relational data. The system design uses a complex object model to freely mix relational and sequence data, while the language design permits declarative queries over both kinds of data. Based on SEQ, we suggest a pragmatic way for existing database systems to incorporate efficient support for sequence data.

1 Introduction

Much real-life information contains logical inter-relationships between data items. One particular class of logical relationships imparts order to the data items. We use the term "sequence data" to refer to data that is ordered due to such a relationship. While traditional relational databases provide no abstraction of ordering in the data model, there has been increasing interest in providing database support for queries based on the logical sequentiality in the data. In earlier work, we described a data model that could describe a wide variety of sequence data, and a query algebra that could be used to represent and optimize queries over sequences [SLR95]. In particular, we observed that the execution of queries over sequences could benefit greatly from algebraic optimizations that exploited the order information [SLR94]. In this paper, we address the issues that had to be addressed when building the SEQ sequence database system based on these ideas.

SEQ is a multi-threaded, client-server database system with support for sequence data as well as relational data. The design of sequence database support is based primarily on the thesis that algebraic query optimization is essential for sequence queries. SEQ therefore provides the $\mathcal{SEQUIN}$ language to specify declarative queries, and an optimization and execution engine to process the queries. This is similar in spirit to the support for relational queries in an RDBMS. An alternative approach that is implemented in some current database systems, is to provide a sequence Abstract Data Type (ADT), with a collection of methods that can be composed to express queries over sequences. We show that this approach makes it difficult to express complex queries, to optimize them, and to evaluate them efficiently. In addition, the design and implementation of SEQ use a novel paradigm to provide support for both sequences and relations.

The implementation of SEQ has been in progress for more than a year. The system uses the SHORE storage manager library [CDF+94] for low-level database functionality like buffer management, concurrency control and recovery. The higher levels provide query processing support for relations and sequences. The system is currently at approximately 35,000 lines of C++ code (excluding SHORE) and will eventually be available in the public domain.

In Section 2, we describe the current state-of-the-art in sequence database support, and the motivation for the design and implementation of SEQ. We describe the high-level system design in Section 3, and the specific support for sequences in Section 4. SEQ is compared with other current systems that support time-series data on issues of performance in Section 4.6. We discuss the integration of sequences and relations in Section 5. Section 6 suggests a pragmatic approach that existing systems can take to improve their sequence query processing functionality and performance. Finally, we present a discussion on related research and future work in Section 7, and concluding remarks in Section 8.

2 Motivation

Many diverse application areas generate large volumes of sequence data; these include financial and business transactions and scientific experiments. Much of this data is ordered temporally, and is called “time-series” data. However, some important sequence data is ordered by other domains like linear positions, physical locations or integer rankings. We wish to efficiently support complex queries over all these kinds of sequence data.

2.1 The State Of The Art

Current database systems provide limited support for sequence data. Most existing support for such data deals with *temporal* sequences. While SQL-92 provides a timestamp data type, there are few constructs that can exploit sequentiality. The Order-By clause in SQL only specifies the order in which answers are presented to the user. Much research in the temporal database community has focused on enhancing relational data models with temporal semantics [TCG+93], but there have been few publicly available implementations. Financial management products like MIM [MIM94] have created a niche market by building special purpose systems for analyzing stock market data. Main-memory based systems like S-Plus [Sta91] perform statistical analysis of sequences, but these systems are not equipped to handle large quantities of data. Most commercial database systems will allow a sequence to be represented as a ‘blob’ which is managed by the system, but 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 [Ill94a] provides database support for time-series data along with relational data. A time-series is an ADT 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 more complex queries. The “ADT-method” based approach to time-series support in this system represents the state-of-the-art in commercial database support for sequence data.

2.2 Desired Functionality in SEQ

To motivate our design decisions, we briefly discuss some of the functionality requirements set forth for SEQ. 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 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 structured records and the positions of an ordering domain.

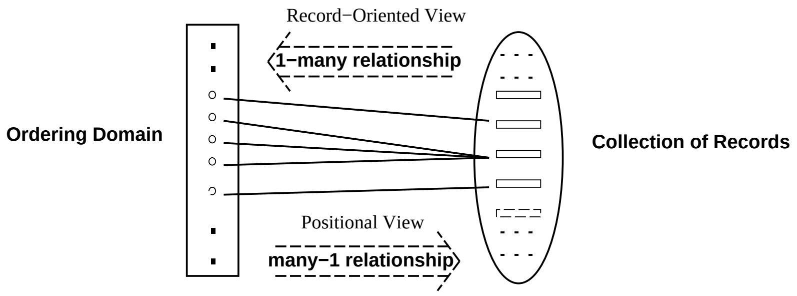

Figure 1: Data Sequence

There are many application domains which deal with sequences of complex values. For instance, satellites produce sequences of images. It is important for the database type system to be capable of extensions to handle such data. Further, despite the fact that we are dealing primarily with sequence data, most applications must also handle a large amount of relational data. After studying some real-world examples, we realized that typically there is some data that is sequential, while there is other "static" data associated with it that is stored in tables. For example, a sequence of bank transactions may record the time, the account number and the transaction details, but the details of the account are maintained in a separate relation whose key is the account number. There are often entire sequences logically associated with individual relational tuples. Trading data from a stock exchange, for example, is often represented as a relation of the stocks traded, and each stock is associated with a specific price history. Entire relations can be associated with a single position in a sequence. As an example of this, consider the information from a hospital on patient visits. There are a large number of visits on any given day, which are recorded in no specific order. However, this data is sequential across days. Administrators might look for seasonality of variations of various kinds of patient visits; this example is based on a real-life application which is described in Appendix A.

In [SLR95], we proposed an algebra of query operators that can be used to specify a certain expressive class of sequence queries. Every operator is compositional; it takes sequences as inputs and produces a sequence as the output. The operators are classified into two distinct categories. The *Record-Oriented* operators treat a sequence as a relation with an extra field in each record to represent the order mapping. In terms of the Figure 1, these operators "view" the sequence mapping from the right (records) to the left (positions). These operators can be directly implemented in a relational database, and several efficient techniques have been proposed for evaluating them in the temporal relational database context [GS89b, LM90]. Of greater interest to us, is the other category of *Positional* operators that concentrate on the relationship between the data records due to their mapping to an ordered domain. These operators "view" the sequence mapping from the left (positions) to the right (records). In the design of SEQ, the algebra of *Positional* operators defined the sequence query functionality requirements. However, extensibility in the operator algebra was desired, so that new Positional operators can be added. While we do not describe the operators in detail in this paper, the $\mathcal{SEQUIN}$ query language is based on this algebra, and its functionality is discussed in a later section.

The most basic implementation requirement was that the system should efficiently process queries over large disk-based sequences. We are primarily concerned with supporting queries, though our implementation should also permit updates. Our earlier qualitative observation [SLR94] that motivated this entire research should be reiterated here: *algebraic query optimization is important for efficiently processing queries over complex collections like sequences.* As we demonstrate quantitatively in this paper, this observation is indeed valid. As practical guidelines, we also required that our design for adding sequence functionality should allow other kinds of interesting collection types (like trees [SLVZ95] and multi-dimensional arrays [MV93]) to be similarly incorporated. Further, the approach we use to support sequence data should ideally be also applicable to other existing database systems (including commercial systems).

3 High-Level System Design

We can categorize the design contributions of SEQ into three components. In this section, we describe two of them: the overall data model design, and its instantiation in the system architecture. The specific design for sequence data support is presented in the next section.

