Query Processing in Temporal Object-Oriented Databases

A thesis submitted to Middlesex University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

Lichun Wang

School of Computing Science
Middlesex University

July 1999

Abstract

This PhD thesis is concerned with historical data management in the context of object-oriented databases. An extensible approach has been explored to processing temporal object queries within a uniform query framework. By the uniform framework, we mean temporal queries can be processed within the existing object-oriented framework that is extended from relational framework, by extending the existing query processing techniques and strategies developed for OODBs and RDBs.

The unified model of OODBs and RDBs in UniSQL/X has been adopted as a basis for this purpose. A temporal object data model is thereby defined by incorporating a time dimension into this unified model of OODBs and RDBs to form temporal relational-like cubes but with the addition of aggregation and inheritance hierarchies. A query algebra, that accesses objects through these associations of aggregation, inheritance and time-reference, is then defined as a general query model /language. Due to the extensive features of our data model and reducibility of the algebra, a layered structure of query processor is presented that provides a uniform framework for processing temporal object queries. Within the uniform framework, query transformation is carried out based on a set of transformation rules identified that includes the known relational and object rules plus those pertaining to the time dimension. To evaluate a temporal query involving a path with time-reference, a strategy of decomposition is proposed. That is, evaluation of an enhanced path, which is defined to extend a path with time-reference, is decomposed by initially dividing the path into two sub-paths: one containing the time-stamped class that can be optimized by making use of the ordering information of temporal data and another an ordinary sub-path (without time-stamped classes) which can be further decomposed and evaluated using different algorithms. The intermediate results of traversing the two sub-paths are then joined together to create the query output. Algorithms for processing the decomposed query components, i.e., time-related operation algorithms, four join algorithms (nested-loop forward join, sort-merge forward join, nested-loop reverse join and sort-merge reverse join) and their modifications, have been presented with cost analysis and implemented with stream processing techniques using C++. Simulation results are also provided. Both cost analysis and simulation show the effects of time on the query processing algorithms: the join time cost is linearly increased with the expansion in the number of time-epochs (time-dimension in the case of a regular TS). It is also shown that using heuristics that make use of time information can lead to a significant time cost saving. Query processing with incomplete temporal data has also been discussed.

Acknowledgement

I would like to acknowledge many people for their roles in making this thesis a reality.

Firstly, I am indebted to my supervisor M. Wing and the director of studies C. Davis for their consistent support and supervision. Secondly, I would like to thank N. Revell and B. Eagelstone for their useful advice and suggestions on my PhD work. Thirdly, I would like to thank F. M. Carrano for the useful discussions on C++ stream processing techniques and I/O cost. Fourthly, I would like to acknowledge H. Thimbleby, A. Blandford, K. Dudman and M. Jones for the useful discussions or carefully reading the thesis. Many thanks are also due to L. Miraziz, J. Tsang, S. Albassan, J. Macdonald and S. Griffin for their help and support.

Finally, I would like to thank my parents, husband and son for their patience and support while I have been doing my PhD.

Contents

Abstract ii

Acknowledgement iii

Contents iv

1 Introduction 1

1.1 Background 1

1.2 Research Project 4

1.2.1 Presentation of Problem 4

1.2.2 Objectives of the Project 5

1.3 Outline of Main Work 6

1.4 Summary of Related Work 11

1.5 Organisation of the Thesis 13

2 Relational, Object-Oriented and Temporal Databases 16

2.1 Introduction 16

2.2 Relational Databases 18

2.2.1 What Is a Relational Database? 18

2.2.2 The Relational Model of Data 18

2.2.3 Strengths and Weaknesses of Relational Databases 22

2.3 Object-Oriented Databases 23

2.3.1 What Is an Object-Oriented Database? 23

2.3.2 An Object-Oriented Data Model and Query Language 24

2.3.3 Why Object-Oriented Databases? 28

2.3.4 Approaches to Object-Oriented Databases 29

2.3.5 Strengths and Weaknesses of Object-Oriented Databases 30

2.4 Temporal Databases 31

2.4.1 Time Dimensions 32

2.4.2 Research on Temporal Databases 33

2.5 Summary 37

3 Query Processing in Databases 42

3.1 Introduction 42

3.2 Query Processing in Relational Databases 43

3.2.1 Optimization Objectives 43

3.2.2 General Processing Strategies 45

3.2.2.1 Query Presentation 45

3.2.2.2 Steps in Query Processing 47

3.2.2.3 General Processing Strategies 48

3.2.3 Optimization Techniques 49

3.2.3.1 Algebraic Optimization 50

(1) Query Transformation 50

(2) Query Improvement 54

3.2.3.2 Plan Optimization 54

(1) Query Evaluation 54

(2) Access Plans 62

3.2.3.3 Semantic Optimization 63

3.3 Query Processing in Object-Oriented Databases 64

3.3.1 Query Models 65

3.3.1.1 A Model of Queries for Object-Oriented Databases 65

3.3.1.2 A Query Algebra for Object-Oriented Databases 66

3.3.2 Query Processing Methodology 68

3.3.3 Optimization Techniques 70

3.3.3.1 Algebraic Optimization 70

3.3.3.2 Path Execution 72

3.3.3.3 Semantic Query Optimization 74

3.4 Processing Temporal Queries 75

3.5 Summary 77

4 A Temporal Object Data Model 79

4.1 Introduction 79

4.2 The Unified Data Model of RDB and OODB 80

4.3 A Temporal Object Data Model 85

4.3.1 Time Space and Temporal Set 86

4.3.2 Chronon, Interval, Span and Lifespan 86

4.3.3 Integrating the Temporal Object with the Unified Model of RDB and OODB 89

4.4 More Application Examples 92

4.4.1 Case Study 1: "The Wood Panel Deformation Measurement Database" 92

4.4.2 Case Study 2: "The Neurological Patient Care Database" 95

4.5 Features of the Temporal Object-Oriented Data Model 100

4.6 Summary 102

5 Query Algebra for the Temporal Object Data Model 104

5.1 Introduction 104

5.2 Predicates 106

5.3 Identity and Equality 109

5.4 Closure 110

5.5 Query Algebra 110

5.5.1 Temporal Unary Set Operations 110

5.5.2 Binary Set Operations 112

5.5.3 Special Operations 114

5.6 Query Examples 120

5.7 Properties of Algebra 122

5.8 Summary 124

6 A Uniform Framework for Processing Temporal Object Queries 126

6.1 Introduction 126

6.2 Optimizer Layering 128

6.3 Query Transformation 129

6.3.1 Relational Rules 130

6.3.2 Temporal Transformation Rules 136
6.3.3 Inheritance Rules 140
6.3.4 Path Transformation Rules 141
6.4 A Decomposition Strategy for Processing Temporal Object Queries 144
6.5 Summary 147

7 Algorithms for Processing Decomposed Query Components 149
7.1 Introduction 149
7.2 Assumptions 151
7.3 Stream Processing Algorithms for Time-Related Operations 152
7.3.1 Stream Processing Algorithm for Time-slice 153
7.3.2 Stream Processing Aggregation Algorithms 155
7.4 Join Algorithms 156
7.4.1 Nested-Loop Forward Join (NLFJ) 157
7.4.2 Sort-Merge Forward Join (SMFJ) 158
7.4.3 Nested-Loop Reverse Join (NLRJ) 159
7.4.4 Sort-Merge Reverse Join (SMRJ) 160
7.4.5 Sorting 162
7.4.6 Summary 163
7.5 Modification of Join Algorithms 164
7.6 Simulation 166
7.6.1 Experimental Database:
"The International Weather Record Database" 167
7.6.2 Simulation Programs 167
7.6.3 Simulation Results and Discussion 171
7.7 Heuristics for Optimization 173
7.8 Summary 175

8 Query Processing with Incomplete Temporal Data 189
8.1 Introduction 189
8.2 Implications of Temporal Data 190

8.3 Interpolation 193
8.4 Query Processing with Incomplete Temporal Data 202
8.5 Summary 204

9 Conclusions and Further Work 205
9.1 Major Contributions of the Work 205
9.2 Conclusions 207
9.3 Future Work 212

List of Figures 214
List of Tables 216

List of Author's Publications Relevant to the Thesis 217

References 219

APPENDIX Selected Published Papers 234
A.1 Processing Temporal Queries in the Context of Object-Oriented Databases
A.2 An Algebra for a Temporal Object Data Model

Chapter 1

Introduction

This introductory chapter outlines the research background, main work, related work and organisation of thesis.

1.1 Background

We are living in a state of information explosion. Organisations have made extensive use of the ability of database management systems to store and manipulate vast amounts of data and knowledge.

There are three major types of databases available today: relational databases (RDBs), object-oriented databases (OODBs) and recently emerged object-relational databases (ORDBs).

Whilst the semantic limitations of RDBs are widely recognised, OODBs have emerged to represent some of the most promising ways of meeting the demands of the advanced database applications, such as computer-aided design (CAD), computer-aided manufacturing (CAM), computer-integrated manufacturing (CIM), computer-aided software engineering (CASE), document and multimedia preparation, office automation and scientific computing.

However, most OODBs offer more restricted query capabilities than those found in RDBs [Kim, 1993; 1994; 1995; Kim *et al.,* 1997]. Typically, the query facilities do not include nested subqueries, set queries (Union, Intersection, Difference), aggregation functions and group by, and even joins of multiple classes, etc., which are fully supported in RDBs [Kim, 1993; 1994; 1995; Kim *et al.,* 1997]. Most OODB designers have not invested a great deal of time in design and development of appropriate

techniques for optimizing object-oriented queries. In other words, OODBs allow the users to create a flexible database schema and populate the database with many instances, but they do not provide a powerful enough means of retrieving objects from the databases.

In addition, query processing techniques have not fully addressed the features arising from key challenges including interoperating OODBs with RDBs, management of multimedia data, spatial data and temporal data, etc., [Kim, 1993; 1994; 1995; Kim *et al.*, 1997].

For example, temporal properties play an essential role in many real world applications. Many advanced database applications require the support for time-varying data. Support for time-varying data in most commercial databases is almost at the level of user-defined time. The vast majority of research on temporal databases is on relational and pseudo-relational database models to incorporate time [Pissinou *et al.,* 1993; 1994; Clifford *et al.*, 1993; Gadia, 1988; Griffiths and Theodoulidis, 1996; Snodgrass, 1995; Stonebraker *et al.,* 1990; Tansel *et al.,* 1993; Tansel and Tin, 1998]. Compared to temporal relational data models, little work has been reported on time in OODBs [Pissinou *et al.,* 1993; Snodgrass, 1995; Ozsu and Szafron, 1998]. Although most OODB proposals include constructors for complex types like lists and arrays that allow the time-stamped entity to be represented as a "blob", which is managed by the system, but interpreted solely by the application program, no facilities for temporal queries are provided [Seshadri *et al.,* 1996]. Only a limited amount of work has been done on temporal query processing and optimization [Leung and Muntz, 1993; Dayal and Wuu, 1992; Seshadri *et al.,* 1996; Pissinou *et al.,* 1994; Snodgrass, 1995; Zurek, 1998] and they are almost all in the context of relational databases.

Query processing remains one of the most difficult problems to be addressed by researchers and developers of OODBs [Ozsu and Blakeley, 1995; Kotz-Dittrich and Dittrich, 1995; Straube and Ozsu, 1995; Yu and Meng, 1998], especially when time-varying data are taken into account.

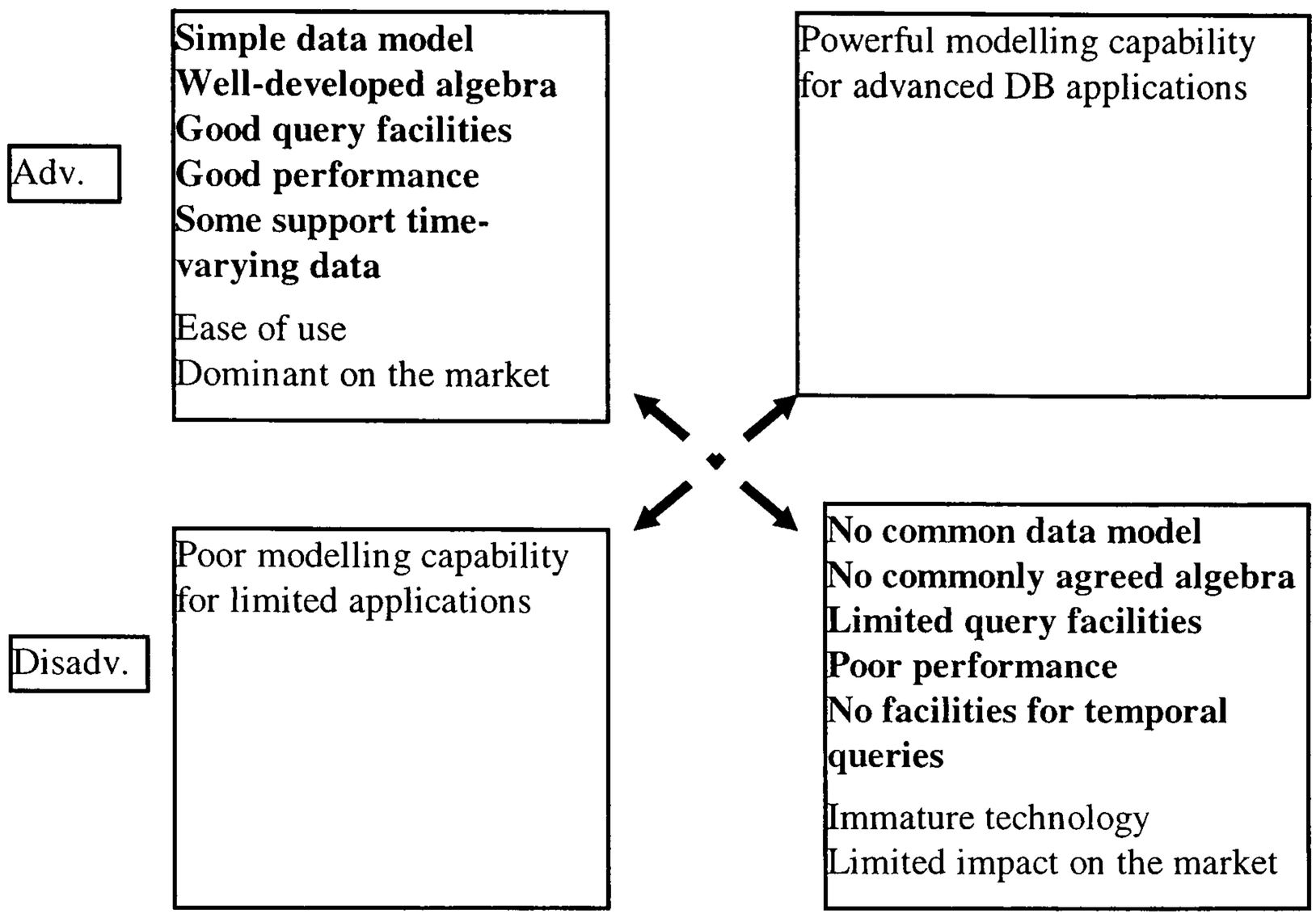


Figure 1-1 Features in OODBs and RDBs.
Items listed in bold are concerned with the query processing related issues.

The features of both RDBs and OODBs, in terms of query processing, are listed in **Figure 1-1.** Obviously OODBs and RDBs are complementary, i.e., the drawbacks of OODBs can be compensated to some extent by the merits of RDBs. It is believed that the basis of the next generation database technology will be the combination of relations and objects [Kim, 1993; 1994; 1995; Kim *et al.,* 1997; Date, 1994; Date, 1996; Date and Darwen, 1998; Darwen and Date, 1995; Stonebraker, 1996; Eisenberg and Melton, 1999]. That is, RDBs will be extended to incorporate the concepts of encapsulation (methods), arbitrary data types, nested objects, and inheritance; or alternatively OODBs will be extended to incorporate an ANSI-SQL compatible non-procedural query language and all the major database features found in today's RDBs (such as, automatic query optimization, etc.). Beyond the unification of relational and object-oriented database technologies, the database field has to address the aforementioned key

challenges such as management of temporal data, etc., to usher in a true revolution in database technology.

Databases thereby emerged that support a dialect of SQL-3, include non-traditional tools, and optimize for complex SQL-3 queries are called object-relational DBMSs [Stonebraker, 1996; 1998]. They are relational in nature because they support SQL; they are object-oriented in nature because they support complex data. In essence they are a marriage of SQL from the relational world and modelling primitives from object world. Examples of object-relational DBMS vendors are UniSQL, Illustra, Omniscience, Hewlee-Packard (with their Odapter for Oracle, also packaged for HP's own Allbase/SQL as Open ODB) [Stonebraker, 1996; 1998], etc.

This thesis will in general not distinguish object-relational from object-oriented databases unless they are explicitly stated, i.e., it accepts the current commonly used classification: object-relational databases are still under the category of object-oriented databases.

1.2 Research Project

The project "Query Processing in Temporal Object-Oriented Databases" is addressing the difficult challenge of management of temporal data. More specifically, it is concerned with historical data management in the context of object-oriented databases and involves the study and development of novel methods for processing temporal object queries. The project initially started in 1993, and it has gone through an intensive research so far and resulted in 13 publications as shown in the list of author's publication at the end of the thesis.

1.2.1 Presentation of Problem

Query optimization techniques are dependent upon the query model or language. The query model, in turn, is based on the data (or object) model since the latter defines the

access primitives that are used by the query model. These primitives, at least partially, determine the power of the query model.

Almost all the object query processors proposed to date uses the optimization techniques developed for RDBs [Ozsu and Blakeley, 1995; Bertino and Martino, 1993; Yu and Meng, 1998]. But these OODBs introduce many new data model features to the database, that cannot be handled by traditional query processing technology (i.e., the query processing subsystems of RDBs). The difficulty is that there is no commonly accepted object data model and there is thereby no universally agreed query algebra on which to base development of the theory and architecture which would provide the basis for optimization [Bertino and Martino, 1993; Ozsu and Blakeley, 1995]. If we have a unified data model of OODBs and RDBs or an object data model that is extended from relational data model, the techniques of query processing and optimization developed for RDBs will be more smoothly and easily extended for OODBs.

