MANAGEMENT OF SEQUENCE DATA

By
Praveen Seshadri

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN – MADISON
1996

Abstract

One of the challenges facing today's database systems is the need to support complex data types, which are of growing importance in new application areas. The thesis addresses this problem, with a specific focus on supporting sequence data.

A large part of the thesis deals with the details of sequences. Issues covered include the model for sequence data, an algebra of operators to query the data, a query language to express the queries, optimization techniques and query processing algorithms. Performance results are presented from an implementation of these ideas, demonstrating the effects of the various optimizations. This detailed exploration of sequence data is one contribution of the thesis.

The second contribution is a solution to the problem of integrating different data types, including sequences and relations, in a general-purpose database system. The thesis discusses the drawbacks of existing solutions, and then proposes a solution based on a novel E-ADT paradigm. This paradigm has been used in the development of the PREDATOR database system, and the implementation brings to light several advantages as well as limitations of this paradigm. The support for sequences has been implemented as a component of this larger system. The conclusion drawn from the sequence implementation is that it is important to provide specialized support for queries over sequences. By extrapolation, similar conclusions may be drawn about support for other complex data types like matrixes and images. The E-ADT paradigm provides a mechanism to integrate all these types within a single general-purpose database system.

The conclusion of the thesis is that database systems should adopt some of the E-ADT ideas so as to provide more efficient support for complex data types like sequences.

Acknowledgements

I do not view the completion of this thesis as a personal achievement; rather it is an occasion to pause, count my blessings, and thank the many people who have helped along the way. While I have tried not to miss out anyone, there will invariably be omissions; friends and associates not acknowledged here, whom I have been fortunate to know during my half-a-decade in Madison. I will always have warm memories of this period (and a few bitterly cold -40 degree ones too!) when my life changed in immeasurable ways. Most importantly, I got married, and my wife Ranjani is primarily responsible for my happiness, health, and indeed my finishing this thesis. While she has put up with me during hacking spells and paper deadlines, she is also my biggest fan and supporter. Where would I be without her?

My advisor, Raghu Ramakrishnan, is the sole reason for my thinking about and committing to a doctoral degree. For those who know the story, he is also responsible for my getting to Wisconsin and working on databases in the first place. It is rare to meet such a nice person in any walk of life, let alone in the highly competitive world of university research. It has been my tremendous good fortune to have Raghu as my advisor, as a teacher, and as a friend. I should also put on record (in a document that he has signed) that in the last set of tennis we played, I won 6-2.

Miron Livny has been my co-advisor on my thesis work, as well as on other research. In my mind though, his primary contribution has been his enthusiasm for my work, and his unflagging words of encouragement whenever he thought that I needed it. While I greatly value my technical discussions with him, I appreciate these intangible aspects of our interaction even more. I do not underestimate the extent to which the database group at Wisconsin has contributed to this thesis and to my background in databases in general. I had the opportunity to work on a project with Mike Carey – the speed at which he grasps what you are saying (and what is wrong with it) is awe-inspiring. Yannis Ioannidis was on my thesis committee, and has always shown interest in my work and my career. So has Jeff Naughton, who in addition has helped me trace my lineage to some of his former students. Both Jeff and Yannis taught me database courses in my first year, and I have always viewed them as examples of how one should handle the pressures of academic life with style. Then there is David DeWitt – David has never taught me a course, or been involved with me on a research project. Nevertheless, he showed interest in my work, encouraged me to search for an academic job, and helped me find such a position. Perhaps more importantly, as are all the other database students at Wisconsin, I am indebted to David for building an environment in which this thesis work was possible. Mark Fedenia of the School of Business was the external examiner on the committee, and gave me detailed suggestions that improved the thesis. Larry Landweber gave me great advice when I was on the job market, and I very much appreciate his help.

One of the best things about UW is that as database students, we got to talk a lot with each other. Mike Franklin was my office mate for my first two years – he still complains about how loudly I used to type. During that time, and even more so later, Mike has given me critical advice that has helped me immensely. He was also a good person to whomp at racketball. When Mike had to leave and go to Maryland, Joe Hellerstein took the empty chair. I really think I might have finished my thesis earlier if we hadn't spent so many hours discussing any interesting research idea that popped into our heads. On the other hand, this was a most enjoyable time, and it was a pity Joey had to graduate and go to Berkeley. Over the last couple of years, I have had several discussions with Jignesh Patel, mostly on object-relational databases, but occasionally on roller-blading, da Bulls, marriage, VW Jettas, and Jim Jarmusch movies. I also had the good fortune to work a little bit with future millionaire Kurt Brown, whose blithe spirit is now sorely missed on the seventh floor.

During my introduction to databases, working on CORAL, S.Sudarshan and Divesh Srivastava welcomed and encouraged this ignorant junior guy, and they have been supportive ever since. Some of the others I had interesting discussions with include Joseph Albert, Tarun Arora, Kevin Beyer, Shaul Dar, Prasad Deshpande, Navin Kabra, Manish Mehta, Renee Miller, Bill Roth, S.Seshadri, Kristin Tufte, Shivakumar Venkataraman, Janet Wiener, and Jie-bing Yu. I am indebted to the SHORE and EXODUS staff whose software I used at various stages. Nancy Hall and Mike Zwilling have both shown the utmost patience with my many questions, and have always helped me out promptly, despite the other demands on their time. My work on sequence data benefited from a NASA grant that supported me for a year and the free software installation provided by Illustra Inc. I should also express my gratitude to the CS lab staff in general, and specifically Jim Luttinen, and, of course, one can only marvel at the efficiency of Marie Johnson, Lorene Webber, and the others who really keep the department going.

During my stint in graduate school, I have enjoyed a warm relationship with the IBM Almaden Research Center; I spent two summers there, and was supported for two years by an IBM fellowship. I had the good fortune to interact with Bobbie Cochrane, Bill Cody, Cliff Leung, John McPherson, Hamid Pirahesh, Beau Shekita, Dave Simmen and many other incredibly nice people at IBM. Further, I was given the opportunity to work on DB2 common server, and to describe my work in papers with Hamid Pirahesh (by far the busiest man on earth) and Cliff Leung.

Finishing a PhD. is in large part a matter of staying positive and cheerful. At least in my case, the credit for this goes to Ranju, and to my friends. Guhan Viswanathan and Vidya have been close, loyal friends from my first day in Madison, as have Madhu Talluri and Sridevi. It is a good thing Pritha Gopalan knows little about databases, because she would have heard a lot more than she anyway did in our weekend conversations. Finally, I hope my mother will be proud of this; somewhere behind each line of code are our summer Meccano projects; somewhere behind each line of this document is the “Lady of Shallot”.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The issues addressed in this thesis deal with the design and development of database query processing systems. These systems allow data to be manipulated and extracted in complex ways.

The most common database systems are called Relational Database Management Systems (RDBMSs); they organize data into sets or multi-sets of identically structured records. The operations supported on relational data are well-known operations on sets and multi-sets of records: union, intersection, cross-product, selection and projection. These operations constitute the *relational algebra* [Cod70], and serve as the established framework to query relational data. There has, however, been an increasing focus on extending database systems to provide efficient support for new and interesting kinds of data like time-series, images, and arrays. This thesis addresses the problem of supporting queries over such new kinds of data within a general-purpose DBMS. A large portion of the thesis deals specifically with *sequence data* (a superset of time-series data) as a specific example of one new data type that should be efficiently supported. A wide range of relevant issues are explored in the context of sequence data, including the data model, query algebra, query language, optimization techniques and evaluation mechanisms. This detailed exploration of a specific data type demonstrates that there are significant benefits to be obtained by providing specialized DBMS support to complex data like sequences.

A DBMS needs to be able to provide such support to several kinds of complex data *within the same system*. The important issue then becomes one of integrating several different types of data together within one common database system, without compromising on the benefits and efficiencies that derive from supporting each individually. Current solutions are based on the concept of Abstract Data Types (ADTs). The important contribution of the thesis in this direction is the *Enhanced-ADT* concept as an architectural paradigm for database system design. The *PREDATOR* database system is a prototype developed to explore the issues involved in building a DBMS based on Enhanced-ADTs, and the support for sequence data is a component of the larger overall system. At a high level, the contribution of PREDATOR is the demonstration that database query processing support can be developed in a modular fashion, with each data type providing a syntax and a semantics for manipulations of its values. The modularity of the design allows the system to scale by adding new data type modules, without unduly compromising on the efficiency of query execution.

The rest of this introductory chapter deals with the relevant background for the material in the thesis. Section 1.1 briefly motivates the need to support sequence data, while Section 1.2 deals with the evolution of extensible database systems. Finally, Section 1.3 details the research contributions of this thesis, and describes the structure of the chapters ahead.

1.1 Why Sequences?

Data over ordered domains such as time or linear position is logically viewed and queried in terms of a sequence abstraction and is often physically stored as a sequence. Databases should (a) allow sequences to be queried in a manner that reflects the ordered semantics of the data, and (b) take advantage of the opportunities available for efficient query processing. However, relational databases are inadequate in this regard. Data collections are treated as sets, not sequences; consequently, expressing sequence queries is tedious, and evaluating them is inefficient.

Example 1.1.1 Consider the following motivating example. A weather monitoring system records information about various meteorological phenomena. There is a sequentiality in the occurrence of these phenomena; the various meteorological events are sequenced by the time at which they are recorded. A scientist asks the query: *"For which volcano eruptions was the strength of the most recent earthquake greater than 7.0 on the Richter scale?".* It is a seemingly innocuous query, but it is difficult to express in a relational query language like SQL and inefficient to evaluate. One possible attempt uses a nested sub-query as follows:

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

Other ways of representing the same query in SQL are not particularly simpler. A conventional relational query optimizer as described in [SAC$^+$79] would probably generate the following query evaluation plan. For every Volcano tuple in the outer query, the sub-query would be invoked to find the time of the most recent earthquake. Each such access to the subquery involves an aggregate over the entire Earthquake relation. The time of the most recent earthquake is used as a join condition to probe the Earthquake relation in the outer query. Finally, the selection condition to check that the strength is greater than 7.0 would be applied. Even the knowledge that the Earthquakes and Volcano relations are sorted by time would not significantly alter the query plan. A more efficient evaluation strategy does however exist; the two sequences can be scanned in lock step (similar to a sort merge join). The most recent earthquake record scanned can be stored in a temporary buffer. Whenever a volcano record is processed, the value of the most recent earthquake record stored in the buffer is checked to see if its strength was greater than 7.0, possibly generating an answer. This query can therefore be processed with a single scan of the two sequences, and using very little memory. The key to such optimization is the sequentiality of the data and the query. □

1.2 Database Systems with Extensible Types

The relational database systems (RDBMSs) of the 1970s supported tabular data containing rows of simple values like integers and strings, and SQL as a query language over this data. While this was adequate to handle most business applications, there were a number of other applications in the area of finance, entertainment, engineering and science that required the storage and manipulation of more complex data structures. One RDBMS solution was to try to "flatten" all complex data into tables which can then be manipulated using SQL – in fact, this approach was used in the example of the previous section. As the example showed, queries may be awkward to pose, and inefficient to evaluate. The other solution is to store the complex data as an uninterpreted "blob". Any manipulations of the complex data have to be done in application code. Neither solution resulted in suitable support for complex data, and by the 1980s, there was a growing move to enhance the type system of an RDBMS to directly support complex data.

Much of the early work in this area came out of the Ingres project at Berkeley. In [SRG83] and [Sto84], Stonebraker proposed that the type system of Ingres could be extended to allow a field of a record to contain a complex value, and even the result of a whole query. However, in order to allow full-fledged extensions to the type system, basic changes to the design of the database system were needed. These design requirements were described by Stonebraker in [Sto86], and led to the development of the new Postgres system [SRH90] as a fully extensible DBMS. An important part of Postgres was the extensible type system that allowed the definition of an arbitrary number of new Abstract Data Types (ADTs). Each ADT was associated with methods that defined its behavior, and that could be invoked within an SQL query. Around the same time, there were other research prototypes like Genesis [BBG+86] and Starburst [HCL+90] that explored other aspects of database system extensibility, but did not focus on the type system.

Other researchers addressed the obvious need for complex data support by developing object-oriented database systems (OO-DBMSs). There are a number of current OO-DBMS commercial products (like O2, Ode, ObjectStore, Versant, etc.), most of which grew out of research systems of the late 1980s. Exodus [CDG+90] was an OO-DBMS toolkit that was used in several research projects. All of these systems allow the definition of persistent database types using a programming language (usually some variant of C++). However, data manipulation was usually expressed by means of procedural code in that language as well, rather than a query in a high-level language like SQL. Observing that this low-level data manipulation interface was inadequate for many applications, some research on OO-DBMSs looked at advanced data models that would directly support certain complex data types like lists, sets, bags, etc. Extra/Excess [VD91] was one such model, as was Melampus [RS91]. Further, the OO-DBMS community gradually agreed on OQL (Object Query Language) [Cat94] as a high-level query language to manipulate object-oriented databases.

As momentum was building behind the OO-DBMS ideas, the relational database community was motivated to address the same problems in a different manner, by leveraging their existing strengths in supporting relational data. Object-Relational databases (OR-DBMSs) [fADF90] were proposed as an extension of RDBMS technology to support object-oriented type systems – in essence, extending the ideas behind the Postgres type system. A commercial version of Postgres called Illustra [Ill94] was developed, whose primary attraction was that it provided the functionality of combining relational data with several kinds of complex data. Several commonly used data types (like images, time-series, etc.) along with their manipulation methods were provided as "datablades". Paradise [DKL+94] is a recent research OR-DBMS that explores the use of complex data types in geographic and scientific databases in a parallel execution environment. The RDBMS community as a whole has in the meantime been attempting to incorporate this approach into its new SQL-3 standard [ISO93]. Committees are debating the issues of exactly which data types should be specially supported, and what methods should be provided on those types. This is the state of the art today, and the conventional wisdom is that OQL and SQL-3 will provide more or less similar features, thereby merging the OR-DBMS and OO-DBMS communities.

1.3 Thesis Outline

One major focus of this thesis is on the support for sequence data in a query processing system. There has been a lot of work on related areas (see Chapter 9), mostly dealing with temporal data in relational databases, and lists in object-oriented databases. The kind of sequence data and queries that this thesis concentrates on are different in flavor from both lists and temporal relational data. Further, practical issues like query optimization and implementation have been studied, and a system has been built to demonstrate the research ideas. These factors distinguish the work in this thesis from the earlier related work. The next few chapters explore the support for sequence data at length. Chapter 2 develops a data model and query algebra for sequences. Chapter 3 studies how sequence queries may be optimized for efficient evaluation. Chapter 4 describes the actual implementation of the database system for sequence data. Performance numbers are used to demonstrate the differences between different storage techniques, and the measured effects of various optimizations. A user-level query language is also presented, that can be used to pose queries against the database.

The second major contribution of the thesis relates to the design of database systems that provide integrated support for different complex data types. Using sequences as an example of one such type, a novel architectural paradigm has been developed and demonstrated. This paradigm of Enhanced Abstract Data Types (E-ADTs) extends the existing state-of-the-art in a significant manner, and has relevance well beyond supporting sequence data. Some of the individual ideas behind E-ADTs have been proposed before, or in parallel with the development of this thesis. However, the work in this thesis puts them together into one comprehensive paradigm. Further, an entire system, PREDATOR, has been built based on E-ADTs , thereby exploring several implementation issues that have not been addressed by the earlier work. Chapter 5 describes the problems with current approaches to integrating complex data types, thereby motivating Chapter 6 which describes the basic E-ADT paradigm, and the architecture of the PREDATOR database system. These chapters step back from the details of sequences, and examine the broader issue of integrating queries over sequences, with queries over other kinds of data. The implementation draws attention to the tradeoffs between design alternatives, and the new interfaces needed in the design of code components. Further, since the conclusion of the thesis is that the next (and indeed, the current) generation of database systems should adopt the E-ADT ideas to support sequence data and other complex data types, Chapter 7 includes migration paths for current systems to evolve in the direction of E-ADTs . Chapter 8 presents some case studies to demonstrate how sequence data can be used along with relational data in PREDATOR, and Chapter 9 concludes the thesis with a brief survey of other research work that is related to the issues explored.

Chapter 2

A Model for Sequence Data

The first question that arises is: "what exactly is sequence data, and what sort of queries does one wish to support over this data?" What is required is a data model for sequences, which is: (a) a mathematical notation for expressing sequence data, and (b) operations on the data that express queries and other manipulations. This chapter presents the $\mathcal{SEQ}$ data model, which captures a large variety of sequence data and queries.

Towards the end of the chapter, the $\mathcal{SEQ}$ model is greatly simplified, and it is this simplified $\mathcal{LSEQ}$ model that the rest of the thesis deals with. The reason for this is that some of the functionality of the $\mathcal{SEQ}$ model is captured by the relational data model with some minor extensions. Consequently, as long as sequences co-exist with relations, the simplified sequence data model suffices. The earlier sections present the entire $\mathcal{SEQ}$ model in detail, since the development of the data model for sequences is itself an individual research contribution [SLR95]. Readers who wish to bypass the details should directly read Section 2.4.1; this describes the $\mathcal{LSEQ}$ model at an intuitive level which is sufficient for the reader to comprehend the rest of the thesis.

2.1 Data Model Fundamentals

Assume the existence of a set T_{Basic} of basic types. During the development of the model, assume that the set T_{Basic} is composed of indivisible atomic types of fixed size (eg., integer, double). The type domain T_i of a basic type t_i is the (possibly infinite) set of values of that type. A *record schema* R is defined as $R \equiv < A_1{:}t_1, \ ... \ , A_n{:}t_n >$ for some finite n. Each of the t_is are types in T_{Basic}, and each A_i is a named *attribute*. The number 'n' of attributes is called the *arity* of the record schema. The *type domain* T_R of R is $(T_1 \times T_2 \times ... \times T_N)$. A *record* of schema R is an element of T_R. Notationally, a record is represented by a list of attribute values between the symbols <>. Given two records $r_a :< a_1, ..., a_m >$ and $r_b :< b_1, ..., b_n >$, the concatenation $< a_1, ..., a_m, b_1, ..., b_n >$ is denoted as $r_a.r_b$. These definitions correspond to the standard notions of typed records. A sequence is now defined as a collection of similar records, ordered by a totally ordered domain like the integers.

Definition 2.1.1 Consider a countable domain D and a total order $\prec$ defined over its elements. Let $\perp \notin D$ be a special value representing a "null" position. The two-tuple $O \equiv < D, \prec >$ is an *ordering domain* iff the following properties hold:

- $\forall p \in D$, let $Next_O$(p) be defined as follows: if $\exists p1 \in D$ such that p $\prec$ p1 $\wedge$ $\nexists p2 \in D$ where p $\prec$ p2 $\prec$ p1, then $Next_O$(p) $\equiv$ p1, else $Next_O$(p) $\equiv \perp$. There must be at most one element $p \in D$ such that $Next_O$(p) $\equiv \perp$.

- $\forall p \in D$, let $Prev_O$(p) be defined as follows: if $\exists p1 \in D$ such that p1 $\prec$ p $\wedge$ $\nexists p2 \in D$ where p1 $\prec$ p2 $\prec$ p, then $Prev_O$(p) $\equiv$ p1, else $Prev_O$(p) $\equiv \perp$. There must be at most one element $p \in D$ such that $Prev_O$(p) $\equiv \perp$.

□

The elements of D are called *positions* of the ordering domain and the notation $p \in O$ is used to denote $p \in O.D$. Operations '$+_O$' and '$-_O$' are defined as follows:

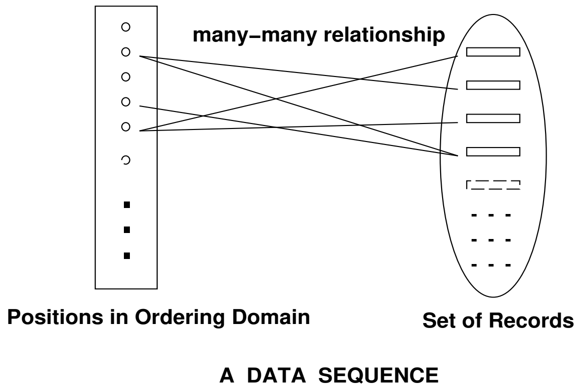

Figure 1: Mathematical Representation of a Sequence

- $\forall p \in O,\, p +_O 0 \equiv p$
- $\forall p \in O,\, p -_O 0 \equiv p$
- $\forall i \in \mathcal{I}, \forall p \in O,\, p +_O i \equiv p1$ if $Next(p) \in O \wedge Next(p) +_O (i-1) \equiv$ p1.
- $\forall i \in \mathcal{I}, \forall p \in O,\, p -_O i \equiv p1$ if $Prev(p) \in O \wedge Prev(p) -_O (i-1) \equiv$ p1.
- $\forall i \in \mathcal{I}, \forall p \in O,\, p +_O i \equiv p -_O (-i)$

An ordering domain is therefore a total order with well-defined predecessor and successor functions.

Definition 2.1.2 Let S be a set of records of schema R, and let O be an ordering domain. A many-to-many relationship O_S from O to T_R is an *ordering of S by O* iff:

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

□

The first part of the definition specifies that every record in S is associated with some position. The second part specifies that no record outside of S is associated with any position.

Definition 2.1.3 A *sequence* Seq is a 3-tuple $< S, O, O_S >$ where

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

□

Figure 1 provides a visual interpretation of this definition of a sequence. Note that each position could be related to more than one record, and vice versa. While every record has to map to at least one position, the converse is not necessarily true, and there could be positions that are "empty", mapping to no records.

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

- $< O, PF >$, where PF is a function that defines for every position p $\in$ O a set of records PF(p) such that $\forall$ p $\in$ O, $\forall$ r $\in$ T_R, r $\in$ PF(p) $\Leftrightarrow$ O_S(p,r).
- $< S, RF >$, where RF is a function that defines for every record r in T_R, a set of positions RF(r) such that $\forall$ p $\in$ O, $\forall$ r $\in$ T_R, p $\in$ RF(r) $\Leftrightarrow$ O_S(p,r) is true.

The first representation is called *Positional* and defines a sequence by the *set of records that map to each position*. The second representation is called *Record-Oriented* and defines a sequence by the *set of positions that map to each record*. While both are equivalent representations of the same sequence, each provides a distinct abstraction of the data that influences the choice of query operators and processing strategies. Existing systems and models are typically oriented to one of these two abstractions of ordered data. However, the $\mathcal{SEQ}$ model recognizes the need to support both classes of sequence operators, since they share a common underlying definition of a sequence as a many-many relationship.

Sequence data is manipulated using an algebra of query operators. A sequence query operator defines a new output sequence using existing input sequences. The number of input sequences defines the *arity* of the operator. It is possible to compose multiple operators, with the output of one operator becoming the input of another operator.

A *query* is an acyclic graph of composed *operators*. The leaves of the graph are base sequences, either stored in the database or obtained from some external source. The Positional representation of a sequence gives rise to a class of operators with a positional flavor. The Record-Oriented representation gives rise to a class of operators with a record-oriented flavor. Each of these classes of operators is now considered separately. The descriptions correspond to templates of operators rather than specific ones. The specific operators are obtained from the templates by filling in certain parameters. An operator M of arity n that uses a parameter P is denoted as M_n[P].

2.2 Positional Operators

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

- Map Operators: M_1[Fn]: If the input sequence of the map operator M is $< O, PF_1 >$, then the output sequence is $< O, PF >$, where for all positions p $\in$ O, $PF(p) \equiv Fn(p, PF_1(p))$.
  A function Fn is applied to each position and the set of input records at that position. The function returns a set of output records that map to the same position in the output sequence. The input and output sequences have the same ordering domain of positions. As examples, Fn could apply a selection condition (eg., check if the first attribute is greater than the third attribute) or a projection operation (eg., retain only the second attribute) to the records in $PF_1(p)$. Further, the projection operation could involve arithmetic (eg. retain the product of the second and third attributes). Note that the selection condition or projection operation could involve the position as well.
- N-ary Operators: M_N[Fn]: If the input sequences of the N-ary operator M are $< O, PF_1 >$, $< O, PF_2 >$, ..., $< O, PF_N >$, then the output sequence is $< O, PF >$, where for all positions p $\in$ O, PF(p) $\equiv$ Fn(p, PF_1(p), PF_2(p), ..., PF_N(p)).
  A function Fn is applied to each position and the sets of records at that position in each of the input sequences. The function returns a set of output records that map to that position in the output sequence. The input and output sequences must have the same ordering domain. Fn could, for instance, be any of the relational operators: union, intersection, cartesian product, set difference or join. For example, the operator could perform the Cartesian product or the join of the sets of input records at every position.

- Offset Operators: M_1[Offset]: If the input sequence of the *offset* operator M is $< O, PF_1 >$, then the output sequence is $< O, PF >$, where for all positions p $\in$ O, $PF(p) \equiv PF_1(p1)$ and $p \equiv p1 +_O Offset$.
  The offset operator defines an output sequence which is identical to the input sequence except that the entire sequence has been "shifted" by the specified number of positions.
- Aggregate Operators: M_1[agg_func,agg_window]: If the input sequence of the aggregate operator is $< O, PF_1 >$, then the output sequence is $< O, PF >$, where for all positions p in O,

$$PF(p) \equiv agg\_func(\{r | p1 \in agg\_window(p) \wedge r \in PF_1(p1)\}).$$

  The *agg_window* specifies a subset of the positions of the ordering domain, based on the current position. The *agg_func* function computes an aggregate over the set union of records in PF_1(p1) at the selected positions p1. For example, an aggregate to compute the running 3-day average of a daily sequence would have *agg_func* $\equiv$ Avg and *agg_window(p)* $\equiv$ {p, p-1, p-2}. Obviously, by suitably defining *agg_func* and *agg_window*, the map and offset operators previously defined become special cases of the aggregate operator. However, one expects that the aggregate operator be instantiated using a limited set of simple aggregate functions (eg.,, Count, Sum, Min, Max, Avg), and hence the separate definitions of Offset and Map operators.

Apart from these operator templates, there are also two other kinds of Positional operators; the *Recursive* operators and the *Zoom* operators. Since these require some detailed explanation, they are dealt with in separate subsections.