3.1 Data Model Design

Object-relational systems like Illustra [Ill94a], Paradise [DKLPY94], and Postgres [SRH92] 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, with other ADTs nested inside it. Relations are the *top-level* type, and all queries are posed in the relational query language.

The SEQ design enhances the ADT notion by supporting "Enhanced Abstract Data Types"(E-ADTs). The enhancements, which are described in this section, belong to two categories: one category improves the performance of queries involving data of the E-ADT , while the other category improves usability by providing the same status to all the types in the system (including relations). Both sequences and relations are modelled as E-ADTs . Consequently, this is a nested model in which a complex object like a sequence can be a field within a relational record, and vice versa.

In the first category of enhancements, each E-ADT may provide support for one or more of the following:

- 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 capabilities that will translate a language expression into a query evaluation plan in some evaluation algebra.
- Query Evaluation: If a declarative language is specified, the E-ADT must provide capabilities to execute the optimized plan.

This design is driven by the need to support sequences as well as relations with declarative query languages and algebraic optimization. The E-ADT paradigm differentiates SEQ 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 collection types like sequences. This is a novel contribution of the SEQ system. We believe that the E-ADT paradigm can be applied to provide database support for *any* complex collection type. At least in the context of sequence data, the merits of this approach over the ADT-method approach can be clearly demonstrated, as we do in this paper.

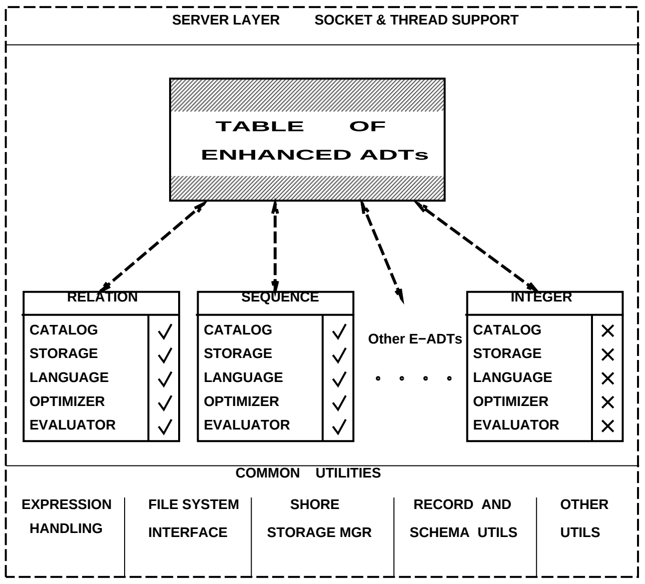

Figure 2: System Architecture

The second category of enhancements allows *any* E-ADT to be the top-level type. Relations are modeled on par with sequences and other E-ADTs . Consequently, even if the entire relation E-ADT were not implemented or not compiled in, the database system would still be able to provide the functionality supported by the other E-ADTs . 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. In order for an E-ADT to be treated as a top-level type, it must provide catalog and storage management features in addition to the language and optimization enhancements:

- Catalog Management: Each E-ADT can provide catalog capabilities that permit certain values to be named, to have statistics maintained and to have schema information stored.
- 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.

3.2 System Implementation

The design philosophy of E-ADTs is carried directly over into the system architecture. SEQ 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. The server is built on top of a layer of common database utilities that all E-ADTs can use. Code to handle arithmetic and boolean expressions, constant values and functions is part of this layer. The primary portion 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; SEQ uses this package to build a multi-threaded server. Multiple clients can connect to the server and have their requests serviced.

The core of the system is an extensible table in which E-ADTs are registered. Each E-ADT may support and provide code for a query language, an optimizer, a query evaluation engine, and storage and catalog management for data belonging to that type. As shown in the figure, some of the basic types like integers do not support any enhancements (though one could argue that a ‘language’ like integer arithmetic is itself a declarative language over the integers). The figure shows two E-ADTs that do support enhancements: sequences and relations.

With this high-level system design, the important question to ask is: how do interactions between the E-ADTs occur? A specific instance of this question that is pertinent to this paper 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 5.

4 The Sequence E-ADT

In this section, we describe how sequences are modeled, the $\mathcal{SEQUIN}$ query language and optimization and implementation techniques.

4.1 Sequence Abstraction

As shown in Figure 1, a sequence models a one-to-many mapping between positions in the ordering domain and a set of records. 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.

In our implementation, the position mapping is maintained as an explicit field of each record. Further, as a simplifying assumption, we restrict each record to be mapped to a single position (the one-to-many model is achieved by making copies of the record with different position fields). While this implementation adequately models much real-world sequence data (like time-series data), we expect to provide support in the future for an interval-based representation of the position mapping (see Section 7).

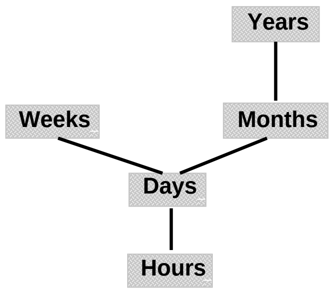

Figure 3: Sample Ordering Hierarchy

An important component of the model of a sequence is the ordering domain. Each ordering domain is modeled as simply another 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. In order to describe such hierarchical relationships, 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.

Although there are different implementations of sequences in the system, they all provide certain common methods as part of a uniform interface:

- *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).

The underlying theme in the implementation of most components of the system is to allow for extensibility by specifying uniform interfaces (using C++ virtual member functions).

4.2 $\mathcal{SEQUIN}$ query language

In this section, we present some important features of the $\mathcal{SEQUIN}$ [1] declarative language for sequence queries. The language is similar in flavor to SQL, an intentional decision aimed at making the language easy for SQL users to adopt. 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: {$\underline{\text{time}}$:hour, $\underline{\text{high}}$:double, $\underline{\text{low}}$:double, $\underline{\text{volume}}$:integer}, where the 'time' field 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          // Desired output per hour.
FROM    Stock1 A                             // Input sequence.
WHERE   A.low < 50;                          // Condition on input record.
```

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[2]. 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.

[1]$\mathcal{SEQUIN}$ is the Sequence Query Interface.

[2]In fact, since this simple query does not really utilize sequentiality, it can be similarly expressed in SQL if each sequence is modeled as a relation.

```
PROJECT avg(A.high - B.high)              // Moving average per hour.
FROM    Stock1 A, Stock2 B                // Positional join of A and B.
OVER    $P-23 TO $P                       // Window for aggregate.
```

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'). Intuitively, the FROM clause merges the sequences by position. 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 sequence view
CREATE VIEW MovAvgStock1 AS (                 // View creation.
       PROJECT avg(C.high) as avghigh         // average renamed as ''avghigh''
       FROM    Stock1 C                       // Input sequence.
       OVER    $P-23 TO $P);                  // Window for aggregate.
// then use the view in the query
PROJECT A.avghigh - B.high                    // Desired output every hour.
FROM MovAvgStock1 A,                          // View in FROM clause.
     Previous(PROJECT D.high                  // Previous() makes a ''most-recent'' sequence
              FROM    Stock2 D                 // Sequence expression.
              WHERE   D.volume > 25,000) B
WHERE $P > 2000;                              // desired range of answers.
```

Note that the sequences in the FROM clause may themselves be defined using another $\mathcal{SEQUIN}$ query block. This 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. This is important in cases like this one where the position attribute is implicitly added to the FROM clause sequences, and hence does not have a field name.