The majority of temporal database research has focused on extensions to a relational foundation in much the same way that object-oriented extensions to a relational model have developed and are still evolving [Simon, 1995]. If the temporal databases are typically extensions of conventional data models, then the temporal aspects should be accessible through extensions to traditional query processing techniques.

The overall problem of temporal query processing in the context of OODBs, then, is how to develop a proper temporal data model (preferably, one that is extended from a unified model RDB and OODB) and algebra, and thereby to expand query optimization and query evaluation techniques of (temporal) RDBs to address and to exploit the new modelling and query extensions in temporal OODBs (TOODBs).

1.2.2 Objectives of the Project

The overall objective of the project is to address the above problem and to devise the techniques or strategies to process and optimize temporal object queries. To fulfil this objective, we have set up the following sub-objectives:

1) Investigate the current research status and available theory and techniques that could be used for our purpose.

2) Define a temporal object data model that is preferably extended from the unified model of relations and objects, and can represent a variety of real world time varying entities, so that we can expand query processing techniques from (temporal) RDBs to address and to exploit the new modelling and query extensions in TOODBs.

3) Develop a query algebra for the defined model. Ideally it should be 'closed' so that relational techniques can be applied.

4) Devise techniques to process temporal object queries.

1.3 Outline of Main Work

Through this thesis, an extensible approach to process temporal queries in the context of OODBs, has been exploited in such a way that temporal queries can be processed within the existing object-oriented framework (that is extended from relational framework), by smoothly extending the existing query processing techniques, making use of the query processing techniques in (temporal) RDBs for TOODBs. There are three key elements in the thesis work that fulfil the objectives set above.

(1) Definition of a temporal object data model

The unified model of OODBs and RDBs from UniSQL/X [Kim, 1993; 1994; 1995; D'Andrea and Janus, 1996; Kim *et al.*, 1997], that is an object data model extended from the relational data model, has been adopted as a starting point. A temporal object data model is thereby defined by incorporating a time dimension into this unified model of OODBs and RDBs, i.e., via defining a time sequence as a temporal object and integrating it into the database schema defined by the unified model of RDB and

OODB. The database schema defined by the temporal object data model forms temporal relational-like cubes but with aggregation and inheritance hierarchies.

Compared to other temporal relational object-oriented data models, the temporal object data model defined here is a temporally grouped model and possesses following distinguishable characteristics:

- Both homogeneity and heterogeneity in the time dimension can be supported;

- The *epoch* (i.e., the time when the temporal object changes its value) represents a transformed time space and can then serve as a convenient token for the analysis of the query processing cost.

- The temporal object-oriented database presented is a superset of object-oriented database (i.e., retaining snapshot reducibility to an OODB) that in turn is a superset of relational database.

The model provides a basis for the expansion of the techniques of query processing and optimization in RDBs to address and exploit new modelling features in OODBs and TOODBs.

(2) Development of an algebra for the temporal object data model

A query algebra, that provides an access to objects through these associations of aggregation, inheritance and time-reference, is then defined as a general query model/language.

The algebra is closed in the sense that the output from one operation can become input to another. The property of closure is important when the query processor uses and extends the query processing and optimization techniques developed for RDBs to process temporal object queries.

In addition, the algebra possesses the properties of reducibility and grouped completeness. By reducibility, we mean that the temporal object algebra can be reduced to the object algebra when time dimension is not taken into account and the object algebra can be further reduced to the relational algebra when aggregation hierarchy and inheritance hierarchy are not taken in to account. This allows query processing techniques developed for RDBs to be used and extended to process temporal object queries. By *grouped completeness,* we mean that it supports a rather strong notion of the "history of an attribute". For example, one can talk about "employee's salary history" as a single object, and ask to see it, or define constraints over it, etc.

The fundamental intent of the algebra is to allow the writing of expressions of a user's query and to serve as a convenient basis for query processing and optimization. The properties of closure, reducibility and grouped completeness pave an extensible approach to processing temporal object queries within a uniform framework of object-oriented query processing.

(3) Techniques and strategies for query processing and optimization

Due to the extensible structure of our data model and the properties of reducibility and closure of the defined algebra, query processing and optimization are exploited. Especially, techniques and strategies to process temporal queries are presented. There are four components:

1) Query transformation rules comprise:

- relational rules;
- temporal transformation rules;
- inheritance rules; and
- path transformation rules.

Relational rules are derived from well-known algebraic optimization techniques in RDBs and play an essential role in query optimization. When the time-dimension is

taken into account, the temporal transformation rules play a role. Inheritance rules are object specific. Path transformation rules are important in developing query processing techniques and strategies for processing (temporal) object-oriented queries (e.g., as in the decomposition strategy and join algorithms described below).

2) A decomposition strategy for processing temporal queries within the object-oriented query framework;

Temporal object queries are represented by the enhanced path that is defined to refer to the path with time-reference in this thesis. A decomposition strategy is devised for such a query. Applying this strategy, an enhanced path is initially divided into two sub-paths: one containing time-stamped class that can be optimized by making use of the ordering information of temporal data, and the other, an ordinary sub-path (with no time-stamped class) that can be further decomposed and evaluated using different algorithms. The intermediate results of traversed two sub-paths are then joined together to create the query output.

The advantages of decomposing the temporal query into sub-query components are that well-known join algorithms can be used to optimize the decomposed query components, and also provide a convenient means to observe and analyse the effects of the time on query processing costs (as pointed by Kim [1995; 1997] and Snodgrass [1995], these effects have not been reported so far but were deemed to be valuable).

3) Implementation of time-related operation algorithms and join algorithms with stream processing techniques (along with cost analysis) for the decomposed query components.

As temporal data often imply the ordering by time, the stream processing approach is a strategy of choice to implement relevant algorithms. With the stream processing techniques, the following algorithms have been implemented:

- Stream processing time-slice algorithms and stream processing aggregation algorithms for the time-related operations;

- Four basic join algorithms: nested-loop forward join, sort-merge forward join, nested-loop reverse join and sort-merge reverse join, and their modifications

These algorithms are presented with the corresponding cost analysis and implemented on a PC using Borland C++ Version 4. To illustrate the efficiency of the above algorithms when the time is present, simulation based on the simulated International Weather Recording Database is provided.

Both cost analysis and the simulation results show that the join time cost is linearly increased with the expansion in the number of time-epochs (it is linearly increased with the expansion of time in the case of regular TS).

4) Heuristics for optimization

To further reduce the time cost, the following heuristics have also been presented as optimization strategies:

- Transform the time-related predicate into the time-slice operation;
- Perform time-slice as early as possible.

The principle of these heuristics is to avoid looking at all of the data. It has been shown in the thesis by both analysis and simulation that utilising the heuristics can lead to a significant time cost saving.

The PhD project has involved the designing of databases for these systems:

- Wood Panel Deformation Measurement System,
- Health-Care Information System, and

- International Weather Record System.

These are taken as case studies to demonstrate the modelling capability of our model and are also used to illustrate our approach.

This thesis also discusses the situation when a user's query requires the data that is not explicitly recorded in the database. It is shown that the techniques of interpolation, or assumption rules that make use of the implications of the temporal data can be used to answer such a query. Again, the efficiency can be improved by making use of the ordering information of the data and also heuristics to reduce the scope of sequence scans.

1.4 Summary of Related work

Although research on temporal databases has been carried out for over fifteen years, there are still some unmet challenges [Snodgrass, 1995; Kim, 1995; Kim *et al*, 1997]. Support for time in conventional relational databases systems is almost entirely at the level of user-defined time (i.e., attribute values drawn from a temporal domain) [Snodgrass, 1995]. The user-defined time support in SQL2 is poorly designed [Snodgrass, 1995; Melton and Simon, 1993]. Although the ISO SQL3 committee in July 1995 voted unanimously to accept a new part: SQL/Temporal (also expected to incorporate object-oriented aspects), with *Period* predefined data type being the first aspect of TSQL2 to become part of SQL3 [Segev, Jensen and Snodgrass, 1995], no details have been revealed as for how it will address the central issues of temporal support [Eisenberg and Melton, 1999]. None of the other object-oriented database standards, including "The Object Database Standard: ODMG 2.0" [Cattell *et al.,* 1997], specifies the management of temporal data.

Research on temporal databases has mainly focused on defining temporal data models by extending existing models [Pissinou *et al.,* 1993, 1994; Clifford *et al.*, 1993; Gadia, 1988; Griffiths and Theodoulidis, 1996; Snodgrass, 1995; Stonebraker *et al.,* 1990; Tansel *et al.,* 1993; Golralwalla *et al.*, 1998; Bohlen *et al.,* 1998; Tansel and Tin, 1998], and there is no commonly accepted consensus temporal data model, nor well-accepted

temporal database algebra [Pissinou *et al.*, 1993; Snodgrass, 1995]. Among the various reports on defining a temporal relational data model and algebra, Clifford [1993] associated lifespans at both attribute and tuple level as timestamps to define the historical relational data model and algebra, Tansel [1993] adopted temporal sets for timestamps and a temporal atom for a time-stamped object, and Gadia *et al.* [1993] provided a parametric data model, treating the attribute value as a function of time. These features of the above work are reflected in our data model and algebra. But because our temporal object data model is defined by incorporating the time-dimension into the unified data model of RDB and OODB via defining a time sequence as a temporal object and representing it into database schema, it is a temporally grouped model, and the *heterogeneity* in time dimension and *grouped completeness* of algebra can be supported. There is no temporal relational algebra that has been shown to be *grouped complete* as stated in [Clifford *et al.*, 1993; Pissinou *et al.*, 1994] whilst the temporally grouped model has been considered desirable [Segev, Jensen and Snodgrass, 1995].

Over two dozen proposals have been made for an object algebra as mentioned in [Ozsu and Blakeley, 1995]. No query algebra thus defined is based on any unified model of RDBs and OODBs, although it has been claimed that an object algebra should extend relational algebra consistently [Ozsu and Blakeley, 1995; Shaw and Zdonik, 19990; Yu and Osborn, 1991]. Few of these object algebras are closed, and none of them consider the temporal dimension or supports temporal data management. Our temporal object algebra reflects some spirits of object algebras [Shaw and Zdonik, 1990; Straube and Ozsu, 1990; Cluet and Delobel, 1994; Alhajj and Arkun, 1993], but in addition to supporting access through aggregation and inheritance associations, our algebra also accesses objects through the time dimension. These are embodied in the nested predicates and (nested) temporal predicates, and are represented by the enhanced path expressions. The properties of closure, reducibility and grouped completeness thus allow an extensible approach to processing temporal object queries.

There is little work reported on temporal query processing and optimization [Leung and Muntz, 1993; Dayal and Wuu, 1992; Seshadri *et al.*, 1996; Pissinou *et al.*, 1994; Snodgrass, 1995; Zurek, 1998] and most of them are in the context of RDBs. As RDBs always require the user to explicitly join two relations, temporal processing in the context

of RDBs has been focused on specific join algorithms, such as temporal join and optimization [Gunadhi and Segev, 1990; Segev, 1993; Zurek, 1998], the strategies of stream processing for temporal joins [Leung and Muntz, 1993], following the bottom up approach. The strategies of stream processing in [Leung and Muntz, 1993] have been adopted in our time-related operation and join algorithms. Seshadri *et al.* [1996] separated the optimizer functionality between the database optimizer and temporal optimizer and provided a paradigm for interaction between a relation and a sequence. Although they have provided a comprehensive approach for sequence data processing, they have not incorporated sequence processing in the relational query processing framework. The idea of separating optimizer functionality has been reflected in our query optimizer. Dayal and Wuu [1992] proposed a uniform approach to processing temporal queries in the context of functional object-oriented data model, but their work did not take into account the query optimization and evaluation in a query processing framework. Also their work is based on a functional model and language, that would lead to a functional optimization that is quite different from the algebraic, cost-based optimization techniques employed in relational as well as a number of object-oriented systems [Ozsu, 1995].

Our work provides a comprehensive approach to processing temporal object queries, from the data model, algebra to query processing techniques, which are required to support time-varying data. We have shown that the temporal object queries can be processed within the existing object framework through extending the existing query processing and optimization techniques. The approach is in contrast to other work in the field.

1.5 Organisation of the Thesis

Apart from Chapter 1 (Introduction) and Chapter 9 (Conclusions and Future Work), this thesis can be divided into three parts as follows. Part I is an overview of background and research status in this area and comprises Chapters 2 and 3. Part II constructs a fundamental part (i.e., the data model and algebra) for query processing and comprises Chapter 4 and 5. Part III comprises Chapter 6 to 8 where the techniques and algorithms are proposed for processing temporal object queries.

Part I

Chapter 2 provides an overview of current status on relational databases, object-oriented databases and temporal databases. It is made clear that the next generation of databases trends to be a combination of relations and objects as strengths and weaknesses of commercial relational and object databases are complementary. Support for time varying data is required by many advanced database applications and temporal databases are still at the stage of research. Chapter 3 gives a survey on query processing techniques in databases. It summarised that relational databases have accumulated a lot of knowledge and experiences on query processing, the problem of query processing in OODBs has not yet been very widely researched and almost all object query processors use the optimization techniques developed for RDBs. Processing temporal queries in the context of relational databases only touched to the operator of join. Little work has been reported on processing temporal queries in the context of OODBs.

Part II

Query processing techniques are dependent upon a data model and algebra/language. Chapter 3 defines a temporal object data model that is extended from UniSQL/X with a time dimension. An algebra for the data model is developed in Chapter 5. The model with the extensible structure and the algebra with the properties of reducibility and closure form a basis for query processing.

Part III

Chapter 6 presents a uniform framework for processing temporal object queries. Within the framework, a set of algebraic transformation rules is specified for algebraic optimization, and a decomposition strategy is proposed for processing the temporal object queries. This addresses a central issue of path optimization in object-oriented databases. Chapter 7 presents algorithms for processing the decomposed query components. The algorithms are implemented with stream processing techniques that

make use of the ordering information for optimization. Both cost analysis and simulation are also provided. Chapter 8 discusses an important aspect in processing temporal object queries: when a user query has no data entry to the database.

In addition, an appendix includes two selected published papers.

Chapter 2

Relational, Object-Oriented and Temporal Databases

This chapter offers an overview of relational databases and object-oriented databases, and outlines existing research effort on temporal databases.

2.1 Introduction

A **database system** is essentially nothing more than a *computerized record-keeping system* [Date, 1995]. The **database** itself can be regarded as a kind of electronic filing cabinet; in other words, it is a repository for a collection of computerized data files. The user of the system will be given facilities to perform a variety of operations on such files, including the following, amongst others:

- Adding new, empty files to the database;
- Inserting new data into existing files;
- Retrieving data from existing files;
- Updating data in existing files;
- Deleting data from existing files;
- Removing existing files, empty or otherwise, from the database.

All database management systems (DBMSs) are distinguished from other programs by their ability to manage persistent data and to access very large quantities of these data efficiently and safely.

Database systems can be based on a number of different data models. In general terms, a data model is a mathematical formalism consisting of a notation for describing data and data structures (information) and a set of valid operations which are used to manipulate those data, or at least the tokens representing them. Currently, the most popular data models are relational and object-oriented data models, resulting in two major types of databases available today: relational databases (RDBs) and object-oriented databases (OODBs). Whilst the semantic limitations of RDBs are widely recognised, OODBs are emerging to represent some of the most promising ways of meeting the demands of advanced database applications, such as computer-aided design (CAD), computer-aided manufacturing (CAM), computer-integrated manufacturing (CIM), computer-aided software engineering (CASE), document and multimedia preparation, office automation and scientific computing [Bertino and Martino, 1993; Cattell, 1994; Kim *et al.,* 1997].

These advanced database applications often require support for time-varying data. The enterprise modelled by a database is rarely static. Often the dynamics of an enterprise represent the most important aspect to be captured within a data model. Recent trends in data modelling emphasise the representation of temporal aspects in database schema and the support of corresponding data manipulation facilities directly by a database management system [Tansel, 1993; Snodgrass, 1995; Kim, 1994; 1995; Kim, *et al.,* 1997].

Most research on temporal databases concerns relational and pseudo-relational database models which incorporate time [Pissinou *et al.,* 1993; 1994; Clifford *et al.*, 1993; Gadia, 1988; Griffiths and Theodoulidis, 1996; Snodgrass, 1995; Stonebraker *et al.,* 1990; Tansel *et al.,* 1993; Tansel and Tin, 1998]. There is, however, an increasing emphasis on the role of time in OODBs [Pissinou *et al.,* 1993; Snodgrass, 1995; Kim, 1994; 1995; Kim, *et al.,* 1997].

In this chapter, after briefly reviewing the basic concepts of relational databases and object-oriented databases, existing research on temporal databases will be outlined. The rest of the chapter is structured as follows. Section 2.2 looks at relational databases. An

overview of object-oriented databases is given in Section 2.3. Existing research on temporal databases is outlined in Section 2.4 and a summary of the chapter is given in Section 2.5.

2.2 Relational Databases

2.2.1 What Is a Relational Database?

A relational database is based on the *relational model of data.* The relational model, in turn, is an abstract theory of data that is based on certain aspects of mathematics (principally set theory and predicate logic).

In Date [1995], a **relational database** is described as "a database that is perceived by its users as a collection of relations or tables". All values in a relation are atomic or scalar (there are no repeating groups). A **relational database management system** (relational system for short) is a system that supports relational databases and operations on such databases, including in particular the operators Select (also known as Restrict), Project, and Join. These operators, and others like them, are all set-level, supporting user requests, e.g., for data retrieval. The optimizer is the system component that chooses an efficient way to implement user requests.