2.2.1 Recursive Positional Operators

These operators define the output sequence at a position in terms of the output sequence at the previous position. The *linear recursive Positional operators* have the general form *M_1[Fn, Fn_window, InitialData]*. If the input sequence is $< O, PF_1 >$, then the output sequence is $< O, PF >$, where

- PF(p) $\equiv$ Fn(p, CurrentData(p), RecursionData(p)),
- CurrentData(p) $\equiv$ $\{r | p1 \in Fn\_window(p) \wedge r \in PF_1(p1)\}$),
- RecursionData(p) $\equiv$ PF(p-1) if $Prev_O(p) \neq \perp$, else RecursionData(p) $\equiv$ InitialData.

These operator templates form a superset of the aggregate operators, since they can express all the aggregate operators with a suitable choice of template parameters. This is a limited form of recursion, because the only recursive data that contributes to PF(p) is PF(p-1). However, this leads to a rich class of queries. Here are some specific examples of the linear recursive operator:

- Previous Operator: Fn_window(p) $\equiv$ {p}, InitialData $\equiv$ {}, and Fn $\equiv$ CurrentData(p) if (CurrentData(p) $\neq$ {}), else Fn $\equiv$ RecursionData(p). This is the "most-recent" operator defining for every position, the mapping from the most recent position that has a non-empty mapping.
- Decay Operator: Fn_window(p) $\equiv$ {p}, Fn $\equiv$ foo(RecursionData(p), CurrentData(p)). Depending on the specification of InitialData and foo, a large variety of decay functions can be specified. The "previous" operator is a special case of a decay operator.
- Recursive Aggregate Operator:(eg. Count with window size N) Fn_window(p) $\equiv$ {p, ..., p $-_O$ N + 1}, InitialData $\equiv$ {0}, and Fn $\equiv$ (RecursionData(p) - count(PF_1(p $-_O$ N)) + count(PF_1(p)) if (RecursionData(p) $\neq$ {}), else Fn $\equiv$ count(PF_1(p)).

- Next Operator: Fn_window(p) $\equiv$ {p, p+1, ..., $+\infty$}, InitialData $\equiv$ {}, and Fn $\equiv$ PF_1(p) if (PF_1(p) $\neq$ {}), else Fn $\equiv$ RecursionData(p) if (PF_1(p-1) = {} $\wedge$ RecursionData(p) $\neq$ {}), else Fn $\equiv$ NextData(p), where NextData(p) $\equiv$ PF_1(p1) and $\forall p2 \in O, p < p2 < p1 \Rightarrow PF_1(p2) = \{\}$.

The linear recursive operators described thus far are limited by the fact that the only recursive information used to define PF(p) is the value of PF(p-1). One extension is to allow additional "state" information to be carried recursively from p-1 to p, thereby adding expressive power to the operators. The *linear recursive-state Positional operators* have the general form *M_1[Fn, Fn_window, InitialData, InitialState, StateFn]*. If the input sequence of M is $< O, PF_1 >$, then the output sequence is $< O, PF >$, where

- PF(p) $\equiv$ Fn(p, RecursionData(p), CurrentData(p), StateData(p)),
- RecursionData(p), and CurrentData(p) are defined as for linear recursive operators,
- StateData(p) $\equiv$ State(p-1) if $Prev_O(p) \neq \perp$, else StateData(p) $\equiv$ InitialState,
- State(p) $\equiv$ StateFn(StateData(p), CurrentData(p), RecursionData(p)).

Here are some specific examples of the linear recursive-state operator; rigorous definitions have been avoided to make the presentation easier to read.

- Nth Previous Operator: Fn_window(p) $\equiv$ {p}, InitialData $\equiv$ {}, and InitialState $\equiv$ {}. The state maintains PF(p1) for the N most recent positions p1 at which PF(p1) was non-empty. Fn $\equiv$ PF(p1) for the smallest p1 whose PF(p1) is maintained in the state. This is an extension of the "previous" operator defined earlier. Analogously, the Nth Next operator can be defined as an extension of the "next" operator.
- Value Aggregate Operator:(eg. Avg of last N non-empty positions) Fn_window(p) $\equiv$ {p}, InitialData $\equiv$ {0}, InitialState $\equiv$ {} and Fn $\equiv$ (RecursionData(p) - count(PF_1(p $-_O$ N)) + count(PF_1(p)) if (RecursionData(p) $\neq$ {}), else Fn $\equiv$ count(PF_1(p)). The count(PF_1(p $-_O$ N)) is derived from the state, which the StateFn maintains as the PF_1(p1) for the last N positions, empty or not.

From the definitions of the various operator templates, it may be apparent that the class of linear recursive-state Positional operators subsumes all the other unary Positional operators described so far. This is because the linear recursive-state operators subsume the linear recursive operators, which in turn subsume the aggregate operators, which in turn subsume the remaining operators. However, the rationale for defining the more primitive operators is that they are commonly used and are more directly amenable to efficient implementation and optimization.

2.2.2 Zooming Operators

In all the operators presented so far, the ordering domains of the input and output sequences have been identical. In general, it is not meaningful to define Positional operators, unless the ordering domains of the various input sequences and the output sequence are related in some way. In this section though, a class of "zooming" operators is considered in which the relationship between input and output ordering domains is not equality. For instance, Figure 3 shows one set of hierarchical relationships between common temporal ordering domains.

Consider a sequence of stock market data records, with values recorded every day. The ordering domain of this data sequence is the domain of days. However, someone querying this sequence might want to view the data at a weekly granularity, thus effectively modifying the ordering domain to weeks instead of days. Since the mapping from days to weeks is well defined, it should be possible to allow such "zoom" operations.

Consider two totally ordered domains O_1 and O_2.

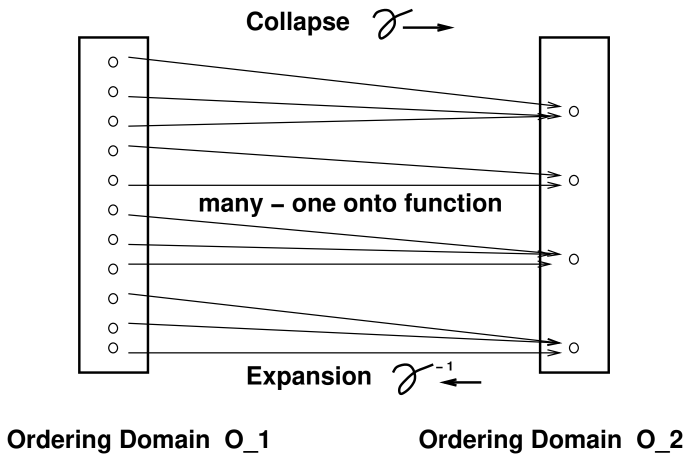

Figure 2: Collapse and Expansion

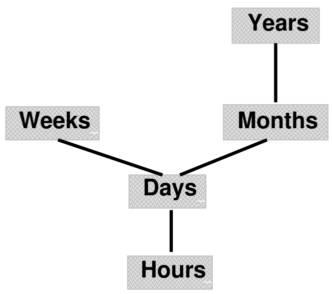

Figure 3: An Ordering Hierarchy

Definition 2.2.1 A *collapse* γ is a many-to-one onto function from O_1 to O_2[1]. The inverse mapping γ^{-1} is called an *expansion* from O_2 to O_1. $\Box$

Figure 2 shows a collapse from O_1 to O_2 and the corresponding expansion from O_2 to O_1. Examples of collapses include days to months, months to years, and feet to yards.

Definition 2.2.2 Consider a sequence $Seq_1 \equiv < O_1, PF_1 >$. Consider another totally ordered domain O_2, and let γ be a collapse from O_1 to O_2. A sequence $Seq_2 \equiv < O_2, PF_2 >$ is a γ-*collapse* of Seq_1 iff:
$\forall p_2 \in O_2,\ PF_2(p_2) \equiv \{\mathrm{r} \mid p_1 \in O_1 \wedge p_2 = \gamma(p_1) \wedge r \in PF(p_1)\ \}$
$\Box$

Definition 2.2.3 Consider a sequence $Seq_2 \equiv < O_2, PF_2 >$. Let γ^{-1} be an expansion from O_2 to another totally ordered domain O_1. A sequence $Seq_1 \equiv < O_1, PF_1 >$ is a γ-*expansion* of Seq_2 iff:
$\forall p_1 \in O_1,\ PF_1(p_1) \equiv PF_2(p_2)$ where $p_2 = \gamma(p_1)$
$\Box$

The Collapse and Expand operators are unary operators based on the definitions of sequence collapse and expansion. These operators take an argument specifying a collapse or expansion function, and derive an output sequence by appropriately collapsing or expanding the input sequence. It is expected that certain "well-known" collapse and expansion functions on the ordering domain are pre-defined, and can be used by the query operators.

Note that γ-expansion(γ-collapse(Seq)) is not equivalent to Seq. This is because a collapse of a sequence does not retain the precise mapping of the original sequence. For example, if a daily sequence is collapsed to a weekly sequence, the information of which record mapped to which day is not retained. More imprecise information about which record mapped to which week is available instead. It is also not necessary that a collapse preserve some notion of the original ordering of the data. For example, consider a collapse from days to the day-of-the-week (Sunday through Saturday). A record X that "happens before" record Y in the original sequence does not necessarily "happen before" record Y in the collapsed sequence. To formalize this notion:

Definition 2.2.4 Let $\prec_1$ and $\prec_2$ be the total orders associated with ordering domains O_1 and O_2 respectively.

[1](i.e., each element of O_1 maps to exactly one element of O_2, many elements of O_1 may map to the same element of O_2, and every element of O_2 is mapped to by some element(s) of O_1.

A collapse γ from O_1 to O_2 is *blocked* iff the following condition holds:

$$\forall p1, p2 \in O_1, (\gamma(p1) = \gamma(p2)) \Rightarrow (\forall p3, p1 \prec_1 p3 \prec_1 p2 \Rightarrow \gamma(p3) = \gamma(p1))$$

A collapse γ from O_1 to O_2 is *order-preserving* iff the following condition holds:

$$\forall p1, p2 \in O_1, p1 \prec_1 p2 \Rightarrow (\gamma(p1) \preceq_2 \gamma(p2))$$

An expansion γ^{-1} from O_2 to O_1 is order-preserving (blocked) iff the collapse γ is order-preserving (blocked). $\Box$

All order-preserving collapses are blocked, though the converse is not necessarily true. The definition of blocked collapse and expansion of ordering domains is similar to the concept of intervallic partitions in [CR86], and the concept of time units in [WJS93]. The definition of order-preserving collapses will be used in the next chapter on optimization techniques.

There is one other variant of an order-preserving Collapse operator that does not have a corresponding Expand operator. This variant is not based on a pre-defined collapse γ. Instead, the block boundaries are data-dependent. As motivation, consider a sequence of the hourly prices of a stock. The desired result of a query is a sequence of price trends for that stock, i.e., a sequence of (lengths of) hour ranges during which the stock price monotonically increased or decreased. The ordering domain of the result of such a *conditional collapse* is the domain $\aleph$ of non-negative integers.

Definition 2.2.5 Consider a sequence $Seq_1 \equiv < O_1, PF_1 >$. A *conditional collapse* γ_{Seq1} is a many-to-one function from O_1 to $\aleph$ such that $\gamma_{Seq1}(p) \equiv \gamma_{Seq1}(p -_{O_1} 1) + 1$ if $Prev_{O_1}(p) \neq \perp$ and $Cond(p, PF_1(p))$ is true, else $\gamma_{Seq1}(p) \equiv \gamma_{Seq1}(p -_{O_1} 1)$ if $Prev_{O_1}(p) \neq \perp$ and $Cond(p, PF_1(p))$ is false, else $\gamma_{Seq1}(p) \equiv 0$ if $Prev_{O_1}(p) = \perp$. $\Box$

The Conditional Zoom operators are based on this definition of a conditional collapse, and are parameterized by the actual input sequence and the $Cond$ condition.

2.3 Record-Oriented Operators

Record-Oriented operators are based on the view of a sequence $< S, O, O_S >$ as a set of positions associated with each record in S. The operators are defined by specifying the RF of the output sequence. There already exists extensive literature that describes a realistic algebra of operators on sets of records; this is the *relational algebra.* There have also been well-documented extensions to the relational algebra to model partitioned aggregate functions [Klu82]. Record-Oriented operators are merely extensions of the relational algebra operators with added semantics to specify the RF mapping of the output set of records. Similar extensions have already received much attention in the temporal database research community, and there is even an extension to SQL proposed to handle such operations [Com94]. Consequently, these operators are not the focus of the research in this thesis, and are not considered after this section. The Record-Oriented operators are:

- Selection Operator: M_1[Condition]: If the input sequence is $< S_1, RF_1 >$, then the output sequence is $< S, RF_1 >$, where S is the union of the records $r \in S_1$ such that Condition(r, RF_1(r)) is true. In other words, this is the standard relational selection operation, where the selection condition on record r can include the positions RF_1(r) mapped to r.
- Projection Operator: M_1[RecProj,PosProj]: If the input sequence is $< S_1, RF_1 >$, then the output sequence is $< S, RF >$, where for every $r_1 \in S_1$, there is an $r \in S$ such that RecProj(r_1) $\equiv$ r and RF(r) $\equiv$ PosProj($RF_1(r_1)$). This is an extension of the standard relational projection operation, that generates a projection of each record, as well as a projection of the set of positions associated with each record. For example, Start-Of (End-Of) is a unary

operator that associates each output record with the first (last) of the set of positions that the corresponding input record maps to.

- Cross-Product Operator: M_2[PosFunction] : If the input sequences are $< S_1, RF_1 >$ and $< S_2, RF_2 >$, then the output sequence is $< S, RF >$, where for every $r_1 \in S_1$ and $r_2 \in S_2$ there is an $r \in S$ such that $r \equiv r_1.r_2$ and RF(r) $\equiv$ PosFunction($RF_1(r_1)$, $RF_2(r_2)$). From the definition of a sequence, no record $r \in S$ can have RF(r) $\equiv$ $\{\}$. Therefore, if PosFunction returns an empty set of positions, the record r is not part of the output sequence. This is therefore an extension of the relational cross-product operator that can have many variants depending on the particular choice of PosFunction. Each of these variants results in a particular form of "sequence-join". For instance, if PosFunction($RF_1(r_1), RF_2(r_2)$) is an intersection of the sets of positions of the two records, the resulting operator is an "Overlap Join". If PosFunction($RF_1(r_1), RF_2(r_2)$) checks for containment of one of the position sets in the other, different variants of the "Contain-Join"[LM90] are derived. Similarly, the Make-Interval operator is a sequence join where PosFunction($RF_1(r_1), RF_2(r_2)$) gives the minimal interval that includes all the positions of both the input records.

- Union Operator: M_2[]: The Union operator computes the union of the two sets of records. If there are identical records in the two sets, the corresponding record in the result maps to the union of the positions in the RF mappings of the original records, otherwise, the RF associated with the records is unchanged.

- Minus Operator: M_2[]: The Minus operators computes the difference between the two sets of records. If there are two identical records in the two sets, the result contains the record mapped to the set difference of the two RF mappings. Otherwise, the RF mapping associated with a retained record is unaffected.

- Aggregate Operator: M_1[agg_part, rec_agg_func, pos_agg_func]: The Groupby Aggregate operators are unary operators defined by three functions. One function *agg_part* is used to partition the input records into disjoint sets. There is one output record corresponding to each partition, whose value is determined by applying the *agg_func* aggregate function to the partition. This output record maps to the set of positions defined by pos_agg_func($\cup$ RF(r_i), $r_i \in$ partition(r)).

2.3.1 Duality of Sequences

The $\mathcal{SEQ}$ model therefore does not merely provide the Positional abstraction or the Record-Oriented abstraction, but allows *both* to coexist simultaneously in specifying a query. This is a recognition of the fact that a sequence is at once "Positional " as well as "Record-Oriented ". This duality has also been recognized in the context of temporal databases by [WJS93], which defines a bimodal temporal calculus based on these two abstractions. In the Record-Oriented abstraction, a sequence can be viewed as a *relation* whose records have some extra attributes that contain the RF mapping information. Given this, the functionality of the RF operators can be obtained by extending the relational operations with the semantics to manipulate the attributes containing the RF mapping. This is the approach that has been taken in the temporal relational database research community, and there is extensive literature on a variety of approaches to actually implement such support [TCG+93]. Since the underlying assumption of this thesis is that support for sequence data will be provided in the context of a general-purpose system, it is reasonable to assume that support for relational data will also be present. It will be possible to treat positional sequence data as a relation (by type-casting it) and vice-versa, thereby allowing a single query to use both Positional and Record-Oriented abstractions. Consequently, the rest of this thesis does not deal with Record-Oriented sequence queries, but focuses instead on the more interesting issue of supporting Positional queries.

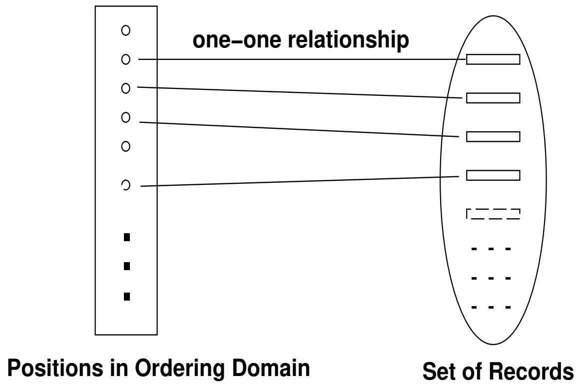

Figure 4: A Sequence in the $\mathcal{LSEQ}$ Model

2.4 Expressing Queries using the $\mathcal{SEQ}$ Model

It is possible to further simplify the Positional $\mathcal{SEQ}$ data model in the context of a general-purpose database system. Recall that a sequence is a many-to-many mapping from positions to records, so that PF(p) is a set of records of atomic types, and RF(r) is a set of positions that map to each record. This can be simplified to a one-to-many mapping by duplicating records that map to several positions, so that RF(r) maps to a single position.

Assume that the type system is augmented to allow an attribute of a record to itself be a set of records. A database system that allows such *nested* structures is usually referred to as a *non-1NF* or *complex object* database. Data models supporting such nested types are becoming increasingly common, especially in object-relational systems [Ill94, Uni93]. In such a complex object system, sequence data could be modeled even more simply as a one-to-one mapping, where each position maps to either no record or one record. That record could itself contain a nested set of records as an attribute.

These two simplifications of the $\mathcal{SEQ}$ data model lead to the $\mathcal{LSEQ}$ model in which a sequence is limited to a one-to-one mapping between positions and records, and there can be "empty" positions. The Positional operators defined in Section 2.2 all apply in the $\mathcal{LSEQ}$ model, though their definitions are simplified because they only need to deal with a single record at every position. Any operators like Collapse that generate sets of records are associated with an aggregate function that creates a single record from that set. The rest of the thesis is based on this $\mathcal{LSEQ}$ model. A brief informal description of the $\mathcal{LSEQ}$ model is presented in the next subsection; there may be some repetition of earlier details, since the subsection is meant for readers who have not read the rest of this chapter;

2.4.1 $\mathcal{LSEQ}$ Model

An *ordering domain* is a countable totally ordered domain (like the integers) with well-defined predecessor and successor functions. The elements of the domain are called *positions.* A *sequence* is a one-to-one mapping from a set of records of similar schema to an ordering domain; this is illustrated in Figure 4. The sequence is represented as $< O, PF >$, where O is the ordering domain and PF(p) is the "position function" which defines the record associated with position p. The schema of a record can include attributes of any data type. Every record is associated with one position, but there could be "empty" positions that map to no record.

Queries are expressed by composing primitive operators from a sequence algebra. Each operator takes one or more input sequences and generates an output sequence. The definition of each operator specifies the ordering domain and the PF of the output sequence in terms of those of the input sequences. There are various categories of operators, which are described by generic templates from which specific operators can be instantiated. A particular implementation will choose to provide a specific subset of these operators. One possible subset of Positional operators along with their arities is listed in Table 1. The $\mathcal{SEQUIN}$ language described in Chapter 4 is based essentially on this subset of operators. Each of these operators is now briefly explained:

Operator	Description	Template
Select_1	Conditional selection	Map
Project_1	Attribute projection	Map
Offset_1	Shift of input	Offset
Compose_2	Compose by position	N-ar¡y
Join_2	Compose with Select	N-ary
MIN,MAX,CNT,AVG_1	Moving window aggregate	Aggregate, Recursive
Next/Previous_1	Shift ignoring empty pos	Recursive
Collapse/Expand_1	Change of ordering domain	Zoom
Conditional Collapse_1	Data-dependent Collapse	Conditional Zoom

Table 1: An $\mathcal{LSEQ}$ Algebra

- *Select:* Filters an input sequence by applying a condition to the record at every position. The output sequence contains those records that satisfy the condition, and they are mapped to the same positions as in the input.
- *Projection:* Strips out some of the attributes of the input sequence records, but leaves the position mapping unchanged. Some of the attributes of the output record could be the result of arithmetic or other expressions.
- *Offset:* Performs a "phase-shift" of the input sequence, but does not modify the records.
- *Compose:* Joins two input sequences by their position, concatenating records from the two sequences that map to the same position.
- *Join:* Performs a compose operation, and additionally applys a selection condition to the output records.
- *Moving-Window Aggregate:* Computes an aggregate function like MIN, MAX, CNT, or AVG over a "window" of the input sequence, which is defined relative to the current output position. For example, "the average of the input records over the last 3 days". Some moving window aggregates may also be defined as recursive operators; the value of the aggregate function at a position can be determined from its value at the previous position by dropping the oldest input value in the window and adding in the newest input value.
- *Previous/Next:* Recursive operators that associate with each output position, the most recent(Previous) or imminent(Next) record in the input sequence. If the input sequence has empty positions in it, these get filled in the output sequence. The recursion arises because, for example, the "previous" record at a position is the input record at that position if one exists, else it is the "previous" record at the preceding position.
- *Collapse/Expand:* Change the granularity of a sequence. A collapse operator coarsens the granularity and an aggregate function is applied to all the records that map to the same coarse granularity position. For example, a daily sequence may be collapsed to a weekly sequence, where the records hold the average value of all days in the week. An expand operator does the converse; an expansion from weeks to days leaves every day with the same record as the week containing it.
- *Conditional Collapse:* These are similar to the collapse operators, except for two differences. The resulting domain is always the domain of non-negative integers, and the boundaries at which the collapses occur are data-dependent. For example, a daily sequence may be collapsed into a sequence of monotonic rising and falling trends.

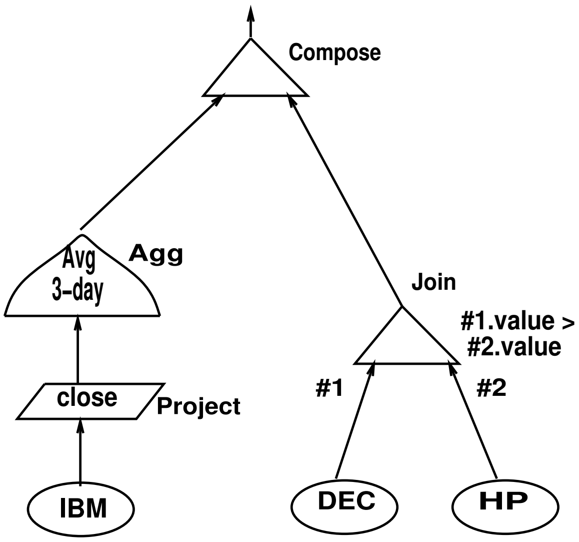

Figure 5: Example of Positional Query

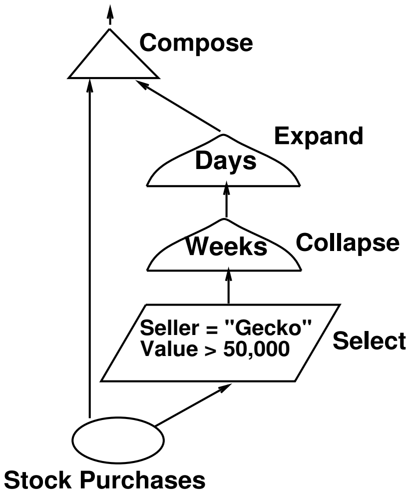

Figure 6: Query Using Collapse and Expand

It is important to note that other operators belonging to the broad categories described in Section 2.2 may also be defined within the $\mathcal{LSEQ}$ model, and optimized using the techniques described in the Chapter 3. As an example of the use of the operators described, Figure 7 shows a graphical representation of the query graph of the introductory "volcanos and earthquakes" Example 1.1.1. The following examples demonstrate other simple sequence queries drawn from the stock market domain. Many of the examples used in the rest of the thesis are based on stock market data. However, they may appear overly simple or contrived to a real stock market analyst. The reason for simplifying the example queries is to demonstrate the underlying concept, rather than explain the real-world significance of the queries. For example, while moving averages are of tremendous significance in technical analysis of the stock markets, they are usually used in pairs (3-day and 9-day average) or triples, rather than individually. Similarly, moving averages are often displaced by an offset rather than directly used. Popular measures like the Relative Strength Index (RSI) involve arithmetic manipulations of the ratio of the number of up closes to the number of down closes over a moving window. This thesis has focused in its presentation on the simpler underlying queries that form the building blocks for the more realistic technical analysis calculations. The reader is referred to [LL92] for a better understanding of technical analysis techniques.