The next query demonstrates the use of the ZOOM clause to exploit the heirarchical relationship between ordering domains[3]. Here is the $\mathcal{SEQUIN}$ query to compute the daily minimum of the volume of Stock1 traded every hour.

```
PROJECT min(A.volume)                     // Min aggregate.
FROM    Stock1 A                          // Input sequence.
ZOOM    days                              // Collapse from hours to 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)                    //  Aggregate to perform.
FROM    Stock1 A                         //  Input sequence.
ZOOM    BEFORE (A.high - A.low < 0.1);   //  Conditional collapse clause.
```

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. While many of the examples presented here are intentionally simple, it is very easy to build complex sequence queries using the view mechanism. Complex sequence queries map to a large graph of operators, and optimization techniques like inter-operator pipelining described in Section 4.5 are crucial for efficiently query processing.

4.3 Running Example

While describing the implementation of the sequence data type, we wish to quantitatively demonstrate the importance of various optimization techniques using performance numbers from the SEQ implementation. It is convenient to use a running example for this purpose. The sequences used in the example 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

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

considered sequences with two different densities: 100% and 20%. For each density, we generated the sequences whose final time-points are shown in Table 1[4].

Density 100%

	Cardinality		
Fields	1K	10K	100K
1	0100/02/15	0101/04/02	0112/07/16
5	0100/02/15	0101/04/02	0112/07/16
10	0100/02/15	0101/04/02	0112/07/16
20	0100/02/15	0101/04/02	0112/07/16

Density 20%

	Cardinality		
Fields	1K	10K	100K
1	0100/08/16	0106/05/09	0163/01/18
5	0100/08/17	0106/05/09	0162/12/10
10	0100/08/09	0106/05/03	0162/11/15
20	0100/08/17	0106/04/17	0162/10/06

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. The Fields value indicates the number of 4-byte integer fields in addition to the timestamp. The values in the fields were integers generated randomly between 0 and 1000. The cardinality of each sequence was either 1000 (1K), 10000(10K) or 100000(100K) records. 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 and the average of the third and fourth execution time was measured as the performance metric.

4.4 Storage Implementation

SEQ supports two repositories for sequence data, the Unix file system and the SHORE storage manager. A sequence can 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, and is useful only when those properties are not required.

The second (and default) sequence data repository is built using the SHORE storage manager library, which supports concurrency control, recovery and buffer management. Data volumes maintained by SHORE can reside either directly on raw disk, or on the file system; our experiments used the latter approach. We studied three alternative implementations of a sequence using SHORE:

1. *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. Further, record-level concurrency is supported by the system. One drawback was that inserts in the middle of a sequence would be difficult. (This problem can be avoided by building a SHORE-supported index over the file on the ordering attribute. Ordered access is now provided by scanning the index, but there is inefficiency due to this level of indirection). Another drawback is that the path length through the file handling code is a large overhead incurred each time the next record needs to be read in a sequence scan. Finally, the storage

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

manager imposes several bytes (at least 24) of storage overhead for every record, in addition to a large space overhead for creating a file.

2. *IdList:* In order to eliminate the space overhead, we tried implementing a sequence 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. Concurrency control is now at the level of the entire sequence. Inserts are easier because SHORE allows new data to be inserted into the middle of a large object. While the space overhead for each file in the File implementation is eliminated, the other drawbacks still remain (primarily, the lack of locality and the storage overhead for records). 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.

3. *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 the various operations within a sequence (like positional lookup, insert and delete) more difficult to implement. Variable length records require additional complex code. However, there are two important benefits to this implementation: the per-record space overhead disappears 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 four 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(*)              // Aggregate to eliminate answer printing time
FROM    <data_sequence>       // Source data sequence.
ZOOM  ALL;                    // Perform aggr over entire sequence.
```

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. 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.

4.5 Query Optimizations

A $\mathcal{SEQUIN}$ query is parsed into a directed acyclic graph of operators, which is then optimized by the query optimizer. The algebra implementation is extensible, and the query optimizer interacts with the operators through various abstract methods. We have described the algebra operators and the optimization techniques in [SLR94]. The query optimizer uses statistics maintained on the sequences to perform some of the optimizations; the

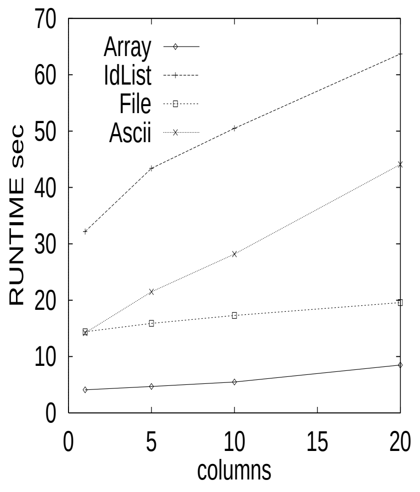

Figure 4: Expmt.1 : Card 100K

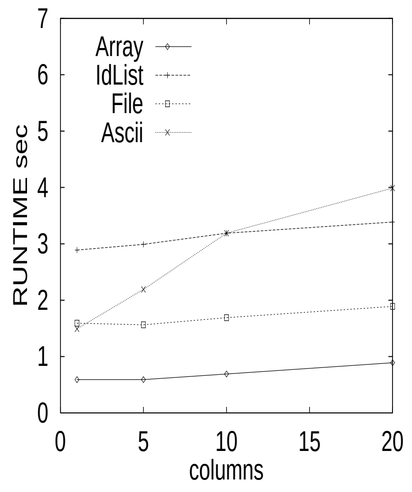

Figure 5: Expmt.1 : Card 10K

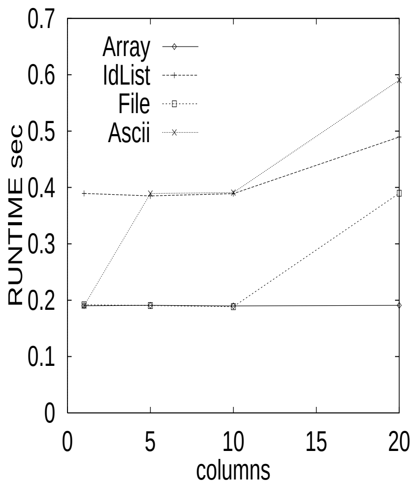

Figure 6: Expmt.1 : Card 1K

statistics include the density of the sequence and the range of valid positions. 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. However, all the experimental queries used are components of meaningful queries posed by users who are currently working with SEQ.

4.5.1 Operator Pipelining

An important optimization principle in SEQ [SLR94] is to try and ensure *stream access* to the stored sequence data as well as intermediate data; i.e., the sequences are read in a single continuous stream. 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. This allows intermediate results in the evaluation to be pipelined between operators, as against being materialized. Consider the simple query below:

```
PROJECT count(*)               // Aggregate to prevent printing many answers
FROM    <data_sequence>        // Source data sequence.
ZOOM  ALL;                     // Perform aggr over entire sequence.
```

This query scans a sequence and performs an aggregate over the entire data to minimize the answer printing costs. This is a portion of a real-life query that looks for a pattern (for example, the value of a field is unchanged on two consecutive days), and counts the total number of occurrences of such a pattern in the sequence. Experiment 2 will show that there is a tremendous penalty to pay for failing to pipeline even such a simple query. Experiment 3 shows that when the query becomes complex, with several nested functions, the relative importance of pipelining becomes even more clearly defined.

Experiment 2: 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 7. 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

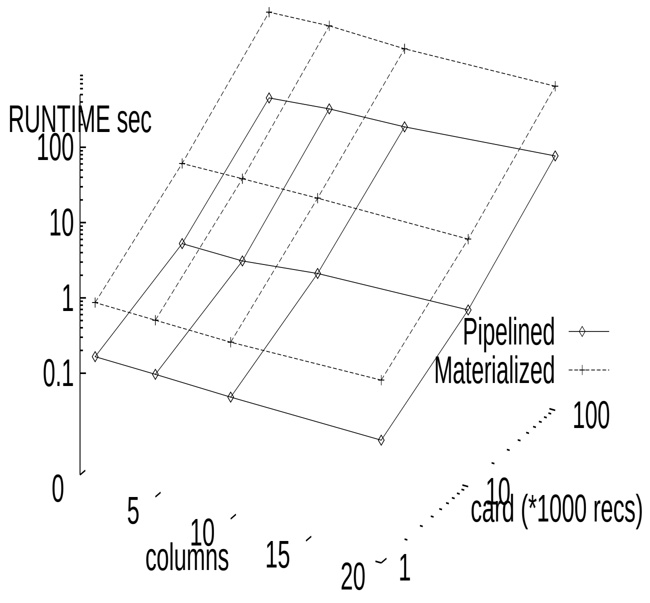

Figure 7: Pipelining: Expmt. 2

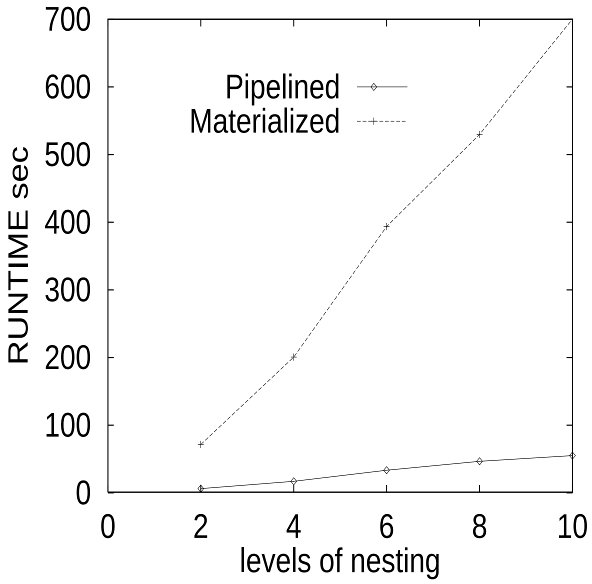

Figure 8: Pipelining: Expmt. 3

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 3: In this experiment, we want to show the effects of increased query complexity on materialized execution. Section 4.2 had several examples of non-trivial queries. By using the view mechanism, many more 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_100%dens) 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(*)                         // Aggr to avoid printing answers
FROM    (PROJECT *                              // Subquery block
         FROM (PROJECT *                           // Another subquery block.
               FROM 100K_10flds_100%dens)) S;      // Source data sequence.
ZOOM  ALL;                               // Perform aggr over entire sequence.
```

Such a query with several levels is an extremely simplified version of a real-life query that steps up an ordering hierarchy (seconds, minutes, hours, days, etc) with an aggregate computed at every level. 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 8. 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.

4.5.2 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 4 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][5]. Experiment 5 demonstrates how the valid-range can be used for optimization.

Experiment 4:

```
PROJECT count(*)                      // Aggregate to avoid printing answers.
FROM 100K_10flds_100%dens S           // Input: 10 fields,100K records,100% density
WHERE S.time > ‘‘<timestamp1>’’       // Selection on the order attribute.
ZOOM ALL;                             // Perform aggregate over entire sequence.
```

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 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).

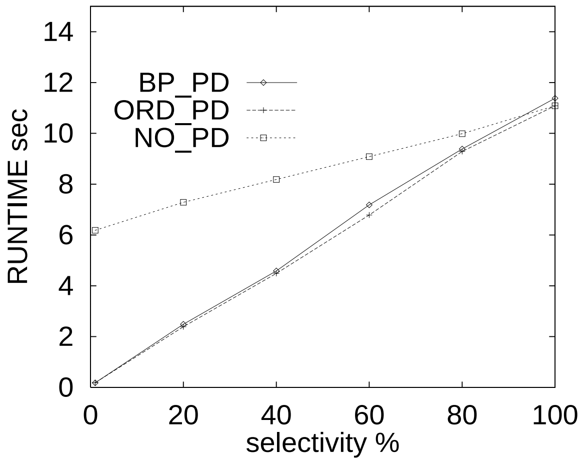

Figure 9: Range Selections At Start: Expmt. 4

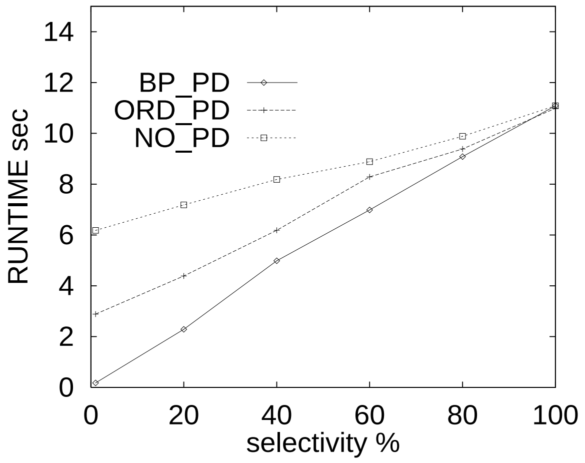

Figure 10: Range Selections At End: Expmt. 4

The results for **at_start** are shown in Figure 9. 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.

[5]Note that this is an important special case of constraint propagation (e.g. [SR93]) and the predicate move-around optimization [LMS94] that have been proposed for relational query optimization.