2.2.2 The Relational Model of Data

Codd's relational model of data was introduced in 1970 [Codd, 1970]. It was based on the first-order predicate calculus (FOPC) and provided a theoretical basis for the development of relational databases.

The relational model of data is motivated by several aims, including: the desire to use formal methods in database design, enquiry and update; the desire to be able to prove the correctness of programs based on non-procedural descriptions; and the urge to meet

the demand that a theory should be as simple as possible while retaining its expressive power.

The relational model is concerned with three aspects of data: data structure (or object), data integrity, and data manipulation (or operators).

A relation, mathematically defined, is any subset of a Cartesian product of sets. Given a list of sets A_1, ..., A_n, their Cartesian product is the set of all bags (a bag is a list wherein elements may be repeated as opposed to a set where repetition is not permitted) of n elements of A where there can be only one element in the bag from each A. Such a bag is called an ordered n-tuple, or just a **tuple**. The relation is sometimes said to be n-ary if there are n attributes. Each A is called a **domain** when viewed as a set of elements from which an attribute may take its values and an **attribute** when viewed as a label for that set. These concepts are illustrated in **Figure 2.1.**

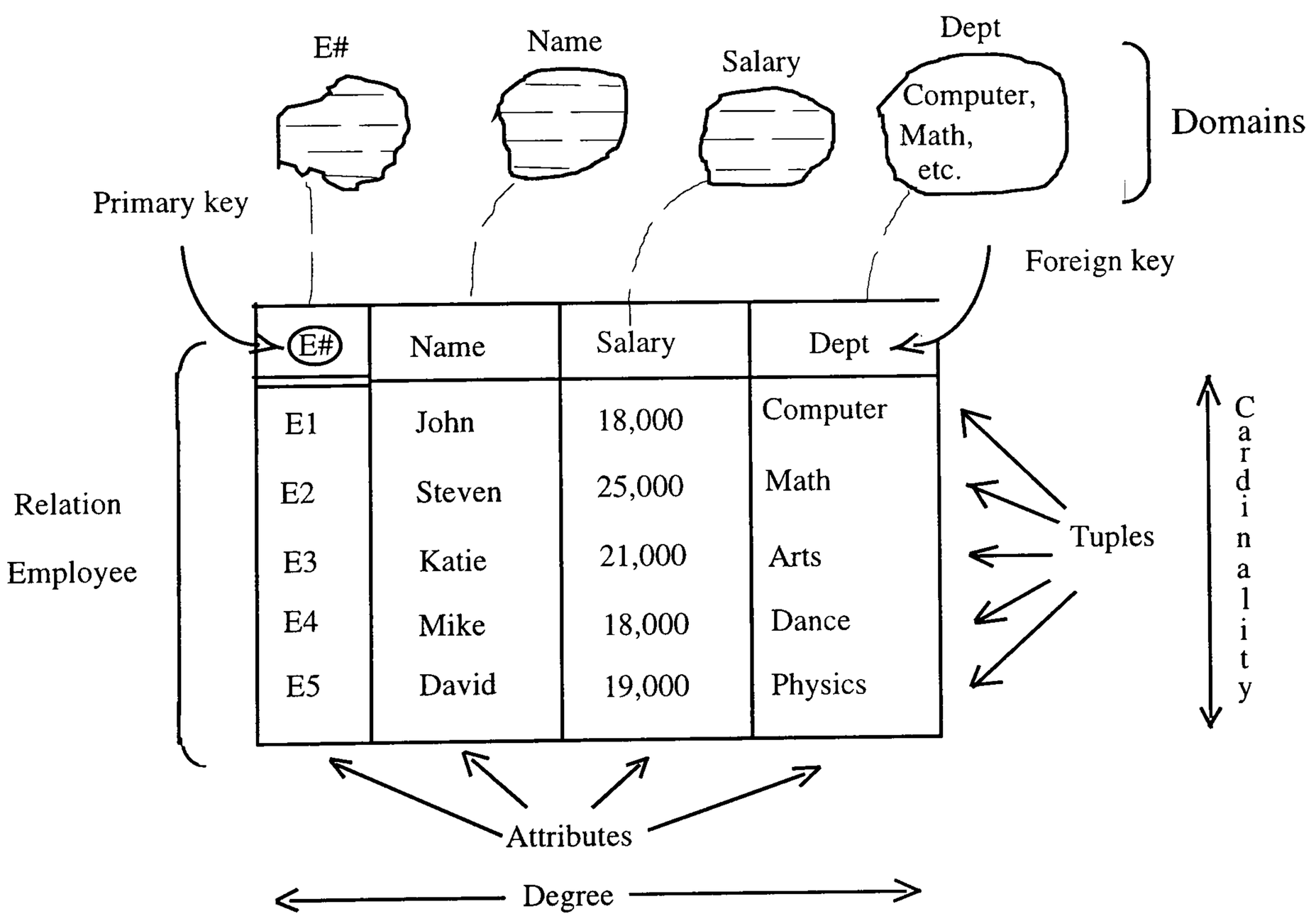


Figure 2.1 An example relation: employee

The next order of structure in database concerns the relationships (links) between relations. These can be are relations themselves. Relation musts conform to certain integrity constraints. Every entity must have specified at least one **primary key** set of attributes that uniquely identifies each tuple at any given time. Furthermore, it must be in the first normal form, i.e., the attribute values cannot be complex structures (repeating groups, lists, and so on) but must be atomic data types (numbers, strings, and so on). The links have two kinds of property: multiplicity and modality. The multiplicity (sometimes called cardinality) of a link may be one-to-one or many-to-one and modality may be necessary or possible. Integrity, multiplicity and modality constrains are usually coded in the application and nearly always in some exogenous procedural language. A **foreign key** is also defined: this is an attribute (or a set of attributes) which is the primary key of some other relation. The integrity rules specify what happens to related relations when a table is subjected to update or deletion operations.

The manipulation part of the model is the means (operators) by which queries and update requests can be expressed. There are essentially two methods, known as the relational calculus and the relational algebra**. The relational calculus** as introduced by Codd [1971; 1972] is a retrieval and update language based on a subset of the first-order predicate calculus. The retrieval is done via a tuple-variable which may take values in some given relation. An expression of tuple calculus is defined recursively as a formula of predicate calculus formed from tuple variables, relational operators, logical operators and quantifiers. Briefly, a tuple variable is a variable that ranges over some relation, i.e., a variable whose only permitted values are tuples of that relation. In other words, if tuple variable T ranges over relation R, then, at any given time, T represents some tuple t of R. For example, the query "Get employee numbers for employees in the computer department", can be expressed in QUEL as follows:

RANGE OF E IS EMPLOYEE

RETRIEVE (E.E#) WHERE E.DEPT="Computer".

The alternative approach is to regard queries and updates as expressed by a sequence of algebraic operations. **The relational algebra** as defined by Codd [1972] is based on five primitive operations: select, project, product, union and difference. *Select* <on predicate> yields those tuples that satisfy the predicate; it may be thought of as a horizontal subset operation. The corresponding vertical subset operation is the *project* that returns a relation consisting of all tuples that remain as (sub)tuples in a specified relation after specified attributes have been eliminated. Product returns a relation consisting of all possible tuples that are a combination of two tuples, one from each of two specified relations. If two tables have the same attributes, their union may be formed by appending them together and removing any duplicate in the primary key. *Difference* returns a relation consisting of all tuples appearing in the first and not the second of two specified relations. The derivative operations such as join, divide, intersection, can be defined in terms of these primitive operations. The join of two relations A and B over a relational operator (dyadic predicate) p is useful though not primitive and can be obtained by building all tuples that are concatenation of a tuple from A followed by one from B such that p holds for the attributes specified (Duplicates are eliminated here). In actual implementation, a query optimizer will usually attempt to select the optimal order of evaluation making use of the referential transparency of algebra.

The algebra and calculus are isomorphically equivalent and thus represent alternatives to one another. The principal distinction between them is as follows [Date, 1995]: whereas the algebra provides a collection of explicit operations, that can be used to tell the system how actually to build some desired relation from the given relations in the databases, the calculus merely provides a notation for formulating the definition of that desired relation in terms of those given relations. At least superficially, it might be said that the calculus formulation is *descriptive* where the algebraic one is *prescriptive*: The calculus simply describes what the problem *is*, the algebra prescribes a procedure for *solving* that problem. Or *very* informally: the algebra is procedural (admittedly high-level, but still procedural); the calculus is non-procedural. Because the algebra and the calculus are precisely equivalent to one another, for every expression of the algebra,

there is an equivalent expression in the calculus; likewise, for every expression of calculus, there is an equivalent expression in the algebra, i.e., there is one-to-one correspondence between the two. The different formalisms simply represent different styles of expression: the calculus is arguably closer to natural language, the algebra is perhaps more like a programming language.

Relational completeness can be regarded as a basic measure of selective or expressive power for database languages in general. Since the algebra and the calculus are both relationally complete, they both provide a basis for designing languages that provide this power of expressiveness. A manipulation language is said to be relationally complete if it is at least as powerful as the algebra (or calculus), i.e., if its expressions permit the definition of every relation that can be defined by means of expressions of algebra (or calculus) [Date, 1995]. Several hybrid languages based partly on the calculus and the algebra exist, The most notable being those based on IBM System R language, SQL (Structured Query Language). SQL has now become the standard language for relational databases.

2.2.3 Strengths and Weakness of Relational Databases

Relational databases have great strengths. The greatest strength of relational model is its basis in a formal theory: first-order predicate logic. This is what makes it possible to have a relationally complete, non-procedural enquiry language such as SQL or QUEL. The logic ensures that certain things about this language can be proved mathematically. Another notable and very real benefit is that they make changes to the data structure relatively easy and they protect users from the complexity with the use of non-procedural enquiry languages, which can be optimized automatically. Performance problems have been gradually overcome. After initial resistance, relational databases have now achieved such wide acceptance in industry that most systems planners no longer consider hierarchical or network solutions.

Weaknesses include the difficulty of dealing with recursive queries; problems related to nulls; lack of support of abstract data types; severe shortcomings in the representation of data and functional semantics; etc.

With regard to query processing, the strengths and weakness of relational databases have been summarised in **Figure 1.1,** in the previous chapter.

2.3 Object-Oriented Databases

2.3.1 What Is an Object-Oriented Database?

An object-oriented database is a database which integrates object orientation with database capabilities and has the potential to provide powerful repositories for advanced database applications [Demuth *et al.*, 1994; Khoshafian, 1993] so that

Object-oriented databases = database capabilities + object-orientation

Object orientation can be loosely defined as the software modelling and development disciplines that make it easy to construct complex systems out of individual components [Khoshafian, 1993]. The essential OO characteristics of various applications of the term tend to have a fuzzy boundary, but generally, what makes something 'object-oriented' are: abstraction/encapsulation, class and inheritance, message-passing, and polymorphism [Jeusfeld and Staudt, 1994]. Taking the basic characteristics of an object-oriented programming language, object orientation is defined as [Graham, 1993; Khoshafian, 1993]:

Object orientation = Abstract data typing/encapsulation

+ Inheritance

+ Object identity

As in relational databases, users are provided with database facilities to perform a variety of operations on the database. According to [Khoshafian, 1993], database capabilities are defined as:

Database capabilities = persistence
+ concurrency
+ transactions
+ recovery
+ querying
+ versioning
+ integrity
+ security
+ performance

2.3.2 An Object-Oriented Data Model and Query Language

A database requires a proper data model that defines general rules for specification of the structures of the data and operations allowed on the data. Unlike the relational data model, there is no universally agreed object-oriented data model, nor is commonly accepted algebra. Several industry consortia, notably the Object Data Management Group (ODMG) and the Object Management Group (OMG), have proposed standards for the object-oriented data model and language. The latest version is ODMG2.0 [Cattell, 1997]. Although these standards are not officially endorsed by ANSI or ISO, they give a good idea of what a basic object-oriented data model and an OODB language would like.

Object Model

The ODMG Object Model is based on the OMG Common Object Model, which in turn is based on a small number of basic concepts:

- objects

- operations
- types
- and subtyping

The OMG Object Model defines a core set of requirements. ODMG adds components (e.g., relationships) to OMG Object Model to support object storage needs.

According to the ODMG Object Model, the key concepts and constructs of the object-oriented data model can be informally (i.e., jargon and lengthy definitions are avoided) represented as follows [Cattell, 1998, Cattell, 1996; Yu and Meng 1998].

An object can be simple (such as an integer, a real number, a character string, etc.) or complex (such as boat, a person, a document, etc.). Complex objects are constructed from simple objects using constructors such as tuple, set, bag (a multiset, or a set that permits duplicate elements), list (in which the order of elements is significant), and array. Each complex object in the database has a system-generated and system-wide unique object-identifier (OID). Each object can be associated with its lifetime, and each object has a structural aspect and a behavioural aspect. The structural aspect describes the organisation of the object's data. It contains a set of attributes, and each attribute has a domain type that specifies the kind of values the attribute takes. The behaviour aspect of an object describes how its data can be acted upon, and is defined by a set of methods. Each method has a signature, which specifies the name of the method, the arguments and their types, and the result type of the method, and a body, which contains the implementation code of the method. The values of an object can be accessed through the use of methods defined upon the object. This is known as encapsulation.

Objects with the same characteristics (i.e., attributes, relationships and methods) are grouped into a class and are defined collectively.

Relationships may exist between the objects of different classes. When the value of an attribute of an object of a class is an object of another class, the OID is used to establish

such a relationship between a pair of classes. OIDs allow the traversal of objects of one class to objects of another class. Therefore a traversal path can be considered as a special kind of attribute, known as complex attribute, whose domain types are class definitions and whose values are OIDs. In addition to one-to-one, one-to-many and many-to-many relationships, numerous semantic relationships, such as the part-of relationship, can be represented by complex attributes. Through complex attributes, links are established to connect objects of different classes. The hierarchy created by these links is known as the composition or aggregation hierarchy.

The set of all instances (or objects) of a class is called the extent of the class. In some cases individual instances of a type can be uniquely identified by the values they carry for some attribute or a set of attributes. These identifying attributes are called keys.

Classes are organised into a class hierarchy. Saying a class C_2 is a subclass of class C_1 or equivalently, saying class C_1 is a superclass of class C_2, has two meanings: (1) the set of characteristics (i.e., attributes, including complex attributes, and methods) of class C_1 is a subset of the set of characteristics of C_2, and (2) the set of objects in C_2 is a subset of the objects in C_1. Semantically, a subclass is a specification of its superclass. As such, a subclass must have (inherit) all the characteristics its superclass has. But a subclass may have additional characteristics. Further, because of specification, each object in a subclass must also be an object in its superclass. A subclass may override the definition of a characteristic inherited from its superclass by redefining the characteristic. When the method is applied to an object, the system decides which implementation to invoke based on the type (e.g., subclass or superclass) of the object. The ability to apply a single method, with different implementations, to objects of different types is called polymorphism. By allowing polymorphism, the name of a method can be chosen based on its functionality rather than on which objects it can operate. The existence of polymorphism dictates that the binding between the signature and a body of a method referenced in a query can be determined not at compile time but rather at run time. This is called late-binding.

Every class has at least one superclass. A class may have multiple subclasses. A class can also have multiple superclasses. If a class has multiple superclasses, then it will inherit characteristics from multiple classes. This is called multiple inheritance. One problem associated with multiple inheritance is the inheritance conflict, which arises when a subclass inherits a characteristic with the same name but different definitions (e.g., different domain types for attribute names and different signatures for methods) or the same name and the same definition but different semantics, from two or more superclasses. The standard method for solving inheritance conflict is to inherit all conflicting characteristics accompanied with proper renaming.

A general OODB schema can be best described as a class aggregation hierarchy since it is typically contains both the class hierarchy and the aggregation hierarchy.

OQL: Object Query Language

Most OODBs provide a declarative database query language. Although OODBs can often be accessed through code written in an object-oriented language such as C++, the use of a query language is considered very important for writing interactive ad hoc queries and for simplifying the C++ code of application programs. Because of the success and popularity of SQL relational language, most proposed object-oriented database languages have adopted a syntax similar to that of SQL. OQL is an OODB query language proposed in ODMG [Cattell, 1998].

OQL is an SQL-like declarative language that provides a rich environment for efficient querying of database objects, including high-level primitives for object sets and structures. OQL is closely based on the query portion of SQL-92 and provides a superset of the SQL-92 SELECT syntax.

OQL also includes object extensions for object identity, complex objects, path expressions, operation invocation and inheritance. OQL's queries can invoke operations in ODMG language bindings, and OQL may be embedded in an ODMG language binding.

OQL maintains object integrity by using an object's defined operations, rather its own operators. OQL is a functional language where operators can be freely composed, as long as the operands respect the type system. This is a consequence of the fact that the result of any query has a type that belongs to the ODMG type model, and thus can be queried again.

The following is an example of an OQL query that appears in the published standard ODMG2.0 [Cattell, 1997]. It is similar to a SQL, but with object extensions:

```
Select c.address
From Persons p,
p.chichen c
where p.address.street="Main Street" and
count (p.chidren)>=2 and
c.address.city!=p.address.city
```

The "dot" notation is used in the query to traverse the data structure. The query inspects all children of all "Person" to find people who live on Main Street with at least two children. It returns only those addresses of children who do not live in the same city as their parents. It navigates from the Person class using the children reference to another instance of the Person class and then to the Address and City classes.

2.3.3 Why Object-Oriented Databases?

As object-oriented databases are extensions of two concepts: object orientation and databases, the potential of object-oriented database lies in the tight integration of these two technologies. Object orientation allows more direct representation and modelling of real world problems. Through object-oriented constructs users can hide the details of implementation of their modules, share objects referentially, and extend their systems by specialising existing modules.

There are three reasons, arising from the requirements of advanced database applications such as CAD, CASE, CAM, document and multimedia preparation, office automation and scientific computing, etc., why we might consider that object-oriented database is needed: to facilitate a clean interface with an object-oriented programming language; to tackle an application that requires the flexibility of relational database but for which the performance of the latter is inadequate; or to tackle totally new kinds of application where message-passing metaphor seems particularly appropriate.