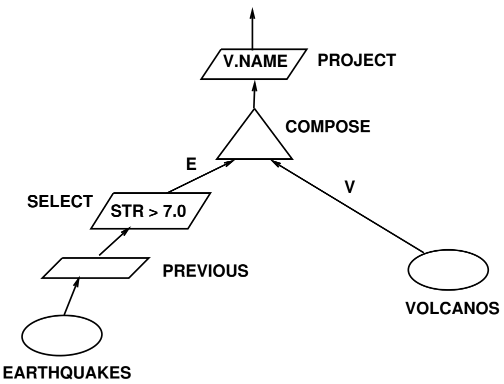

Figure 7: Volcano & Earthquake Query

Example 2.4.1 Assume that daily sequences of stock data are available for IBM, DEC and HP. A query could ask for the 3-day average of the close of IBM stock values when the value of DEC is greater than that of HP. The query graph is visualized in Figure 5. The Join operator (a combination of a Compose and a Selection operator) on the right side of the query determines the positions at which the value of DEC was greater than the value of HP. The left side of the query generates the sequence corresponding to the 3-day moving average of the close of IBM. Finally, the Compose operator ensures that the answer sequence contains the value of the moving average for only the desired days. □

Example 2.4.2 Assume that a sequence of stock market transactions has been recorded on a daily basis. The following query needs to be posed: select all daily stock purchases during all weeks in which Gordon Gecko sold more than $50,000 of stock in a single transaction. Figure 6 shows the representation of the query. A selection is applied to the sequence of stock purchases to select the transactions of Gecko of value greater than 50,000. This sequence is then collapsed to a weekly sequence, in which a week that maps to no records implies Gecko had no major transaction that week. This is then expanded back to a daily sequence, in which all days belonging to such weeks map to a record. In essence, this provides a selection of those days that are of interest to the query. Finally this is composed with the original stock sequence, thereby producing the desired answer. Project operators have been omitted in order to keep the figure simple. □

Chapter 3

Optimization for Sequence Queries

Now that there is a model for sequence data and queries, the next step is to develop a mechanism for the evaluation of these queries in a database system. Experience from relational databases has shown that for any specific relational query, there are several possible evaluation strategies of widely varying efficiency. Similar behavior may be expected of sequence queries. The purpose of "query optimization" is for the system to choose a query evaluation strategy (or "plan") that attempts to minimize the query execution time. This chapter discusses optimization techniques for sequence queries based on the $\mathcal{LSEQ}$ data model, with some minor restrictions on the kinds of queries that may be posed. Much of this material was presented in [SLR94]. The topics covered are:

- The *scope* of an operator, and stream-access evaluation : Section 3.1.
- Heuristic optimization techniques, and sequence access modes : Section 3.2.
- Query plan generation and cost formulas : Section 3.3.
- Common sub-expression optimization : Section 3.4

Evaluation Model

To place the subsequent discussion of query optimization in context, a brief description of the basic evaluation model is necessary. Unless otherwise specified, a sequence query is a tree of operators. This definition of a query is more restrictive than an acyclic graph of operators, as defined in Chapter 2. It turns out that it is simpler to optimize tree queries. However, graph queries are also addressed in Section 3.4. The execution of a query generates some specified range (or ranges) of the answer sequence. The base sequences are stored on disk, and there is a cost associated with accessing them. The cost of accessing disk-resident data in its physically sequential order is significantly less than the cost of accessing it randomly. While sequence data need only be ordered logically, it is also often physically accessible in the same order. In fact, the most common kinds of sequences (temporal data) are generated in temporal order, and hence can easily be stored in the same order. A bounded amount of main-memory is available for query evaluation. The difference between random and sequential access to main-memory is typically not significant when compared to the cost of disk accesses. Evaluation proceeds at a centralized site; no parallelism is considered in query processing until Section 3.5.

Notation

For a sequence $S \equiv < O, PF >$, the notations S(p) and S($\{p_1, ..., p_n\}$) are used as shorthand for S.PF(p), and $\{r | 1 < i < n \wedge r \in PF(p_i)\}$ respectively. Where the ordering domain is implicit, the subscripts of '$+_O$', '$-_O$', '$Prev_O$', and '$Next_O$' are dropped. A derived sequence defined by an operator M whose inputs are sequences S_1 through S_n is represented as M(S_1, ..., S_n). Each operator is associated with an "operator function" OpFunc, that defines the output sequence in terms of the input sequences. For the output sequence of the operator M, PF(p) $\equiv$ M.OpFunc(S_1, ..., S_n, p).

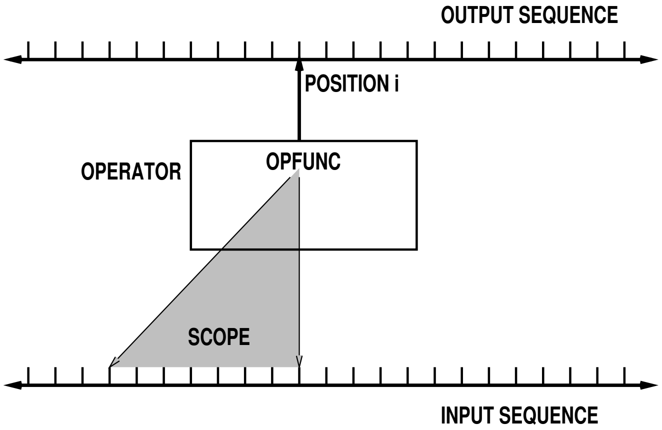

Figure 8: Operator Scope

3.1 Operator Scope

The first concept presented is the *scope* of an operator, which defines the portion of each input sequence needed to compute the value of the output sequence at a specific position. The operator scope is central to sequence query optimization and is used for all of the following:

- To propagate meta-information through a query tree.
- To apply query equivalence transformations.
- To determine run-time access and caching strategies.
- To determine strategies for handling common sub-expressions.
- To determine replication strategies for partitioned parallel execution.

Definition 3.1.1 Consider an operator Op(S_1, ..., S_n) that defines $S \equiv < O, PF >$. The *scope* of Op is a function $Scope(k,p)$ that returns a minimal set of positions that satisfies the following property for each position p. If $S_{1'}, ..., S_{n'}$ is defined as follows:

- $\forall k \in \{1..n\}, \forall j \in O, S_{k'}(j) \equiv S_k(j)$ if j $\in Scope(k,p)$, else $S_{k'}$(j) is not defined.

Then, the following equality must hold:

- $Op.OpFunc(S_1, .., S_n, p) \equiv Op.OpFunc(S_{1'}, .., S_{n'}, p)$

□

Definition 3.1.2 Scope_δ(k,p) $\equiv$ (Scope(k,p) - Scope(k, p - 1)) if $Prev(p) \neq \perp$, else Scope_δ(k,p) $\equiv$ Scope(k,p). □

When the specific input sequence is obvious from the context, the 'k' argument is omitted from Scope(k, i) and Scope_δ(k, p). The OpFunc notation is also overloaded so that the output record PF(p) at p is represented as OpFunc(S_1, .., S_n, p), and also as OpFunc(S_1(Scope(1,p), .., S_n(Scope(n,p)), p).

Figure 8 illustrates a generic operator defining an output sequence based on an input sequence. The operator function OpFunc accesses the current input record and the last seven input records in order to derive the output record at a particular position p. These positions form Scope(p), and Scope_δ(p) depends on the value of $Scope(p-1)$.

The following properties of the scope of an operator may be identified for each input sequence:

- The scope *size* at position p is the number of positions in Scope(p). The scope size could be fixed for all p or could vary with p. A special case is the unit scope of size one. For example, a selection operator has a fixed scope of size one, a 3-day moving average has a scope of size 3, and a decay operator has a variable size scope.

- The scope *sequentiality* indicates how the scopes at successive positions overlap. If $\forall p, \forall i \in Scope\_\delta(p), \forall j \in Scope(p-1)$, we have $i > j$, then the scope is sequential. For example, the scope of a 3-day moving average is sequential, while the scope of a collapse operator from days to day-of-the-week is not.

- The scope *overlap* at position p is defined by $Scope(p) \cap Scope(p-1)$ if $Prev(p) \neq \perp$, else it is the empty set. It can also be defined as $Scope(p) - Scope\_\delta(p)$.

These properties of the scope of an operator can be used to reason about the optimization of a complex query tree created by the concatenation of several such operators. Importantly, they can be used to determine access and caching strategies.

3.1.1 Stream-Access and Operator Scope

Sequence query optimization techniques exploit the possibility of accessing the stored data in the sequence order. Such an access is called *stream-access*. Given a complex query, it is possible to guarantee, based on the scopes of the operators in the query, that all the base sequences are accessed no more than once in a *stream-access* fashion during the evaluation of the query. Obviously, this is trivial (and uninteresting) if there were no bound on the memory usage during query evaluation. This section describes how such a guarantee can be made with a bounded usage of memory.

The model of sequence query evaluation associates a cache (a randomly accessible main-memory buffer) with each query operator. Caches can store input records for efficient subsequent retrieval. There is some mechanism (usually an index) provided for accessing the cached records associatively by position. Repeated access to cached records does not contradict stream-access to the stored sequence data. The definitions of recursive-state operators includes “state” data that is maintained by the operators. The results of this section are limited to those operators whose state data is of bounded size; all the example recursive-state operators in Section 2.2.1 fall in this category[1].

Definition 3.1.3 A query evaluation has the *bounded-cache* property if the size of the cache (number of input records stored) at every operator is bounded by a constant determined independent of the actual data in the input sequences. □

It should be noted that a bounded-cache evaluation does not imply that a bounded amount of memory is used during query evaluation. It merely requires that a bounded number of records stay cached in memory during evaluation. If there is some upper bound on the size of each record (based on the types of values in the record), then such an implication may be made.

Definition 3.1.4 A query evaluation has the *stream–access* property if it performs no more than a single scan of its input (base) sequences in sequence order. □

Theorem 3.1.1 *If every operator in a query tree has a sequential, fixed-size scope on all its inputs, then the query has a bounded-cache stream–access evaluation.*

Proof: The proof of the theorem is in two parts.

---

[1]A recursive-state operator that computes the number of unique records since the beginning of the sequence would not fall in this category. This is because its state would have to contain the unbounded set of all earlier records.

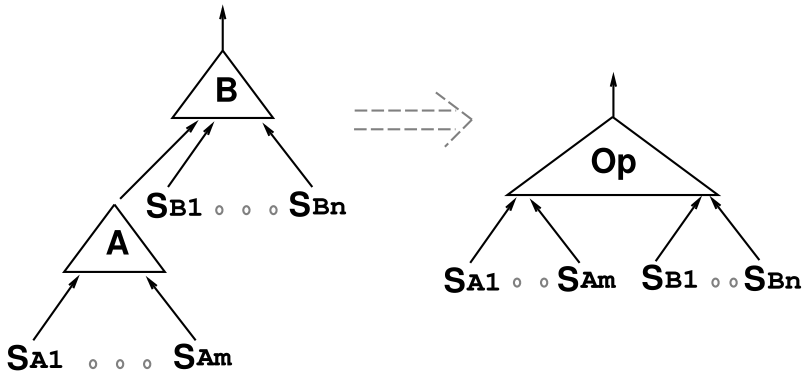

Figure 9: Complex Operator Construction

Part A: For any particular operator with sequential, fixed-size scope on all its inputs, the output sequence can be generated in sequence order with no more than one scan of its inputs in sequence order. This is the per-operator bounded-cache stream-access property.

Part B: If every operator in the query has a bounded-cache stream access evaluation, then the entire query has a bounded-cache stream access evaluation.

The proof of Part A is by constructing an evaluation mechanism for each individual operator that generates the output sequence in its sequence order. The operator scans the input sequences in their sequence orders. For every position in the output sequence, the scope of each input sequence at that position is cached. Since the scopes are all of fixed sizes, the bounded-cache property is not compromised. When moving on to the next output position, some new input records may need to be read. However, since the scopes are sequential, a FIFO cache replacement policy guarantees that the scopes of the input sequences at the new position will reside in the cache. Consequently, this is a bounded-cache stream access evaluation for each individual operator, and the output sequences are generated in their sequence orders.

The proof of Part B is by induction over the nodes in the query tree. The base case for the induction is the proof of Part A. The basic idea is that a concatenation of two operators with fixed-size sequential scopes on their inputs (and hence with bounded-cache stream-access evaluations) is equivalent to a single complex operator with a bounded-cache stream access evaluation, and fixed-size sequential scopes on its inputs. By inductive application to the entire query, the query itself is equivalent to a single giant complex operator with a bounded-cache stream access evaluation. The important part of this proof is the inductive step where a complex operator is constructed from two concatenated operators, each with bounded-cache stream access evaluations.

Consider two operators A and B composed as shown in Figure 9. Let A(S_{A1}, .., S_{Am}) $\equiv$ S_A, and let B(S_A, S_{B1}, .., S_{Bn}) $\equiv$ S_{out}. The composition forms a complex operator Op(S_{A1}, .., S_{Am}, S_{B1}, .., S_{Bn}). The scope is defined by $\forall i, 1 \leq i \leq m$, Op.Scope(Ai, p) $\equiv$ $\{j \mid k \in$ B.Scope(A, p) $\wedge$ j $\in$ A.Scope(Ai, k)$\}$, and $\forall i, 1 \leq i \leq n$, Op.Scope(Bi, p) $\equiv$ B.Scope(Bi, p). The operator function OpFunc(S_{A1}(Scope(A1, p)), .., S_{Bn}(Scope(Bn, p)), p) $\equiv$ B.OpFunc(S_A(Scope(p)), S_{B1}(Scope(p)), .., S_{Bn}(Scope(p)), p).

From this definition of complex operator Op, the properties of its scope can be proven:

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

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

At this stage, since Op has been shown to have fixed-size sequential scopes on all its inputs, it is guaranteed to have a bounded-cache stream access evaluation (by the proof of Part A). However, as a practical matter, it may be difficult to actually construct the OpFunc for this complex operator, since it may require merging code from A and B. Instead, as is demonstrated next, the execution of the complex operator can be performed in a bounded-cache stream-access fashion using the existing operator code.

Operator B is evaluated in the standard bounded-cache stream-access fashion, attempting to perform a stream-access on all its inputs. When it sets up a stream-access on input S_A, this causes the stream access evaluation of operator A. The evaluation of A is a stream-access bounded-cache evaluation, and generates its output, S_A, in sequence order. S_A is consumed by operator B in sequence order. □

Operator	Scope(p)	Size	Sequential	Scope_overlap
Select	{p}	1	Yes	{}
Project	{p}	1	Yes	{}
Offset(N)	{p}	1	Yes	{}
Compose	{p}	1	Yes	{}
Join	{p}	1	Yes	{}
MovAggr(N)	{p, ..., p-N+1}	N	Yes	{p-1, ..., p-N+1}
Rec. MovAggr(N)	{p}	1	If incr.	{}
Rec. Previous	{p}	1	Yes	{}
Rec. Next	{p, ..., $+\infty$}	1	Yes	{p, ..., $+\infty$}
Expand	{p}	1	Yes	{p}
Collapse	Depends on γ	Var	If ord-pres.	{} if ord-pres.
Cond. Collapse	Var	Var	Yes	{}

Table 2: Scope Properties of $\mathcal{LSEQ}$ Algebra

In light of the importance of stream-access, the Positional operators defined in Table 1 are now characterized by their scopes. Table 2 shows the scopes and scope properties of the different operators. The Map and N-ary operators have fixed-size sequential scopes, independent of their parameters. For the Offset operator, the scope is always of fixed size (one). If the offset parameter is a constant, then the scope is also sequential. Otherwise, it need not be. For example, if the offset is 3 at some positions, and 5 at other positions, the scope is not sequential. For the aggregate operator, the scope is defined by the *agg_window* function. In the definitions of the recursive operators (both with and without state) the scope on the input sequence at p is defined by *Fn_window(p)*; depending on the definition of *Fn_window* in any specific operator, the scope might be fixed-size and sequential. For the "previous", and recursive aggregate operators, this scope is fixed-size and sequential. However, for the "next" operator, the scope is not of fixed size. Since the output of a recursive operator also acts as an input sequence, it is also important to note that the scope over this recursive input is sequential and of fixed size(one). The Collapse operators do not have fixed-size scopes, unless the underlying collapse between ordering domains is totally regular; the Expand operators always have fixed-size(one) scopes. The scopes are sequential only if the underlying collapse is order-preserving. Finally, the Conditional Collapse operators have variable-sized scopes which are sequential.

It appears that Theorem 3.1.1 can guarantee bounded-cache stream-access evaluations for only some of these operators. Luckily, while fixed-size sequential scopes are a sufficient condition for stream-access, they are not a necessary condition. In the next section, some variants are explored that still can guarantee stream access.

3.1.2 More on Scope and Stream-Access

In some cases, "broadening the scope" of an operator can guarantee bounded-cache stream-access.

Definition 3.1.5 A *scope cover* of an operator is a function ScopeCover such that for all positions i, and for all input sequences S_k, Scope(k,i) $\subseteq$ ScopeCover(k,i). $\Box$

Lemma 3.1.1 *If every operator in a query tree has a sequential, fixed-size scope cover on all its inputs, then the query has a bounded-cache stream–access evaluation.*

The proof is similar to that of Theorem 3.1.1. Consider the example of the Offset operator which computes an offset of 3 at some positions, and 5 at other positions. The scope is of fixed size (one), but is not sequential. However, a valid scope cover is $SC(p) \equiv \{p-3, p-4, p-5\}$, which is fixed-size and sequential. As another example, consider a Collapse from a daily sequence to a monthly sequence: the scopes are not of fixed-size (since the number of days in each month varies) but the scopes are certainly sequential. A fixed-size sequential scope cover includes the last 31 input days in the scope for any output month, since no month has more than 31 days; this guarantees stream-access.

This approach still does not deal with operators that have unbounded scope sizes, like the conditional collapse and the "next" operators. One observation that helps is that there are two reasons for caching the input records in the scope of an output position p:

- To actually compute the output record at p.
- To carry over the records in Scope_overlap(p+1).

Assume that while Scope(p) is of variable size, *Scope_overlap(p)* is of a fixed size. Further assume that the output record at p can be computed on the fly while reading the input records. An operator that computed the count of the number of records in the scope would fall into this category, while a count of the number of distinct values in the third attribute of the records would not. Such an "on-the-fly" operator is called an *incremental* operator. In this case, as long as the Scope_overlap is of fixed size, and the scope is sequential, there is a bounded-cache stream access.

Lemma 3.1.2 *A query has a bounded-cache stream–access evaluation if every operator in the query tree satisfies either of the following conditions on each of its input sequences:*

- *The scope is sequential and of fixed size, or*
- *The operator is incremental, and the scope is sequential, and the Scope_overlap is of fixed size.*

The ScopeCover_overlap(p) may be defined based on the definition of ScopeCover(p), just as Scope_overlap(p) is defined based on Scope(p).

Lemma 3.1.3 *A query has a bounded-cache stream–access evaluation if every operator in the query tree satisfies either of the following conditions on each of its input sequences:*

- *There is a scope cover which is sequential and of fixed size, or*
- *The operator is incremental, and there is a sequential scope cover, and the scope cover overlap is of fixed size.*

The Conditional Collapse operator falls in this category, assuming that the aggregate function used is an incremental aggregate like Count, and not a holistic aggregate like Count_Distinct. While the "next" operator does not directly fall into this category (the Scope_overlap is not bounded), the semantics of the operator guarantees that the number of non-empty positions in the Scope_overlap is exactly of size 1! Consequently, the "next" operator also has a bounded-cache stream-access evaluation.

For the rest of this chapter, the assumption is that all the operators in a query fall into one of these categories. It is for this reasonably broad class of queries that query optimization strategies have been developed, and implemented.

Sequence	Schema	Span	Density
IBM	(pos, price, close, volume, comment)	200 .. 500	0.95
DEC	(pos, price, close)	1 .. 350	0.7
HP	(pos, price, close, comment)	1 .. 750	1.0

Table 3: Example Sequence Data

3.2 Query Optimization Techniques

Sequence query optimizations fall into three categories:

- Those that use meta-information in the input sequences to perform global query optimizations.
- Those that heuristically transform the query into an equivalent query that is more efficient to execute.
- Those that choose evaluation plans from a large number of options using a cost-based decision process.

3.2.1 Using Meta-Information

For base sequences, the following kinds of meta–information can be maintained. These are similar to the properties associated with a Time Sequence in [SS87, SK86].

- Start and end positions, which determine the *valid range* or *span* of the sequence. Any position outside of this range is mapped to no record.
- A *density* which specifies the fraction of the positions within the valid range that are non-empty.
- Other statistical information about the base sequences, including the distributions of values in the columns (used to determine the selectivity of predicates) and the distribution of empty positions.
- Available access paths to base sequences, and the costs of access along these paths. Two types of access are considered: stream-access, and *probed* or *indexed* access which retrieves the record at a specific position.

For the purposes of this section, consider the following three daily stock market sequences: the IBM sequence, the DEC sequence and the HP sequence in Table 3.

3.2.2 Global Span Optimization

In Figure 10.A, the query asks for the price of DEC stock when the close of the IBM stock was greater than the close of HP stock. The composition of the IBM and HP sequences followed by a selection condition on their "close" values is condensed into a single "join" operator. Note that the span of DEC is from position 1 to 350, IBM is from 200 to 500 and HP is from 1 to 750. The span of a derived sequence can be computed based on the spans of the input sequences to the operator and the scope of the operator. This query is equivalent to the query in Figure 10.B where the span of the final output sequence is restricted to the period from position 200 to position 350. Similarly, based on the operator scope, if the span of the output sequence is known, the spans of the inputs may be modified, while retaining equivalence to the original query. This leads to Figure 10.C where a smaller range of each base sequence is accessed.

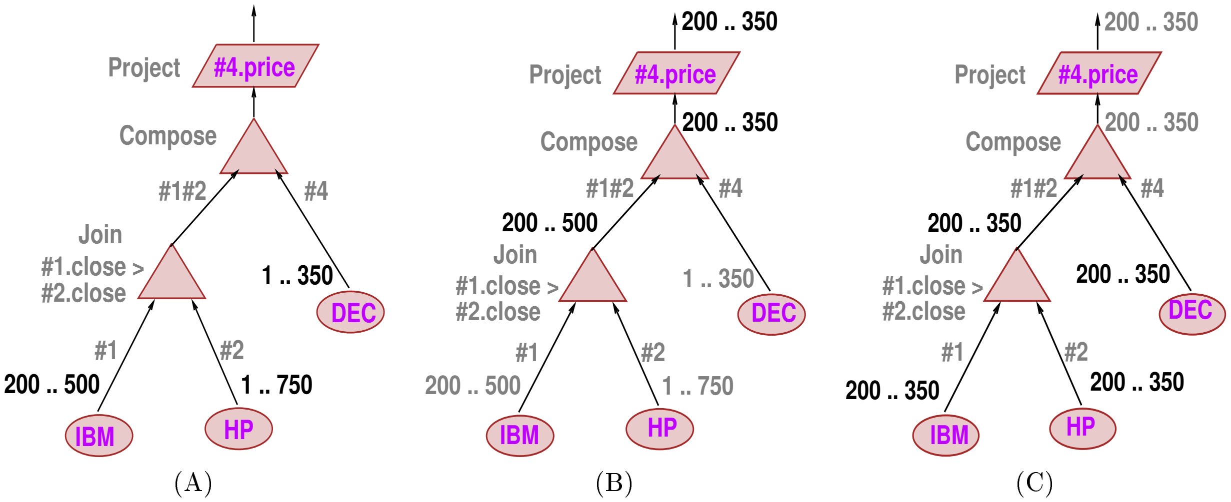

Figure 10: Using Span Information

3.2.3 Query Transformations

Depending on the actual operators used in a query, it may be possible to transform it into an equivalent query that is more efficient to evaluate.

Definition 3.2.1 Two sequence queries Q1 and Q2 are *equivalent* if they both have the same input sequences, and generate identical output sequences for all possible instantiations of the data in the input sequences. □

The transformations developed in this section deal with a narrower definition of equivalence between queries. Remember from the proof of Theorem 3.1.1 that an entire query can be treated as a single complex algebraic operator.

Definition 3.2.2 Two sequence queries Q1 and Q2 are *scope-equivalent* if they both have the same input sequences, the same scopes on the input sequences, and the same operator function. □

Various transformations may be applied to a query tree to generate a query tree that is scope-equivalent. It is typically difficult to reason about the correctness of transformations that affect the entire query tree. However, an important class of transformations acts at the level of individual pairs of operators. The following proposition notes that such local transformations can be used to transform the entire query.

Proposition 3.2.1 *Consider a "local" transformation that modifies a sub-tree of a query, but does not affect the rest of the query tree. If the transformed sub-tree is scope-equivalent to the original sub-tree, then the transformed query is scope-equivalent to the original query.* □

Depending on the particular set of basic operators chosen in a model, various local equivalence transformations may be specified. The transformations listed here (and proved in the appendix) apply to the sample operators presented in Section 2.4.1. Most of the transformations are analogous to well-known relational query transformations. In describing the equivalences, an attribute of an input sequence whose value is used by an operator is said to *participate* in the definition of the output sequence.

1. $Select_{cond1}(Select_{cond2}(\text{A})) \equiv Select_{cond1 \wedge cond2}(\text{A})$.

2. $Project_{attrs1}(Project_{attrs2}(\text{A})) \equiv Project_{attrs1}(\text{A})$, if $attrs1 \subseteq attrs2$.

3. $Project_{attrs}(Select_{cond}(\text{A})) \equiv Select_{cond}(Project_{attrs}(\text{A}))$ if all the attributes that participate in the selection condition $\in attrs$.

4. $Project_{attrs}(\text{Op(A, ...)}) \equiv Project_{attrs}(\text{Op}(Project_{attrs.A}(\text{A}), ...))$ if all the attributes of A that participate in Op $\in attrs.A$.

5. $Select_{cond1 \wedge cond2}(\text{Compose(A, B)}) \equiv Select_{cond2}(\text{Compose}(Select_{cond1}(\text{A}), \text{B})$, if all the attributes that participate in cond1 are from A.

6. Compose(A, B) $\equiv$ Compose(B, A).

7. Compose(A, Compose(B, C)) $\equiv$ Compose(Compose(A, B), C).

8. $Offset_L(\text{Op}(S_1, ..., S_k)) \equiv \text{Op}(Offset_L(S_1), ..., Offset_L(S_k))$ if the following conditions hold. The output sequence of Op and the input sequences $S_1, ..., S_k$ have the same ordering domain, and $\forall k, \forall p$, Op.Scope(k, p) $\equiv \{C_{k1}+\text{p}, .., C_{kn}+\text{p}\}$, where each value C_{kj} is a constant independent of p, and Op.OpFunc is independent of p.