The results for **at_end** are shown in Figure 10. 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 5:

```
// Define a view sequence which applies a selection to the base sequence
CREATE VIEW ViewSeq AS (PROJECT A.fld2
                        FROM a100K_10_100 A
                        WHERE A.fld1 > 900);
// Merge the sequence S3 with the ViewSeq shifted by a specified offset
PROJECT count(*)
FROM 100K_5flds_100%dens 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. Such a query might arise when looking for similarity between two sequences with a phase lag (for instance, in a soil study experiment that looks for cause-effect relationships among readings from different monitors). 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 11. The valid-range propagation optimization is able to recognize such optimization opportunities in all SEQ queries.

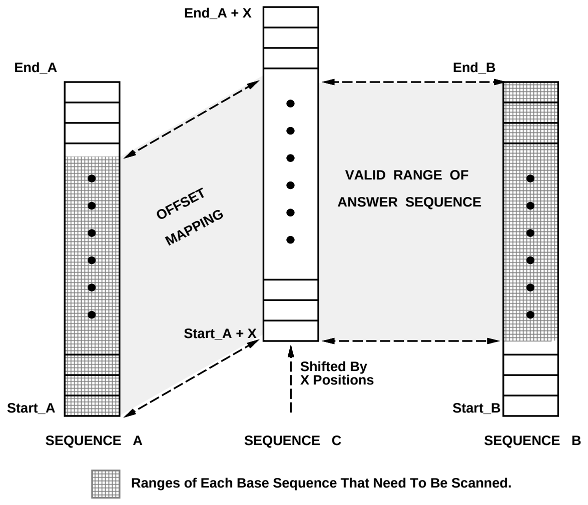

Figure 11: Range Propagation: Intuition

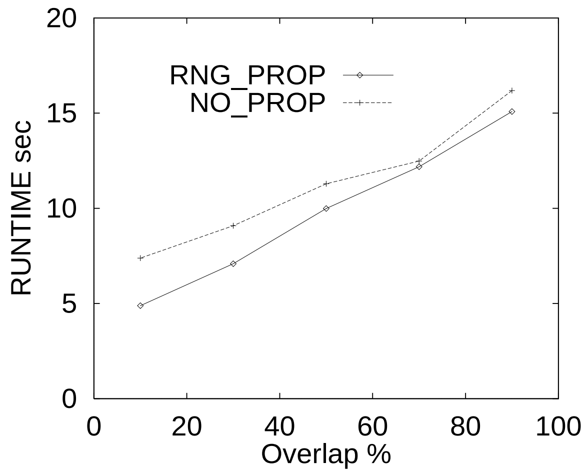

Figure 12: Range Propagation: Expmt. 5

We varied the overlap from 90% of the valid-range to 10% of the valid-range, and executed the query with (RNG_PROP) and without (NO_PROP) the valid-range optimization. The results are shown in Figure 12. The smaller the overlap between the two sequences, the better is the relative performance of RNG_PROP. The difference between the two lines is a measure of the work saved in scanning the sequence.

4.5.3 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 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 6 demonstrates the importance of exploiting symmetric aggregates.

Experiment 6: We considered queries of the form

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

// Perform  a scan over the view sequence, with a count aggregate
// to minimize time for printing answers.
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_100%dens and 100K_10cols_20%dens 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 13. 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. Remember that the sequences are ordered using hourly timestamps. 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

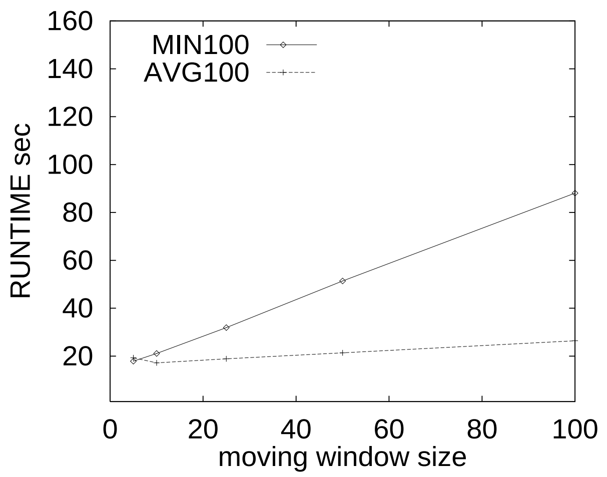

Figure 13: Expmt.6: 100% Density

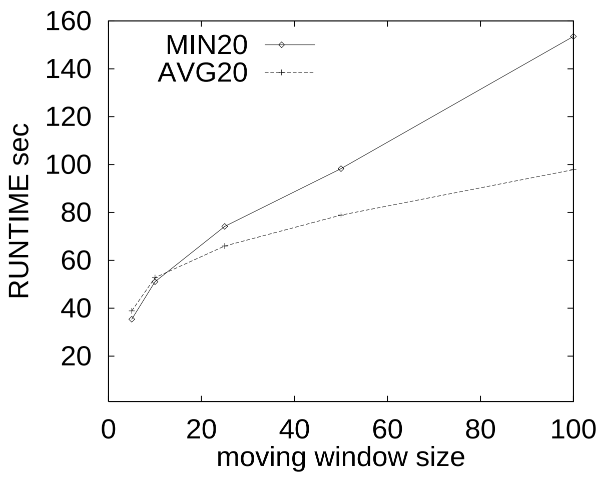

Figure 14: Expmt.6: 20% Density

accounted for. Further, since all interaction with the ordering domains occurs through virtual member functions, the overhead for window computation becomes more significant. This overhead could be avoided if this were a special-purpose system for temporal sequences only.

The results for the 20% density sequence are shown in Figure 14. Note that a moving aggregate over a sequence with holes generates many more records than exist in the input sequence. For example, 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 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.

4.5.4 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.

```
// Define the view for the moving average of a stock over the last 24 hours
CREATE VIEW MovAvgStock1 AS
     (PROJECT avg(S.high) as avghigh
      FROM    Stock1 S
      OVER    $P-23 to $P);

// Ensure that the moving average has not changed much
// from the value in the previous hour.
PROJECT *
FROM MovAvgStock1 T1, Offset(MovAvgStock1, 1) T2
WHERE T1.avghigh - T2.avghigh < 10.
```

Figures 15 and 16 show two possible query graphs that can be constructed from this query. The query graphs use operators from the 'Positional' sequence algebra described in [SLR95]. The meaning of each query graph is

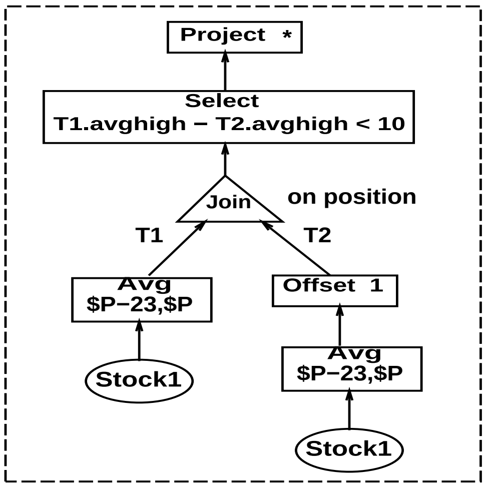

Figure 15: Graph 1: Repeated Computation

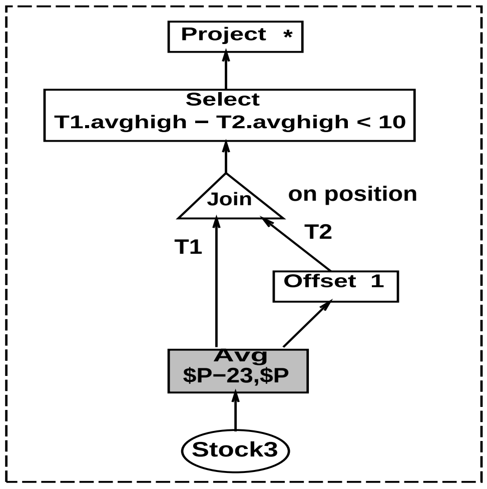

Figure 16: Graph 2: Common Sub-Expression

obvious. The difference between the two query graphs is that one uses a common sub-expression, while the other doesnt. 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 15). The other option is to compute the sub-expression once, store the result, and repeatedly access the stored result. For sequence queries, we have shown 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; this is a novel optimization! While we do not have the space to describe the logic of the analysis technique in this paper, we instead demonstrate its effects.

Experiment 7: We ran the very same query shown above (except that the Stock1 sequence was replaced by 100K_10flds_100%dens). 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 17. 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.

4.6 Comparison with the ADT-Method Approach

Some current systems like Illustra [Ill94a] support sequences (more specifically, time-series) as ADTs with a collection of methods providing query primitives. We call this the "ADT-method" approach to sequence database support. 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. We discuss some recent research [CS93] that relates to this subject in Section 6.2. We now compare this approach with SEQ, based on the performance comparisons presented in Section 4.5. Note that the queries used in the comparison are designed to make the experiments easy to understand, and are not intended as a benchmark, although we believe

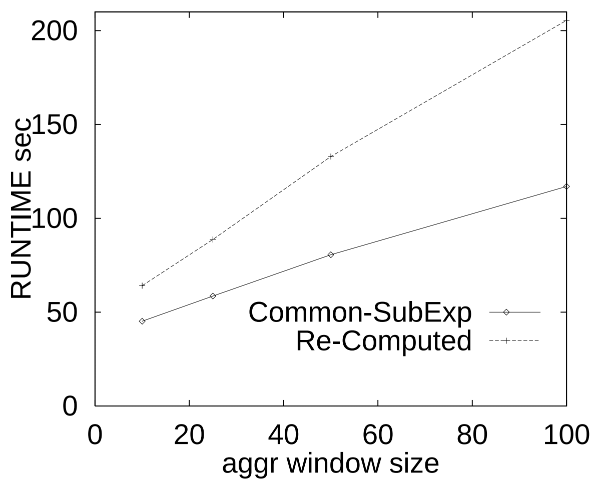

Figure 17: Common Subexpressions: Expmt. 7

that they are representative of a large class of meaningful sequence queries. All of them are simple abstractions of queries that were generated by SEQ users in the domains of medicine, soil sciences and stock analysis.

The pipelining optimization (Experiments 2 and 3) coupled with stream access is a very important algebraic optimization in SEQ. In the ADT-method approach, pipelining is not possible without inter-function optimization. The simple query of Experiment 2 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. Experiments 2 and 3 showed that materialization can perform an order of magnitude worse than pipelining.

The optimizations that propagate valid ranges and selection predicates (Experiments 4 and 5) once again require the ability to push range selections from one function to another. Consequently, ADT-method based systems do not exploit these optimizations. In Experiment 4, the NO_PD algorithm corresponds to the performance of an ADT-method based query where the user has not explicitly pushed the selection predicate into the scan of the sequence.

Note that an ADT-method approach cannot identify common sub-expressions without inter-function optimization, let alone take advantage of them to optimize query execution. Experiment 7 showed that the common sub-expression optimization could reduce query execution time by almost a factor of two. An ADT-method based system could store the results of all method invocations (i.e., perform function caching), thereby avoiding the cost of repeatedly executing the sub-expression. On the other hand, the materialization costs could be significant.

The optimization of symmetric moving window aggregates could be incorporated into an ADT-method system. This does not require any optimization across function boundaries. The table below summarizes these points of comparison.

Ideally, we would have liked to present a performance comparison with an ADT-method based system to quantitatively demonstrate the effects of the underlying design differences. We chose Illustra [Ill94a] for this comparison because it is a commercial database system that provides time-series support using the ADT-method approach, and it uses a time-series model that is similar to our sequence data model. While we did carry out this performance study comparing SEQ and Illustra, our license agreement does not allow us to publish performance numbers without their permission; Illustra did not give us the necessary permission. Consequently,

Optimization Category	Implemented in SEQ	ADT-Method Systems
Pipelined Operators	Yes	Not Implemented
Range Propagation	Yes	Not Implemented
Common Sub-Expressions	Yes	Not Implemented
Moving Window Aggregates	Yes	Can Be Implemented

Table 2: Summary of Optimization Comparison

to get a quantitative feel for the magnitude of the performance differences between the algebraic and ADT-method approaches, we have had to rely upon the "internal" performance results presented in Section 4.5.

5 Combining Sequences and Relations

We now return to the issue of how sequences and relations interact in SEQ. 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 4.2. 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.

5.1 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)[6] with nested query expressions in the sequence query language ($\mathcal{SEQUIN}$). Appendix A contains a real-life example of the opposite case: a query over relational data embedded within a sequence.

Let us consider the SQL query to find for each stock, the number of hours 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
                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

[6]We have implemented a version of SQL as the query language of the relational E-ADTs. Our SQL version allows functions registered with the DBMS to appear in the SELECT and WHERE clause.

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 world'', S.stock_history)
FROM   Stocks S;
```