2.3.4 Approaches to Object-Oriented Databases

When looking at the recent developments in the database area, we can identify different approaches to object-oriented databases:

Extended relational approach

This approach bases its extensions and changes on the existing relation model and its implementation in various products and prototypes. The goal is to extend the relational framework with additional concepts in an "evolutionary" manner. That is, by building on available experience, new concepts and new functionality are added to the model and to corresponding existing relational database systems. The products that try to comply with SQL-3 fall into this category.

Object-oriented approach

Another approach takes object-oriented concepts as implemented in object-oriented programming languages as the basis for their extensions. Incrementally, database features and database functionality are added to provide the same capabilities as relational systems, such as a (declarative) query language and transaction management. In some cases this has resulted in user interfaces and implementations that differ considerably from those of relational database systems. The products that try to comply with ODMG fall into this category.

Logic based approach

The third approach relies heavily on concepts and implementation techniques that have developed for logic programming and deductive systems in general. The use of rules or calculus-oriented languages and deductive-based techniques as an important technique for evaluating complex requests reflects the logic-based approach. In many ways, this approach complements and extends the relational approach as far as the model and the language are concerned.

2.3.5 Strengths and Weaknesses of Object-Oriented Databases

OODBs have a number of advantages over RDBs. The main benefits, so far as commercial systems are concerned, is the possibility of capturing application semantics, thus better enabling reverse engineering and prototyping. A further important benefit is that of extensibility. OODBs remove the 'impedance mismatch' between application and query languages. Compared with RDBs, they reduce the need to perform expensive joins when objects are used in an application. This makes them potentially much more efficient for applications involving complex objects. They add support for long transactions and automatic version control. Some offer dynamic schema evolution and support for multimedia and group work.

However, there are unsolved problems concerning non-procedural query languages, query optimization and locking. Critics of the object-oriented approach frequently point to the theoretical limits of optimization as a major drawback of the object-oriented approach as compared with the relational approach [Unland *et al.,* 1992].

There are still many problems with current-generation object-oriented databases. Products are immature. There is no universally agreed formal model behind them, nor is commonly accepted algebra. OODBs achieve many of the aims of semantic data models, but are not yet as structurally rich. They also omit many of the features of knowledge management. There is a small, but growing number of commercial OODB products. These have been mostly applied to applications where complex objects

predominate, such as CASE tools, multimedia databases, geographic information systems and CAD/CAM systems. Commercial applications are in their infancy.

OODB is an immature but rapidly maturing technology of great significance to information technology in general. Several industry consortia, notably OMG and ODMG have been working on to propose standards for OODBs. OODBs are evolving toward a new generation of systems combining semantic models, expert systems, object-orientation and hypermedia technology. There are several ways to achieve benefits. One of them is the combination of relations and objects [Kim, 1993; 1994; 1995; Kim *et al.*, 1997; Date, 1995; Date and Darwen, 1998; Stonebraker, 1996; 1998; Eisenberg and Melton 1999].

The strengths and weaknesses of OODBs have also been included in **Figure 1.1.**

2.4 Temporal Databases

Time is an important aspect of all real-world phenomena. Events occur at specific points of time; objects and the relationships among objects exist over time. The ability to model this temporal dimension of the real world and to respond within time constrains to changes in the real world as well as to application-dependent operations is essential to many computer application, such as accounting, banking, econometrics, geographical information systems, inventory control, law, medical records, multimedia, process control, reservation systems, scientific dada analysis, etc.

Nearly every database product today requires users' intervention to handle the temporal property. Temporal databases add the property of time to the underlying data. Temporal database systems will move this property into the DBMS environment itself, automatically storing multiple time-sensitive versions of data objects, and additionally, providing facilities to retrieve data by time-oriented queries. Temporal databases are basically research vehicles rather than truly commercial applications. This subsection outlines the research effort on temporal databases.

2.4.1 Time Dimensions

In the context of databases, two time dimensions are of general interest [Snodgrass and Ahn, 1986; Snodgrass, 1995]: valid time and transaction time.

The **valid time** of a fact is defined [Jensen, *et al.*, 1994] as the time when the fact is true in the modelled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user. Valid time can also be in the future, if it is expected that some fact will be true at a specified time in the future [Snodgrass, 1995].

A database fact is stored in a database at some point of time, and after it is stored, it may be retrieved. The **transaction time** of a database fact is defined [Tansel, 1993] as the time when the fact is stored in the database. Transaction times are consistent with the serialization order of the transactions. Transaction-time values cannot be after current time. Also, because it is impossible to change the past, transaction times cannot be changed. Transaction times can be implemented using transaction commit times.

These two dimensions are not homogeneous; transaction time has different semantics from valid time. These two dimensions are orthogonal, though there are generally some application-dependent correlations between two times. A data model supporting neither dimension is termed snapshot, as it captures only a single **snapshot** in time of both the database and enterprise that the database models [Jensen *et al.,* 1993]. A data model supporting valid time is termed, logically, a *valid time* model; one that supports transaction time is termed a *transaction-time* model; and one that supports both valid and transaction time is termed a *bitemporal* model. *Temporal* is a generic term implying some kind of time support.

While valid time may be bounded or unbounded (it is at least bounded in the past), transaction time is bounded on both ends. Specially, transaction time starts when the database is created (before the creation time, nothing was stored) and does not extend

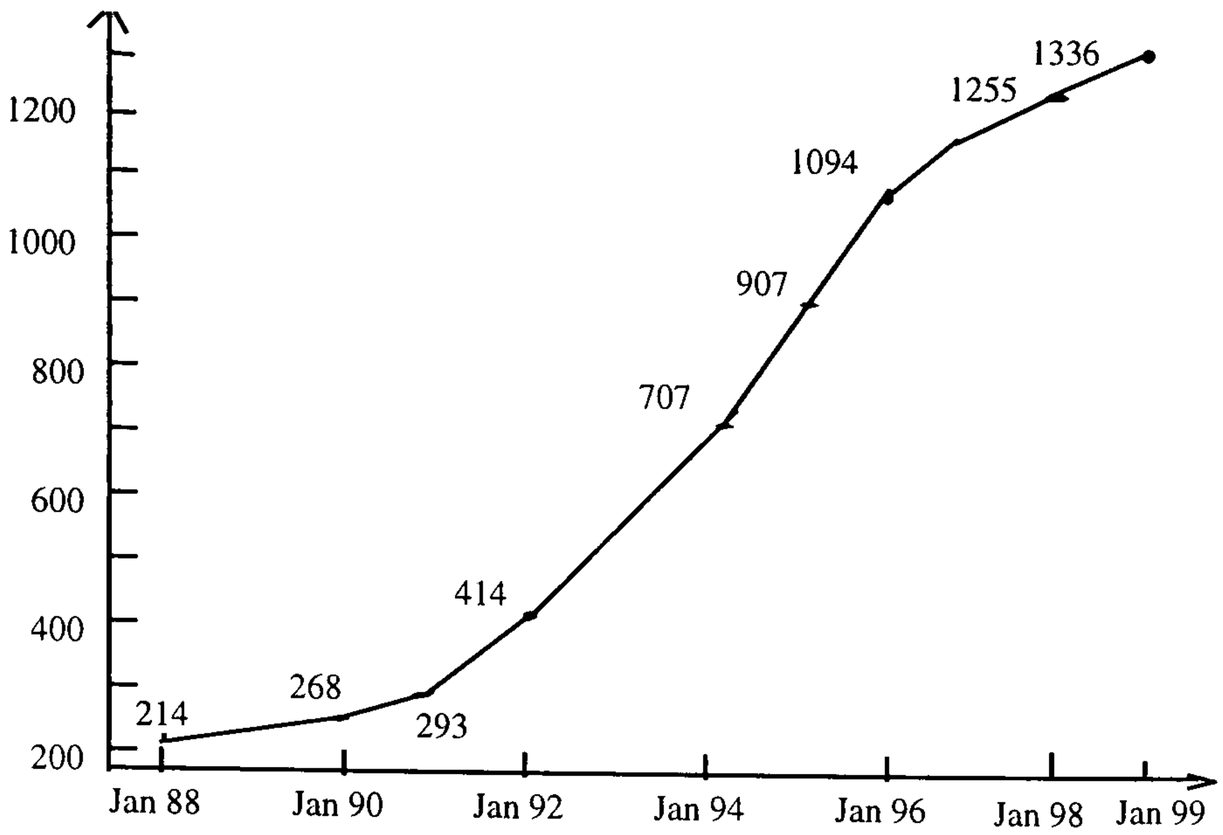
past the present (no facts are known to have been stored in the future). Changes to the database state are required to be stamped with the current transaction time. Hence, transaction time and bitemporal relations are *append only*.

There is a third kind of time that might be included: **user-defined time**. This term refers to the fact that the semantics of these values are known only to the user and are not interpreted by the DBMS, in contrast to valid and transaction time, whose semantics are supported by the DBMS.

A **temporal database** is a database supporting some aspect of time, not counting user-defined time. There are two components to temporal data management: historical data management and version management. The former refers to valid time dimension whilst the latter refers to transaction time dimension.

2.4.2 Research on Temporal Databases

The research on temporal databases has been an active area for at least fifteen years. There are six bibliographies on temporal databases: Tsotras and Kumar, Temporal Database Bibliography Update, ACM SIGMOD Record, 25(1): 41-52, March 1996; Kline, An Update of Temporal Database Bibliography, ACM SIGMOD Record, 22(4): 66-80, Dec. 1993; Soo, Bibliography on Temporal Databases ACM SIGMOD Record, 20(1): 14-23, March 1991; Stem and Snodgrass, A Bibliography on Temporal Databases, IEEE Database Engineering, 7(4): 231-239, Dec., 1988; McKenzie, Bibliography: Temporal Databases, ACM SIGMOD Record, 15(4): 40-52, Dec. 1986 and Bolour, Anderson, Dekeyser, and Wong, The Role of Time in Information Processing: A Survey, ACM SIGMOD Record, 12(3): 27-50, 1983. According to the sixth bibliography [Tsotras and Kumar, 1996], the growth of temporal database papers is superlinear, as shown in **Figure 2.2**, demonstrating that the area remains vibrant.



Source: adapted from (Tsotras and Kumar, 1996)

Figure 2.2 Temporal database papers

Since the significant events of the first book on temporal databases [Tansel, 1993], the ARPA/NSF-sponsored International Workshop on Infrastructure for Temporal Databases that held in Arlington, Texas, in June 1993 (the report of that workshop was published in ACM SIGMOD Record, 23(1), March 1994), and two consensus glossary of temporal database concepts published [Jensen *et al.*, 1994; Tansel, 1993], many ideas and concepts have been made clear and a great progress has been made. Another International Workshop on Temporal Databases was held in Zurich, Switzerland, Set 1995. A consensus extension to SQL-92, the Temporal Structured Query Language, or TSQL2, was developed and published in the new book: The TSQL2 Temporal Query Language, edited by R. Snodgrass [1995]. The ISO SQL3 committee in July 1995 voted unanimously to accept a new part: SQL/Temporal, also expected to incorporate object-oriented aspects, with *Period* predefined data type being the first aspect of TSQL2 to become part of SQL3 [Segev, Jensen and Snodgrass, 1995].

Temporal data model

Research on temporal databases has mainly focused on defining temporal data models by extending existing models [Pissinou *et al.,* 1993, 1994; Clifford *et al.*, 1993; Gadia, 1988; Griffiths and Theodoulidis, 1996; Snodgrass, 1995; Stonebraker *et al.,* 1990;

Tansel *et al.,* 1993; Goralwalla, *et al.,* 1998]. Even so, there is no commonly accepted consensus data model, nor is there well-accepted temporal database algebra [Pissinou *et al.*, 1993; Goralwalla, *et al.,* 1998].

The majority of work on adding time to data models is based on the relational and object-oriented data models (here some post-relational models are also described as object-oriented data models). **Table 2.1**[*] lists most of the temporal relational data models that are defined in the literature. Some models are defined only over valid time or transaction time; others are defined over both. The last column gives a short identifier that denotes the model, the table is sorted on this column. **Table 2.2** classifies the extant temporal object-oriented data models. Models with "arbitrary" in the third and fourth columns support time with user- or system-provided classes; hence anything is possible. N/A denotes "not applicable".

With regard to the valid time, valid times can be represented with single chronon identifiers (i.e., event timestamps), with intervals (i.e., as interval timestamps), or as valid-time elements, which are finite sets of intervals. Valid time can be associated with entire tuples or with individual attribute values. A third alternative, associating valid time with sets of tuples (i.e., relations) has not been incorporated into any of the proposed data models, primarily because it leads to high data redundancy.

There are two types of temporal data model: the temporal data models: *temporally ungrouped* and *temporally grouped.* These models which employ tuple-time-stamping are termed *temporally ungrouped* whereas those models that employ individual attribute time-stamping are termed *temporally grouped* [Pissinou *et al.,* 1994]. The temporally grouped was considered as a desirable property [Segev, Jensen and Snodgrass, 1995].

Temporal query languages

A data model consists of a set of objects with some structure, a set of constraints on those objects, and a set of operations, specifically temporal query languages, on those

[*] Table 2.1-2.4 are adapted from [Ozsoyoglu and Snodgrass, 1995].

objects. Several dozen temporal query languages have been proposed. **Table 2.3** lists the major temporal relational query language proposals to date. The underlying data model refers to **Table 2.1**. Most of temporal relational query languages have a formal definition. Some of the calculus-based query languages have an associated algebra that provides a means of evaluating queries. **Table 2.4** lists the object-oriented query languages that support time. Note many nested relational query languages and data models, such as HQuel, HRDM HTQuel, TempSQL, etc., have features that might be considered object-oriented. A few proposals provide algebras for their query languages. It is rare for a temporal object-oriented query language to have a formal semantics.

The related topics of temporal reasoning (also termed inferencing or rule-based search) are usually excluded from the scope of temporal databases. Temporal reasoning typically uses artificial intelligence techniques to perform more sophisticated analyses of temporal relationships and intervals, generally resulting in much lower query-processing efficiency.

While the expressive power of *ungrouped completeness* was generally accepted as a desirable property for TSQL as it avoids the creation of nulls, *grouped completeness* was also considered to be a useful quality [Pissinou *et al.,* 1994]. G*rouped completeness* implies the support of a rather strong notion of the "history of an attribute". For example, one can talk about "Employee's salary history" as a single object, and ask to see it, or define constraints over it, etc. In temporal RDBs, as stated in [Clifford *et al.*, 1993; Pissinou *et al.,* 1994], there is no algebra that has been shown to be *grouped complete*.

Database design and optimization

In contrast to the flurry of activity in query language and data models, there is a dearth of results in temporal database design and temporal query optimization, in part because there is no commonly accepted consensus data model or query language upon which to base research and development.

There is a little work reported on temporal query processing and optimization [Leung and Muntz, 1993; Dayal and Wuu, 1992; Seshadri *et al.*, 1996; Pissinou *et al.*, 1994; Snodgrass, 1995; Zurek, 1998], whose work was almost in the context of RDBs. This will be discussed in the next chapter.

2.5 Summary

Two major types of databases available today are relational databases and object-oriented databases. Strengths and weaknesses of RDBs and OODBs are complementary, i.e., the weaknesses of OODBs can be compensated to some extend by the merits of RDBs. Therefore it is a commonly accepted that the next generation of database technology will combine relations and objects.

Due to pressing requirements to include time within databases from the user community, substantial effort is being made on temporal databases. Most temporal databases are based on relational and object-oriented data models (and compared with temporal relational models, little work has been done on adding time into object-oriented data models).

Surprisingly, in spite of both this substantial activity and the pressing requirements from the user community, there are no widely used commercial temporal database management systems. A primary reason for the absence of technology transfer from research to practice is the lack of a commonly accepted consensus data model or query language. In contrast to the flurry of activity in query language and data models, there is a dearth of results in temporal database design and temporal query optimization, in part because, again, there is no commonly accepted data model or query language upon which to base research and development.

Finally, we believe that developing a good temporal object-oriented database (probably the combination of RDB and OODB that addresses the temporal aspect) needs a proper temporal object data model.