Two successive projections or two successive selections can be combined, and selections and projections may commute. The first three rules above express these transformations. Rule 4 says that a projection can be "pushed through" any operator as long as the attributes that participate in the operator remain after the projection. Rule 5 shows how a selection can be pushed through a compose operator (just as relational selections can be pushed through relational joins). Since compose operators are similar to relational joins, they have the commutative and associative properties shown in Rules 6 and 7. Finally, in Rule 8, offset operators (which merely phase-shift a sequence) can be pushed through any operators that are unaffected by relative shifts in the sequence mapping.

There are few transformations involving operators of non-unit scope like an aggregate or a recursive operator. This is because there is no one-to-one mapping between input records and output records[2]. The non-unit scope operators therefore break up the query into a tree of *query blocks*, with the output sequence of a block feeding into the input of another block. The operators of non-unit scope form singleton blocks, while the unit-scope blocks contain compose, select, project and offset operators. The compose operators can be reordered according to Rules 6 and 7, and since there are a number of possible orders (exponential in the number of sequences being composed) and different evaluation methods, optimization techniques based on costs and statistics should be used. The next section examines the compose operator in greater detail.

3.2.4 Compose Operator and Access Modes

Consider a Compose operator with two input sequences as in Figure 11. The query finds the DEC data on days when the close of IBM was greater than 25. In order to generate the output sequence, one option is to retrieve each record of the DEC sequence, and search for the corresponding record from the other input sequence (labeled #1 in the figure). Another possibility is to retrieve each record in sequence #1 first, and to then look up the corresponding record from the DEC sequence. These represent two variants of an *Indexed Compose* method which streams one input sequence and probes the other. The *Streamed Compose* method streams both sequences in sequence order in lock-step, performing the composition at common positions. These methods are analogs of the nested-loops and sort-merge relational join methods. A hybrid of the two methods is also possible. Consider a Streamed Compose method in which sequences DEC and #1 are being streamed in lock-step. When a new DEC record is read that

[2]Exceptions to this occur when a Max (conversely, Min) window aggregate is followed by a selection that checks if the max value is greater (less) than a constant. In these special cases, the selection may be commuted with the aggregate operator.

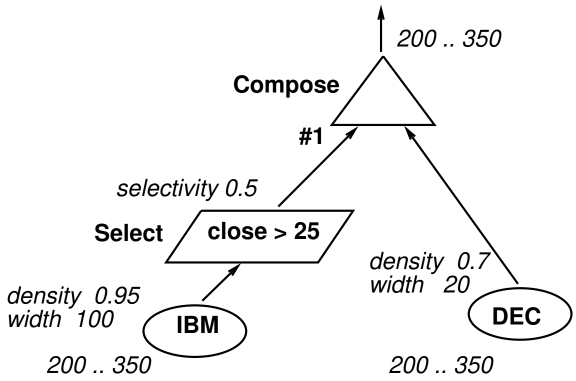

Figure 11: Compose Access Modes

is ahead of the latest #1 record, the straightforward technique is to keep reading the #1 sequence till the position catches up with that of the DEC sequence. However, there might be a large number of records read unnecessarily until this happens. Instead, it is possible to "jump-ahead" in the scan of the #1 sequence to the appropriate position by pushing the current position of DEC down into the computation of #1. The logic for this is similar to the propagation of valid ranges or selection predicates on the position attribute. Such a Compose method is a hybrid, because while it is still Streamed, the positions retrieved on one stream are used to "index" the other stream.

Access mode refers to the manner of access to a sequence. While this chapter has concentrated so far on stream-access, it is not necessary that a stream access evaluation implies an optimal evaluation. The basic operation for a stream access is to get the record at the *next* non-empty position. This is as opposed to the basic operation for an indexed access, which is to get the record at a specific position. While the per-record access cost of a stream access is usually significantly less than that of an indexed access, this may not be the case if the data is not physically organized to favor stream access (for example, if it is not physically sorted in the logical sequence order). As another example to suggest where indexed access might be useful, there may be very few IBM records whose 'close' attribute is greater than 25. Consequently, it may be preferable to stream the #1 sequence, and use an indexed access to the DEC sequence for the few positions that are of interest. Finally, the result of the query may only be desired at a single position or a small set of positions. An indexed access to these positions may be preferred over a stream-access to the entire sequence. The next section describes how suitable access modes and operator methods may be chosen as part of the overall optimization of the sequence query.

3.3 Query Plan Generation

This section presents an algorithm for the optimization and evaluation of sequence queries. A (derived or base) sequence may be queried by asking for its records at (a) specific positions, or (b) all positions in a range. In both cases, the query is associated with a "position" sequence that has records at those positions for which an answer is desired. Figure 12 shows a query specification. The sequence being queried is labeled SQT(Sequence Query Tree) to signify that this is a derived sequence represented by a query tree. The *Start* operator is a special operator that initiates query evaluation by invoking a stream access on its input. The query optimizer needs to produce a query evaluation plan that will generate the output sequence.

The query optimization algorithm explores the space of possible query evaluation plans by performing the following steps:

Step 1 - Query Specification: The query is specified as a Sequence Query Tree (SQT) in the query template. The

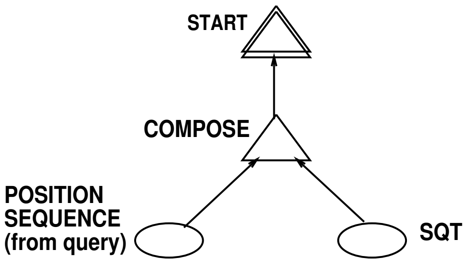

Figure 12: Sequence Query Template

Query Position Sequence is also specified as part of the template. If the input sequences or any of the operators are themselves represented by SQTs, they are expanded out. This is similar to the policy of expanding view definitions used in relational database systems [Kim82]. At this stage, the query is a hierarchical tree of operators, with base sequences at the leaves of the tree and the Start operator at the root.

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

Step 2.a - Bottom Up Annotation: The query tree is adorned with schema and meta–information in a bottom-up traversal. The purpose of this step is to perform type checking of the query and to define the meta–information associated with the derived sequences in the query. The span information from the input sequences of every operator is used to determine the span of its output sequence. Further, the densities of the input sequences are used to determine the density of the output sequence. The semantics of each operator plays a part in the density propagation. Given the definition of aggregate operators, an empty output position occurs only if all the input positions in the scope are empty. Similarly, for a Compose operator, the density of the output sequence is a function of the input sequence densities, the selectivity of the join predicates and the correlation in the empty positions of the input sequences.

Step 2.b - Top Down Annotation: This step propagates span information down the tree using a top-down traversal starting from the root. The spans of the input sequences of each operator are modified based on the span of its output sequence.

Step 3 - Query Transformations: Various heuristic transformations are applied to the query tree, based on the equivalence rules presented in Section 3.2.3. For example, it is a good heuristic to propagate selections, projections and offsets as far down the query tree as possible.

Step 4 - Identification of Query Blocks: As described in Section 3.2.3, the operators with non-unit scope divide the query into "blocks". At this stage, these query blocks are identified and ordered in a partial ordering as follows: if the output sequence of a query block A is an input for another block B, then $A < B$ in the block ordering.

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

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

The approach taken to generate a plan for each block is similar in spirit to the plan generation algorithm for SQL query blocks described in [SAC$^+$79], which is referred to as the Selinger algorithm. A brief explanation of the Selinger algorithm is in order. An SQL query block consists of a list of relations to join, a list of selection predicates on the relations, a list of join predicates and a list of projection attributes. The primary emphasis is on choosing an efficient order to join the relations involved in the query. While evaluating the cost of each join, different join methods are considered. At the first stage, the cheapest way of accessing each individual relation is determined for each "interesting" order. Interesting orders are sort orderings on columns that might benefit Group-By or Order-By operators at higher levels, or that might help in a subsequent sort-merge join. At the next stage, one additional relation is joined to the existing relations to produce all joined pairs of relations. If there are multiple methods for computing a particular join, only the most efficient method is retained for each interesting order. This process is repeated until a join order has been chosen for all relations in the query.

The approach to sequence query optimization is similar, but differs in the details. As explained in Section 3.2.3, each query block is either an operator of non-unit scope, or consists of a set of compose operators on input sequences. The input sequences may be base sequences, or the outputs from lower blocks. The order of evaluation of the positional joins and the method used to evaluate each of them remains important to determine. However, issues like access modes need to be introduced into the algorithm. Further, the ordering domain of the sequences provides an implicit interesting order for the query as a whole, and so the optimizer does not need to worry about interesting orders. The plan generation phase for each block provides evaluation plans and cost estimates for the output sequence of the block accessed in both stream and indexed modes. This information is then used to generate plans for the higher blocks.

The algorithm for processing a single query block is now presented, along with sample cost formulas that can be used to determine the costs of various plans. The specific cost formulas used depend on the system environment.

Access Costs to Base Sequences

The stream-access cost of a base sequence is determined by the size of the valid range of the sequence, the density of the sequence and the access paths available. The cost is measured as a product of the number of pages to be accessed and the cost of each access via a particular access path. The indexed-access cost is the product of the number of positions in the valid range of the sequence and the average cost of accessing one record at a given position via whatever indexed access mechanism is available.

Blocks with Compose operators

Consider the Compose of two sequences S1[A1,a1,d1] and S2[A2,a2,d2], where A1(A2) corresponds to the stream access cost, a1(a2) to the indexed access cost and d1(d2) to the density of sequence S1(S2). The costs of the plans for the output sequence are evaluated as follows:

- Stream Access Cost $\equiv minimum(A1 + d1 * a2, A2 + d2 * a1, A1 + A2) + d1 * d2 * output\_span * K$, where the value K is the constant cost associated with a single application of the "join" predicates on the two records being composed. The value $d1 * d2 * output\_span$ is the number of times the predicates need to be applied, assuming a uniform distribution of the empty positions in each sequence.

- Indexed Access Cost $\equiv minimum(a1 + d1 * a2, a2 + d2 * a1) + d1 * d2 * output\_span * K$.

Plan generation for other operators like projections and selections is straightforward, and are not explained here. As in the Selinger join optimization algorithm, the number of sequences composed is increased in steps of one until finally all the input sequences to the block have been joined and there are two access plans generated corresponding to the cheapest ways of accessing the output sequence in stream and in indexed modes. The pseudo-code for this optimization technique is shown in Figure 13. This is a dynamic programming algorithm that searches an exponential space of options. In practice, it might be acceptable to search a smaller space of options (for example, a polynomial space, as suggested by the KBZ join optimization algorithm [KBZ86]).

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

Figure 13: Block-wise Plan Generation

Blocks with Non-Unit Scope

The $\mathcal{LSEQ}$ model considered several types of operators with non-unit scope: aggregates, recursive operators, collapse operators, etc.

For a fixed-size scope operator like an aggregate, stream-access is possible if the entire scope is cached. The cost is the stream-access cost for the input sequence, plus the cost of storing and removing each record from the cache, plus the cost of accessing the cache for each output record, plus the computational cost. As described in Section 2.2.1, some moving-window aggregates may also be modeled as recursive operators. If this option is explored, then there is the additional cost of storing the most recent output record, and manipulating it by "dropping" old input values from the computed values. However, the computational cost may be lowered. As is shown in Section 4.4.3, each of these approaches to aggregates can be the best choice in certain situations. For operators with unlimited scopes (like "previous", decay, etc.), caching the entire scope is not an option, and the recursive method is the only suitable choice.

For indexed access, some reasonable estimate needs to be made of the number of input positions from the scope that will have to be accessed on average to satisfy each indexed request. The indexed access cost is the indexed access cost of the input sequence multiplied by this estimate, which can be based on the density of the input sequence.

Algorithmic Analysis

The Selinger algorithm has been extensively studied and is known to generate the class of "left-deep tree" join plans. [GHK92] contains a detailed analysis of this algorithm. The optimization algorithm presented here also explores the class of left-deep query trees within each block. The entire query evaluation plan however is not restricted to be a left-deep tree because the tree may be bushy *across* query blocks. The execution of the optimization algorithm for a block with N compose operators has:

Property 3.3.1

a) A time complexity in terms of the number of plans considered $\equiv O\left(N * 2^{N-1}\right)$

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

3.4 Common Sub-Expression Optimization

This chapter on query optimization has so far assumed that a query is a *tree* of operators. A more general model permits a query to be a *acyclic graph* of operators; there could be an operator or a base sequence that acts as the input for more than one operator. Each such shared use of a sequence forms a *common subexpression* (CSE) in the query graph. This section examines the impact of common subexpressions on the query optimization mechanisms. It should be emphasized that common subexpressions are not rare, especially when base sequences are considered as well. Sequence data is often available as a single sequence of recordings, out of which several derived sequences are created within a query, before they get finally put together to form the query result.

There are two straightforward mechanisms used in relational query processing to deal with a common subexpression. One is to duplicate the shared portion of the query graph, effectively converting the graph to a tree. The effect, however, is that there is duplication in the work performed in query evaluation. Another option that can be used for CSE operators is to compute the shared portion of the query graph once, store the results, and then access the stored data as many times as is needed. This is better than the previous approach only when the amount of computational work performed in the shared portion of the graph is greater than the cost of writing out the temporary sequence and reading it back again. There has been research that describes how common subexpressions may be identified and used in this manner in several different environments [Sel88, GM81]. In a sequence query, however, there is a possible third option that is more efficient. The next section describes how a common subexpression can be computed only once in a stream access fashion, without writing out its result as a temporary sequence.

3.4.1 Stream Access and CSE

Consider a shared node in the query graph, which acts as input to several other operators. Some of these accesses are stream accesses and some are indexed accesses (query optimization has been performed just as described earlier). For indexed accesses, the access patterns are not predictable (i.e., only the occasional position is asked for). Consequently, these accesses might as well be treated as independent, and the shared portion of the query graph is duplicated.

For the stream accesses however, the access patterns are predictable; the entire shared derived sequence must be computed (at least over its valid range). The goal is to coordinate the stream accesses to the shared CSE so that it need only be computed once, and some finite number of the output records can be cached in main-memory. The problem is that when there are multiple stream-accesses on the shared operator during query execution, *the different accesses might be reading different portions of the sequence*! This arises because some of the users of the shared sequence (and their parents) can possess non-unit scopes. One way to characterize this effect is by asking: "in order to generate the output record at position p, what are the positions in the CSE sequence that are required?" Assume that there are two operators that use the CSE sequence, and that the positions needed by each user are the ranges $[p1_1, ..., pn_1]$ where $p1_1 > pn_1$, and $[p1_2, ..., pm_2]$ where $p1_2 > pm_2$ respectively. By the usual stream access mechanism, each of the users will ensure that they maintain a cache of the size of its required scope range. Without loss of generality, assume that $p1_1 > p1_2$, i.e. user 1 "leads", and user 2 "trails". There are two cases to consider:

- If $p1_2 \geq pn_1$, then the scope of the leading user overlaps the scope of the trailing user. Consequently, when stream access is requested by the trailing user, "new" records can be retrieved from the *cache of the leading user.*
- If $p1_2 < pn_1$, then the scopes of the two users do not overlap. However, consider the effect of enlarging the cache of the leading user by $pn_1 - p1_2$ records. This strategy essentially converts this case to the previous case, and once again, a single pass stream-access evaluation of the CSE is possible.

The critical information in performing this optimization is knowledge of the range of positions accessed by each of the users of the CSE. This information is implicit in the scope of every operator. For each operator, let Scope_Lead(k, p) be the difference between p and the maximum position in Scope(k,p). Here is the algorithm to identify what (if any) modification to the stream-access caches need to be made for the stream-access evaluation of a CSE.

- Find V_{CSE} which is the (unique) cut vertex operator for the CSE operator.
- Let there be $M \geq 2$ unique paths $P_1, ..., P_M$ from V_{CSE} to the CSE in the query graph.
- For each path P_i, let $Lead_i \equiv \Sigma(Scope\_Lead\ at\ each\ node\ along\ P_i)$. This sum represents the total scope lead along a path.
- Let $Lead_{max}$ be the maximum value of $Lead_i$ for all paths P_i. $Lead_{min}$ is determined similarly. The difference $(Lead_{max} - Lead_{min})$ represents the maximum difference in the stream access positions at the CSE. Once this has been identified, it is possible to check to see which of the above two cases apply. $Lead_{max}$ corresponds to $p1_1$ and $Lead_{min}$ corresponds to $p1_2$.

Note that if one of the paths involves a recursive operator, there are effectively an infinite number of distinct paths (corresponding to loops through the recursive operator) from V_{CSE} to the CSE. Consequently, shared stream-access to the CSE operator is not possible. Intuitively, this is because the presence of a recursive operator means that it might have to go arbitrarily far back in its input sequence.

3.4.2 Other Changes Due to CSEs

Heuristic transformations typically do not occur across a CSE boundary. For example, it is difficult to push a selection through a CSE operator. This is because the same selection would have to be present at all the users of the CSE operator. In cases where heuristics suggest that it is very important to perform a transformation, the CSE can be duplicated, and the transformation can then be performed.

The span propagation algorithm must be modified in the presence of CSEs. The upward propagation phase remains unaltered, but in the downward propagation phase, several operators may try to limit the span of their shared input sequence. In this case, the span should be limited to the union of the spans of the individual operators.

One issue that was overlooked during the previous section was how the issue of shared CSEs interacts with cost-based optimization. The basic optimization technique was founded on a knowledge of the costs of accesses to input sequences, and the use of these costs to find cheap plans at higher levels. However, with CSEs, the decision of whether or not to use stream access, which is made at the higher levels, affects the number of stream accesses to the shared CSE. This in turn affects the actual cost of each stream access, which is really the cost of a stream access amortized over the number of users leveraging off that same access. A thorough solution to this problem requires the generation of a large number of plans (exponential in the number of users of the CSE), one for every combination of shared and unshared accesses. This is impractical, and therefore the alternative approximate solution is to perform the cost-based optimization as though there were no sharing of the CSE, and introduce the shared access after the final plan is generated.

3.5 Scope and Partitioned Parallelism

While the focus of this work has not been on parallel query execution, it turns out that the scope of an operator is an important property that helps to parallelize a sequence query operator in a "shared-nothing" environment. A well-established mechanism to achieve intra-operator parallelism in a database query is to partition the input data across all the parallel nodes, and to run instances of the operator in parallel on the individual partitions. For example, this is a standard evaluation strategy for a parallel relational join.

In a sequence query, partitioning cannot be performed as cleanly. Since the operators have sequential scopes, hash-partitioning (which randomly distributes adjacent records) is not applicable to operators of non-unit scope. Range-partitioning (which sends a range of consecutive records to the same site) is a better choice. However, input records at the partition boundaries may be required for processing at the adjacent site as well. The operator scope specifies exactly how many such records are needed at the adjacent site, and these boundary records need to be replicated at that site as well. Such an approach has been studied in the specific context of a "band-join" [DNS91], and in the context of a "temporal-join" [LM92]. However, the characterization of all operators by their scope provides a general framework in which to parallelize all sequence queries.

Recursive operators present a challenge to this form of partitioned parallelism, because their definition is essentially sequential. However, techniques like "parallel-prefix" computation [HS86] could be used to execute such operators.

Chapter 4

A Sequence Database System

This chapter discusses the implementation of the SEQ sequence database system, a component of the larger PREDATOR DBMS which provides support for several kinds of complex data. For the purposes of this chapter, the presence of the other types of data and the integration between them is ignored, and the focus is on the support for sequences. However, one point to be noted is that the basic type system used in the DBMS is extensible, and so the attributes of the sequence records can belong to any abstract data type registered with the system.

SEQ provides the $\mathcal{SEQUIN}$ language to specify declarative queries over sequences, and an optimization and execution engine to process the queries. The effects of several design choices and optimization techniques are demonstrated via experiments on a synthetic sequence database. Much of this material was presented in [SLR96b].

The system is described in four stages:

- System design goals.
- Implementation of a sequence.
- The $\mathcal{SEQUIN}$ query language.
- Performance demonstration of optimization techniques.

4.1 Design Goals

The goals of the SEQ system can be described in terms of the kind of data and manipulations supported.

Kind of Sequence Data: The most common sequence 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. It is desirable to efficiently support all these kinds of sequence data. Further, these sequences are typically large and stored on disk.

Kind of Sequence Queries: The system should efficiently process sequence queries that are capable of stream-access evaluation. While supporting queries is the primary concern, the implementation should also permit future extensions to handle updates to sequence data.

In terms of the overall implementation, system modularity and extensibility is emphasized. This is reflected in the design of several components of the system.

4.2 Sequence Implementation

The sequence implementation is based on the $\mathcal{LSEQ}$ data model. An important component of the model of a sequence is the ordering domain. Each ordering domain is modeled as a data type with some additional methods that make it an *ordered* type. *LessThan(Pos1, Pos2), Equal(Pos1, Pos2)* and *GreaterThan(Pos1, Pos2)* allow comparisons to be made among positions. *NumPositions(Pos1, Pos2)* counts the number of positions between the two specified end points. *Next(StartPos, N)* and *Prev(StartPos, N)* compute the Nth successor and predecessor of the starting position. All ordering domains are registered in an extensible table. Additionally, the hierarchical relationship between various ordering domains needs to be captured. A table of *Collapses* is maintained, where 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.

In each sequence, the position mapping is maintained as an explicit attribute of each record. 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 stream-access of the sequence in the sequence order. Any positions in the domain that are not mapped to a record are ignored in the scan.

- *GetElem(Pos).* This acts as an indexed access and finds the record at the specified position in the sequence (or fails if none exists at that position).

In the next section, the goal is to describe some possible choices of storage techniques for sequences, and to quantitatively demonstrate the differences between them. It is convenient to develop a benchmark database for this purpose, so that the same data can later be used to demonstrate the performance impact of various optimization techniques.

4.2.1 Synthetic Benchmark Database

The sequences in the database were generated synthetically; while a real-life data set could have been used 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). Sequences with two different densities were considered: 100% and 20%. The cardinality of each sequence was either 1000 (1K), 10000(10K) or 100000(100K) records. For sequences of each density, the final time-points are shown in Table 4; the entries in the table are approximate since they only show the last day, not the last hour.

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

Table 4: 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, which is textually represented in the format 'yyyy-mm-dd hh:mm:ss'. Different sequences were generated with 1, 5, 10 and 20 fields in addition to the timestamp. The values in the fields were 4-byte integers generated randomly between 0 and 1000. All experiments were performed on a SUN-Sparc 10 workstation equipped with 24MB of physical memory. In all the experiments, the queries contain a final count aggregate over the entire sequence, thereby minimizing any time spent in printing answers. Each query was executed four times in succession, the maximum and minimum execution times were excluded, and the average of the other two times was used as the performance metric.

4.2.2 Storage Implementation

SEQ supports two repositories for sequence data, the Unix file system and the SHORE storage manager [CDF+93], with the ability to add further repositories as well.

The 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; the experiments used the latter approach. Three alternative storage methods have been implemented 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 allows a sequence to be implemented as a SHORE file. One advantage is that coding this implementation requires minimal effort. The major drawback is that the storage manager imposes several bytes (at least 24) of space overhead for every record, in addition to a large space overhead for creating a file. While concurrency control is available at the record level, inserts in the middle of a sequence are difficult to implement without a separate index.

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

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 many sequences are irregular (i.e., have empty positions), a compressed array representation is used, in which no space is wasted for an empty position. While this dramatically reduces space utilization for data sets of very low density, some operations within a sequence (like positional lookup, insert and delete) become more expensive to implement. Variable length records require additional complex code (and the current implementation of array sequences only handles fixed size records). However, there are two important benefits to this implementation: the per-record space overhead is minimal and there is physical sequentiality for the records of a sequence. With fixed-size records in a mostly-query environment, this should be the implementation of choice.

While the default repository is on top of SHORE, a sequence can instead be stored as an ascii file on the Unix file system. Much real-world sequence data currently exists in this format, and it may be more expedient to directly run queries off this data, instead of first loading it into the database. Of course, the file system repository does not provide any of the database properties of concurrency control, recovery, etc.

Experiment 4.1 This experiment measured the time taken to scan and count the number of records in each of the example sequences stored using each of the four implementation techniques. Since only the count is returned, this measured time does not include the time for printing an entire answer sequence. A scan is the most basic sequence

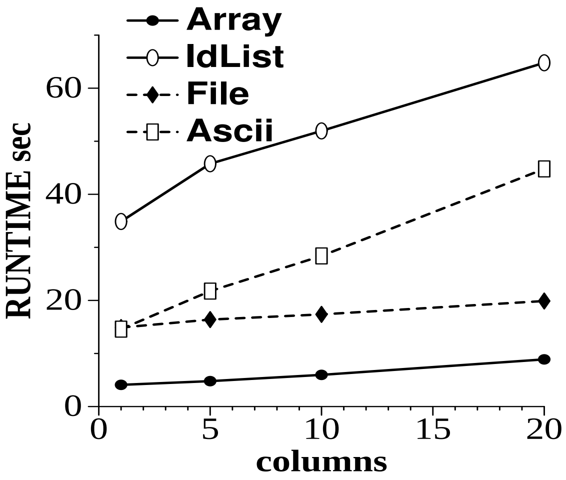

Figure 14: Storage Methods : Card 100K

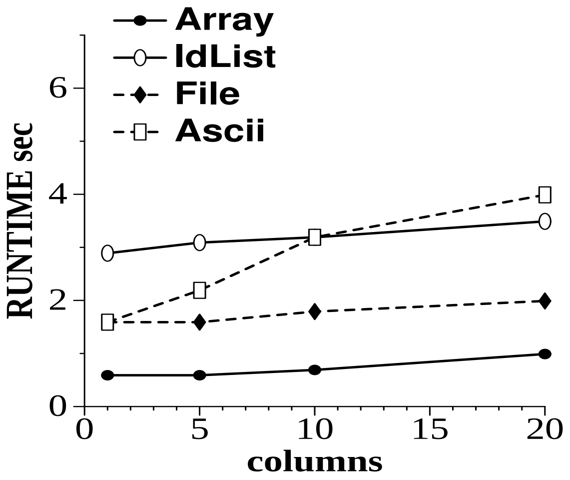

Figure 15: Storage Methods : Card 10K

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).