However, this nested query language paradigm requires enhancements to the implementation of three relational system components: type checking, optimization and evaluation.

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 .

The evaluation of the query can involve many nested evaluations of the embedded query plan. This is similar to the usual evaluation of functions in a language like SQL, and to the nested evaluation of correlated sub-queries in SQL. There are many optimizations like function caching [Hel95] and magic decorrelation [SPL94] that have been proposed in those contexts and similar optimizations could be performed in SEQ too. The current implementation, however, uses a simple value-at-a-time nested evaluation.

Another issue that arises in the nested design is the granularity at which statistics are maintained. Since a declarative query is expressed over the nested data, its optimization could depend on the meta-data available. In the example, the question is: should individual statistics be maintained for each nested stock_history sequence? Our solution was to maintain aggregate statistics for all nested instances of each data field, instead of individual statistics. In this example, all the stock_history sequences for the different stocks would have common statistics maintained. This design represents a compromise between the need to maintain statistics and the overhead to do so. Other implementations may choose to maintain either more or less statistics. We do not have sufficient experience yet with SEQ to decide exactly what the ideal compromise is.

5.2 Discussion of the Nested Design Paradigm

To the best of our knowledge, our nested design paradigm based on E-ADTs is a novel contribution. It allows query languages to interact in an extensible manner, while insulating the changes in one language from the other languages in the system. We should emphasize that the nested paradigm is not merely a proposal; all the features described in this paper have been implemented[7]. We now discuss some of the possible arguments against the nested paradigm.

[7]The only exceptions are the cast functions between sequences and relations which are being implemented at the time of this submission.

With respect to the language design, some researchers consider it undesirable to have multiple languages used within the same query. We would however invite comparison with the existing solution, which uses the composition of ADT methods instead. As we have seen, this is a bad idea from the point of view of query optimization and execution efficiency. Further, in terms of ease of use, function composition is merely an expression language that is not well-suited to query expressions over collection types.

It has also been suggested that we might have devised some extension of SQL that provided sequence functionality while staying within one language. While we did indeed attempt this initially, we were quickly dissuaded because many changes were needed. Not only is this approach unlikely to be incorporated in existing systems, it is also a solution that cannot be extended beyond sequences to other complex data types that may need such functionality. We believe therefore that it is most practical to leave SQL relatively unchanged, and introduce the sequence functionality in the manner that we have proposed.

One other concern is that some opportunities for optimization across query blocks may not be exploited because they cross E-ADT boundaries. Although optimizations like function caching and magic decorrelation may help (as discussed in Section 5), there is no denying that there is some truth in this concern. Some systems like Starburst [PHH92] perform a number of heuristic query transformations that modify the blocks of a query expressed entirely in SQL. However, it is a fact that even for relational queries in SQL, there is little if any cost-based optimization performed across query blocks. Consequently, we felt that this was an acceptable price to pay for the other benefits of the nested design paradigm. If our future experience with SEQ indicates that important opportunities for cross-E-ADT optimization exist, we might need to augment the interaction between the different E-ADT optimizers (possibly using techniques similar to those of [PHH92]).

6 Comparison with Existing Systems

An important goal of our work is to find ways to incorporate efficient sequence data support into existing database systems. In this section:

- We show a real example of how sequence queries are phrased in an existing ADT-method based system (Illustra [Ill94a]). While queries in $\mathcal{SEQUIN}$ are declarative, queries based on the functional composition of methods have a more procedural flavor and are often harder to express.
- We suggest practical improvements that existing database systems can make to improve their support for sequence data.

6.1 Sample Query Using ADT-Methods

The basic model of a time-series in Illustra is similar to the model in the SEQ implementation. Consider a simple sequence query which finds the moving average of a sequence S after first excluding some of the records. Expressed in $\mathcal{SEQUIN}$, the query looks like:

```
PROJECT avg(S.field3)                    // Find the average.
FROM    S                                // Input sequence S.
WHERE   S.field2 > S.field1              // Condition on records.
OVER    $P-23   TO $P;                   // Moving window for average.
```

Here is how it would need to be written using ADT methods [Ill95]:

```
-- the Filter function gives the filtered time-series, with only those records
-- that satisfy the qualification, other positions will be null
CREATE function Filter(TimeSeriesOf(ts))
RETURN TimeSeriesOf(one_float)
as
RETURN PutSet(TimeSeriesCreate('hours'),
              (SELECT t, field3
               FROM Transpose(ts)
               WHERE field2 > field1));

-- compute the 24-hour moving average over the modified time-series
AggregateBy('TimeSeriesRunningAvg($1, 24)',
            'hours',
             Filter(S))
```

The time-series is converted to a relation (using the Transpose function) to perform the desired selections, and the result is converted back to a time-series. Notice that an entire copy of the time-series is made in order to perform the selection. This switch to SQL is needed because the selection condition cannot be expressed directly using the time-series functions. Finally, the moving average of the result is performed. The user needs to write part of the query using SQL and part using time-series functions.