Table 2.1 Temporal relational data models

Data Model Name	**Citation**	**Temporal Dimension(s)**	**Identifier**
Accounting Data Model	Thompson, 1991	Both	ADM
--	Snodgrass and Ahn, 1989	Both	Ahn
Temporally Oriented Data Model	Ariav, 1986	Both	Ariav
--	Bassiouni and Llewellyn, 1992	Valid	Bassiouni
--	Bhargava and Gadia, 1993	both	Bhargava
Bitemporal Conceptual Data Model	Jensen *et al.*, 1994	both	BCDM
Time Relational Model	Ben-Zvi, 1982	both	Ben-Zvi
DATA	Kimball, 1978	transaction	DATA
DM/T	Jensen *et al.* 1991	transaction	DM/T
Homogeneous Relational Model	Gadia, 1988	Valid	Gadia-1
Heterogeneous Relational Model	Gadia and Yeung, 1988	valid	Gadia-2
Parametric Data	Gadia and Nair, 1993	valid	Gadia-3
Historical Data Model	Clifford and Warren, 1983	valid	HDM
Historical Relational Data Model	Clifford, 1993	valid	HRDM
--	Jones *et al.*, 1979	valid	Jones
--	Lomet and Salzberg, 1993	transaction	Lomet
Temporal Relational Model	Lorentzos and Johnson, 1988	valid	Lorentzos
--	Lum *et al.* ,1984	valid	Lum
--	Mckenzie and Snodgrass, 1991	both	McKenzie
Temporal Data Model	Navathe and Ahmed, 1989	valid	Navathe
--	Sadeghi, 1987	valid	Sadeghi
--	Sarda, 1990	valid	Sarda
Temporal Data Model	Segev and Shoshani,1987	valid	Segev
--	Snodgrass, 1987	both	Snodgrass
--	Tansel, 1993	valid	Tansel
-	Tansel and Tin, 1998	valid	Tansel and Tin
TS-TDM	Segve and Shoshani, 1993	valid	TS-TDM
Time Oriented Databank Model	Wiederhold *et al.*, 1975	valid	Wiederhold
--	Yau and Chat, 1991.	both	Yau

Table 2.2 Temporal object-oriented data models

Data Model Name	**Citation**	**Temporal Dimension(s)**	**Transaction Timestamp representation**	**Identifier**
--	Caruso and Sciore, 1988	Both	chronon	Caruso
IRIS	Beech and Mahbod, 1988	transaction	chronon, identifier	IRIS
--	Kim *et al.*, 1990	transaction	version hierarchy	Kim
MATISSE	ADB, 1992	transaction	chronon, identifier	MATISSE
OODA-PLEX	Wuu and Dayal, 1992	arbitrary	arbitrary	OODA-PLEX
OSAM*/T	Su and Chen, 1991	valid	N/A	OSAM*/T
OVM	Kafer and Schoning, 1992	transaction	identifier	OVM
Postgres	Stonebraker *et al.*, 1990	transaction	interval	Postgres
--	Sciore, 1991	arbitrary	arbitrary	Sciore-1
--	Sciore, 1995	both	chronon	Sciore-2
TEDM	Chu *et al.*, 1992	valid	identifier	TEDM
TIGUKAT	Goralwalla and Ozsu, 1993	both	identifier	TIGUKAT
TMAD	Kafer and Schoning, 1992	valid	N/A	TMAD
Temporal Object-Oriented Data Model	Rose and Segev, 1991	both	temporal element	TOODM
Temporal Object Data Model	Wang *et al.*, 1996	valid	arbitrary	TODM

Table 2.3 Temporal relational query languages

Name	Citation	Underlying Data Model	Based on	Formal Semantics	Underlying Algebra
HQL	Sadeghi, *et al.*, 19987	Sadeghi	DEAL	partial	Sadeghi, 1987
HQuel	Tansel, 1991	Tansel	Quel	yes	Tansel, 1986
HSQL	Sarda, 1990b	Sarda	SQL	no	Sarda, 1990a
HTQuel	Cadia, 1988	Cadia-1	Quel	yes	Cadia, 1988
Legol 2.0	Jones, *et al.*, 1979	Jones	relational algebra	no	N/A
TDM	Segev and Schoshani, 1987	Segev	SQL	no	
Temporal Relational Algebra	Lorentzos and Johnson, 1988	Lorentzos	relational algebra	yes	N/A
Temp SQL	Yau and Chat, 1991	Yau	SQL	yes	
Time-By-Example	Tansel, 1989	Tansel	QBE	yes	Tansel, 1986
TOSQL	Ariav, 1986	Ariav	SQL	no	
TQuel	Snodgrass, 1987	Snodgrass	Quel	yes	McKenzie and Snodgrass, 1991
TSQL	Navathe and Ahmed, 1989	Navathe	SQL	no	
TSQL2	Snodgrass, 1995	TSQL2	SQL-92	yes	Soo, *et al.*, 1994
--	Thompson, 1991	ADM	relational algebra	yes	N/A
--	Bassiouni and Llewellyn, 1992	Bassiouni	Quel	yes	
	Ben-Zvi, 1982	Ben-Zvi	SQL	yes	
	Jensen *et al.*, 1991 Jensen and Mark, 1992	DM/T	relational algebra	yes	IM/T Jensen *et al.*, 1993
	Gadia, 1986	Gadia-2	Quel	no	
	Clifford and Warren, 1983	HDM	ILs	yes	
	Clifford and Croker, 1987	HRDM	relational algebra	yes	N/A
-	Tansel and Tin, 1998	Tansel and Tin	Relational calculus: TRC		TRA
SQL/TP	Toman 1998	Toman	SQL/92	yes	
	McKenzie and Snodgrass, 1991	McKenzie	relational algebra	yes	N/A

Table 2.4 Temporal object-oriented query languages

Name	**Citation**	**Underlying Data Model**	**Based on**	**Implemented**	**Underlying algebra**
MA-TISSE	ADB, 1992	MA-TISSE	SQL	yes	
OODA-PLEX	Wuu and Dayal, 1992b	OODA-PLEX	DA-PLEX		Wuu and Dayal, 1992a
OSQL	Beech and Mahbod, 1988	IRIS	SQL	yes	
OQL	Kafer and Schoning, 1992a	OVM	SQL	yes	
OQL/T	Su and Chen, 1991	OSAM*/T	OSAM*/OQL		TA-algebra
Orion	Kim *et al.*, 1990	Kim	SQL	yes	
PIC-QUERY	Gardenas, *et al.*, 1993	TEDM	PIC-QUERY	yes	
Post-quel	Stonebraker, *et al.*, 1990	Postgres	QUEL	yes	
TMQL	Kafer and Schoning, 1992b	TMAD	SQL		
TQL	Ozsu *et al.*, 1995	TIGUKAT	SQL	yes	
TOO-SQL	Rose and Segev, 1993b	TOODM	SQL	yes	Rose and Segev, 1993a
TOSQL	Rose and Segve, 1991	TOODM	SQL		Rose and Segev, 1993a
VISION	Caruso and Sciore, 1988	Caruso	meta-functions	yes	
GCH-OSQL	Comic *at al.*, 1998	GCH-OODM	SQL		
	Sciore, 1991	Sciore-1	annotations		
	Sciore, 1995	Scior-2	EXTRA /EXCESS		Carey *et al.*, 1988

Chapter 3

Query Processing in Databases

This chapter reviews the basic query processing techniques and strategies used in relational databases, looks at how the query processing is handled in object-oriented databases and points out the current status in processing temporal queries.

3.1 Introduction

Query processing is the procedure of selecting the best plan or strategy to be used in responding to a database request. The plan is then executed to generate a response. The component of the DBMS responsible for generating this strategy is called a query processor. In the database literature, query processing is also referred to as query optimization, and the process here is better described as improvement as the optimization done in practical systems does not necessarily find the best strategy. The optimal strategy may be too difficult to evaluate and on average may not be dramatically different from the one afforded through a heuristic strategy.

The greatest innovation of the relational model of data was declarative queries and associated techniques for automated evaluation that were made possible. Optimization problems have been the focus of a great deal of theoretical and applications research, and much research is still being carried out in this field.

The same cannot be said for OODBs. The problem of optimization of object-oriented queries has not yet been very widely researched. Almost all the object query processors proposed to date use the optimization techniques developed for RDBs [Ozsu and

Blakeley, 1995]. The techniques developed for RDBs can be adapted to developing optimizers for OODBs. Most OODBs designers have adopted this approach. But these OODBs introduce many new data model features to the database, that cannot be handled by traditional query processing technology (i.e., the query processing subsystems of RDBs). Thus query processing in OODBs remains a big challenge. In the context of this thesis, it is significant that there is little work that has been reported on processing temporal queries, especially in the context of OODBs.

This chapter will introduce query processing in RDBs, in more detail, in Section 3.2. How query processing is handled in OODBs will be briefly discussed in Section 3.3. Few reports on processing temporal queries that have been published will be outlined in Section 3.4. Finally a summary will be given in Section 3.5. The discussion of this chapter will be based on centralized database systems.

3.2 Query Processing in Relational Databases

A feature that relational systems have introduced to database management has been a query system that includes a declarative language and system support to process queries efficiently. In many relational database management systems, SQL query commands entered by a user go through a process called query optimization before being executed. This process is carried out by a part of the DBMS known as the query optimizer. The query optimizer's job is to find the best strategy for actually carrying out the user's request.

3.2.1 Optimization Objectives

Economic necessity requires that optimization procedures either attempt to maximize the output for a given number of resources or to minimize the resource usage for a given output. Query optimization tries to minimize the response time for a given query language and mix of query types in a given system environment.

The total cost to be minimized is the sum of the followings [Jarke and Koch, 1984]:

Communication Cost: The cost of transmitting data from the site where they are stored to the sites where computations are performed and results are presented. These costs are composed of costs for the communication line, which are usually related to the time the line is open, and the costs for the delay in processing caused by transmission. The latter, which is more important for query optimization, is often assumed to be a linear function of the volume of data transmitted.

Secondary Storage Access Cost: The cost of (or time for) loading data pages from secondary storage into main memory. This is influenced by the volume of data to be retrieved (mainly relating to the size of intermediate results), the clustering of data on physical pages, the size of the available buffer space, and the speed of the devices used.

Storage Cost: The cost of occupying secondary storage and memory buffers over time. Storage costs are relevant only if storage becomes a system bottleneck and if it varies from query to query.

Computation Cost: The cost for (or time of) using the central processing unit (CPU).

The structure of query optimization algorithms is strongly influenced by trade-offs amongst these cost components. In the long-range distributed DBMSs with relatively slow communication lines, communication delays dominates the costs, whereas the other factors are relevant only for local suboptimization [Jarke and Koch, 1984]. Telecommunication lines can be bottleneck resources in the distributed DBMSs [Bell and Grimson, 1992]. In centralized systems, the costs are dominated by the time for secondary storage accesses although the CPU costs may be quite high for complex queries. In locally distributed DBMSs, all the factors have similar weights, which results in very complex cost functions and optimization procedures.

With regard to the centralized databases, communication costs are not considered because in such systems communication requirements are independent of the evaluation

strategy. For the optimization of single queries, storage costs are usually also assumed to be of secondary importance. They are considered only for the simultaneous optimization of multiple queries. There remain the costs of secondary storage accesses (usually measured by the number of page accesses) and CPU usage (often measured by the number of comparisons to be performed).

A number of common ideas underlying most techniques developed to reduce these costs [Jarke and Koch, 1984] attempt to (1) avoid duplication of effort, (2) use standardized parts, (3) look ahead in order to avoid unnecessary operations, (4) choose the cheapest way to execute elementary operations, and (5) sequence them in an optimal fashion.

3.2.2 General Processing Strategies

3.2.2.1 Query Representation

A query is a language expression that describes data to be retrieved from a database. Queries posed by users, while suited to people, are not always in a form convenient for internal system use. The query processor re-structures the user query, transforming it from some query language supported by the DBMS into a standard internal form that it can manipulate. Queries can be represented in a number of forms. In the context of query optimization, an appropriate query representation form must fulfil the following requirements: It should be powerful enough to express a large class of queries, and it should provide a well-defined basis for query transformation. According to Jarke and Koch [1984], there are four different query representation forms, each of which has been used in a number of approaches to query optimization.

These forms are relational calculus, relational algebra, query graphs (object graph and operator graph) and tableau.

The relational calculus is a notation for defining the result of a query through the description of its properties. As mentioned in Chapter 2, relational calculus is founded on a branch of mathematical logic called predicate calculus.

The relational algebra, as mentioned in Chapter 2, is a collection of operators on relations. These operators fall into two classes, that is, traditional set operators, such as *Cartesian product (*), union (∪), intersection (∩),* and *difference (–),* and special relational algebra operators, such as *select (σ), project (π), join (▷◁),* and *division (÷).* As described previously, for every expression of the algebra, there is an equivalent expression in the calculus; likewise, for every expression of the calculus, there is an equivalent expression in the algebra. There is a one-to-one correspondence between the two.

Query graphs are used in query optimization for the representation of queries or query evaluation strategies. Two classes of graphs can be distinguished: object graphs and operator graphs.

Nodes in object graphs represent objects such as (relation) variables and constants. Edges describe predicates that these objects are fulfilling. Object graphs contain the properties of the query result and are therefore closely related to the relational calculus.

Operator graphs describe an operator-controlled data flow by representing operators as nodes that are connected by edges indicating the direction of data movement. An operator graph depicts how a sequence of operations can be performed. Operator graphs have been used for representation of algebra expressions. Equivalence transformations such as the earlier application of the selection operation can be used to modify the graph. The graph clearly shows what the effect of such a transformation would be. For most simple queries, the graph resembles a tree. The graph can be used to discover redundancies in query expressions.

Tableaus are tabular notations for a subset of relational calculus queries, characterised by containing only AND-connected terms and no universal quantifiers. Thus tableau queries are a particular kind of conjunctive queries. Tableaus are specialised matrices, the columns of which correspond to the attributes of the underlying database schema. The first row of the matrix, the summary, serves the same purpose as the target list of a relational calculus expression. The other rows describe the predicate. The symbols appearing in a tableau are distinguished variables (corresponding to free variables), nondistinguished variables (corresponding to existentially quantified variables), constants, blanks, and tags (indicating the range of relation).

The algebra and query graph will be used for our discussion in this thesis.

3.2.2.2 Steps in Query Processing

Query processing approaches in the literature can be divided in two classes, which can be described as bottom up and top down. Researchers have found that the overall query optimization problem to be very complex. Theoretical work began with a bottom-up approach, studying special cases, such as the optimal implementation of important operations and evaluation strategies for certain simple subclasses of queries. Subsequently researchers attempted to compose larger building blocks from these early results. A top-down approach incorporates more knowledge about special case optimization opportunities within the general procedures. At the same time, the general algorithms themselves have been augmented by combinatorial cost-minimization procedures for choosing amongst strategies.

The top-down approach follows the following steps:

Step 1 Find an internal query representation into which user queries can be easily be mapped that leaves the system all necessary degrees of freedom to optimize the evaluation.

Step 2 Apply a logical transformation to the query representation that (1) standardises the query, (2) simplifies the query to avoid duplication of effort, and (3) improves the query to streamline the evaluation and to allow special case procedures to be applied.

Step 3 Map the transformed query into alternative sequences of elementary operations for which a good implementation and its associated cost are known. The result of this step is a set of candidate "access plans".

Step 4 Compute the overall cost for each access plan, choose the cheapest one, and execute it.

The first two steps of this procedure are to a large degree data independent and thus can often be handled at compile time. The quality of steps 3 and 4, that is, the richness of access plans generated and optimality of the choice algorithm, heavily depends upon knowledge about the values in the database.

The consequences of data dependence are twofold. First, if the database is volatile, steps 3 and 4 can be done only at run time. This means that the possible gain in efficiency must be traded off against the cost of the optimization itself. Second, a meta-database (e.g., an augmented data dictionary) must maintain general information about the database structure and statistical information about the database contents.

3.2.2.3 General Processing Strategies

Query processing strategies presented in **Figure 3.1** [Desai, 1990] use general techniques for query modification. These techniques include:

1) Expressing the query in an equivalent but more efficient form;
2) Substituting a query involving *n*-relations by a group of simpler queries (query decomposition);

3) Replacing a query involving views to one expressed on the base relations; and
4) Adding additional predicates to the query to enforce security.

In addition, query processing strategies take into account the characteristics of the data and the expected sizes of both the intermediate and final results. Strategies are also included to enhance the query response time or reduce the cost of evaluating the query. It is unlikely that details of database statistics such as the precise sizes of relations, number of distinct values in each attribute of every relation, etc., can always be maintained. However, the query processing procedures estimate these values and use them in preparing a strategy for optimizing the query evaluation. The estimation cannot be exact and the optimization of costs may be computationally infeasible. Therefore, it is usual to employ heuristic strategies.

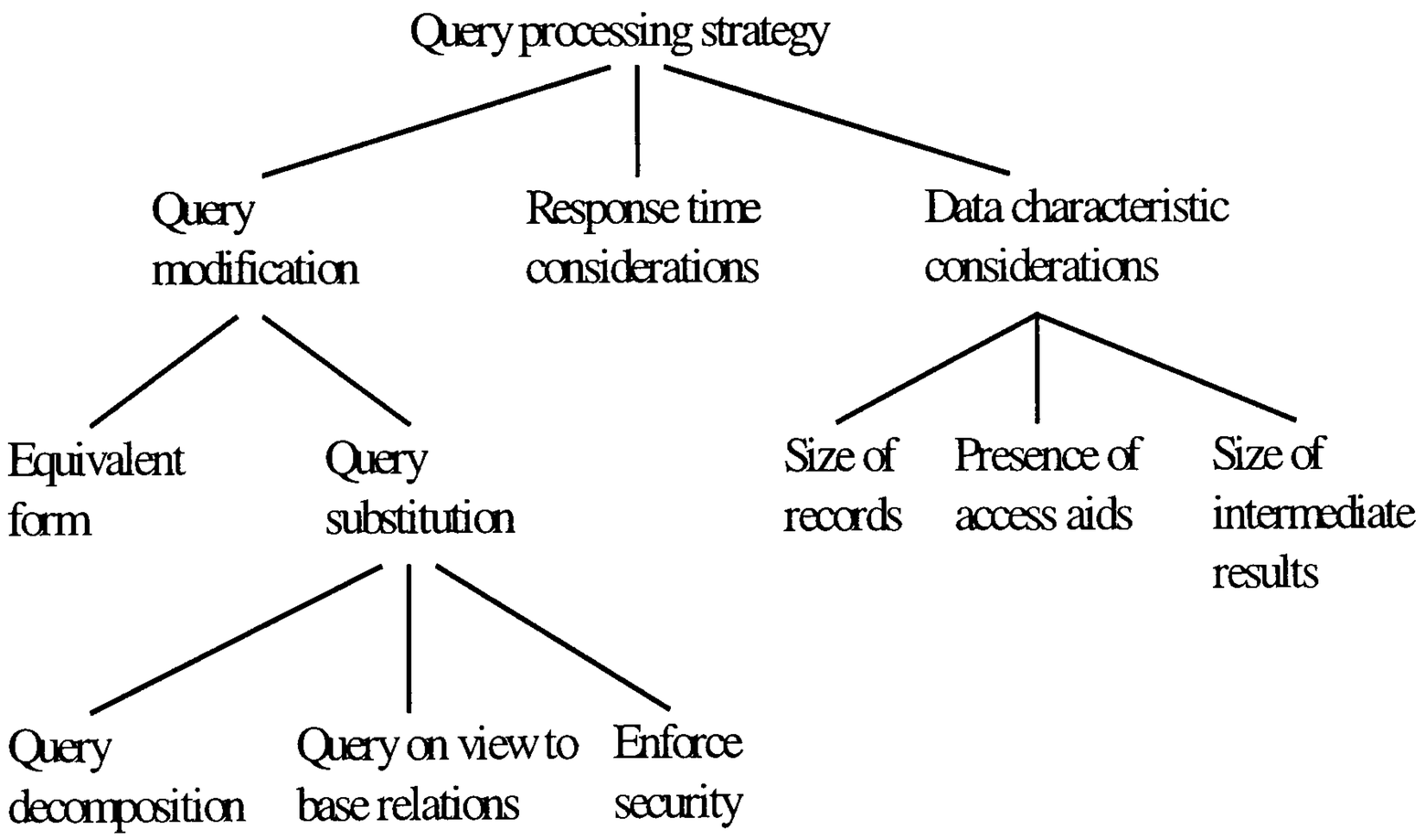


Figure 3.1 Query processing strategies

3.2.3 Optimization Techniques

There are two basic kinds of optimization found in query processors: algebraic manipulation and cost-estimation strategies [Ullman, 1989]. Algebraic simplification of

queries is intended to improve the cost of answering the query independent of the actual data or the physical structure of the data. This is also called *algebraic optimization.* The general term "query transformation" or "query rewrite" falls into this category. This kind of optimization takes place at Step 2 in query processing process.