Figures 14 and 15 show the results for the sequences of cardinality 100K and 10K respectively. In both 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. The SHORE File implementation is worse than SHORE Array because of the file handling overhead per record. IdList is the worst SHORE implementation primarily because of the added cost of converting from logical to physical identifiers. The Unix ascii file implementation is the most sensitive to the width of the data records because each attribute needs to be parsed at run-time to convert it from ascii to binary format. Indeed, if binary data were stored in a Unix file, it would probably be the most efficient to scan. □

The purpose of this experiment is to determine what should be the default storage implementation for sequences. The results indicate that the SHORE Array should be the default, and this is the implementation used for all the remaining experiments. The data is 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 were turned off to model a query-only environment.

4.3 $\mathcal{SEQUIN}$ query language

The $\mathcal{SEQUIN}$ [1] query language is the interface through which queries are posed to the system. The language syntax is similar in flavor to SQL, an intentional decision aimed at making the language easy for SQL users to adopt. It is based on the $\mathcal{LSEQ}$ algebra of Positional operators presented in the earlier chapters, and there is a well-defined translation from any $\mathcal{SEQUIN}$ query to a sequence query graph. As always, the result of a query is an answer sequence. Except where noted, the constructs in $\mathcal{SEQUIN}$ all result in queries that have stream-access evaluations.

The overall structure of a $\mathcal{SEQUIN}$ query is:

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

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

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

Example 4.3.1 The first query estimates the monetary value of Stock1 traded in each hour in which the low price was 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; 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. There is no output record for positions at which the WHERE clause condition fails; these are empty positions in the output sequence. The sequence in the FROM clause can be associated with a correlation name ("A" in this case) for convenience. 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. □

Example 4.3.2 Now consider finding the 24-hour moving average of the difference between the high prices of the two stocks.

```
PROJECT avg(A.high - B.high)        // 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'). 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 be of the form $\$P + / - < constant\_integer >$, where the special \$P symbol represents 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. There is an extensible table of aggregate functions in the system, and any of these may be used instead of *avg*. □

Example 4.3.3 This example shows a rather complex query to demonstrate 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[2].

[2]The value 200 is used here for simplicity. In reality, hourly times are typically represented using a longer time-stamp format.

```
// 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          // 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 maps every position to the record at the most recent non-empty position in the input. Similarly, *Next* defines a sequence in which the empty positions 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 queries like this one where the position attribute is implicitly added to the FROM clause sequences, and hence does not have a field name. □

Example 4.3.4 This query demonstrates the use of the ZOOM clause to exploit the hierarchical relationship between ordering domains. The word "zoom" is used because the action of moving down or up through the ordering hierarchy is similar to zooming in and out with a lens. 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.
```

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. □

Example 4.3.5 The final example shows how the ZOOM clause can perform conditional collapses. Suppose that just as in the previous query, one wants 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). □

The section has omitted discussion of some other features of $\mathcal{SEQUIN}$ including DDL features, a construct to re-define the ordering field of a sequence, mechanisms to cast relational data to a sequence and vice-versa, and update constructs. Some of these features have yet to be implemented as well in SEQ.

4.3.1 Generating a Query Graph

A $\mathcal{SEQUIN}$ query can be parsed into a directed acyclic graph of algebraic operators. Base sequences in the FROM clause form the leaves of the graph. The Previous, Next and Offset modifiers in the FROM clause are represented by their corresponding operators. All the sequences in the FROM clause act as inputs to a N-way Compose operator. If there is a nested query block in the FROM clause, it is recursively parsed to form a query graph, and this query graph is treated as an input of the Compose operator. This operator acts as the input to a Selection operator that applies the predicates in the WHERE clause. If there is no OVER or ZOOM clause, the PROJECT clause expressions are part of a Projection operator, which is the result of the query. If there is an OVER clause, the PROJECT clause expression along with the OVER clause window is represented by an Aggregate operator. If there is a ZOOM clause instead, the PROJECT clause expression together with the ZOOM clause is represented by a Collapse or Expand operator.

4.4 Demonstration of Optimizations

Several of the optimization techniques from Chapter 3 have been implemented in SEQ; costs of disk accesses and various operations are yet to be modeled. This section describes the effects of five categories of implemented optimizations. The queries in the experiments have been kept as simple as possible, in order to isolate the effects of each optimization.

4.4.1 Range Propagation

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 4.2 demonstrates the benefits of propagating such selections into the sequence scans. The other source is from statistics on the valid ranges of positions in each sequence. These valid ranges can be propagated through the entire query as described in Section 3.2.2. Experiment 4.3 demonstrates how the valid-range can be used for optimization.

Experiment 4.2 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.

```
PROJECT count(*)               // Aggregate to reduce answer
FROM 100K_10flds_100dens S     // 10 fields,100K records,100% density
WHERE S.time > "<timestamp1>"  // Selection on the order attribute.
ZOOM ALL;                      // Perform aggregate over entire 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.

The effect of varying the predicate selectivity from 1% to 100% was studied. The experiment was run 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).

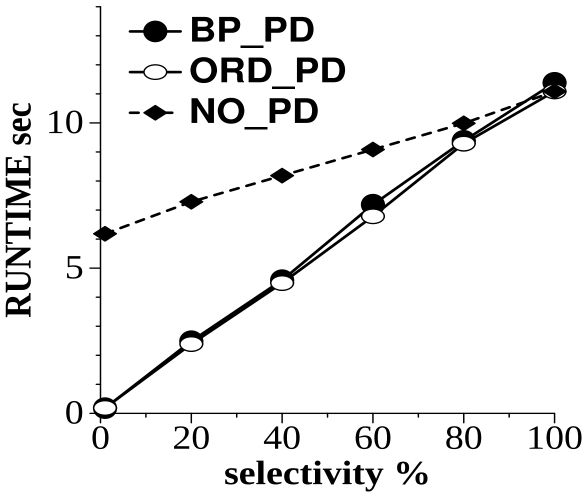

Figure 16: Range Selections At Start

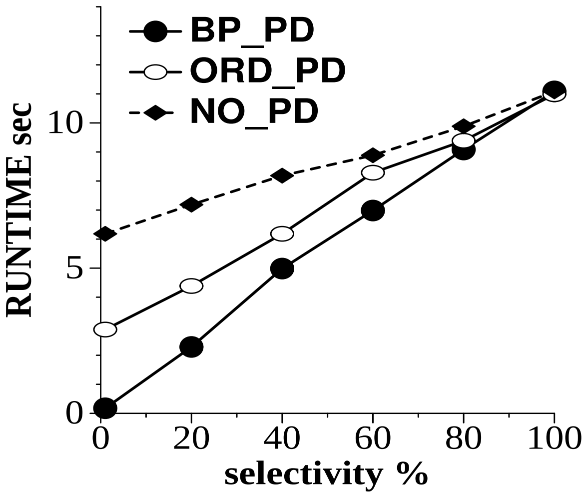

Figure 17: Range Selections At End

The results for **at_start** are shown in Figure 16. The predicate selectivity is shown on the X-axis, and the query execution time is on the Y-axis. While there is no difference between BP_PD and ORD_PD (since the predicate is at the start of the window), NO_PD performs much worse because the entire sequence is scanned. As the selectivity increases, all the algorithms become more expensive because there is additional work being done in the final count aggregate. The slopes of the NO_PD line indicates the increase in the amount of work done in the count aggregate. The slopes of the BP_PD and ORD_PD lines indicate the extra data being scanned in addition to the extra work being done in the count aggregate.

The results for **at_end** are shown in Figure 17. 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 4.3 This query joins two sequences on position; however, one of the sequences is first shifted by some specified number of positions. Each of the base sequences in this query has 100K records spanning an identical range (see Table 4). 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 18.

```
// 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 B with the ViewSeq shifted by a specified offset
PROJECT count(*)
FROM 100K_5flds_100dens B, Offset(ViewSeq, <offset_distance>) C;
```

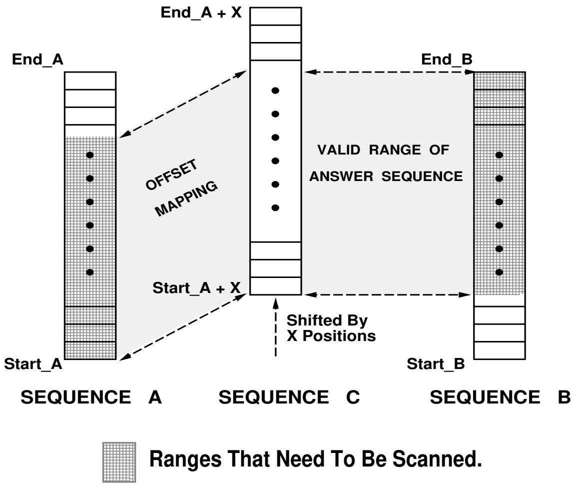

Figure 18: Range Propagation: Intuition

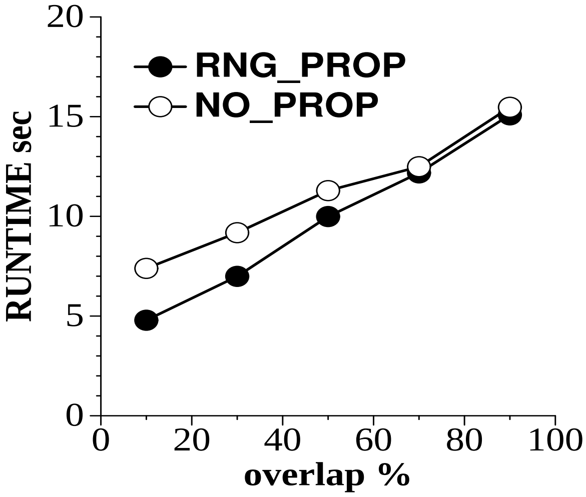

Figure 19: Range Propagation: Results

The overlap was varied from 90% of the valid-range to 10% of the valid-range, and the query was executed with (RNG_PROP) and without (NO_PROP) the valid-range propagation. The results are shown in Figure 19. 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. □

4.4.2 Compose Optimizations

The optimization of the Compose operator is of some significance, as described in Section 3.2.4. There are two primary choices to be made: the order in which each of the inputs to the Compose operator is accessed, and their access modes. Experiment 4.4 demonstrates the differences in performance that can result from various specific choices.

Experiment 4.4 This query involves the composition of two sequences, one of which has a selection condition on it. The selectivity of the condition is varied by modifying the constant in the inequality. As the selectivity changes, the density of the S2 sequence changes, and different choices become appropriate for the Compose operator combining S1 and S2. The performance of three evaluation strategies was measured: STR is stream-access to S1 and S2, HYB is hybrid (i.e., indexed-stream) access to S1 and S2, and HYB-NOORD is hybrid access without reordering the input sequences based on their densities. The results are shown in Figure 20. The selectivity of the predicate varies along the X-axis, while the query execution time is on the Y-axis.

```
PROJECT count(*)                // Aggregate to reduce answer
FROM 100K_5flds_100dens S1,     // Input without selection
     (PROJECT *                 // Input with selection
      FROM 100K_10flds_100dens S
      WHERE S2.fld1 > ``constant'') S2
ZOOM ALL;                       // Perform aggr over entire sequence.
```

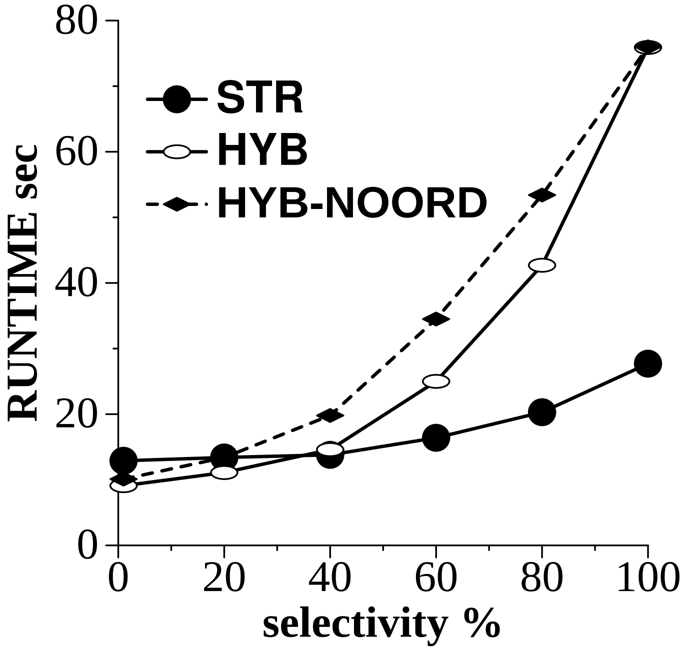

Figure 20: Compose Optimization: Results

As expected, STR is the most stable approach, because it must always scan the entire two sequences. Its cost grows only slowly as the selectivity changes, since there is more work performed in the final count aggregate. HYB is a better approach at lower selectivities, since there is significant benefit to propagating the current position of the less dense sequence (the one with the condition) to the other composed sequence. However, as the selectivity crosses 10%, the cost of performing the "jump-ahead" indexing outweighs the savings in data scanned, and performance is worse than for STR. HYB-NOORD is worse than HYB because the denser sequence (without the condition) is treated as the first input sequence. Consequently, there are more propagations of the current position down the less dense sequence; many of these are not useful, but they increase the query execution time. At 100% selectivity, there is no difference between HYB and HYB-NOORD, because both do exactly the same amount of propagation of current position. For both strategies, the "indexing" is a wasted overhead because there is never any input record that can be skipped. Consequently, there is a major difference between STR and the hybrid strategies. This is similar to the differences between sort-merge join and indexed nested loops join in the RDBMS world. The conclusion of this experiment is that, depending on the densities of the sequences involved, there can be significant performance differences between the various evaluation strategies for the Compose operator. □

4.4.3 Moving Window Aggregates

All aggregate functions in SEQ 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[3]. This is called the *symmetry* property of an aggregate function. In order to exploit the symmetry property in an extensible manner, SEQ requires each aggregate function to provide two more methods: *IsSymmetric()* and *Drop(record)*. Experiment 4.5 demonstrates the importance of exploiting symmetric aggregates.

Experiment 4.5 Consider 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. The example query is the simplest form of a moving aggregate (with a final count operator as usual to eliminate the time for printing answers). This experiment was restricted to only the 100K_10cols_100dens and 100K_10cols_20dens sequences. The window size was varied from 5 to 100, while the aggregate function AVG was used in symmetric (AVG_SYM) and non-symmetric(AVG_NOSYM) modes.

The results for the 100% density sequence are shown in Figure 21. Notice that the performance of AVG_NOSYM grows linearly with the size of the aggregation window. This is because the entire aggregation window has to be processed for each AVG aggregate computed. In comparison, the performance of AVG_SYM is almost independent of the size of the aggregation window. The slight dependence of AVG_SYM on the window size has an interesting reason. Given a particular timestamp, it is more expensive to compute the 100th previous timestamp, than the 10th previous timestamp. Simple arithmetic cannot be applied to temporal ordering domains because the variable number

[3]It is possible to model these aggregates similarly by maintaining tree data structures that allow efficient recomputation when individual records are dropped and added. However, this is a lot more complicated to code and evaluate than aggregates like Count, etc.

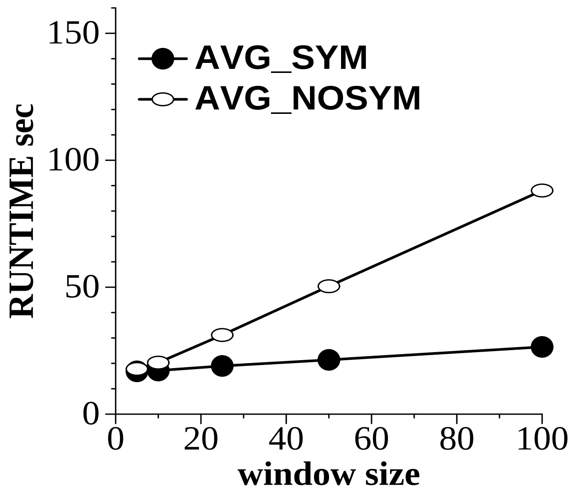

Figure 21: Aggregates: 100% Density

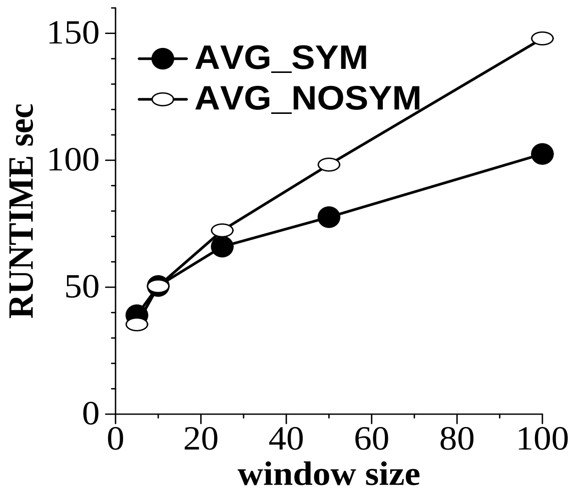

Figure 22: Aggregates: 20% Density

of days in a month has to be accounted for. Further, since all interaction with the ordering domains occurs through virtual member functions, the overhead for window computation becomes more significant. The 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 22. Note that a moving aggregate over a sequence with holes generates many more records than exist in the input sequence. Assume that there is an input record at hour 100 and the next record is at hour 102. A 3-hour moving aggregate sequence has a value at hour 101 as well, because there is at least 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 AVG_SYM and AVG_NOSYM grows more slowly with the size of the aggregation window. The relative difference between AVG_SYM and AVG_NOSYM at window size 100 for density 20% is about the same as the relative difference at window size 20 for density 100%. This is to be expected, because the ratio of the densities of the two sequences is also 100:20. □

4.4.4 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 over the last 24 hrs
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.
```

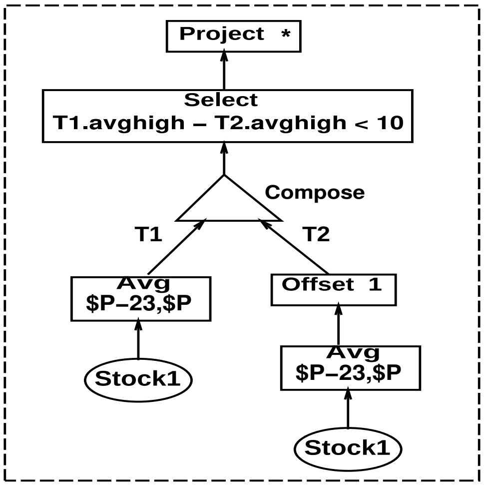

Figure 23: Repeated Computation

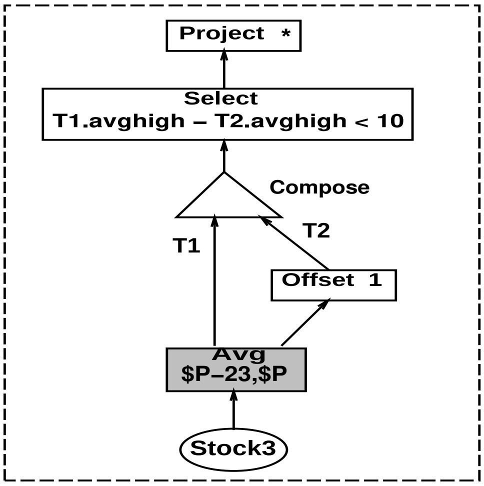

Figure 24: Common Sub-Expression

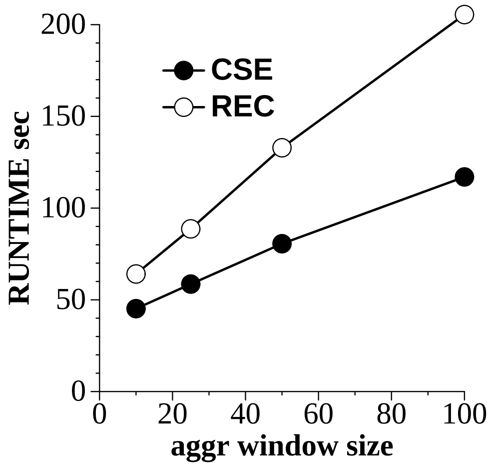

Figure 25: Common Subexpressions

Figures 23 and 24 show two possible algebraic query graphs that can be constructed from this query. The difference between the two query graphs is that one uses a common sub-expression (CSE), while the other does not. As described in Section 3.4.1, two evaluation options are to recompute the CSE, or to materialize and repeatedly access its result. A third stream-access option is possible in SEQ because of scope-based optimization techniques.

Experiment 4.6 The very same query shown above was run, except that the Stock1 sequence was replaced by 100K_10flds_100dens. The size of the aggregation window was varied 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 (CSE) and with repeated evaluation (REC). The results are shown in Figure 25. 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. With a large window size, the cost of query execution is dominated by the cost of executing the common subexpression. Recomputation is therefore almost twice as expensive as computing it once in a stream-access fashion. While the possibility of materializing the intermediate result has been ignored, the next experiment will show that materialization is very inefficient in general. □

4.4.5 Operator Pipelining

An important optimization principle in SEQ is to try and ensure *stream access* to the stored sequence data as well as to intermediate data; i.e., each sequence is read in a single continuous pipelined stream without materializing it. Consider the simple query below that scans a sequence and computes an aggregate over the entire data:

```
PROJECT count(*)              // Aggregate to reduce answer
FROM    <data_sequence>       // Source data sequence.
ZOOM  ALL;                    // Aggr over entire sequence.
```

Experiment 4.7 will show that there is a tremendous penalty to pay for failing to pipeline even such a simple query between the Scan operator on the base sequence and the Count operator. Experiment 4.8 shows that when the query becomes complex, with several nested operators, the relative importance of pipelining becomes even more clearly defined[4].

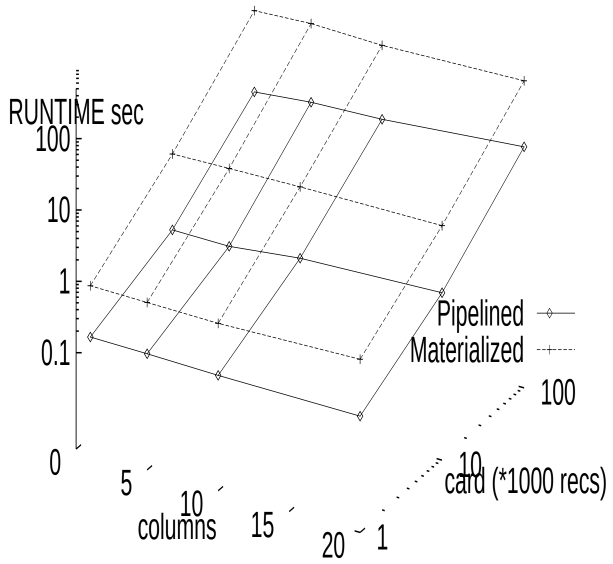

Figure 26: Varying Record Size

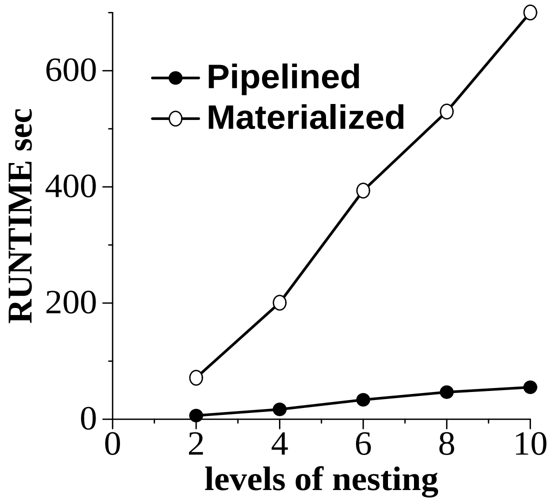

Figure 27: Varying Levels of Nesting

Experiment 4.7 The query shown above was run 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 26. 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, only the results for the 100% density sequences are shown (the 20% density results are similar). Notice that materialization increases the cost by almost an order of magnitude! □

Experiment 4.8 This experiment shows the effects of increased query complexity on materialized execution. 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, a query is used that can be varied in a controlled manner. One particular data sequence (100K_10flds_100dens) is considered and the number of levels of operators in the query is varied from 2 to 10. For instance, with 4 levels, the corresponding $\mathcal{SEQUIN}$ query is

[4]While these results are obvious based on similar results in relational query processing, the later chapters will show that in order to apply these optimizations (and derive these performance improvements), the new design paradigm of E-ADTs is required.