This example query is difficult to comprehend, and it reflects the point that a procedural query language based on function composition can be more awkward to use than a declarative language like $\mathcal{SEQUIN}$. In just the same way, it is usually easier to express a complex query in SQL than in relational algebra.

6.2 Adapting Existing Database Systems

We have noted repeatedly in this paper that current ADT-based database systems do not perform inter-function optimizations (and thereby miss out on a number of optimizations that SEQ can perform). Further, there is usually only one implementation for each function, corresponding to only one evaluation strategy for each operation. SEQ on the other hand can consider various alternatives implementations for algebraic operators.

Let us consider how a traditional system with ADT-methods could overcome these problems. A recent research proposal suggests that rewrite rules could be specified to consider semantics associated with "foriegn functions" [CS93]. The paper proposes that queries involving relations stored outside the DBMS can be optimized using declarative rules that specify how the query can be rewritten into equivalent forms. A similar approach can perhaps be used to rewrite functional expressions into equivalent expressions. It is not clear that such an approach would be feasible to implement, or that it would be efficient to optimize and execute. In the OQL language design [Cat94], methods are allowed to have multiple implementations, and the system has the right to choose the best one. If these two research proposals are combined and implemented together, a functional expression could be recognized by the system as a composition of algebraic operators that can subsequently be transformed and optimized. Indeed, our performance results indicate that such optimizations can significantly increase execution efficiency. We note that this approach effectively makes a functional expression a *declarative* language expression (because the system is free to execute it in any way that produces the desired result). This is not very different from the SEQ approach, and primarily differs in that the declarative language syntax takes a functional form, instead of looking like $\mathcal{SEQUIN}$. This approach requires more research before it can be widely implemented, and parts of it (especially, the rewrite rules for function commutativity and associativity) may require substantial development effort.

An alternative is to try to incorporate some of the SEQ techniques into existing systems. The SEQ design for sequence data support has three components: the $\mathcal{SEQUIN}$ query language, the E-ADT paradigm, and the loosely-coupled architecture of E-ADTs in the SEQ implementation. It is, of course, unrealistic to require existing database systems to completely redesign their code in order to support sequences in exactly the same fashion. Therefore, we now consider ways in which such systems could adopt some aspects of the SEQ approach.

If issues like usability and performance are important, our conclusion is that relying upon methods of a sequence ADT for queries is not sufficiently expressive or efficient; a sequence language must be supported, and carefully optimized. At the very least, our performance results strongly suggest that a system that relies upon ADT-methods for querying sequences must make an effort to perform inter-function optimizations.

However, it is important to recognize that a sequence (sub)language can be embedded in SQL without placing relations and sequences on par as collection data types (as is done in SEQ). For example, the interaction between SQL and the sequence query language can take place using a special function call that interprets a sequence query string passed as its argument (just as the SEQ implementation does using the implicit $\mathcal{SEQUIN}$ function). The optimization of sequence query strings, of course, can be carried out using the techniques presented in [SLR94] and in this paper (it may well be that further improvements or variations are possible, but at the least, we believe that our results provide a sound basis for further work). Implementing this proposal in a system with good support for ADTs is a matter of a few man-months of work, and our results indicate that the resulting performance difference may be measured in orders of magnitude. Since many current DBMS products provide some notion of abstract data types (with support for complex composite structures as in Illustra, or simply binary large objects interpreted by user-level code, as in several other systems), we think this is a viable approach for current systems to take.

7 Related Research and Future Work

In this section, we discuss related research, and some topics of ongoing activity and planned future work on SEQ.

7.1 Related Research

In Section 2.1, we surveyed the sequence processing capabilities of existing database systems. We now review some research related to this area. Research work directed at modelling time-series data [SS87, CS92] 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 [SS88]. The dual nature of sequences (Positional versus Record-Oriented) is also recognized by the temporal query language of [WJS93]. The extensive work on temporal database modelling, query languages, and query processing [TCG+93] is mostly complementary to our work, because it involves changes to relations and to SQL [TSQL94], not to the sequence E-ADT . However, it would be interesting to study how time-ordered sequences can be efficiently converted into relations with time-stamps, and vice-versa. We intend to eventually support the temporal relational functionality described in the temporal database benchmark [Jen+93].

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 to match regular patterns over sequence data [GW89b, GJS92], and the proposal of [GJS92] has been implemented as an event recognition system. This work is complementary to ours, since SEQ is oriented to more traditional database queries, and currently does not have meaningful pattern-matching capabilties, though we plan to address this shortcoming soon (see Section 7.2).

The AQUA algebra [SLVZ95] can model a variety of "pattern-match" queries over different collection types including lists. The idea is to map any query over complex and possibly nested data into a uniform algebra. The algebraic expression can then be optimized using transformation-based techniques. This is an interesting alternative to the loosely-coupled design used in SEQ, and we would like to explore whether the AQUA approach can yield better optimization of sequence queries.

Sequences are also of interest to the data mining and on-line analytical processing (OLAP) communities. There has been some work on mining sequential patterns in databases [AS95], and finding similarity between sequences [FRM94]. In OLAP queries [CCS93], time is often an important dimension of the data, and moving window queries are common. These areas represent possible application domains to which SEQ could perhaps be adapted in the future.

There is much research work related to the E-ADT paradigm. The issues regarding support for ADTs in database systems were explored in [Sto86]. There has been extensive work on nested data models (especially nested relational models [Hul87]), and there is even a commercial database system, UniData [Uni93], based on such a model. Object-oriented systems like O2 [BDK92] also support a nested model with composite objects. The recently proposed OQL query language [Cat94] for OO databases allows collection types to be nested, and permits nested queries over them. The entire query is expressed in OQL, and there are few special query constructs for lists. However, the idea of enhancing ADTs with query language and query processing capabilities seems to be unique to SEQ. The loosely-coupled architecture with multiple top-level collection types with different query languages also appears to be novel.

7.2 Future Work

The SEQ implementation is being constantly expanded with new functionality. In the current implementation, every record in the sequence maps to a single position. We plan to allow each record to map to an interval of positions instead. This can be viewed as a compressed storage representation, and the query evaluation need not be modified. On the other hand, queries can be evaluated more efficiently if this interval information is used. We plan to explore this issue of sequence storage techniques, along with lightweight indexing techniques to efficiently access a specific position in a sequence.

There are many sources of sequence data that pose special challenges to the system implementation. The most exciting of these are real-time sequences (where the implementation of query evaluation may have to be modified to use one thread to read each real-time sequence), sequences stored on tape (where stream access becomes absolutely critical for performance) and multi-dimensional sequences (where the zooming features may have to be enhanced to allow queries that drill down and up the dimensions).

With respect to the query language and algebra operators, we expect to add further functionality as our experience with applications grows. We have identified the need for and are currently implementing a generic operator that performs linear recursion on the positions of a sequence (note that currently implemented operators for Previous and Next are specific examples of such a generic operator). This is useful for computing decay functions over a sequence, and can also be used for recursive pattern match queries (expressing regular patterns).