The second class of optimization strategies considers the issues such as the existence of indexes, to select from among alternatives the strategy that is best for the data and structure at hand. Techniques involved here are called *query evaluation techniques.* This sort of query optimization is also called *execution plan generation* or *plan optimization.* This kind of optimization takes place at Steps 3 and 4 in the query processing process.

In addition, further significant gains in efficiency can be obtained by using higher-level information, particularly information about the semantics of a database. The resulting optimization is called semantic optimization.

3.2.3.1 Algebraic Optimization

(1) Query Transformation

Queries can be expressed in a number of different representational forms. Additionally, a number of semantically equivalent expressions may exist for each query, even within a given language. The transformation of a given expression into an equivalent one by means of well-defined rules is useful in Step 2 of query processing. The goals of query transformation are threefold:

1) the construction of a standardized starting point for query optimization (standardization),
2) the elimination of redundancy (simplification), and
3) the construction of expressions that are improved with respect to evaluation performance (improvement).

Basic transformation laws

These concern transformation that can be made without the benefit of any information on the relations and their schemes. They are based on the associative and commutative laws of relational algebra. Assure that *R, S, T*... are relations on relational schemes *R, S, T*... and *C, C_1, C_2*... are conditions. Also, ∅ is an empty relation, that is, a relation with cardinality of zero, defined on an appropriate relation scheme. The basic transfomation laws are presented as follows [Desai, 1990]:

$R \cup S \equiv S \cup R$	*commutative law*
$R \cap S \equiv S \cap R$	*commutative law*
$R \triangleright\triangleleft R \equiv R$	
$R \cup R \equiv R$	*idempotent law*
$R \cap R \equiv R$	*idempotent law*
$R - R \equiv \varnothing$	
$R \cup \varnothing \equiv R$	
$R \cap \varnothing \equiv \varnothing$	
$R \triangleright\triangleleft \varnothing \equiv \varnothing$	
$R - \varnothing \equiv R$	
$\varnothing - R \equiv \varnothing$	
$R \triangleright\triangleleft S \equiv S \triangleright\triangleleft R$	*commutative law*
$R * S \equiv S * R$	*commutative law*
$R \triangleright\triangleleft (S \triangleright\triangleleft T) \equiv (R \triangleright\triangleleft S) \triangleright\triangleleft T$	*associative law*
$R * (S * T) \equiv (R * S) * T$	*associative law*

Transformation (heuristic) rules

Based on the above basic query transformation laws, heuristics like those regarding the general query processing strategies presented in the previous section can be applied to

control the transformation of the queries into equivalent but more efficient ones. Some example rules follow:

(a) Perform *select* before a *join* or *Cartesian product*. *Select* reduces the cardinality of the relation and, as a result, reduces the subsequent processing time.

Consider $\sigma_C(R \triangleright\triangleleft S)$. If the attributes involved in the condition C are in the scheme of R and not in S, that is, $attr(C) \in R$ and $attr(C) \notin S$, then

$$\sigma_C(R \triangleright\triangleleft S) \equiv \sigma_C(R) \triangleright\triangleleft S$$

If the attributes involved in the condition C are in the scheme of S but not in R, i.e., $attr(C) \in S$ and $attr(C) \notin R$, then

$$\sigma_C(R \triangleright\triangleleft S) \equiv R \triangleright\triangleleft \sigma_C(S)$$

If the attributes involved in the condition C are in the same scheme of R and S, i.e., $attr(C) \in R$ and $attr(C) \in S$, then

$$\sigma_C(R \triangleright\triangleleft S) \equiv \sigma_C(R) \triangleright\triangleleft \sigma_C(S)$$

If $C=C_1 \wedge C_2$ and the attributes involved in the condition C_1 are from R, i.e., $attr(C_1) \in R$, and the attributes involved in the condition C_2 are from S, i.e., $attr(C_2) \in S$, then

$$\sigma_C(R \triangleright\triangleleft S) \equiv \sigma_{C_1}(R) \triangleright\triangleleft \sigma_{C_2}(S)$$

If $C=C_1 \wedge C_2 \wedge C_3$ and the attributes involved in the condition C_2 are only in R, i.e., $attr(C_2) \in R \wedge attr(C_2) \notin S$, the attributes involved in the condition C_3 are only in S, i.e.,

$attr(C_3) \in S \wedge attr(C_3) \notin R$, and the attributes involved in the condition C_1 are in R and S, then

$$\sigma_C(R \rhd\lhd S) \equiv \sigma_{C_1}(\sigma_{C_2}(R) \rhd\lhd \sigma_{C_3}(S))$$

The above equivalencies also apply when the *Cartesian product* operation is substituted for the *join*.

(b) **Combine a number of unary operations**. Consider the evaluation of $\pi_X(\sigma_Y(R))$, where $X, Y \in R$. Both the *select* and *project* operations can be done on the tuples of R simultaneously, requiring a single pass over these tuples and singular access to them. Similarly,

$$\sigma_{C_1}(\sigma_{C_2}(R)) \equiv \sigma_{C_1 \wedge C_2}(R)$$

$$\pi_X(\pi_Y(R)) \equiv \pi_{X \cap Y}(R)$$

$$If\ X \subseteq Y,\ then\ \pi_X(\pi_Y(R)) \equiv \pi_X(R)$$

(c) **Convert the *Cartesian product* with a certain subsequent *select* into a *join***. Consider the evaluation of $\sigma_Y(R*S)$, where Y is, let us say, $A\theta B$ and $A \in R$, $B \in S$. In this case, the *Cartesian product* can be replaced by a *theta join* as follows:

$$R \underset{A\theta B}{\rhd\lhd} S$$

(d) **Use associative and commutative rules for *joins* and *Cartesian products*:**

$$R \rhd\lhd S \equiv S \rhd\lhd R$$

$$R \rhd\lhd S \rhd\lhd T \equiv R \rhd\lhd (S \rhd\lhd T) \equiv (R \rhd\lhd S) \rhd\lhd T \equiv (T \rhd\lhd S) \rhd\lhd R \equiv \ldots$$

$$R * S \equiv S * R$$

$$R * S * T \equiv R * (S * T) \equiv (R * S) * T \equiv (R * T) * S = \ldots$$

The order of the *join* and *product* is very important as it can substantially affect the size of the intermediate relations and, therefore, the total cost of generating the result relation.

(2) Query Improvement

A query can be improved in a number of ways before its evaluation is performed. Improvements are basically concerned with minimizing, if not altogether removing, redundancy from expressions and results. Elimination of redundancy is equivalent to pruning the query operator tree. The rules represented in the previous section are used to find equivalent expressions.

3.2.3.2 Plan Optimization

(1) Query Evaluation

Having found the best equivalent form of a query, the next step is to evaluate it. We classify query evaluation approaches according to the number of relations involved in the query evaluation. Thus we distinguish between the approach to be used when the query expression involves one, two, or many relations. These are known as one-variable, two-variable, and *N*-variable expressions, respectively. A number of different query evaluation strategies have been proposed. Here we look at some commonly implemented techniques.

(a) One-variable expressions

A one-variable expression involves the selection of tuples from a single relation. Let us consider the SQL query

Select $a_1,\ldots,a_k$

from R

where p

The simplest approach would involve reading in each tuple of the relation and testing it to ascertain if it satisfies the required predicates. This is illustrated below.

Sequential access

Use sequential access to read in every tuple of the relation. If the tuple satisfies the qualification conditions, include the project of the tuple on the target list attributes in the result relation. The algorithm is given below:

result:=∅{empty}
for every r in R do
if satisfies (p,r)
then result:=result+<r.a_1...a.$_k$>

where *<r.a_1...a.$_k$>* represents the tuple obtained by concatenating the projects of *r* onto the attributes in the target list.

If the relation has *n* tuples that are blocked as *b* tuples/block, then for sequential access to the tuples, the number of block access is *[n/b]*. In dealing with large relations, this is an inefficient approach. For example, if relation *R* has 400000 tuples, and there are 400 tuples per block of secondary storage devices, reading in all tuples of *R* would involve access to 400,000/400=1,000 block access.

Access aid

The number of tuples needing to be accessed could be reduced if the relation is sorted with respect to one or more attributes. In such cases, if the predicates involve one or more attributes on which the relation is sorted, then only some of tuples need be accessed. Use of indexes can provide faster access to the required tuples.

If the relation has an index, it may be used to improve evaluation performance when access is required to a subset of tuples. Such indices could be on one attribute or they may involve a combination of attributes.

(b) Two-variable expressions

A two-variable expression involves either two distinct tuples from the same relation or two distinct relations. Here we concentrate on the latter case. One of the most common (and expensive) binary operations is the *join* operation. In this section we consider how the *join*, for instance $R \triangleright\!\!\triangleleft S$, can be evaluated. There are two basic methods for join: nested-loop method and sort and merge method.

Nested-loop method

The nested loop method is a simple method in which every pair of tuples from the participating relations are accessed and tested for the join condition. The algorithm in the form of pseudo-code is sketched below.

```
for i:= 1 to |R| do          (* outer loop *)
        begin
        get ith tuple of R
        for j:=1 to |S| do     (* inner loop *)
        perform join  of the ith tuple of R with the jth tuple of S
        end        (* inner loop *)
        end        (* outer loop *)
```

The total number of secondary storage accesses required, assuming that each tuple requires an access, is given as $|R|+|R|*|S|$. The first term represents the access to the tuple of the outer relation and for each such tuple, all the tuples of the inner relation must be accessed. It is preferable to have the smaller relation in the outer loop. Even in the case of small relations, the value $|R|+|R|*|S|$ is quite large. The order of algorithm is $o(n^2)$.

We can substantially improve the performance of the nested loop method by considering physical device characteristics. Data is accessed from secondary storage in chunks called blocks or pages. So the first improvement to the algorithm would be to move away from comparing a single tuple of the outer relation with a single tuple of inner, to comparing all tuples in a block of the outer relation with those from a block of the inner one. This strategy requires that there be space in the main memory for those blocks. The modified algorithm for a blocked nested loop is given below.

```
for each B blocks of R do      (* outer loop *)
        Begin
        read B blocks of R
        for each block of S do  (* inner loop *)
                begin
                read block of S
                for each tuple in the B blocks of R do
                for each tuple in the block of S do
                        if join condition is satisfied
                        then
                        join the tuple of R with the tuple of S;
        end   (* inner loop *)
end   (* outer loop *)
```

Suppose we use blocked (or paged) accesses with the blocking factors of relation R and S represented by bf_R and bf_S, respectively. B blocks of memory are available to store the blocks of relation R (the outer relation). Then the outer loop involves reading B blocks of relation R (the outer relation). Then the outer loop involves reading B blocks at a time. Each tuple in the block of the inner relation can be compared with tuples from these B blocks of the outer relation. This results in the total number of secondary memory accesses given by the following expression:

$$[|R|/bf_R] + [(1/B)*[|R|/bf_R]] * [|S|/bf_s].$$

If one of the relations (let us say R, the smaller of the two) can be kept entirely in memory, then the number of disk accessed required is $[|R|/bf_R] + [|S|/bf_S]$.

Sort and merge method

Relations are assumed to be sorted in the sort and merge method. If they are not sorted, a preprocessing step in the query evaluation sorts them. These sorted relations can be scanned in ascending or descending order of the values of the join attributes. Tuples that satisfy the join predicate are merged. The process can be terminated as indicated in the algorithm below [Desai, 1990].

Algorithm *Sort-Merge to Include a Many-to Many Relationship*

Input: *R,S, the two relations to be joined on attributes A and B, respectively.*
Output: *T, the relation that is join of R and S (concatenation of attributes of R and S, including the attributes A and B).*

```
begin {sort-merge}
T:=empty
sort R by A values and S by B values in ascending order
read (R)
read (S)
while not (eof(R) or eof(S) do      (* main while loop *)
begin
        while not(eof(R) or eof(S) or R↑.A≠S↑.B) do   ( * find a join value * )
if R↑.A<S↑.B
        then read (R)
        else read (S)
if not (eof(R) or eof(S))
        then
                begin (*join a R tuple with one or more S tuples *)
                n:=0
```

```
R_current.A:=R↑.A
while S↑.B=R_current.A and not (eof(S)) do
        begin
        n:=n+1
        U[n]:=S↑
        read(S)
        end
    while R↑.A=R current.A and not (eof(R)) do
        begin
        for i:=1 to n do
        T:=T+R↑ || U[i]↑
        read(R)(*does another tuple of R join with the tuples whose
        pointers are in array U? *)
        end
    end
end (*main while loop *)
    end ( *sort-merge* )
```

In the algorithm, we join the relation R with relation S and the join predicate is $R.A=S.B$. We assume that the relations have been sorted in ascending order with respect to the attributes A and B and that sufficient space for an appropriate number of buffers is available. The tuples are placed in the buffers by the file manager and the algorithm reads the tuples from these buffers. $R\uparrow$ and $S\uparrow$ are pointers that point to the corresponding tuples in the buffers. We assume that once the last tuple in a buffer has been read, the buffer is refilled. If the joining attributes are not the primary key of the relations, a many-to-many relationship could exist via the joining attributes. We use an array U where pointers to tuples of relation S that have the same attribute value as the current tuple of R are sorted. These tuples join with the current tuple of relation R and allow a single pass over the tuples of both the relations. A tuple whose pointer has been stored in this array locks the tuple so that the buffer containing it is not released. An attempt to read past the last tuple in the relation would raise the *eof* (end of file)

condition. The algorithm could be easily modified to include cases where the join involves more than one attribute.

The number of accesses for algorithm is given by:

$$[|R|/bf_R] + [|S|/bf_S] + R_{CS} + S_{CS}$$

where R_{CS} and S_{CS} are the costs of sorting the relations, assumed to be equal to the number of accesses required during the sorting of the relations R and S, respectively. The sort costs depend on memory availability and the number of runs produced in the initial sort stage. For example, if we have enough memory to perform a *max(N,M)*-way merge [Desai, 1990], where the number of accesses required for the join is as follows:

Initial read: $[|R|/bf_R] + [|S|/bf_S$ *blocks*

Writes of sorted runs: $[|R|/bf_R] + [|S|/bf_S]$ *blocks*

Read in merge phase: $[|R|/bf_R] + [|S|/bf_S]$ *blocks*

Writes of the join: $[|T|/bf_T]$ *blocks*

Note that T is the result relation and bf_T is the blocking factor for it. Similar calculations can be done for other memory sizes.

If the relations are already sorted on the joining attributes, the merge-sort method is an efficient method for evaluating a join.

Join selectivity and use of indexes

Consider the join:

$$R \underset{R.A=S.B}{\triangleright\triangleleft} S$$

Join selectivity of a relation R in a natural join with a relation S denoted by ρ_{RS} is the ratio of distinct attribute values for the same attribute A participating in the join to the total number of distinct values for the same attribute in R, that is, $|R[A]|$. Similarly, ρ_{SR} is the join selectivity of the relation S in a natural join with the relation R.

Under the uniform distribution assumption, $\rho_{RS}{}^{*}|R|$ tuples of R and $\rho_{SR}{}^{*}|S|$ tuples of S would be involved in a natural join of relation R with S. The use of join selectivity statistics is an alternate and practical method of estimating the size of the join.

If the relation S has an index on the join attribute and if we assume uniform distribution, then the number of accesses required is given by $|R| + \rho_{SR}{}^{*}|S|$, where ρ_{SR} is the join selectivity. The method of performing the join is as follows. We read in the tuples of R and for each attribute value of $R.A$ we consult the index for S to determine if any tuples from S are involved in the join. If so, these tuples of S are retrieved and joined with the corresponding tuples of R. The tuples of S required to be retrieved would be $\rho_{SR}{}^{*}|S|$.

Should the records of relations R be blocked, the number of block accesses is given by $|R|/bf_R$. If the records of relation S are stored in blocks, the optimal number of block accesses required to access k records of S (where $k=\rho_{SR}{}^{*}|S|$) that is randomly distributed in a file of n records ($n=|S|$) and stored as m blocks ($m=|S|/bf_S$) is given by the following expression:

$$y(k, m, n) = m * \left[1 - \prod_{i=1}^{k} \frac{n - n/m - i + 1}{n - i + 1} \right]$$

However, if indexes exist on the joining attributes for both relations, the use of these indexes provides a more efficient method of evaluating the join. In this case, we can determine if a given value that exists in one of the relations is also present in the order. If so, then the required tuples could be read and joined to produce the result tuples.