```
PROJECT count(*)              // Aggr to avoid printing answers
FROM    (PROJECT *            // Subquery block
         FROM (PROJECT *      // Another subquery block.
               FROM 100K_10flds_100dens)) S;
ZOOM  ALL;                    // Aggr over entire sequence.
```

The results with and without the pipelining optimization are shown in Figure 27. The X-axis shows the number of levels of nesting in each query, while the Y-axis shows the query execution time. 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. □

It should be noted that the materialization code in SEQ is written in a straightforward fashion, extending the temporary sequences one record at a time. Bulk loading techniques could instead be used to reduce the materialization overhead, and this will lower the relative differences between materialization and pipelining. However, in the current implementation, the generation of temporary intermediate sequences is a computationally intensive process rather than a disk-bound process. In fact, because the SHORE volumes are created on top of UNIX, written temporary data (which is relatively small) stays in the file system buffers instead of being flushed to disk. Consequently, the costs are not significantly affected by changes in the database buffer pool size. In a more efficient implementation, increasing the buffer size would decrease the materialization overhead. All the same, as in any database system, there is a significant cost in breaking the execution pipeline.

4.5 Possible Extensions

In the current SEQ implementation, every record in the sequence maps to a single position. One extension is 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. Another issue to explore is the development of 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 should 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).

At this stage, the thesis has described sequence database support in isolation. To place this work in the context of the larger problem of supporting multiple types of complex data, let us step back from the details of sequences, and examine the issues at a higher level of abstraction. At this higher level, sequences are merely another data type, coexisting with images, matrices, documents, and other complex data types, and indeed with integers, doubles, strings and other simple atomic data types as well. While the previous chapters have shown the potential benefits of special support for sequences, similar conclusions could be drawn about the positive effects of special support for the storage and manipulation of any complex data type. Today's object-relational database systems based on abstract data types lack the architecture to incorporate this kind of special support for complex data types. The next chapter explains the shortcomings of today's OR-DBMS, thereby motivating the new architecture proposed in the subsequent chapters.

Chapter 5

Object-Relational ADT Technology

The problems with the support for complex data types in today's OR-DBMSs are described in this chapter. While sequences are certainly an example of such a complex data type, a simpler and easily understood data type is the matrix. For the purposes of this chapter, the simple matrix operations serve to demonstrate the important issues. It also helps to demonstrate that the limitations of OR systems apply to data types other than sequences.

Consider an OR-DBMS based on Abstract Data Types (ADTs), which permits functions or methods to be defined and registered with the database system. These functions can then be used within the WHERE or the SELECT clause of an SQL query. A matrix ADT needs to be provided so that the DBMS may be used by scientists to store some experimental data. The functions provided with the matrix ADT are Multiply(M1, M2), Transpose(M1), and Clip(M1, Range). The first two functions have their obvious mathematical meanings, while the third function extracts a portion of the matrix as the result. Consider a single relation Exp of experimental data with a schema : { Station : char[30], Humidity : matrix }. The Humidity attribute is a 100X50 matrix of readings where the dimensions correspond to Temperature and Pressure. The Temperature varies from 1 to 100 degrees, while the Pressure varies from 500 mbar to 1500 mbar in steps of 20 mbar. Using this example, the following issues are studied:

- How is the Humidity matrix defined?
- How is the matrix data stored?
- How are queries involving the matrices expressed?
- How are these queries evaluated?

5.1 Defining the Matrix

The first question to ask is: how exactly is the schema specified? In writing "Humidity : matrix", what have been omitted are the types of the entries, the matrix dimensionality, the sizes of each dimension, the step size in each dimension, and the names Temperature and Pressure. How can one specify these in an ADT based system? Typically, there is a matrix "type constructor" which allows the user to define a specific sub-type of matrix to be used. In this case, the matrix type constructor would have to take all this information as arguments and generate a new specific type. The attribute declaration would look something like:

```
create type MyMatrix
 matrix(double,                      // the type of entries
        2,                           // dimensions
        {"Temperature", 1, 100, 1}, // name, start, end, step
        {"Pressure", 500, 1500, 20} // name, start, end, step
              )
Humidity : MyMatrix
```

While MyMatrix has exactly the same functions that can operate on it as any other matrix, it is entered as a new ADT into the table of types defined by the system. If this distinction is made between ADT constructors and actual ADTs, then it would be fair to say that almost all complex data belong to ADTs created using ADT constructors. This is true for images, time-series, matrixes, arrays, and even for something as simple as a string (a Char[30] is constructed using the Char type constructor and the length argument 30). However, in all these cases, the ADT instances generated by the application of an ADT constructor do not support any functions beyond what is supported by the generic ADT (as in this case, MyMatrix has the same functions that all matrices do). Therefore, in this thesis, "ADT" often refers to the generic ADT (matrix) that provides the functionality, rather than the specific ADT (MyMatrix) that is used at the level of type checking.

Returning to the example, the question that arises is : "Why does a new ADT need to be created for each specific kind of matrix, if they all support the same functions?". Indeed, since there are an infinite number of such specific matrix ADTs, they have to be created as needed by users. Consider another matrix almost identical to MyMatrix with the dimensions called A and B instead of Temperature and Pressure. Is this really a different type? What if the step size is slightly different, or the limits of the dimensions are different? Any practical system needs to draw the line somewhere, and there may be further specializations that distinguish between two uses of the very same type.

A more intuitive way to think of this example is that the Humidity attribute is of type matrix, and there is some additional meta-information that specifies the structure, properties, etc., of the matrix. This is analogous to thinking of all tables in an RDBMS as relations, with meta-information for each relation specifying the schema and other properties. The term "parametric types" [CW85] has also been used in this context to describe a type which is parameterized by additional meta-information, though this is not a feature commonly supported by an OR-DBMS.

5.2 Storing the Matrix

There may be several different implementations of a matrix. For example, there is the issue of whether the matrix is stored in a row-major or a column-major format (in fact for multi-dimensional matrices, there are many more possibilities), or whether it is broken into tiles [SS94]. Further, different implementation techniques may be used depending on the size of the matrix. Depending on the sparseness of the matrix, compression may be used. How can one specify what storage implementation is to be used? The ADT solution is to add this information to the type constructor. Therefore, MyMatrix stored in column-major format is a different type from MyMatrix stored in row-major format. This definition of types is certainly not derived from programming language type theory, since this is a storage issue, not a logical issue. Once again, the abstraction of meta-information stored with the type is much more intuitive to address issues like storage formats.

5.3 Expressing and Evaluating Matrix Queries

Now consider how a query may be posed against this experiment database.

```
SELECT Clip(Transpose(E.Humidity), 700, 800, 30, 60)
FROM Exp E
WHERE E.station = ‘‘Erehwon’’
```

There are different algorithms that may be applied to implement the transpose function. Their relative merits depend on the size of the matrix, its sparseness, the size of main memory available and the row or column-major storage technique. In a typical ADT-based DBMS, code for the Transpose function is written assuming a generic matrix. The same code is executed for all matrices; therefore it must take into account the factors that affect the algorithmic choices. Some of these factors may be known from the MyMatrix type instance of the argument (in this case, the size and storage technique may be known). However, the sparseness of the matrix is often a property of the *data*, not of the type. Consequently, if the function is to make the right decision, it must apply its decision rules for each function application depending on the characteristics of its particular argument during that invocation. Typically, database functions ignore such information and apply the same algorithm whatever the argument. However, proponents of the ADT approach may argue that this is merely for reasons of convenience rather than because of any real flaw in the paradigm. It is even conceivable that the available memory is passed by the system to the function as a special argument and is used at run-time to decide which specific algorithm to apply.

If the Clip function is applied after the Transpose function, its arguments are the range 700-800 on the first dimension (Pressure) and 30-60 on the second dimension (Temperature). Note that the dimensions were switched as a result of the Transpose function. On the other hand, it is also possible to evaluate the Clip function *before* the Transpose function. That expression evaluation might look like: Transpose(Clip(E.Humidity, 30, 60, 700, 800)). There are cases where this may be more efficient.

Now consider the case where the expression involving the matrix is a little more complicated. Specifically, let the expression be:

```
Transpose(Multiply(A, B))
```

There are several different and equivalent ways of evaluating this. The most obvious way is to first multiply the two matrices and generate the intermediate result; then compute the transpose of the intermediate matrix product. If the functional expression is interpreted procedurally (which is more or less the current state of the art in database ADT technology), this is the specific evaluation strategy chosen too. However, it is obvious that there are several other possibilities: for example,

```
Multiply(Transpose(F.Humidity), Transpose(E.Humidity))
```

Also, it may not be necessary to materialize the entire intermediate result of each function. Depending on the row-major/column-major properties of the data, it may be possible to stream or pipeline the intermediate results into the computation of the next function. In general, for any reasonably complex expression, there can be a wide range of possible evaluation strategies, each with different costs depending on the statistics of the underlying data. Ideally, the system should decide automatically among these choices in a cost-based manner. In current OR database systems, there is no framework for these choices to be considered. Instead, the user's specification of the functional expression also serves as a procedural specification of the evaluation strategy.

5.4 Summary

To summarize the discussion of this chapter:

1. In order to conveniently specify matrixes, meta-information should be maintained with each matrix definition. However, this facility is limited in current OR-DBMS systems.
2. In order to manipulate matrixes, they have to be embedded inside relations and queried within SQL. Since the top-level query language is SQL, there is no way to directly execute some expression involving matrices without embedding it inside an SQL expression.
3. A functional syntax is used for matrix manipulations, which is not always aesthetic or convenient. For example, expressing a complex arithmetic operation in a purely functional syntax is awkward. It would also be nice for the user to be able to use the index labels (Temperature and Pressure) meaningfully inside a matrix expression.

4. The functional expressions over matrixes are evaluated in a *procedural* fashion — in contrast to the traditional database technique of keeping queries as *declarative* as possible. Consequently, query evaluation can be extremely inefficient. The inefficiency arises from two factors: the inability to transform the functional expression into an equivalent expression that is more efficient, and the inability to consider "global" evaluation strategies like the pipelining of intermediate results.

The matrix ADT has been used as an example, with a simple "query language" over it. More complex query languages for matrixes and multi-dimensional arrays have been proposed [LMW96], and the same query processing problems could be demonstrated with these languages. Similar arguments could be made using sequences as the data type of interest. Any attempt at integrating sequences into a general purpose DBMS must address these problems. However, a suitably general solution to the problems developed in the context of sequences can also be applied to matrixes and other complex data types.

Chapter 6

Enhanced Abstract Data Types

One would like to make expressions involving ADT objects *more declarative*, and consequently more efficient. Further, this should occur without sacrificing the goal of *extensibility* – i.e., it should be possible to add new data types without causing dramatic changes to the rest of the system. Figure 28 demonstrates three possible approaches if one wants to support complex data in relational database systems. Each approach has the following components in the figure: a relational portion of the query expression, two separate subexpressions dealing with different ADTs, and interfaces between the ADT subexpressions and the relational expression. Fully shaded regions represent declarative computation, while hatched regions represent procedural computation.

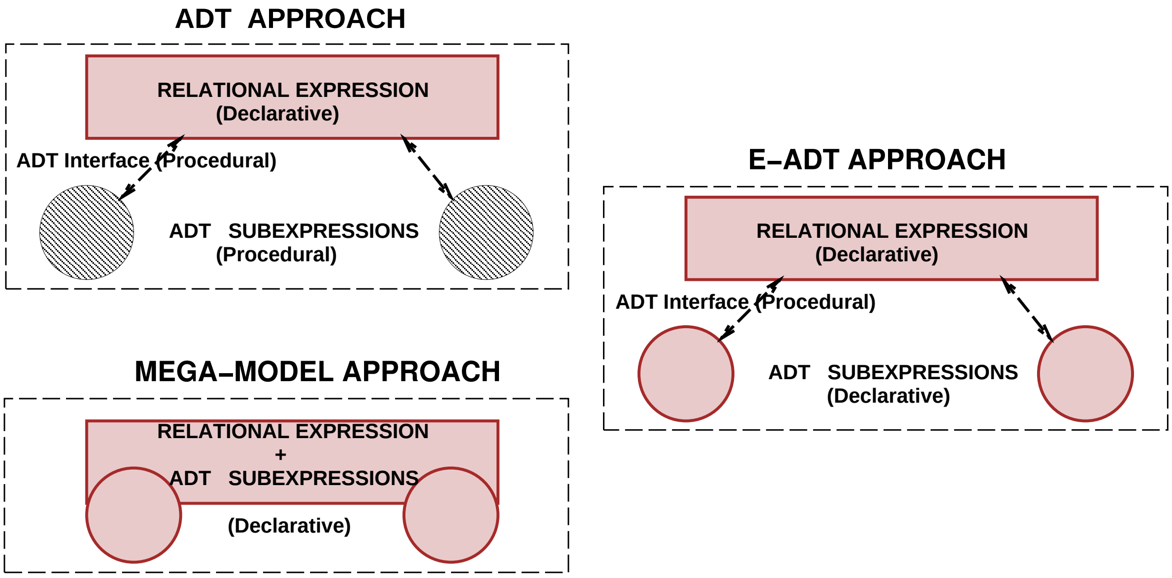

Figure 28: Alternatives Approaches for Complex Data Support

Current OR-DBMSs allow the relational portion of the query to be declarative, but the other parts of the query dealing with ADT manipulations are procedural. Another proposed approach is to construct a "mega-model" that can represent all the various kinds of complex data, and queries over such data. A system based on this model would require a single unified algebra that can represent all operations over the various types of data (examples of such proposals are the Aqua algebra [SLVZ95] and the higher-order monadic algebra used in the Kleisli programming environment [Won94]). While in these proposals, the entire query expression can be declarative, the important questions are whether such a system is extensible, and whether it is practical. If there are already several types in the system and a new type (like a matrix) needs to be added, how can all the interactions between matrixes and the other types be specified, and how can this information be used? Obviously, extensibility is compromised by requiring that each data type fully understand its interactions with all other data types. Further, it is not clear that such queries will be easy to pose, or practical to optimize and evaluate.

The Enhanced ADT (E-ADT) approach proposed in this thesis (and described in detail in [SLR96a]) lies between the two extremes of the ADT approach and the "mega-model" approach. Each of the subexpressions of the query corresponding to a specific data type is declarative. However, across subexpressions, the interaction is mostly procedural. In the example of matrices embedded inside relations, the expression involving the matrices (Transpose(Multiply(A,B))) is treated as a declarative matrix manipulation, but the interaction between this subexpression and the rest of the SQL query is procedural. The extensibility of the ADT approach is retained by giving up some of the declarativeness of the mega-model approach.

6.1 The Enhancements

The E-ADT (Enhanced Abstract Data Type) paradigm enhances the notion of an ADT in five primary ways. Each ADT:

1. Performs storage management for its objects.
2. Maintains catalog information for its objects.
3. Provides a query language.
4. Provides a query algebra and optimization techniques.
5. Provides an evaluation engine.

These enhancements are in addition to the existing ADT features, and do not all have to be provided by every E-ADT . In this chapter, the E-ADT paradigm is described in the context of the PREDATOR database system, which is based on E-ADTs.

6.1.1 Data Model Design

PREDATOR has a nested data model in which a complex object like a sequence or a matrix can be a field within a relational tuple, and vice versa. The type "relation" is also modeled as an E-ADT . Notions of object-orientation and inheritance are orthogonal to the issues addressed by this work, and consequently have been ignored. Each E-ADT may provide support for one or more of the following:

- *Storage Management:* Each E-ADT can provide multiple physical implementations of values of its type. The particular implementation used for a specific value may be specified by the user when the value is created, or determined automatically by the system.
- *Catalog Management:* Each E-ADT can provide catalog capabilities so that schema information can be stored and statistics maintained on values of that E-ADT . Further, certain values may be named, and a name mapping is maintained.
- *Query Language:* An E-ADT can provide a query/manipulation 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, the sequence E-ADT provides $\mathcal{SEQUIN}$, and the matrix E-ADT may provide its own language). As a special case, the language provided can have a functional syntax so that E-ADT manipulations look like the functional expressions of an OR-DBMS.
- *Query 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.

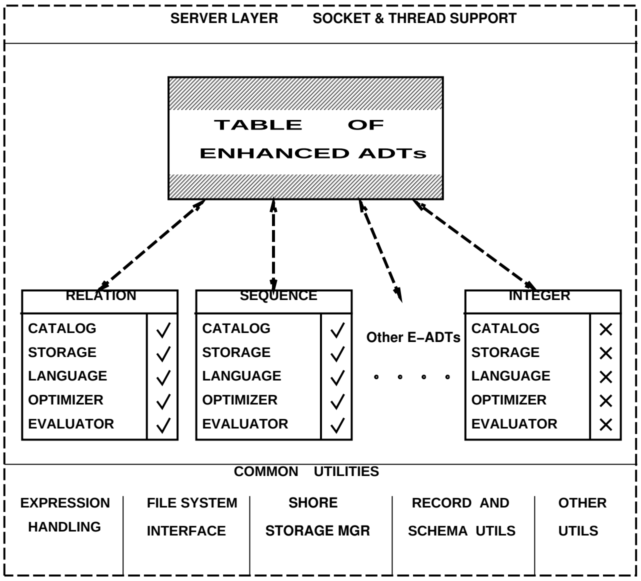

Figure 29: System Architecture

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

The E-ADT paradigm can be applied to provide database support for *any* complex data type. The ability to name objects belonging to different E-ADTs allows any E-ADT to be the top-level type. Relations are modeled on par with 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.

6.1.2 System Implementation

The design philosophy of E-ADTs is carried directly over into the system architecture. PREDATOR is a client-server database in which the server is a loosely-coupled system of E-ADTs . The high-level picture of the system is shown in Figure 29. 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. An important portion of the utility layer is the SHORE Storage Manager [CDF+93], which provides facilities for concurrency control, recovery and buffer management for large volumes of data. It also provides a threads package that interacts with the rest of the storage management layers; PREDATOR uses this package to build a multi-threaded server. 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. In the interest of extensibility, all E-ADTs have identical interfaces. 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 (although 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 all the enhancements: sequences and relations. A matrix E-ADT would also be in this category (though it has not been implemented yet).

With this high-level system design, the important question to ask is: how do interactions between the E-ADTs occur? The interactions are explained using two examples: matrixes and relations (from the previous chapter), and relations and sequences.

6.2 Matrixes and Relations

The example from Chapter 5 is repeated here again:

```
SELECT Clip(Transpose(E.Humidity), 700, 800, 30, 60)
FROM Exp E
WHERE E.station = ‘‘Erehwon’’
```

The experiment data is modeled as a relation, and the tuples of the relation have a field called Humidity which is a matrix. Another possible design is to have named matrices (for example, a named matrix called Erehwon). Assume for simplicity that the query language for matrices is called MQL, and has a functional syntax exactly as is used in the query. In PREDATOR, the syntax of the query has to be slightly modified so that the matrix sub-expression is "marked" as belonging to the MQL query language.

```
SELECT MQL("Clip(Transpose($1), 700, 800, 30, 60)", E.Humidity)
FROM Exp E
WHERE E.station = "Erehwon"
```

The change is purely syntactic, and helps the system (and the user) identify the sub-expressions in the query that belong to each data type. For example, an equivalent syntax for the expression is shown below:

```
DEFINE FUNCTION Foo (matrix H) as
    MQL {Clip(Transpose(H), 700, 800, 30, 60)};
SELECT Foo(E.Humidity)
FROM Exp E
WHERE E.station = "Erehwon"
```

The specific syntax chosen is not important; what matters is that the system be able to identify the boundaries between sub-expressions belonging to different languages. Once the query has been specified by the user (for concreteness, using the first syntax shown above), there are various stages of query processing, which are explained in the subsequent example.

6.3 Sequences and Relations

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. Since the sequence data is nested within the relational data, it is appropriate for the user to think of the relational E-ADT as the top-level type. A query will therefore be posed in the relational query language (SQL) with nested query expressions in the sequence query language ($\mathcal{SEQUIN}$). Consider the SQL query to find for each stock, the number of hours after hour 3500 when the 24-hour moving average of the high price was greater than 100.

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

The SQL query has the usual SELECT clause target list of expressions. One of these expressions is a $\mathcal{SEQUIN}$ query, which appears as a function to the SQL parser. 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. The SQL language parser does not know about the grammar of the embedded language, and merely treats the $\mathcal{SEQUIN}$ subquery as a function call whose first argument is some string. For the SQL parser, this query is treated in the same manner as the following query would be:

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

This nested query language paradigm requires enhancements to the three stages of query processing: type checking, optimization and evaluation.

Parsing and Type Checking: Each E-ADT provides a language parser by means of the following virtual method:

```
ExprParse(const char* ExprString,
          const ArgEnv* ArgEnv,
          BasicType& ReturnType,
          GenericParse*& ParseStructure)
```

The ExprString is the textual string containing the query expression. The ArgEnv represents the types of any values passed to the expression because it is embedded in a nested scope. The method returns the type of the expression, as well as a generic ParseStructure that will only be interpreted by the E-ADT that creates it.

As part of the type-check of the SQL query, the type of the $\mathcal{SEQUIN}$ function is also checked. This causes the ExprParse method of the sequence E-ADT to be invoked, thereby parsing the sequence subexpression. It is no longer sufficient for the SQL parser 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. The return type of the $\mathcal{SEQUIN}$ query expression is a sequence as usual.

Query Optimization: Each E-ADT is responsible for optimizing its own query expressions. In this example, the SQL optimizer is called on the outer query block. The optimization of an SQL query follows certain standard cost-based techniques in searching among a space of evaluation plans. Any sub-expressions need to be first optimized before the SQL query can be optimized. Since the sequence sub-expression is introduced in the guise of a function, each optimizer must be sure to 'plan' any function invoked. Planning a function like $\mathcal{SEQUIN}$ causes the invocation of the following virtual method of the sequence E-ADT :

```
ExprOptimize(const GenericParse* ParseStructure,
             GenericPlan*& PlanStructure);
```

This method takes the parse structure generated as a result of parsing the sequence expression and returns a generic structure representing an evaluation plan for that expression. Note that the PlanStructure is interpreted only by the sequence E-ADT , thereby conforming to the extensibility requirements. The only well-known method of the PlanStructure is a Cost() method that returns an estimated execution cost. As will be discussed in Section 7.4.2, a common cost model is needed to integrate the optimizations of multi-E-ADT queries. In this example, the SQL optimizer uses the knowledge of the cost of the sub-expression to determine the best plan for evaluating the SQL query. For example, if the sequence subexpression is very expensive, it may be preferable to apply it as few times as possible.

Query Evaluation: The evaluation of the query is relatively straightforward. The SQL query plan starts to be executed by the relational query execution engine. A virtual method provided by each E-ADT performs this evaluation.

```
ExprEvaluate(GenericPlan* PlanStructure,
             const ValueEnv* ArgEnv,
             void *ReturnValue);
```

The method executes the optimized plan provided. The SQL query evaluation plan does not need any arguments passed to it. However, while executing the portion of the query corresponding to the SELECT clause, it invokes the sequence subexpression. As far as the SQL plan is concerned, this is some unknown function which it does not know how to execute. However, it does know what the arguments to the expression are (S.Stock_History) and the type of the return value of the expression. It also has a handle on a query evaluation plan for the function. It repeatedly invokes the function plan, passing it the plan structure and the current argument, and uses the return value to continue with its computation. Each function plan invocation in turn calls the ExprEvaluate() method of the sequence E-ADT which causes the sequence subexpression to be evaluated.

6.4 Comparison with An Existing System

Illustra [Ill94] is a commercial OR-DBMS supporting sequences (more specifically, time-series) as ADTs with a collection of methods providing query primitives. This section qualitatively compares this approach with PREDATOR, based on the performance experiments presented in Section 4.4. While a quantitative performance comparison with Illustra [Ill94] was performed, those results cannot be published under the terms of the license from Illustra. However, the experiments of Chapter 4 that demonstrate the effects of each individual optimization indirectly indicate the expected performance of Illustra, which lacks several of the optimizations.

The basic model of a time-series in Illustra is similar to the model in the SEQ implementation. Consider the same sequence query used as an example in Section 6.3. Here is approximately how it would be written using ADT methods:

```
SELECT S.name,
       count(filter("time > 3500",
                  filter("high > 100",
                       mov_avg(-23, 0,
                            project("time, high",
                                 S.stock_history)))))
```

```
FROM   Stocks S;
```

Note that a query language based on function composition can be more awkward to use than a high-level language like $\mathcal{SEQUIN}$. In this example, some of the arguments to the project() and filter() methods need to be provided as strings to be interpreted by the functions. The more important observation is that while queries in $\mathcal{SEQUIN}$ are declarative, queries based on the functional composition of methods have a procedural semantics. When a query expression involves the composition of more than one of these methods, little or no inter-function optimization is performed, and each individual method is evaluated separately.

The pipelining optimization coupled with stream access is a very important algebraic optimization in SEQ. With the ADT approach, pipelining is not possible without inter-function optimization. Experiments 4.7 and 4.8 showed that materialization can perform an order of magnitude worse than pipelining. The optimizations that propagate valid ranges and selection predicates (Experiments 4.2 and 4.3) require the ability to push range selections from one function to another. Consequently, ADT based systems do not exploit these optimizations. The ADT approach cannot identify common sub-expressions without inter-function optimization, let alone take advantage of them to optimize query execution. Experiment 4.6 showed that the common sub-expression optimization could reduce query execution time by almost a factor of two.

The ADT approach of Illustra is unable to apply optimization techniques that could result in overall performance improvements of approximately two orders of magnitude. It is important to stress that these differences are symptoms of a basic design difference between PREDATOR and Illustra, and are not merely due to a more efficient implementation. In order for Illustra and other similar OR-DBMSs to derive some of the efficiencies of the PREDATOR approach, they will need to adopt some or all of the system design techniques used in PREDATOR. These design techniques are discussed in detail in the next chapter.

6.5 Summary of E-ADT Ideas

To summarize, a query is broken into components that correspond to sub-expressions in each E-ADT query language. The sub-expressions may be nested within each other. Query parsing, optimization and evaluation begins at the highest level of the query. The interfaces across the different subexpressions are procedural, and are similar to standard embedded function invocations. However, within each subexpression, the evaluation is declarative, based on a cost-based optimization. This approach is “loosely-coupled” in the sense that there is little interaction between the various E-ADTs in the system. The primary interaction arises due to the shared table of types, the common cost model, and the use of the interface methods for handling nested expressions. There are also some other methods that contribute to the “virtual” interface of E-ADTs . These are listed in detail in the appendix. Mainly, these methods provide catalog management, so that nested complex objects may be modeled. For instance, while defining the experiment relation Exp, the embedded matrix field Humidity needs to be defined. This happens via the invocation of a method on the matrix E-ADT with arguments specifying the name of the new field (Exp.Humidity), and the textual string with which the user specifies the schema of the Humidity matrix.

Chapter 7

The EADT paradigm

The purpose of this chapter is to analyze the E-ADT paradigm. Some of the topics covered include implementation issues, possible problems and their solutions, and open research issues. There is also a discussion of migration paths that a current OR-DBMS could take to partially acquire the E-ADT functionality. Finally, there is the interesting possibility of extending the E-ADT paradigm to heterogeneous database system integration.

7.1 Implementation Issues

Some interesting implementation issues arose while building the PREDATOR DBMS based on the E-ADT paradigm. Several issues have already been discussed in the previous section, including the need to augment compile-time type information with meta-information, and the need to treat relations as just another data type. Here are some other issues that have not yet been discussed.

7.1.1 Reentrant Components

The various components of the query processing system of each E-ADT (mainly the parsing, optimization and evaluation functions) need to be reentrant. This is because in a complex object model, a specific instance of a data type may be defined in terms of another instance of the same type. For example, a relation might contain a nested relation, or it may contain a nested sequence of relations. Queries over such data might be expressed in SQL, with embedded expressions in any other query language, possibly SQL again. Consequently, the SQL parser needs to be reentrant, and the reentrancy requirement holds for the query optimization and evaluation components as well.

7.1.2 Ubiquitous Identifiers

Since the interaction between the different types is so loosely-coupled, every E-ADT needs to maintain identifiers on various kinds of objects. At the very least, if the data values are large, then identifiers for them are used instead to store within records. This leads to the need to perform persistent reference counting for these identifiers (this has not been implemented yet). Identifiers are also needed for parse structures and query plans. For example, when a subexpression in $\mathcal{SEQUIN}$ is planned from within a SQL query, a handle (i.e. identifier) to a query plan is returned. This handle is stored within the SQL evaluation plan. When the SQL query is executed, it invokes the subexpression by asking the sequence E-ADT to evaluate the plan identified by the stored handle. In the current implementation, query parsing, optimization and evaluation happen in one pass, and so the data structures like parse tree and evaluation plan are maintained in memory. The addresses of these structures serve as their identifiers. However, a full-fledged implementation will also have to consider the case where queries are pre-optimized, and this will require that query plans also have persistent identifiers.

7.1.3 Common Utilities and Plug-In Types

One important aspect of the extensible system design is that it should be possible to "plug-in" a new E-ADT or take out an existing E-ADT without adversely affecting the other data types in the system. Indeed, this is true in the PREDATOR system for all the data types[1]. Therefore, if the relational E-ADT is not compiled into the system, it can still support sequences, and vice-versa.

At the same time, it is important that as far as possible, code should be re-used between the different E-ADTs . The role of the Utilities layer of the system is to provide many useful abstractions that can be used by E-ADT builders to quickly and efficiently supply the interfaces that they need. The SHORE storage manager is part of the Utilities layer and provides persistent storage that all E-ADTs can use. Similarly, there are utility functions to handle boolean expressions, manipulate algebra trees, create records, and perform arithmetic. A rule-based optimizer engine could also be supplied as part of this layer. Ideally, most of the code complexity of the system would reside in the utilities layer, while most of the "intelligence" of the system would reside in the E-ADT methods. As the system continues to develop with new data types, it should be possible to have a more concrete sense of the extent to which such a separation can be made.

7.1.4 Nested Evaluation

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 [SPL96] 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.

7.1.5 Granularity of Statistics

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? In PREDATOR, average statistics are maintained 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 decision also allows for the compile-time generation of a single "optimized" plan for all invocations of the nested sequence subexpression.

7.2 Adapting Existing Database Systems

Most current relational database systems are slowly incorporating portions of the ADT technology used by object-relational systems. Consider a relational database system that has the notion of an ADT and has mechanisms to allow functions to be extensibly defined. The E-ADT proposal requires reasonably extensive changes to the existing system code base, which may not be feasible. In order for such a system to adopt some subset of the features of the E-ADT paradigm, various possible migration paths are now suggested.

[1]The sequence E-ADT depends on the existence of the integer E-ADT , but beyond this small exception, there is complete independence among the types.

7.2.1 Minimal Extension

The minimal extensions suggested here are typically already performed to some extent in object-relational systems.

- As a first step, each ADT may be allowed to have multiple implementations. The choice of implementation may be specified by the user or decided automatically by code provided as part of the ADT. This functionality is already present in the Illustra database system [Ill94]. Of course, there may be some ADTs for which a single implementation may suffice. On the other hand, for the matrix ADT, there may be a row-major and a column-major implementation, in addition to different implementations for sparse and dense matrices.

- The next step is to use the knowledge of the implementation of the data type inside each function evaluation. For example, the Transpose function can first check if its argument is in column-major or row-major format and then invoke different code for each case. Note that this form of optimization is performed at run-time for each independent invocation of the function.

7.2.2 Exploiting Heuristics

The next set of enhancements try to exploit heuristics for optimization. For instance, assume that the database system is associated with a rule-based engine that can apply heuristic transformation rules. Each ADT could provide rules that specify heuristic transformations that should be applied to expressions involving functions of that ADT. For example, there could be a rule for the matrix ADT that says :