At the level of generic nested E-ADT queries, we believe that there are opportunities for set-oriented optimization using techniques recently proposed for relation query optimization [SPL94, Hel95]. This work also has relevance independent of sequence data, because it can be applied to query processing in all languages that support nesting, and to query processing in heterogenous databases. The importance of cross-E-ADT optimizations also needs to be determined.

8 Conclusions

We have presented the design and implementation details 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 demonstrate the effects of algebraic optimization by means of performance comparisons on the system implementation. Further, we compare the merits of our approach with the alternative approach based on ADT-methods, that some current systems use.

The SEQ system supports sequence data as well as relational data. To take advantage of our algebraic approach to sequence database support, the system uses 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. Based on our implementation, we suggest a pragmatic way for existing database systems to incorporate efficient support for sequence data.

9 Acknowledgements

The persistent data support for SEQ was built on top of the SHORE storage manager developed at the University of Wisconsin. Mike Zwilling was very patient in tracking down SHORE 'problems' that almost always turned out to be bugs in SEQ. Illustra Information Technologies, Inc. give us a free version of their database and time-series datablade, and free access to their user-support personnel. David DeWitt and Mike Carey gave helpful advice and support 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 . Dave Simmons with the UW Family Medicine Department provided us with the data and application discussed in Appendix A.

References

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. In *Proceedings of the IEEE Conference on Data Engineering*, March 1995.

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

[CCS93] E.F. Codd, S.B. Codd, C.T. Salley. Beyond Decision Support. In *Computerworld*,Vol.27, No.30, 26 July 1993.

[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.

[CS93] Surajit Chaudhuri and Kyuseok Shim. Query Optimization in the Presence of Foreign Functions. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 526–541, 1993.

[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.

[Cat94] R.G.G. Cattell, Editor. The Object Database Standard:ODMB-93. Morgan-Kaufman Inc., Los Altos, CA, 1994.

[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.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *Proceeding of the ACM SIGMOD Conference on Management of Data*, pages 419-430, May 1994.

[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.

[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.

[GW89b] S. Ginsburg and X. Wang. Pattern Matching by Rs-operations: Towards a Unified Approach to Querying Sequenced Data. In *Proceeding of the ACM SIGMOD Conference on Management of Data*, 1992.

[Hel95] Joseph M. Hellerstein. Optimization and Execution Techniques for Queries With Expensive Methods Ph.D. Thesis, University of Wisconsin, August 1995.

[Hul87] Richard Hull. A Survey of Theoretic Research on Typed Complex Database Objects. In J.Paradeans editor, *Databases*, pages 193-256. Academic Press, London, 1987.

[Ill94a] Illustra Information Technologies, Inc. Illustra User's Guide, June 1994. 1111 Broadway, Suite 2000, Oakland, CA 94607.

[Ill95] Illustra User Support. Personal Communication. October 1995.

[Jen+93] C.S. Jensen, et al. The TSQL Benchmark. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, June 1993.

[LM90] Cliff T.Y. Leung and Richard R. Muntz. Query Processing for Temporal Databases. In *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, CA, February 1990.

[LMS94] A.Y. Levy, I.S. Mumick, and Y. Sagiv. Query Optimization by Predicate Move-Around. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, Santiago, Chile, 1994.

[MV93] D. Maier and B. Vance. A Call to Order. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Washington, DC, May 1993.

[PHH92] Hamid Pirahesh, Joseph Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of ACM SIGMOD '92 International Conference on Management of Data, San Diego, CA*, 1992.

[RS87] L. Rowe and M. Stonebraker. The POSTGRES Data Model. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, Brighton, England, 1987.

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

[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.

[SLVZ95] Bharati Subramaniam, Theodore Leung, Scott Vandenberg and Stanley Zdonik. The AQUA Approach to Querying Lists and Trees in Object-Oriented Databases. In *Proceedings of the IEEE Conference on Data Engineering*, March 1995.

[SPL94] Praveen Seshadri, Hamid Pirahesh, and T.Y.Cliff Leung. Decorrelating Complex Queries. To appear in *Proceedings of the Twelfth International Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

[SR93] Divesh Srivastava and Raghu Ramakrishnan. Pushing Constraint Selections. In *Journal of Logic Programming*, 16(3–4):361–414, 1993.

[SRH92] M. Stonebraker, L.A. Rowe, and M. Hirohama. The implementation of POSTGRES. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.2, March 1992.

[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, 1988.

[Sta91] Statistical Sciences, Inc. S-Plus User's Manual. Statistical Sciences, Inc., Seattle, WA.

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

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

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

[Uni93] UniSQL User's Guide (Release 2.1). UniData Inc., Denver, 1993.

[VD91] S.L. Vandenberg and D.J. DeWitt. Algebraic Spport 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.

A Clinic Visit Data Example

The Family Medicine Department at the UW-Madison collects patient visit information from 18 clinics around Wisconsin. The operating information from each clinic is available as a flat-file with records of the form <day, patient_id, diagnosis_id, treatment_id, auxiliary_info>. There are separate tables that map the patient_id/diagnosis_id/treatment_id into the actual information on the patient/diagnosis/treatment. A sample query compares the seasonality of cases of gastro-enteritis during the last 5 years, with the seasonal patterns of appendicitis. SEQ would model the data as follows:

```
CREATE SEQUENCE Visit_Info AS
   { date   day,
     visits table {patient_id    integer,
                   diagnosis_id  integer,
                   treatment_id  integer}
   };
```

In other words, the Visit_Info sequence contains a nested relation for every day, which in turn holds the visit records for that day. There are various possible definitions of 'seasonality' query. One definition of the query is as follows: find the 7-day moving average of the number of visits per day of gastro-enteritis and appendicitis. Now compute the sequence of differences between the two averages every day and find the average of these values every month over the last 5 years. Following the nesting of the data, $\mathcal{SEQUIN}$ is the top-level E-ADT query language to be used, and there can be nested SQL expressions. Here is the $\mathcal{SEQUIN}$ query to perform this query:

```
// 7-day moving average of the gastro-enteritis records per day
CREATE VIEW GE_Visit_Avg as
 (PROJECT avg($SQL(``SELECT count (*)
                    FROM $1 R, DIAGNOSIS D
                    WHERE R.diagnosis_id = D.id
                      AND D.name = 'gastro-enteritis''',
                   V.visit)) as avg_val
     FROM Visit_Info V
     OVER $P-6 TO $P);
// 7-day moving average of the appendicitis records per day
CREATE VIEW AP_Visit_Avg as
 (PROJECT avg($SQL(``SELECT count (*)
                    FROM $1 R, DIAGNOSIS D
                    WHERE R.diagnosis_id = D.id
                      AND D.name = 'appendicitis''',
                   V.visit)) as avg_val
     FROM Visit_Info V
     OVER $P-6 TO $P);
// define the desired monthly aggregate sequence query
CREATE VIEW Query_Seq as
  (PROJECT avg(AP.avg_sal - GE.avg_sal)
   FROM AP_Visit_Avg AP, GE_Visit_Avg GE
   ZOOM month);
// select the desired range from the sequence
PROJECT *
FROM Query_Seq
WHERE $P > ``01/01/1990'' and $P < ``12/31/1995'';
```

The query formulation has been broken into a number of views so as to be easily comprehended. Note the use of a nested relational query, within a sequence query. While this query may appear extremely complex, the sequentiality in the query allows it to be executed in a single scan of the sequence.