Only those tuples that involved in the join are required, and therefore only $\rho_{RS} * |R|$ tuples of R and $\rho_{SR}{}^{*}|S|$ tuples of S are retrieved. The total cost of the join, however, includes the cost of retrieving the indexes.

Hash method and join indexes

The use of hash and join indexes to implement the join operation provides more efficient join algorithms [Desai, 1990; Date, 1995].

(c) N-variable expressions

An *n*-variable expression involves more than two variables. The strategy used here is to try to avoid accessing the same data more than once. One method of evaluating such expressions is to simultaneously evaluate all terms of the query. Therefore, if a number of terms in the query require unary operations on the data accesses, these could be done in parallel. If the data accessed participate in binary operations, these binary operations are partially evaluated.

General *n*-variable queries can be reduced for evaluation by either **tuple substitution** or **decomposition** [Desai, 1990; Wong and Youssfi, 1976; Youssfi and Wong, 1979; Rowe and Stonebraker, 1985].

The presence of access aids and the commonality of attributes can be used to advantage in the evaluations of multiple variable queries.

(2) Access Plans

Once the method of evaluating various operations is determined, the steps involved in combining the query components to deduce the final results have to be planned. Generating an optimal access plan is a stepwise process done in conjunction with the query transformation operation.

The techniques for the efficient evaluation of query components can be used as building blocks of a general query evaluation algorithm. Generating an optimal access plan is

then the combination of these blocks into an efficient evaluation procedure. The inputs of such a procedure are a logically preprocessed query, the existing storage structures and access paths, and a cost model. The output is an optimal (or at least heuristically "good") access plan. The procedure consists of the following steps.

(a) Generate all reasonable logical access plans for evaluating the query. A logical access plan describes a sequence of operations or of intermediate results leading from existing relations to the final result of a query.

(b) Augment the logical access plans by details of the physical representation of data (sort orders, existence of physical access paths, statistical information).

(c) Choose the cheapest access plan by applying a model of access and processing costs.

3.2.3.3 Semantic Optimization

The conventional approach to query optimization, as mentioned above, is to use low-level information such as statistics about various processing costs to access individual tables in a relational database. Significant gains in efficiency can be achieved by using such information.

Over a number of years researchers in the database area have indicated that additional gains in efficiency can be obtained by using higher-level information, particularly information about the semantics of a database. A transformation that is valid only because a certain integrity constraint is in force is called a semantic transformation, and resulting optimization is called semantic query optimization (SQO). SQO can be defined [Date, 1995] as the process of transforming a specified query into another, qualitatively different, query that is however guaranteed to produce the same result as the original one, because the data are guaranteed to satisfy a certain integrity constraint.

3.3 Query Processing in Object-Oriented Databases

One of the basic facilities of database management system is to be able to process declarative user queries. As pointed out by Ozsu and Blakeley [1995], the first generation of OODBs did not provide declarative query capabilities. However, the last decade has seen significant research in defining a query model (including calculi, algebra and user languages) and in techniques for processing and optimizing the queries. Many of the current commercial products provide at least rudimentary query capabilities. The techniques developed for processing object-oriented queries are essentially extended from those of RDBs [Ozsu and Blakeley [1995].

There is no standard formulation of "the" query optimization problem among object-oriented database researchers, because data models and query languages differ as well as query execution engines, their facilities and execution costs. Query processing capabilities in most current OODB products and advanced prototypes are limited in their expressivity or the sophistication of their query optimization and processing techniques [Maier *et al.*, 1994]. In the current crop of systems, queries are generally limited to selecting a subset from a set of existing objects with conditions that are given as a conjunction of path comparisons. There is seldom post-processing of selected elements as part of the query, nor is the combination of the elements from different collections generally supported. Query optimization consists largely of detecting opportunities to apply indices. Often methods are excluded from consideration during the query processing, or limited to those procedures that can themselves be expressed as queries. Dynamic binding of operations to methods is generally inefficient or lacking, thus limiting query processing abilities on heterogeneous collections. Only few of them support querying against bulk types other than sets. Even so, where queries can be posed against ordered collections, there is no facility for constructing auxiliary access paths on such data structures.

Although the proposals and solutions made here draw from experience with the relational model, they all heavily emphasise the object-oriented aspect.

3.3.1 Query Models

3.3.1.1 A Model of Queries for Object-Oriented Databases

The work in [Banerjee *et al.,* 1988; Kim *et al.,* 1989; Kim 1989; Jenq *et al.*, 1989], which covers optimization for ORION, presented a rather comprehensive query model which is consistent with object-oriented concepts embodied in the object-oriented data model. The model takes into account the semantics of class hierarchy and nested objects, and as such is inherently richer than the relational or nested relational model of queries. The model restricts the target of a query to a single class or a class hierarchy rooted at that class. This is an important restriction, since this excludes operations comparable to relational joins and set operations. However, the model explicitly takes into consideration some of the important consequences of object-oriented concepts. First, it allows the user to use the directed graph model of the definition of the target class for specifying a query; predicates may be applied to any attributes of any classes on the graph. This is similar to the nested-relational extensions of relational selection operation. Second, a query may be directed against a single class or a class hierarchy rooted at the class. This is important, since a class hierarchy captures the IS-A relationship between a class and all its subclasses; and as such instances of a class may be regarded as belonging to the class and all classes on the superclass chain starting from the class. In fact, the domain of an attribute of a class is the specified class and all direct and indirect subclasses of the class. The model proposed in [Banerjee *et al.,* 1988] and elaborated somewhat in [Kim *et al.*, 1989] is based on the view that a query model may be defined as a subschema of the database schema; the database schema is reduced to a query model by applying the selection and projections. It is the first query model which made a serious efforts to capture the semantics of object-oriented concepts. However, the model defined only limited types of queries concerning a single class or a class hierarchy rooted at that class. Further the model contained some important oversights, notably in its treatment of the projection operation, and the directionality of the arcs in the class-aggregation hierarchy.

Kim [1989], firstly, provided a considerably more rigorous treatment of the single-operand query, and corrected the mistakes in the model given [Banerjee *et al.,* 1988]. Secondly, he significantly extended the model to provide a formal basis for a query which involves more than one operand, namely, object-oriented equivalents of the relational join. The query model was defined for a set of operations pertinent to relational database systems, namely, selection, projection, join, and set operations. Although he uses relational technology, the semantics of these operations are rather different from those used in relational systems.

3.3.1.2 A Query Algebra for Object-Oriented Databases

A major issue in development of query algebra is the potential for optimization. Many object algebras have been proposed for OODBs (e.g., Alhajj and Arkun, 1993; Beeri and Kornatzky, 1990; Blakeley *et al.,* 1993; Peters *et al.,* 1993; Shaw and Zdonik 1990; Straube and Ozsu 1990; Vandenberg and Dewitt 1991).

However, unlike the relational algebra, there is not a commonly accepted object algebra, nor is it clear how an object algebra should be developed and what trade-offs should be made between elegance, optimizability and expressiveness.

In many object-oriented database applications, the advantages of using a well-chosen family of algebra operations as the basis of a query model may outweigh the restrictions imposed on the expressive power of the model. This approach supports the ability to write programs that work independently of physical structures. When arbitrary programs are used as queries, end-users may need to know about the physical data structures used. They must write code which depends on the particular structure selected, leaving no opportunity for the physical structure to be tuned as database usage becomes clearer. In addition, using algebraic operations provides more opportunities for query optimization. Queries can be formulated in many equivalent forms and optimized by equivalence preserving transformations. The algebraic approach can also provide an important property of a query model- the closure property. This property guarantees that each operation on an object

(objects) produces a new object which has exactly the same status as the original one (ones), namely that all the operations of the object algebra are potentially applicable to the new object. By having this property, the result of a query can be used as the input for other queries or can be stored as user's view. Unfortunately, most existing query models for OODBs don't preserve closure [Ozsu and Blakeley, 1995].

The object algebra of [Manola and Dayal, 1986; Orenstein *et al.,* 1986; Dayal *et al.,* 1985; Dayal *et al.*, 1985] called the PDM algebra, was developed for the PROBE database. The PDM algebra is a modified relational algebra operating upon functions. In particular, an entity type (such as PERSON) is treated as a unary function which, when evaluated, returns a set of entities of that type. Formal arguments of PDM functions are labelled, and can be declared to be in, out, or both so that functions can return multiple values. PDM functions can be extensional (i.e., stored as relations) or intensional (i.e., computed from a subroutine). The appearance of a function in an algebra expression means that the function is to be executed with some actual arguments substituted for its formal arguments. An algebra expression not only returns a function but also may produce assigned variables if the functions appearing in that expression have out arguments. In such a way, multiple results can be obtained from one expression and then serve as a context for the evaluation of subsequent functions.

Osborn's object algebra [1988; 1989a; 1989b] was developed for a general object-oriented data model. The algebra is defined on three generic classes: atomic, aggregate and set objects. Relational algebra operations are extended. Also included are Naming, DeepCopy and Apply operations. Apply serves an iterator on set objects. DeepCopy creates a complete copy of an object without sharing any sub-objects with the old one.

Straube and Ozsu's object algebra [Straube and Ozsu, 1990; Straube, 1991; Ozsu, 1991] was developed to provide a formal basis for object-oriented query processing. To support encapsulation, the algebra allows only one object equality, namely the identity test. An object calculus is also provided. The translation from the algebra to the calculus is complete, but the translation from the calculus to the algebra is only partial.

Shaw and Zdonik's object algebra [Shaw and Zdonik, 1989; Zdonik, 1989; Shaw and Zdonik, 1990; Shaw and Zdonik, 1989] supports *i*-equality *(i>0),* where *i* indicates how "deep" the equality-test should go into two complex objects. DupEliminate and Coalesce operations are included to manipulate object identities. Besides *i*-equality, another equality called *id*-equality is introduced to compare, on two *i*-equal objects, the structures implied by the identities associated with their attribute values. The algebra operation access objects only through the external interface defined by their types. Results of queries are collections of existing objects or collections of tuples built by the query. By including parameterized types, the algebra can be statically type-checked while maintaining the ability to construct dynamic relationships between existing objects.

3.3.2 Query Processing Methodology

A query processing methodology similar to relational DBMSs, but modified to handle the difficulties rising from the new features typical of object-oriented models, can be seen in OODBs. Straube and Oszu [Straube and Ozsu, 1990; Straube, 1991; Ozsu, 1991] proposed such a methodology, which has been adapted and depicted in **Figure 3.2.**

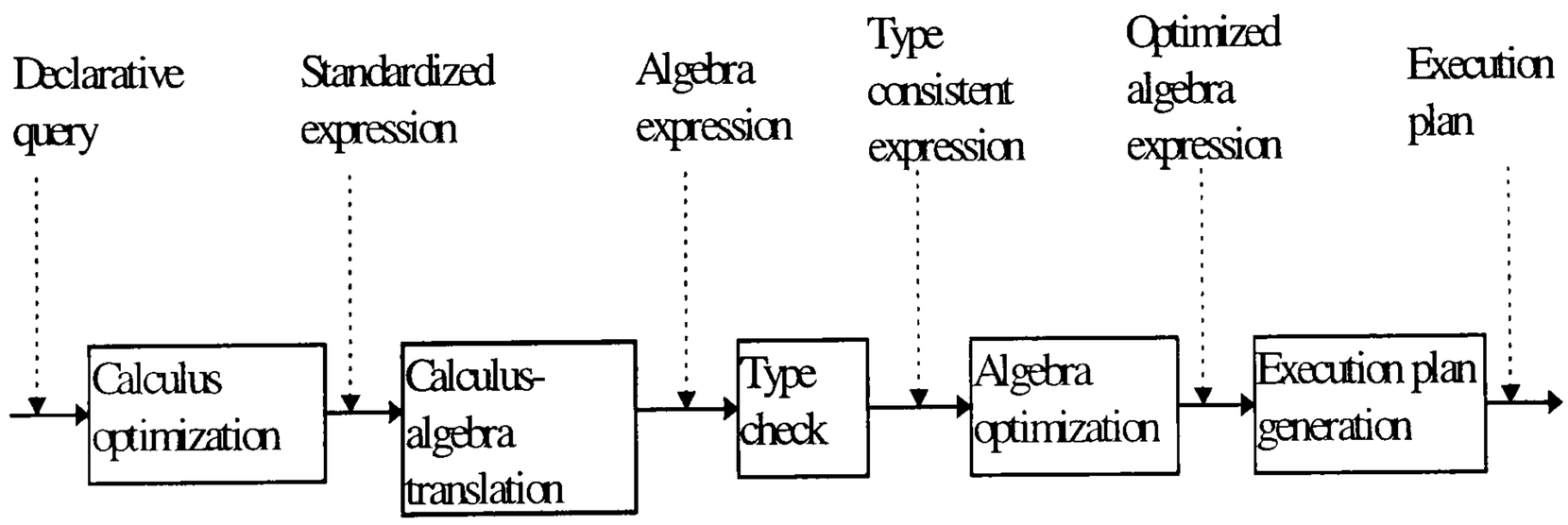


Figure 3.2 Object query processing methodology

The steps in the methodology are as follows. Queries are expressed in a declarative language that requires no user knowledge of object implementations, access paths, or

processing statistics. The calculus expression is first reduced to a standardized form by eliminating duplicate predicates, applying identifiers and rewriting. The standardized expression is then converted to an equivalent object algebra expression. This form of the query is a nested expression that can be viewed as a tree whose nodes are algebra operators and those leaves represent extents of classes in the database. The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects that do not support the requested function. This is not as simple as type checking in general programming languages since intermediate results, which are set of objects, may be composed of heterogeneous types. The next step in query processing is the application of equivalence-preserving rewrite rules [Freytag, 1987] to the type consistent algebra expression.

The separation of the algebraic optimization step from the execution plan generation step follows the distinction that is made between "query rewrite" and "plan optimization" [Haas *et al.,* 1989]. Query rewrite is a high-level process in which general-purpose heuristics drive the application of transformation rules. Plan optimization, on the other hand, is a lower-level process that transforms a query into the most cost-effective access plan, based on a specific cost model and knowledge of access paths and database statistics. This methodology clearly separates the various concerns and provides extensibility to query processor. However, it faces one serious problem: the combinatorial cost of analysing the large number of plans that are generated. The algebraic optimization step generates a family of equivalent query expressions based on the transformation rules defined for algebra. The execution plan generation step creates a number of alternative mappings from each of these expressions to the object manager interface calls. Therefore, the number of alternatives that need to be considered may become quite high. One alternative followed in Starburst [Haas *et al.,* 1989] is to use heuristic rules to control query rewrite so that a single query expression is generated as input to the plan optimization step. Cost-based optimization approaches, on the other hand, merge these two steps into one and consider the alternative execution algorithms as part of search space.

This methodology assumes the existence of a fully specified calculus-based language and an object algebra. There are only a few calculi that have been defined for OODBs [Abiteboul and Beeri, 1987; Peters *et al.*, 1993; Straube and Ozsu, 1990] and a few object logics with declarative query facilities [Kifer and Wu, 1989; Maier, 1986]. There are a large number of declarative user languages (e.g., Blakeley [1991], Carey *et al.* [1988], Kifer *et al.* [1992], Orenstein *et al.* [1992]), but these generally do not have a formal calculus. Many algebras have been defined with a variety of operations (e.g., Alhajj and Arkun [1993], Beeri and Kornatzky [1990], Blakeley *et al.* [1993], Peters *et al.* [1993], Shaw and Zdonik [1990], Straube and Ozsu [1990], and Vandenberg and Dewitt [1991]).

As far as the author is aware, the methodology of Straube and Oszu has never been implemented.

3.3.3 Optimization Techniques

Optimization techniques for object queries fall into two categories. The first is the cost-based optimization of queries based on algebraic manipulations. Algebraic optimization techniques have been extensively studied within the context of the relational model. The work on relational DBMS has benefited greatly from the availability of a universally accepted algebra definition. Despite over two dozen proposals, there is no universally accepted object algebra, making it difficult to generalise research results.

The second is the optimization of path expressions that represent traversal paths between objects and are unique to OODBs, distinguishing object-oriented from relational query processing.

3.3.3.1 Algebraic Optimization

Algebraic optimization is well-understood for relational systems; there, an algebraic expression is given for the semantics of (e.g. SQL) queries, algebraic equivalencies have been specified, and heuristic rules have been discovered which are beneficial when applying those equivalencies for the transformation of queries. However, the "objects" in

question are very simply structured (i.e., they are sets of tuples). Thus, while the equivalencies (and rules) carry over to new data models, they are by no means sufficient. Namely, algebraic optimization has to take complex objects and type hierarchies into account.

CoOMS [Demuth *et al.*, 1994] is a structurally object-oriented database system. It generalises algebraic optimization from the relational data model to a structurally object oriented data model. Algebraic optimization takes idempotence, commutativity, associativity, and distributivity properties of operation into account. Some optimization rules (based on these properties) carry over from the relational algebra, while other rules are related to the specific features of the data model of CoOMS; thus they specify inheritance, subobject, and navigational rules. An algebra is especially useful to represent queries for optimization and evaluation, as rules and equivalence known from the relational algebra carry over.

Cluet and Deloleb [1992; 1994] proposed a formalism that unifies optimization based on classes extensions (path) and algebraic query rewriting. The method introduces types in algebraic expressions and reduces complex expressions representing selection, projection or join criteria. Their approach "unifies" algebraic and type-based rewrite techniques, permits factorization of common subexpressions, and supports heuristics to limit rewriting. They exploit type information to decompose initial complex arguments of a query into a set of simpler operators and rewrite path expressions ("pointer chasing") into joins.

Lanzelotte and Valduriez [1991] presented a similar attempt to optimize path expression within an algebraic framework using an operator called implicit join.