```
Clip(Transpose(X), A, B, C, D) ==> Transpose(Clip(X, C, D, A, B))
```

Note that this rule is a *heuristic* and assumes that it is better to apply the Clip function early (this is analogous to the selection pushdown heuristic used in relational query optimization). The rule is applied at "compile time", which is a significant change. A recent research proposal [CS93] suggests that rewrite rules could be applied in this manner (though the purpose of the proposal was to optimize access to external databases).

There are a couple of important limitations to this extension. While the example could be specified as a relatively simple rule, the typical kinds of rules used in query optimization are much more complex. For instance, even a straightforward rule like pushing a selection through a relational join is difficult to express in this high-level manner. Various properties of the selection condition and the join condition need to be checked. In fact, most practical implementations of rule-based relational query optimizers use procedural code to implement each optimization rule. It is reasonable to expect that optimizations of expressions involving other kinds of complex data will also require rules of similar complexity. In fact, the experience with optimizing sequence queries bears out this expectation.

The other limitation arises due to the fact that these rules are applied as heuristics, and not in a cost-based manner. In order for the rules to be treated as equivalences which can then be applied in either direction, the system needs to be able to supply cost estimates of the queries that result from applying the rules. Estimating the cost requires a cost model and cost formulae for the various functions. This leads to the next possible extension.

7.2.3 Applying Cost-Based Transformations

The two limitations of heuristic rules may be eliminated by (a) allowing the rules to be specified using complex procedural code, and (b) allowing the rules to be associated with cost metrics. A rule-based optimizer engine like Volcano [GM93] could then be used to apply the rules.

Since the rules require procedural code, they need to be written by a "database implementor" rather than by a naive user. This is keeping with the model of implementing important data types as vendor-supplied "datablade" libraries. In order for the rules to be applied in a cost-based manner, cost formulas for each of the functions need to be specified. While the common cost model could be based on CPU and I/O usage (just as in relational query optimization), the following question arises: to what extent does the cost formula depend on properties of the arguments to the function? Note that at least one of the arguments will be a value of the complex data type (for example, a sequence). The cost formula has to determine *at compile time* what the cost of executing the function will be at run-time. This is obviously a difficult problem without any knowledge of the properties of the data value. The equivalent problem in relational query optimization might be to find the optimal join order without any information about the implementations of and statistics on the relations being joined!

What if the optimization rules were applied at run-time using the known properties of the run-time arguments? While this would indeed be feasible, the issue becomes one of performance. Note that each expression involving the composition of ADT functions may be invoked several times within the same query. It is typically not desirable to require that each specific expression invocation be independently optimized at run-time.

Obviously, therefore, this extension has minimal benefit unless the system can provide *compile time* statistics about the values that will be passed as arguments. Using the example of Section 5, statistics need to be maintained on the implementation and sparseness of the Humidity matrices. If this information is made available to the optimization rules, they can be applied in a cost-based manner. Further, instead of performing run-time decisions about the specific implementations used for each function, compile-time decisions can be made. However, this added enhancement of providing compile time statistics for each complex data type requires a relatively major change to the database system.

7.2.4 Maintaining Compile Time Statistics

Note that in the example, there need to be some common statistics maintained for all the Humidity matrices. For example, that all of them are stored in row-major fashion, are dense, and have specific ranges along each dimension. Of course, the schema of the matrices (columns are Temperature and Pressure, and the matrix entries are of type double) is also maintained. On the other hand, if it is not possible to maintain common statistics (for instance, some of the matrices are stored row-major, while others are column-major), then this inhibits the ability to perform compile-time optimization.

At the level of the system code, there is one major change that is necessitated by the desire to maintain compile time meta-information on the various objects. At an intuitive level, the type name "sequence" is no longer sufficient information to uniquely define the format of the data. For reasons of type checking, the schema of the sequence must also be specified. Further, for optimization purposes, the statistics must also be specified. In an extensible system, this implies that *at every stage of the code where the type of a attribute or expression is stored or passed as an argument, the meta-information must also accompany the type information.* This involves a major code change, over and above the work involved in collecting statistics on the various complex data types. Note that not only must the statistics be collected, they must also be maintained, thereby necessitating a catalog structure for each new data type.

Further, in order to be totally extensible, the meta-information on a specific data type like Sequence must only be interpreted by routines provided as part of that type! This is reasonable, because it is only the optimization techniques for sequence expressions that will need to use the sequence meta-information.

If this major change is performed, then cost-based optimization can be performed using transformation rules. There are still some limitations though. One problem is that there may be a large space of possible equivalent queries, and there is really no guarantee that good alternatives are tried. In fact, rule-based optimization is still a topic of ongoing research for relational queries, so it is not clear that this is necessarily a practical solution. Another limitation is that the same optimization mechanism is being imposed on all expressions involving all data types. However, it might be desirable to optimize sequence expressions more exhaustively than the matrix expressions within the same query. Such control mechanisms for rule-based optimizers may be difficult to specify. Finally, rules can be used to specify a certain category of "equivalence" optimizations. However, something simple like streaming intermediate results between two functions, does not fall into this category. In the expression Clip(Transpose(X), A, B, C, D), the transposed matrix X does not need to be materialized in full. On the other hand, the result of the Clip needs to be materialized if it is to be stored as a field within a relational tuple. How does one exploit such optimizations?

The basic feature that is lacking is the recognition of distinct subexpressions that are composed totally of operations of one data type. If such sub-expressions can be recognized, then the entire subexpression can be treated declaratively (allowing for streaming of intermediate results). Other optimizations like common sub-expression recognition can also be performed. This is the subject of the next enhancement.

7.2.5 Declarative Expression Boundaries

The goal of this enhancement is to recognize that a complex functional expression is really a declarative expression involving a specific data type. As a trivial example of this, there is usually only one implementation for each function, corresponding to only one evaluation strategy for each operation. In the OQL language design [Cat94], methods are allowed to have multiple implementations, and the system has the freedom to choose the best one. This is a small first step towards declarative functional expressions.

Recognizing the expression boundaries has benefits independent of the maintenance of compile-time statistics, or indeed the specification of transformation rules. For example, the ability to stream intermediate results can result in a big improvement by itself. One simple way for existing systems to mimic this is to provide a special "EVALUATE" function along with each data type. The EVALUATE function will take a string argument (which represents the complex expression), and other arguments which will serve as parameters to the expression. The entire declarative semantics of the expressions can now be encapsulated within the EVALUATE function. If relations were implemented as just another data type, then the EVALUATE function would take an SQL string as an argument. This would allow relational expressions to be nested within any other kinds of expressions.

This use of the function mechanism provides a pathway to treat expressions over data types as declarative queries. With this as a basic enhancement, it is not very difficult to now permit any suitable language syntax for the string argument that denotes the query expression. For example, when implementing support for sequences, it was convenient to use the Sequin query language for the sequence query subexpressions.

7.2.6 Multiple Top-Level Types

So far, the assumption has been that SQL is the top-level query language, and there are various subexpressions within an SQL query that manipulate objects of other complex data types. Since catalogs are already being maintained for each complex data type (see the previous section), it is not much additional work to allow these catalogs to maintain named objects. For example, this allows named sequences or named matrices to exist along with named relations. Therefore, a query involving named matrices can be expressed in the matrix query language without having to use SQL. Further, an SQL expression can be embedded within a query in some other language.

In order to migrate to this totally uniform model, relations need to be treated as merely another data type (like matrices, sequences and even integers). This is another major change to the system code, because the assumption of the special status of relations is usually implicit in many aspects of the system.

7.3 Limitations of E-ADTs

In this section, some possible objections to the E-ADT paradigm are discussed. The first concern is that adding a new data type has now become a complicated process, because so much new functionality (parsing, optimization, evaluation, catalogs and implementations) have to be added. It should be noted that all of these features are purely optional. Although the system supports E-ADTs , it still involves strictly no additional work to add the traditional style of ADTs (with methods, but little else). For data types representing large complex data with complex manipulations (like matrices and time-series), the improvements in performance (shown to be orders of magnitude in the case of time-series) will be well worth the additional few thousand lines of code.

The next question to ask is whether the mega-model approach (express the entire query in some unified algebraic model and then optimize it) would find better evaluation plans than the E-ADT approach. At first glance, it would seem that the answer is obvious: placing the query in a unified framework is bound to find at least as good an evaluation strategy as the E-ADT approach, if not a better one, because the space of possible options is strictly greater. Of course, one loses the advantages of extensibility, but one might gain performance improvements. On closer examination, it becomes evident that simply because there is a larger space of plans does not mean that the space is searched, and that all the plans in it will be considered. For practical reasons, all database systems limit the possible plans considered, so that even within a totally relational query, there are many plans that are not ever considered. In light of this observation, using a mega-model algebra does not necessarily mean that a better plan will be found in practice. There is, however, a legitimate concern that there may be some inter-E-ADT optimizations that might be important. Some such possible optimizations could perhaps be implemented as extensions to the basic E-ADT mechanism. 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, this appears to be an acceptable price to pay for the other benefits of the nested design paradigm. Future experience with PREDATOR might indicate though that important opportunities for cross-E-ADT optimization exist.

It has been suggested that one could have devised some extension of SQL that provided sequence functionality. While this was indeed initially attempted, too many changes to SQL 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. Therefore it is most practical to leave SQL relatively unchanged, and introduce the sequence functionality via E-ADTs .

7.3.1 Extensions of the E-ADT Paradigm

Consider an SQL query which has an embedded expression which is also in SQL. It might be possible to recognize that it is the very same E-ADT on both sides of the expression interface, and merge the two expressions. The resulting merged expression might result in more efficient performance (possibly using techniques like those suggested in [PHH92]). Extrapolating from this, if there are two similar E-ADTs like relations and sequences involved in a query, it might be possible to propagate information from one E-ADT expression to the other. For example, a selection condition might be pushed across the interface.

The main source of procedural behavior in the E-ADT paradigm is at the interfaces between subexpressions. The evaluation mechanism across the E-ADT interface is *tuple-at-a-time*, i.e., the subexpression is evaluated once for each combination of arguments. Tuple-at-a-time execution is typically a source of inefficiency in database systems. However, this problem is identical to similar problems in nested SQL queries and heterogeneous queries. If each expression also provides a "set-at-a-time" plan (i.e., given a set of arguments, instead of a single one, evaluate the expression for each of the arguments), then this provides a set-oriented interface across E-ADTs . Recent work [SHR+96] has described how cost-based optimization can be used by a database system to determine how best to utilize set-oriented plans. The same techniques can be applied to E-ADT queries as well.

The occurrence of the same E-ADT subexpression (or portion of it) in multiple places within a query could be another possible source of inefficiency. For example, within an SQL query, the very same matrix subexpression could occur within both the WHERE clause and the SELECT clause. This is a standard problem with nested expressions, and is not really caused by the E-ADT paradigm. The very same problem would arise with the functional expressions of the standard ADT approach. Possible solutions to the problem are based on recognizing common subexpressions in different parts of the query. In fact, the E-ADT approach may well be better suited to handling this, since the E-ADT optimizer could maintain its state across multiple invocations on different subexpressions in the query. In this way, it can detect common subexpressions in the query.

7.4 From Structural to System Heterogeneity

So far, the E-ADT paradigm has been developed to support complex data in an extensible and yet efficient manner. Since an E-ADT provides a language, optimizer, evaluation engine, catalogs and storage management, it is reasonable to characterize each E-ADT as a mini-system by itself. If many mini-systems can co-exist in a loosely coupled manner, how difficult is it to extend this further to support system heterogeneity? In other words, can this paradigm be extended to support heterogeneous database systems. This section tries to address this issue at the logical level, though these ideas have not been implemented yet.

7.4.1 Types Versus Systems

The first point to note is that while it was convenient to introduce the concept of E-ADTs as declarative data types, there is really no compelling reason to tie each declarative language to a single data type. Each language is really supported by a *database engine*, though all the database engines operate within the same address space on the same system. If the database system is enhanced with a shared table of database engines (in addition to the table of E-ADTs), then there is no basic change in the paradigm proposed; except that each engine can now support a language manipulating more than one type. More importantly, there may be several engines providing languages for the very same E-ADT !

With this change, how should the data management (catalog and storage management) be modified? One option is for each engine to perform its own data management, though this implies that engine A cannot access the data managed by engine B. One could possibly go a further step and allow each engine to be registered with all the E-ADTs whose data it manages. Therefore, as long as the table of E-ADTs is common to all the engines, it may even be possible for engine A to process the data stored by engine B. So far however, the assumption is that all these engines are part of the same system, sharing the type table, cost model and utilities layer.

7.4.2 Shared Components

To move to a heterogeneous system environment, the shared components need to be examined. All the systems will have to make sure that their type systems agree (at least on the interacting types). Since arguments may be passed from one system to another, there must be a common data format for values of these types. Some common interchange format for data (like XDR) will need to be used. Transaction issues become much more complicated, and if any of the systems being integrated are existing commercial systems, it may not be possible to modify them to introduce some form of two-phase commit. Consequently, strict transaction semantics may not be possible (this is a difficult problem anyway in heterogeneous systems [Moh95]). Finding a common cost model across multiple database systems is a difficult research area [DKS92], and some approximate cost model must be used. Despite these problems, it should be possible to connect an external database system into the PREDATOR system without any large-scale modifications to the system code.

7.4.3 Contribution of E-ADTs to System Heterogeneity

Not addressed here are many of the really important heterogeneous database system issues like schema and data integration, and query optimization. However, what the E-ADT approach does offer is a lower level mechanism for integrating multiple systems. The user of a heterogeneous database might be presented with a single unified abstraction and database schema against which to pose queries. Within the system, the queries may be mapped into subexpressions belonging to the various component systems and expressed in the corresponding languages. This proposal contrasts with the popular approach of providing heterogeneity by writing "wrappers" over data sources [CGMH+94, CCH+95]. In the wrapper-based approach to integrating diverse systems, each system provides a common abstraction of its data, usually using an object-oriented model. This common abstraction is provided by implementing a wrapper around each system. The wrapper understands expressions in a unified query language and can translate these expressions into the language provided by the underlying system. Since all the systems being integrated provide a common wrapper interface, expressing queries across the systems is not difficult. In contrast, the approach suggested by E-ADTs is that systems do not necessarily have to provide uniform wrappers. This intelligence can be provided at the system that interacts with the users.

Chapter 8

Case Studies

Many of the previous chapters have included examples of the use of sequence queries in real-life contexts. In this chapter, three additional case studies are presented. One deals with queries over medical data, while another is a financial data analysis application. The third case study demonstrates that sequence queries can be used for pattern matching over streams of data.

8.1 Clinic Visit Data Example

The Family Medicine Department at the UW-Madison collects patient visit information from clinics around Wisconsin. The operating information from each clinic is available as a flat-file sequence 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, and treatment_id into the actual information on the patients, diagnoses, and treatments respectively. A sample query compares the seasonality of cases of gastro-enteritis during the last 5 years, with the seasonal patterns of appendicitis. PREDATOR would model the data as follows:

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

The Visit_Info sequence contains a nested relation for every day, which in turn holds the visit records for that day. In addition, there are three tables Patient, Diagnosis and Treatment defined. There are various possible definitions of the '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. Despite the use of views, the entire example is a single sequence query, rather than a collection of several queries. Note the use of a nested relational query, within a sequence query. While this query may appear extremely complex, the optimization strategies developed in SEQ allow it to be executed in a single scan of the VisitInfo sequence.

The actual data from the Family Medicine clinics is a little more complicated. In reality, each patient can receive several treatments for the same diagnosis, so the *treatment_id* field of the visits nested relation is itself a nested relation. Further, this schema only handles the data from one clinic; however there are many (18, actually) clinics located around Wisconsin. Consequently, at the highest level, the data could either be modeled as 18 separate sequences, or as a single table which contains nested VisitInfo sequences. This leads to a nested schema in which a top-level relation of clinics has nested sequences of visit histories, which have nested relations of visits on any day, which in turn have nested relations corresponding to the treatments performed for any particular patient visit! PREDATOR can handle such data and queries over all the different levels of the nested hierarchy.

8.2 Financial Data Analysis

A hypothetical stock market analyst wants to determine how various portfolios have performed over the last few years. Each portfolio is a set of stock names, each associated with the number of shares of that stock and the purchase price per share. There are also data sequences available representing the daily price of each individual stock over the last few years. All these stock histories are stored in a table(StockData), which associates each history with the name of the stock. The desired answer is a "performance" sequence for each portfolio over the period of interest. Here is how this data and query could be modeled in PREDATOR. The top-level language used is SQL.

```
// define the PortFolios table
CREATE TABLE PortFolios AS
   { pfname      char[30],
```

```
     portfolio table {stockname  char[10],
                      number     int,
                      buyprice   float}
   };

// define the StockData table
CREATE TABLE StockData AS
   { stockname   char[30],
     history sequence {date  day,
                       avgprice float}
   };


// This view finds for each portfolio, a set of sequences,
// each corresponding to the performance of each stock across
// the period of time being tracked. This is not exactly
// the query desired, because it presents a number of
// performance sequences for each portfolio.
CREATE VIEW DetailedAnswer as
   (SELECT PF.pfname,
       $SQL(``SELECT  $SEQUIN(``PROJECT S.day,
                                        $2*(S.avgprice - $3)
                                FROM $1 S'',
                              SD.history,
                              PF.number,
                              PF.buyprice) as perf_seq
              FROM $1 PF, StockData SD
              WHERE PF.stockname = SD.stockname'',
            PF.portfolio) as per_stock_perf
    FROM  PortFolios PF);

// If one aggregate performance sequence is needed per portfolio,
// then the individual sequences need to be summed up. While there
// are different ways of doing this, the most elegant way is to
// use an aggregate function defined on a set of input sequences.
// This will be just another aggregate function in SQL, like MIN,
// MAX, COUNT, etc. However, the mechanism by which this can be
// defined using SEQUIN has not yet been worked out.
SELECT DA.pfname, $SQL(``SELECT SEQSUM(P.perf_seq)
                          FROM $1 P'',
                       DA.per_stock_perf)
FROM  DetailedAnswer DA
```

8.3 Stream Pattern Matching

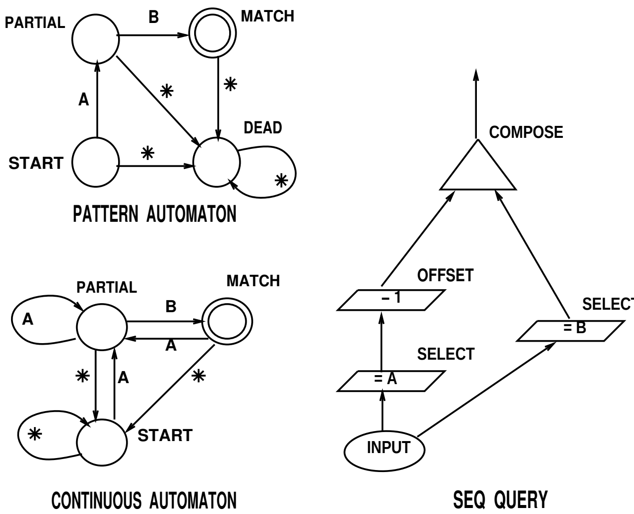

Figure 30: Regular Automaton and $\mathcal{LSEQ}$ Query for Pattern "A B"

[GJS92] presents a model and implementation of a system to recognize composite event patterns. The system provides an event expression language that creates composite events from basic events. The goal of the system is to detect *all* occurrences of these composite events in a sequence of basic events; this is called a *continuous match* query, since a pattern of events can be matched starting potentially at any position in the input sequence. The input event sequence has no empty positions. The pattern itself can be as complicated as an arbitrary regular expression. The approach used in [GJS92] is to translate the event expression into a finite state automaton that can then be used to evaluate the expression in a "memoryless" fashion. Because the pattern has to be matched at every possible starting position, the automaton needs to be "restarted" on every input. This can be done by introducing an empty transition from every state to the start state. Therefore, even if the automaton corresponding to the original pattern is acyclic, the actual automaton used does involve cycles. As an example, Figure 30 shows an automaton representing the simple pattern "A B" (the Pattern Automaton), and the automaton used to continuously match it on an input sequence (the Continuous Automaton). Transitions in the automata labeled $*$ indicate a transition on any symbol for which an explicit transition out of the state is not drawn. Note that when the continuous automaton sees an A, it requires a transition to state PARTIAL from *every* state, not just the start state (this is because the automaton has to match *every* occurrence of "A B" in the input sequence).

Such a query can be expressed in the $\mathcal{LSEQ}$ model augmented with a positional Union operator.

Theorem 8.3.1 *Given a pattern P represented by a deterministic finite state automaton without cycles, the $\mathcal{LSEQ}$ model can express a continuous match of P over sequence S.*

The proof, which is presented in the appendix, is by constructing a $\mathcal{LSEQ}$ query equivalent to P. The optimization techniques developed for sequences can sometimes prove more efficient in a database context than using regular automata! Consider the pattern "A B" that needs to be continuously matched on a sequence. Figure 30 shows the $\mathcal{LSEQ}$ query for it constructed from the pattern automaton. The $\mathcal{LSEQ}$ representation of the query composes a sequence of B events (derived from the Select "= B" operator) with a sequence of A events offset by one position.

Consider the possibility that the selection condition "= A" is expensive to compute, but is usually satisfied, while the selection condition "= B" is cheap, but rarely satisfied. The regular automaton (since it is memoryless) must check the condition "= A" for every input record. It would therefore incur the cost of a large number of unnecessary but expensive condition checks. In general, for a pattern "A B", each component (A or B) could be either rare or common, and could be either expensive or cheap to compute. This rough classification gives rise to 16 possible permutations. The regular automaton strategy will probably be the best for 4 out of those 16 permutations (for the cases where A is rare and cheap). However, for the remaining 12 cases, its performance may be poor. *The $\mathcal{LSEQ}$ query however can be optimized and executed in the manner best suited to each of those 16 permutations!* At the Compose operator, the choice could be made to first check the condition "= B", and if that condition were satisfied, then check that the previous record satisfied condition "= A".

Chapter 9

Conclusions

To conclude the dissertation, this chapter presents a brief survey of related work in the area of sequence database support. This has been categorized into:

- Systems (commercial and research) that support sequence data.
- Proposals (mostly research) that deal with sequences or temporal data.
- Systems and proposals that support the inclusion of complex data types in a database system.

9.1 Systems Supporting Sequences

Financial management products like MIM [Log] and FAME [FAM90] provide 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. A proprietary time-series management system has been developed at the Union Bank of Switzerland [DDS95], but it is not publicly available. The ODE object-oriented system has an event recognition system that allows complex event patterns to be identified over event streams [GJS92]. These are all examples of special-purpose database support for sequences.

Current general-purpose database systems provide limited support for sequence data. The Order-By clause in SQL only specifies the order in which answers are presented to the user. Most existing support for such data deals with *temporal* sequences. SQL-92 provides a timestamp data type, but there are few constructs that can exploit sequentiality. While many temporal queries can be expressed in SQL-92 using features like correlated subqueries and aggregation, these are typically very inefficient to execute. 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.

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 [Ill94] provides database support for time-series data as an ADT.

9.2 Research Proposals for Sequence Data

Since temporal sequences are among the most commonly occurring sequences in real life, there has been much research on temporal databases. The reader is referred to [Soo91] for a bibliography of recent work. Extensive research on temporal databases has dealt with temporal models [JSS92], algebras [SM89] and query processing [TCG+93]. Most of the work has focused on extending existing data models to support temporal data, or extending existing query languages to allow temporal queries to be expressed. Typically, the relational or object-oriented model is extended by associating a "timestamp" with each tuple or object. The timestamp represents the positions in time at which the tuple or object was "true". The associated query language is extended with predicates that access the timestamp. This is mostly complementary to this thesis because it involves changes to relations and to SQL [Com94]. However, it would be interesting to study how time-ordered sequences can be efficiently converted into relations with time-stamps, and vice-versa.

Research directed at modeling time-series data [SK86, SS87, CS92, SP90, SSV92] provided initial direction to this thesis. The model of a time-series in [SS87] is similar to that of the $\mathcal{LSEQ}$ model, and an SQL-like language was also proposed. Implementation issues were discussed in the context of how the model could be mapped to a relational data model.The Tangram Stream Processor [SP90] uses transducers and stream processing to query sequences in a logic programming context. These ideas were extended with parallelism in the SVP model [SSV92]. The dual nature of sequences (Positional versus Record-Oriented) is recognized by the temporal query language of [WJS93].

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 been languages proposed that model ordered data in addition to sets [DV92], as well as formal languages for list-based complex objects (see [CRSG94] as an example of such a language). Much of the formal work in this area was motivated by [MV93] which stressed the importance of supporting ordered data. There have also been languages proposed to match regular patterns over sequence data [SX92, GJS92]. This work is complementary to this thesis, since SEQ is oriented to more traditional database queries, and currently does not have powerful pattern-matching capabilities. On the other hand, SEQ has explored practical implementation issues that the more theoretical work did not address.

Some of the optimization techniques used in this thesis are similar to other proposals in the literature. For instance, the concept of operator caches is similar to the notion of "working–memory" in [LM93, SP90]. [SP90] studies stream processing techniques using a logic language as the underlying data engine. While the emphasis was not on a cost-based query optimization for sequence queries in the database context, the emphasis on stream processing is similar. [GS89a] argues, as does this thesis, that special statistical information needs to be maintained for temporal data. However, research on optimization and processing strategies has been relatively limited. An extension of a relational query optimizer to handle scientific data was presented in [WG93]. There has been some work on query optimization based on temporal relational data models[GS89b, LM93, NG93]. For instance [GS91, LM93] propose efficient stream access techniques of processing various types of temporal "joins", and [GS89a] proposes an optimization framework for temporal data based on such techniques. This work is of significance in supporting the Record-Oriented sequence operators of Chapter 2. In terms of physical implementation, various access structures have been proposed for temporal relational data[EWK90, LS89] and some performance studies have been conducted[AS86, EKW90].

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 [APWZ95], 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.

9.3 Complex Data Types

There is much research work related to the E-ADT paradigm. The issues regarding support for ADTs in database systems were first explored in [SRG83] and [Sto86]. This work lead to the development of the Postgres research DBMS [SRH90] and its commercial incarnation, Illustra [Ill94]. More recently, several commercial database systems including DB2 C/S V2 [Cha96] have begun providing some support for type extensibility. Among research systems, the Paradise client-server DBMS [DKL+94] is exploring issues related to the development of ADTs for spatial, geographic, and scientific data. The problems being addressed include the parallelization of operations over ADTs like scientific arrays [TD96]. Since the traditional ADT support for functions and methods proves insufficient when one wants to parallelize query execution, the Paradise system explores extensions to ADTs that enhance parallelism. From the E-ADT viewpoint, parallel execution is simply yet another possible benefit of declarative ADT expressions. Consequently, the work on parallel execution of ADT methods fits in neatly with the E-ADT paradigm, and would perhaps even be easier to develop within a system based on E-ADTs.

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 idea of enhancing ADTs with query language and query processing capabilities seems to be unique to PREDATOR. The loosely-coupled architecture with multiple top-level collection types with different query languages also appears to be novel. The notion of breaking a query into regions, each of which is optimized separately is similar in flavor to the "blackboard" architecture for relational query optimizers proposed in [MDZ93]; however, that work was not directed at supporting operations on complex data types.

As an interesting counterpoint to the approach taken in this thesis, there have been efforts that try to find a holistic solution to the problem of querying new data types. Instead of breaking a query into many parts with local query optimization, these approaches try to find a global integrated solution. Obviously, this requires that the entire query be modeled somehow in an integrated framework. AQUA [SLVZ95] is an algebraic framework that has been proposed for this purpose, while CPL/Kleisli [Won94] is a framework based on comprehensions as a query language and a higher-order monadic algebra. The primary difficulty is that the integrated solution must be capable of modeling all possible queries and query processing strategies for all possible data types. There is ongoing research on both AQUA and Kleisli to address this and other issues.

9.4 Conclusions

The thesis has proposed a novel database system paradigm that allows several types of complex data to be efficiently supported within a DBMS. The PREDATOR database system has been built to demonstrate this E-ADT paradigm, and several implementation issues have been discussed in the thesis. The motivation for the research on E-ADTs was the need to support queries over sequence data; indeed sequences are the primary example implemented to demonstrate the E-ADT concept. Support for sequence data has been investigated at various levels. The $\mathcal{SEQ}$ data model was developed as a framework for queries over sequences. The $\mathcal{SEQUIN}$ query language provided the syntax for queries. Various optimization strategies were proposed, based primarily on the concept of the scope of sequence query operators. Finally, the implementation of the SEQ component of PREDATOR demonstrated the effects of these optimizations. The interaction between sequences and relations in PREDATOR has further demonstrated that the E-ADT paradigm is practical, and that it leads to significant performance improvements. Based on these results, the conclusion is that current database systems should take steps to adopt the E-ADT design concept. The thesis has suggested migration paths to help them move in this direction.

Bibliography

[APWZ95] Rakesh Agrawal, Giuseppe Psaila, Edward Wimmers, and Mohammed Zait. Querying Shapes of Histories. In *Proceedings of the Twenty First International Conference on Very Large Databases (VLDB), Zurich, Switzerland*, 1995.

[AS86] Ilsoo Ahn and Richard Snodgrass. Performance Evaluation of a Temporal Database Management System. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 96–107, 1986.

[BBG+86] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, C. Twichell, and T. Wise. GENESIS: A Reconfigurable Database Management System. Technical Report TR-86-07, University of Texas at Austin, March 1986.

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

[Cat94] R.G.G. Cattell. *The Object Database Standard:ODMB-93.* Morgan-Kaufman, 1994.

[CCH+95] M. Carey, W. Cody, L. Haas, W. Niblack, M. Arya, R. Fagin, M. Flickner, D. Lee, D. Petkovic, P. Schwarz, J. Thomas, M. Tork-Roth, J. Williams, and E. Wimmers. Querying Multimedia Data From Multiple Repositories By Content: The Garlic Project. *Proceedings of the IFIP Working Conference on Visual Database Systems, Lausanne, Switzerland*, March 1995.

[CCS93] E.F. Codd, S.B. Codd, and C.T. Salley. Beyond Decision Support. *ComputerWorld*, 27(30), July 1993.

[CDF+93] 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 *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, pages 526–541, 1993.

[CDG+90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In *Readings in Object-Oriented Databases.* Morgan-Kaufman, 1990.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *"Proceedings of the IPSJ Conference, Tokyo, Japan"*, October 1994.

[Cha96] Don Chamberlin. *Using the New DB2.* Morgan Kaufmann, 1996.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Com94] TSQL2 Language Design Committee. TSQL2 Language Specification. *ACM SIGMOD Record*, 23(1):65–86, March 1994.

[CR86] James Clifford and Ahobala Rao. A Simple, General Structure for Temporal Domains. CRIS 154, Center for Research on Information Systems, Stern School of Business, New York University, October 1986.

[CRSG94] L.S. Colby, E.L. Robertson, L.V. Saxton, and D.V. Gucht. A Query Language for List-Based COmplex Objects. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Minneapolis, MN*, pages 179–189, 1994.

[CS92] Rakesh Chandra and Arie Segev. Managing Temporal Financial Data in an Extensible Database. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 238–249, 1992.

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

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[DDS95] Werner Dreyer, Angelika Kotz Dittrich, and Duri Schmidt. Using the CALANDA Time Series Management System. In *Proceedings of ACM SIGMOD '95 International Conference on Management of Data, San Jose, CA*, 1995.

[DKL+94] D.J. DeWitt, N. Kabra, J. Luo, J.M. Patel, and J. Yu. Client-server paradise. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB), Santiago, Chile*, September 1994.

[DKS92] W. Du, R. Krishnamurthy, and M.C. Shan. Query Optimization in Heterogeneous Database Management Systems. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), Vancouver, Canada*, pages 277–291, 1992.

[DNS91] David DeWitt, Jeffrey Naughton, and Donovan Schneider. An Evaluation of $Non-Equijoin$ Operations. In *Proceedings of the International Conference on Very Large Databases*($VLDB$), 1991.

[DV92] S. Danforth and P. Valduriez. A FAD for Data-Intensive Applications. *IEEE Transactions on Knowledge and Data Engineering*, 4(1), February 1992.

[EKW90] Ramez Elmasri, Yeung-Joon Kim, and Gene Wuu. Efficient Implementation Techniques for the Time Index. In *Proceedings of the Seventh IEEE Data Engineering Conference*, April 1990.

[ES94] Ramez Elmasri and S.B.Navathe. *Fundamentals of Database Systems.* Benjamin/Cummins Publishing Co., 1994.

[EWK90] Ramez Elmasri, Gene Wuu, and Yeung-Joon Kim. The Time Index : An Access Structure for Temporal Data. In *Proceedings of the International Conference on Very Large Databases*($VLDB$), pages 1–12, 1990.

[fADF90] The Committee for Advanced Database Function. Third-Generation Database System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, 1990.

[FAM90] FAME Software Corporation, 6869 Marshall Road, Dexter, MI 48130. *User's Guide to FAME*, 1990.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, pages 419–430, May 1994.

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

[GJS92] N.H Gehani, H.V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 327–338, 1992.

[GM81] J. Grant and J. Minker. Optimization in Deductive and Conventional Relational Database Systems. In H. Gallaire, J. Minker, and J. M. Nicolas, editors, *Advances in Data Base Theory*, volume 1, pages 195–234. Plenum Press, New York, N.Y., 1981.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth IEEE Conference on Data Engineering, Taipei, Taiwan*, 1993.

[GS89a] 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.

[GS89b] Himawan Gunadhi and Arie Segev. Event-join Optimization in Temporal Relational Databases. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, 1989.

[GS91] Himawan Gunadhi and Arie Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of the International Conference on Data Engineering*, 1991.

[HCL+90] L. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[Hel95] Joseph M. Hellerstein. *Optimization and Execution Techniques for Queries With Expensive Methods*. PhD thesis, University of Wisconsin, August 1995.

[HS86] W.D. Hillis and G. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12), December 1986.

[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.

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

[ISO93] ISO-ANSI. *ISO-ANSI Working Draft: Database Language SQL2 and SQL3*, 1993. X3H2; ISO/IEC JTC1/SC21/WG3.

[JSS92] C.S. Jensen, M.D. Soo, and R.T. Snodgrass. Unification of Temporal Relations. Technical Report 92-15, Computer Sciences Department, University of Arizona, July 1992.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Non-Recursive Queries. In *Proceedings of the Twelfth International Conference on Very Large Databases*, pages 128–137, 1986.

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7, September 1982.

[Klu82] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[LL92] Charles LeBeau and David Lucas. *Computer Analysis of the Futures Market*. Irwin Professional Publishing, 1992.

[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, California, February 1990.

[LM92] Cliff Leung and Richard Muntz. Temporal Query Processing and Optimization in Multiprocesor Database Machines. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), Vancouver, Canada*, pages 383–394, 1992.

[LM93] Cliff T.Y. Leung and Richard R. Muntz. *Stream Processing: Temporal Query Processing and Optimization*, chapter 14. Benjamin/Cummings, 1993.

[LMW96] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data, Montreal, Canada*, 1996.

[Log] Logical Information Machines. *MIM User Manual*.

[LS89] David Lomet and Betty Salzberg. Access Methods for Multiversion Data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 315–324, 1989.

[MDZ93] Gail Mitchell, Umeshwar Dayal, and Stanley Zdonik. Control of an Extensible Query Optimizer: A Planning-Based Approach. In *Proceedings of the Nineteenth International Conference on Very Large Databases (VLDB), Dublin, Ireland*, pages 517–528, 1993.

[Moh95] C. Mohan. An Overview of the Exotica Research Project on Workflow Management Systems. In *Proceedings of the 6th International Workshop on High Performance Transaction Systems, Asilomar*, September 1995.

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

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

[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.

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

[RS91] Joel Richardson and Peter Schwartz. Aspects: Extending Objects to SUpport Multiple Independent Roles. In *Proceedings of ACM SIGMOD '91 International Conference on Management of Data, Denver, CO*, 1991.

[SAC+79] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD '79 International Conference on Management of Data*, pages 23–34, 1979.

[Sel88] T. Sellis. Multiple Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[SHR+96] P. Seshadri, J. Hellerstein, R. Ramakrishnan, H. Pirahesh, T.Y.C. Leung, D. Srivastava, S. Sudarshan, and P. Stuckey. Cost-Based Optimization for Magic: Algebra and Implementation. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data, Montreal, Canada*, 1996.

[SK86] Arie Shoshani and Kyoji Kawagoe. Temporal Data Management. In *Proceedings of the Twelfth International Conference on Very Large Databases*, pages 79–88, 1986.

[SLR94] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence Query Processing. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, May 1994.

[SLR95] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proceedings of the Eleventh IEEE Conference on Data Engineering, Taipei, Taiwan*, pages 232–239, March 1995.

[SLR96a] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Case for Enhanced Abstract Data Types. Technical Report TR-1319, University of Wisconsin-Madison, Computer Sciences Department, July 1996.

[SLR96b] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of the Twenty Second International Conference on Very Large Databases (VLDB), Bombay, India*, September 1996.

[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 Eleventh IEEE Conference on Data Engineering, Taipei, Taiwan*, March 1995.

[SM89] Richard Snodgrass and Edwin McKenzie. An Evaluation of Algebras Incorporating Time. Technical Report TR-89-22. Computer Science Department. University of Arizona., September 1989.

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

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

[SPL96] Praveen Seshadri, Hamid Pirahesh, and Cliff Leung. Complex Query Decorrelation. In *Proceedings of the Twelfth IEEE Conference on Data Engineering, New Orleans, Louisiana*, February 1996.

[SRG83] M. Stonebraker, B. Rubenstein, and A. Guttman. Application of Abstract Data Types and Abstract Indices to CAD Data Bases. In *Proceedings of the Engineering Applications Stream of Database Week*, San Jose, CA, May 1983.

[SRH90] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[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.

[SS94] Sunita Sarawagi and Michael Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Proceedings of the Tenth IEEE Conference on Data Engineering*, 1994.

[SSV92] D. Stott Parker, Eric Simon, and Patrick Valduriez. SVP - a Model Capturing Sets, Streams, and Parallelism. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), Vancouver, Canada*, pages 115–126, 1992.

[Sta91] Statistical Sciences, Inc., Seattle, WA. *S-Plus User's Manual*, 1991.

[Sto84] Michael Stonebraker. QUEL as a Data Type. In *Proceedings of ACM SIGMOD '84 International Conference on Management of Data, Boston, MA*, June 1984.

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

[SX92] S.Ginsburg and X.Wang. Pattern Matching by Rs-operations: Towards a Unified Approach to Querying Sequenced Data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1992.

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

[TD96] Kristin Tufte and David J. DeWitt. Design and Implementation of HDF in the Paradise DBMS. Copy from kristint@cs.wisc.edu, 1996.

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

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

[WG93] Richard Wolniewicz and Goetz Graefe. Algebraic Optimization of Computations over Scientific Databases. In *Proceedings of the Nineteenth International Conference on Very Large Databases (VLDB), Dublin, Ireland*, pages 13–24, 1993.

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

[Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, U.Pennsylvania, 1994.

Appendix A

Appendices

A.1 E-ADT $C++$ Interface

```
class EADT {
public:
// This serves as an index into the global table of ADTs!!
virtual BasicType TypeId() const;
// This is the name of the type (used while parsing schemas)
virtual char* TypeName() const;

// Maximum "size" of any object of this type, given the specific
// meta-information. This size is the maximum number of bytes needed
// to store it in a memory location. This is used to allocate
// memory locations to hold the results of an expression returning a
// value of this E-ADT. If the objects are really large, they will not
// be stored directly. Instead, they will be stored separately, and an
// ID for them will be stored in the specified memory location.
// The object size in this case is the size of the id. This decision
// (id or not) is made by the EADT.
virtual int MaxObjectSize(const void* MetaInfo) const;
// size of a particular object of this type: when stored in a tuple
virtual int TypeSize(const char* Object, const void* MetaInfo) const

// Extract an object of this Type from ObjectRef into the
// preallocated result object. Increments the ref count, if only
// a handle is copied. Returns size of resultant object.
virtual int TypeCopy(const char* ObjectRef, char *ResultObject,
                     const void* MetaInfo) const;
// Makes a copy of the ObjectRef and allocates memory for the new copy.
virtual char* TypeCreate(const char* ObjectRef,
                         const void* MetaInfo) const;
// Makes a copy that can be stored (in memory representations need to
// be converted to disk representations).
virtual int TypeStoreCopy(const char* ObjectRef, char *ResultObject,
                          const void* MetaInfo) const;
// Destroy an object of the E-ADT.
// This could involve decrementing ref counts.
```

```
virtual void TypeDestroy(char *Object, const void* MetaInfo) const;
// Should a type object be stored in its record field, or should it be
// put at the end of the record with an in-record pointer to it instead?
virtual Boolean StoreInField() const;

// Check for equality between two objects of this E-ADT
virtual Boolean Equals(const char* Obj1, const char* Obj2,
                       const void* MetaInfo1,
                       const void* MetaInfo2) const;

//
//  Textual I/O for the E-ADT
//
// Make a readable text string version of the object.
virtual ErrCode WriteText(ostream& OutStream, const char* Object,
                          const void* MetaInfo) const;
// Read a text version of the object into memory allocated for it.
virtual ErrCode ReadText(istream& InStream, char*& ObjectRef,
                         const void* MetaInfo) const;

//
// Catalog maintenance for objects of this E-ADT. Note that the
// meta-information is interpreted solely by this E-ADT.
//
// Does the EADT maintain catalogs or not.
virtual Boolean MaintainsCatalogs() const;
// Find meta-information for a specific named object
virtual Boolean GetMetaInfo(const char* ObjName, void* &MetaInfo) const;
// Copy the meta-information.
virtual ErrCode CopyMetaInfo(void* MetaInfo, void*& Copy) const;
// Ask the E-ADT to reclaim the meta-information.
virtual ErrCode ReclaimMetaInfo(void* MetaInfo) const;
// Associate Name in the catalogs with the meta-information
// provided in textual form.
virtual ErrCode AddMetaInfo(const char* Name, const XxxBool Named,
                            const char* MetaInfoString);

//
//  Support for functions/methods on E_ADTS. Three E-ADT
//  methods are provided that act as ‘‘meta-functions’’
//
// Function that provides type checking for allowed methods
// of the ADT. During type checking, the return parameter is
// ReturnType. To allow for variable methods (methods with the
```

```
// same name but either with different type signatures), the actual
// expression list must also be passed. For such cases, it is
// usually sufficient in PL to pass the list of argument _types_.
// However, in our case, we must also support function invocations
// in which the first argument is a constant expression string, and
// the function type signature depends on what is in the string.
// Therefore, the actual list of constant argument expressions needs to
// be passed in. A generic parse structure is returned, and this can be
// queried for both the return type and the meta-information.
virtual ErrCode FuncTypeCheck(const char* FuncName,
                              ValueExprList *ArgList,
                              int NumArgs,
                              BasicType& ReturnType,
                              GenericParse*& FuncParse) const;

// Function that optimizes method execution. This takes the
// generic parse structure and returns a generic plan.
// Therefore, it is only this E-ADT that actually interprets the
// generic parse structure.
virtual ErrCode FuncOptimize(const GenericParse* FuncParse,
                             GenericPlan*& FuncPlan) const;

// Function that executes all allowed methods for this ADT
// If this function is applied to some object, the object is
// passed as the first argument.
virtual ErrCode FuncExec(const GenericPlan *FuncPlan,
                         ValueEnv* ValueEnv,
                         void *ReturnValue) const;

//
//  Support for cast functions from one E-ADT to another
//
// Perform type checking for type cast.
virtual XxxBool CastCheck(const BasicType CastType,
                          const void* MetaInfo) const;
// Execute type cast.
virtual XxxErrCode CastExec(BasicType CastType, const void* MetaInfo,
                            char* OldObject, char* NewObject) const;

//
// Support for a declarative language interface
//
// This is the name of the language supported (example, ‘‘SQL’’)
virtual char* LanguageName() const;
```

```
// Function that parses parameterized expressions in the ADT language.
// It returns a data structure that contains a parsed representation
// of the ADT expression. It also specifies the return type of the
// expression.
virtual ErrCode ExprParse(const char* ExprString,
                          const ArgEnv* ArgEnv,
                          BasicType& ReturnType,
                          GenericParse*& ParseStructure);
// Function that optimizes a parsed expression in the ADT language.
// It returns a data structure that contains the optimized execution
// plan (or plan identifier) for the ADT expression.
virtual ErrCode ExprOptimize(GenericParse* ParseStructure,
                             GenericPlan*& PlanStructure);

// Function that evaluates an optimized plan.
virtual ErrCode ExprEvaluate(GenericPlan* PlanStructure,
                             const ValueEnv* ArgEnv,
                             void *ReturnValue);
};
```

A.2 Proofs of Equivalence Transformations

The informal proofs of the equivalence transformations of Section 3.2.3 are now presented. Most of these transformations are analogous to well-known transformations of relational queries [ES94], and the proofs follow from the proofs of the corresponding relational transformations.

- Rule 1: Analogous to splitting relational selections.
- Rule 2: Analogous to splitting relational projections.
- Rule 3: Analogous to relational projection pushdown through selection.
- Rule 4: Analogous to relational projection pushdown.
- Rule 5: Analogous to relational selection pushdown through a join.
- Rule 6: Analogous to relational join commutativity.
- Rule 7: Analogous to relational join associativity.
- Rule 8: Since Op.OpFunc is independent of p, it is only dependent on the records in the input scopes. Since the scope positions are defined as constants added to the current position, a phase shift in the input records is equivalent to a phase shift in the output records.

A.3 Proof of Regular Expression Theorem

Proof: Theorem 8.3.1 is proved in two parts. The first part is an algorithm that constructs a $\mathcal{SEQ}$ query based on the deterministic automaton DFA. The algorithm is described as a graphical construction of the $\mathcal{SEQ}$ query and is best understood by applying each step to the simple pattern automaton in Figure 30. Consider a deterministic finite automaton with no cycles. Let the pattern be expressed as a star-free regular expression from the alphabet of symbols $\{a_0, a_1, ..., a_l\}$. Let the states be $\{S_0, S_1, ..., S_m\}$ and the labeled transitions between states be $\{e_0, e_1, ..., e_n\}$ where each transition e_i is a triple $< S_{from}, S_{to}, a_{input} >$. S_0 is the start state.

Step 1 Create a node N_i corresponding to each state S_i of the DFA. Each node will correspond to a sequence.

Step 2 For each transition $e_t =< S_0, S_j, a_k >$ out of the start state, draw an edge from node N_0 to a newly created Select operator Op with selection condition "$= a_k$". Draw an edge from the output of the Op to node N_j. (This step is applied to transition $< Start, Partial, A >$ in the pattern automaton of Figure 30.)

Step 3 For every other transition $e_t =< S_i, S_j, a_k >$ out of the start state, draw an edge from node N_0 to a newly created Select operator Op1 with selection condition "$= a_k$". Draw an edge from the output of Op1 to a newly created Compose operator Op2. Draw an edge from N_i (corresponding to state S_i) to a newly created Offset -1 operator Op3. Draw an edge from the output of Op3 to the other input of the Compose operator Op2. Finally draw an edge from the output of Op2 to the node N_j (corresponding to state S_j). (This step is applied to transition $< Partial, Match, B >$ in the pattern automaton of Figure 30.)

Step 4 For any node N_i with more than one incoming edge, introduce a newly created Union operator Op and redirect all the incoming edges to point to Op. Draw an edge from the output of Op to node N_i. This ensures that each node has no more than one incoming edge. If there is only one final state S_f, then the output edge from the corresponding node N_f is the result of the query. If there is more than one final state, create a new Union operator Op. For every final state S_f, draw an edge from S_f to Op. The output of Op is the result of the query. Delete all nodes, edges and operators that do not lie on some path from the start node N_0 to a node N_f corresponding to some final state S_f. For every node N_i, remove N_i and connect its input edge to its output edges. Replace node N_0 by the input sequence.

The second part of the proof has the following stages:

- It shows that every state in the pattern DFA corresponds to a derived sequence in the $\mathcal{SEQ}$ query. State S_i corresponds to the derived sequence represented by the edge formed on removing N_i.
- It shows that if there is a pattern in the input sequence starting at position p_1 and ending at position p_2 that is in the language of the pattern DFA, then the query answer sequence will have a record at position p_2.
- It shows that if the query answer sequence has a record at a position p_2, then there is some earlier position p_1 such that the input sequence from p_1 to p_2 is a string in the language of the pattern DFA.
- It performs an inductive proof on the length of the input pattern matched.

□