Blakeley *et al.* [1993] proposed an object-algebra operator called materialize (*Mat*), to enable algebraic optimization of path expressions (e.g., e.dept.site, where e, dept, and site are classes and constitute a class-aggregation hierarchy). The purpose of *Mat* is to represent the computation of each interobject reference (i.e., path link) explicitly, allowing

a query optimizer to express the materialization of multiple components as a group using a single *Mat* operator or individually using a *Mat* operator per component. Therefore *Mat* indicates to the optimizer where path expressions are used and where algebraic transformations can be applied.

Unlike RDBs, the use of "select pushdown" or "predicate pushdown" is no longer advantageous in all situations in OODBs. Strategies like "practical predicate placement" [Hellerstein, 1994], "predicate move-around" [Levy *et al.*, 1994], "caching predicate method" [Hellerstein and Naughton, 1996], etc., have been proposed. These are often discussed in relation to query execution. (Query processing in OODBs does not always support the separation of algebraic optimization and plan generation).

3.3.3.2 Path Execution

RDBs benefit from the close correspondence between the relational algebra operations and the access primitives of the storage system. Therefore, the generation of the execution plan for a query expression basically concerns the choice and implementation of the most efficient algorithms for executing individual algebra operators and their combinations. In OODBs, the issue is more complicated due to the difference in the abstraction levels of behaviourally defined objects and their storage. A query-execution engine requires three basic classes of algorithms on collections of objects: (collection scan, indexed scan and collection matching). Collection scan is a straightforward algorithm that sequentially accesses all objects in a collection. Indexed scan allows efficient access to selected objects in a collection through an index. It is possible to use an object's field or the values returned by some method as a key to an index. Also, it is possible to define indexes on values deeply nested in the structure of an object (i.e., path index). Set-matching algorithms take multiple collections of objects as input and produce aggregate objects related to some criteria. Join, set intersection and assembly are examples of algorithms in this category.

Path indexing

Indices are crucial in database systems to expedite the evaluation of queries that retrieve a small subset of data from a large database. Many indexing techniques designed to accelerate the computation of path expressions have been proposed.

Bertino *et al.* [Bertino, 1994; Bertino and Guglielmina, 1993; Bertino and Foscoli, 1995] have discussed a number of indexing techniques specifically tailored for object-oriented databases. They present indexing techniques supporting an efficient evaluation of implicit joins among objects. However, of the several techniques developed, none is optimal for both retrieval and update costs. Techniques providing lower retrieval costs, such as path indices or access relations, have greater update costs compared to techniques, such as multi-index. However, these have greater retrieval costs. These research also introduce an indexing technique that provides integrated support for queries on both aggregation and inheritance graphs. This indexing technique is currently being extended to deal with multi-valued attributes.

Set-matching

OODBs significantly reduce the need for explicit joins. The *select* operation allows its predicate to be applied on a contiguous sequence of attributes along a branch of the class-aggregation hierarchy, where the path expression is used to represent this sort of predicate. The attribute/domain link between a class R and the domain S of one of the attributes A of R creates the *join* between the class R and S, in which the attribute A of the class R and identifier OID, which is defined by the system and which can be considered as an attribute of class S, are join attributes. This sort of query is generally called *implicit join.* Assume two sets of objects R and S stand in a many-to-one relationship from R to S. R and S are stored as separate disk files and the objects in R contain an OID to their related objects in S. Various algorithms have been proposed to execute such a join.

Pointer-based join algorithms [Shekita and Carey, 1990] are used when objects in *R* are to be retrieved first. *R* always plays the role of the inner set because the direction of the pointer is from *R* to *S*. When the objects of *S* are retrieved first, the standard (relational) join algorithms can be used. Shekita and Carey [1990] showed that when *R* is significantly larger than *S*, standard hybrid-hash may outperform pointer-based hybrid-hash. Therefore, OODBs can benefit by supporting both algorithms.

The assembly operator [Keller *et al.,* 1991] is a generation of the pointer-based hash-join algorithm.

Gardarin *et al.* [1996] and Tang *et al.* [1996] analysed the costs of path execution using both the navigational operator and join, and suggested that both object navigation and set-oriented join should co-exist as neither dominated the other. This confirms the results previously stated, that converting implicit joins to explicit joins during the optimization phase may yield better execution plans [Blakeley *et al.,* 1993; Ozsu and Blakeley, 1995; Ozkan *et al.,* 1995].

Multiple path expressions

A query may involve multiple path expressions. Ozkan [1995] proposed a heuristic based approach for optimizing such queries involving multiple path expressions.

3.3.3.3 Semantic Query Optimization

Semantic Query Optimization (SQO) uses the semantic knowledge about objects to transform a query into more efficient expression. In the field of relational databases, much research on SQO has been carried out. OODBs may support many different ways of query processing. It seems likely that a SQO system can be inserted as a preprocessing system for OODB query optimization systems. Class instances have attributes that are instances of other complex classes, and different traversal mechanisms to find the target attribute

values for a query are required. Taking into account this structural property of an OODB, new heuristics which are different from those applied to relational databases are needed.

Sung and Park [1991] presented a new semantic query processing technique in an object-oriented database system. The query technique takes advantage of semantic data integrity constraints to generate more efficient access plans. Semantic information relating to the target objects of a given query is utilized in a suitable way, either by eliminating the unnecessary part of the query or by transforming the given query into a more efficient form. Heuristics which guide the query processor into generating efficient access plans using semantic knowledge underpin the SQO process.

Pang, Lu and Ooi [1991] described their initial results in a study of query optimization in an object-oriented database, where semantic query transformation is used to preprocess the query and the semantically optimized query is then translated into a query evaluation plan which comprises method invocations that can be evaluated directly by the system.

3.4 Processing Temporal Queries

Temporal query optimization is substantially more involved than conventional-query optimization for several reasons [Ozsoyoglu and Snodgrass, 1995]. Temporal-query optimization is more critical, and it is thus easier to justify substantial effort in this area, compared with conventional optimization. The relations over which temporal queries are defined may be larger, and often grow monotonically, implying that unoptimized queries take longer and longer to execute. It is reasonable to expand effort in the optimization of queries on such data and to allow greater execution time in the performance of the optimization. The predicates used in temporal queries are more difficult to optimize [Leung and Muntz, 1990; 1993]. In traditional database applications, queries generally specify equality predicates (hence the prevalence of equijoins and natural joins); if an inequality predicate is involved, it is rarely in combination with other such predicates. In contrast, in temporal queries, joins with a conjunction of several inequiality predicates appear more frequently. Optimization techniques in conventional databases focus on

equality predicates and often implement inequality joins as Cartesian products, with their associated inefficiency.

On the other hand, there is greater opportunity for query optimization when time is present [Leung and Muntz, 1993]. Time advances in one direction; the time domain is continuously expanding, and the most recent time point is the largest value in the domain. This implies that a natural clustering on sort order will manifest itself, which can be exploited during optimization and evaluation. Query optimization can also consider time-oriented integrity constraints. $t_{start} < t_{end}$ holds for every time-interval tuple t.

Unfortunately, there is little work reported on processing temporal queries, and the work that has been published is often described in the context of relational databases.

As RDBs always require the user to explicitly join two relations, temporal processing in the context of RDBs has focused on specific join algorithms, following the bottom up approach. Gunadhi and Segev [1990] and Segev [1993] specified a temporal join and proposed an approach for optimization. Leung and Muntz [1990; 1993] proposed the strategies of stream processing for processing temporal joins (inequality join and semijoin). These strategies ware later extended to parallel processing strategies in multiprocessor database machines [1992]. Zurek [1998] presented a framework for parallel temporal joins.

Seshadri *et al.* [1996] proposed separating general database optimizing from temporal optimizing and provided a paradigm for interaction between a relation and a time sequence (here temporal data is represented by a time sequence). Although they have provided a comprehensive approach for sequence data processing, they have not incorporated sequence processing in the relational query processing framework.

Dayal and Wuu [1992] proposed a uniform approach to processing temporal queries in the context of a functional object-oriented data model. But their work did not take account query optimization and evaluation in a query processing framework. In addition, their work is based on the functional model and language. This leads to functional optimization that is

quite different from the algebraic, cost-based optimization techniques employed in relational, as well as a number of object-oriented systems [Ozsu, 1995].

It is worth noting that most object-oriented database proposals include constructors for complex types like lists and arrays that allow time-stamped entity to be represented as a "blob", which is managed by the system, but interpreted solely by the application program; no facilities for temporal queries are provided [Seshadri *et al.*, 1996].

3.5 Summary

Query processing and optimization problems in relational databases have been the focus of a great deal of theoretical and applied research. Much research is still being carried out in the area of RDBs, and a large body of knowledge, gathered over a period of almost two decades, on relational query processing and optimization techniques and strategies has accumulated. The great success of query processing in RDBs is attributed to the relational model of data that provides declarative queries and associated techniques for automated evaluation.

The problem of optimization of object-oriented queries has not yet been very widely researched. Critics of the object-oriented approach frequently point to the theoretical limits of optimization as a major drawback of the object-oriented approach as compared with the relational approach [Unland *et al.,* 1992; Ozsu and Blakeley, 1995; Kim, 1993; 1994; 1995; Kim *et al.,* 1997].

Almost all the object query processors proposed to date uses the optimization techniques developed for RDBs, as pointed out by Ozsu and Blakeley [1995]. The lack of universally accepted object data model and algebra makes it difficult to generalize research results. In general, it is impossible to achieve the same degree of optimization as in a relational language. It is important to develop extensible approaches to query processing that allows experimentation with new ideas as they evolve.

Although extensive activity in defining temporal data models by extending existing models, little work has been done on temporal query processing and optimization. Even so, it is mostly in the context of relational databases and follows a bottom-up approach: usually focusing on a specific algorithm such as a join. Because adding time creates multiple tuple versions for the same object, ignoring time will result rapid performance degradation due to ever-growing overflow chains or accumulated facts. As pointed out by Kim [1993; 1994; 1995; Kim *et al.*, 1997] and Snodgrass [1995], empirical studies are needed to compare storage and query evaluation strategies that support time-varying data.

At the moment, considerable research is needed to deal with optimization and execution of object-oriented queries when time is taken into account. Query processing in TOODBs remains a big challenge.

Chapter 4

A Temporal Object-Oriented Data Model *

This chapter defines a temporal object data model, which has been adapted from the unified model of RDB and OODB in UniSQL/X to which a time dimension has been added to form temporal relational-like cubes. Aggregation and inheritance hierarchies are also retained. The characteristic of the model will be analysed and a number of case studies are given to illustrate the model.

4.1 Introduction

A data model is a prescription of a way of representing data, and a prescription for a way of manipulating such a presentation. Defining a data model has always been the start point for developing a database.

The vast majority of research on temporal database systems has focused on developing a temporal data model by the incorporation of time elements into existing database models [Tansel *et al.*, 1993; Snodgrass, 1995; Ozsoyoglu *et al.,* 1995; Stonebraker *et al.,* 1990; John and Patrick, 1992; Pissinou *et al.*, 1993; Goralwalla *et al.,* 1998]. Compared with temporal relational models, little work has been reported on time in object-oriented databases (OODBs), although there is a significant increase in the work on defining temporal object database models recently. An OODB is a database system based on object-oriented data model concepts. One approach in introducing time into an object data model is to extend the semantics of a pre-existing snapshot model to incorporate time directly [Snodgrass, 1995].

*The model presented in this chapter has initially published in paper 6, and its modified version has been properly described in paper 1. The case studies have been presented in papers 8, 9, 10, 11, and 12. The *List of Author's Publications* includes paper 1-13.

However, there is currently no commonly accepted object data model, and definitions of temporal object-oriented data models vary.

The unified model of RDB and OODB from UniSQL/X [Kim, 1993; 1994; 1995; Kim *et al.*, 1997; D'Andrea and Janus, 1996] is extended from relational data model and has an potential in making use of relational database techniques to process object queries. In this chapter, the author will adopt the unified model of RDB and OODB from UniSQL/X as a snapshot object data model, and then incorporate within it a time dimension so that we can make use of the research results of temporal extensions to RDBs for (temporal) OODBs.

The remainder of this chapter is organised as follows. Section 4.2 describes the unified data model of RDB and OODB. Section 4.3 presents the temporal object-oriented data mode by defining the temporal object and integrating it into the unified model of RDB and OODB. Section 4.4 gives a number of application examples as case studies. Features of the data model are outlined in Section 4.5 and a summary of the chapter is given in Section 4.6.

4.2 The Unified Model of OODB and RDB

The unified data model of RDB and OODB from UniSQL/X [Kim, 1993; 1994; 1995; D'Andrea and Janus, 1996] extends the relational data model in three important ways, each reflecting a key object-oriented concept: (1) nested predicates; (2) inheritance; (3) methods. The mechanism for such an extension follows the basic tenet of an object-oriented system or programming language that the value of an object is also an object. We will use the example database schema in **Figure 4.1** to describe these extensions where each node is a relation (synonymous with a class). A node is divided into three levels, the first of which contains the name of the relation, the second the attributes and the third the methods or procedures attached. Two nodes *C* and *C'* may be connected by either a thin arc, indicating that *C'* is the domain of an attribute of *A* of *C* (or that *C'* is the class of the result of a method of *C*)--resulting in the aggregation hierarchy; or a thick arc, indicating that *C* is the superclass of *C'*--resulting in the inheritance hierarchy. Arrows indicate the directions of connection.

A RDB consists of a set of relations (tables), and a relation in turn consists of rows (tuples) and columns. A row/column entry in a relation may have a single value, and the value may belong to a set of system-defined data types (e.g., integers, string, float, date, time, money, etc.). The user may impose further restrictions known as integrity constraints, on these values (e.g., the temperature of a city may be restricted to between -80^{0}C and 80^{0}C). The user may then issue a nonprocedural query against a relation to retrieve only those tuples of the relation, the values of whose columns satisfy user-specified conditions. Further, the user may correlate two or more relations by issuing a query that joins the relations on the basis of comparison of values in user-specified columns of relations.

The first extension of UniSQL/X allows the value of a column of a relation to be a tuple of any arbitrary user-defined relation, rather than just an element of a system-defined data type. This means that the user may specify an arbitrary user-defined relation as the domain of a column of a relation. In **Figure 4.1,** the column *Weather* of *WEATHER-RECORD* (or the

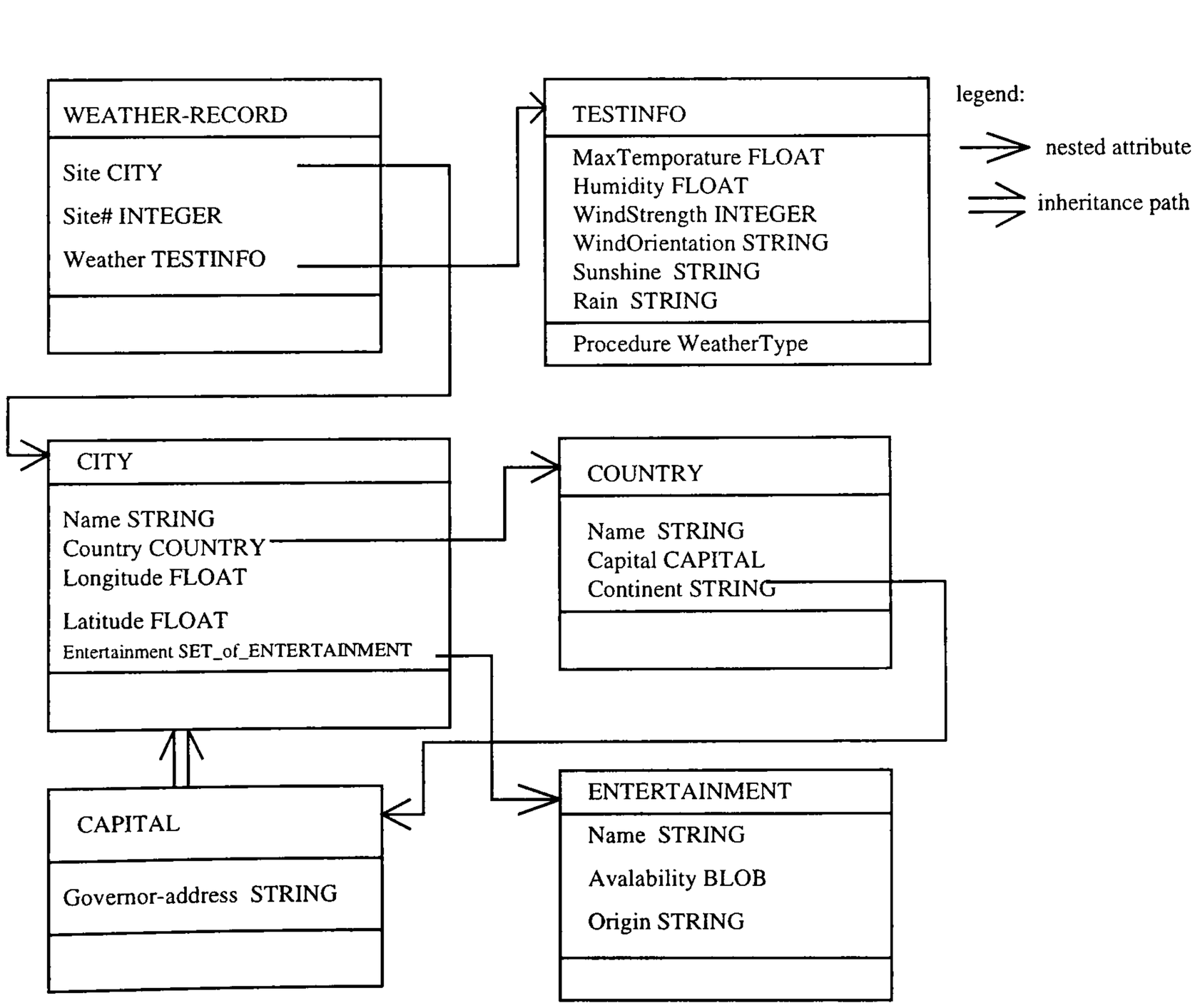


Figure 4.1 An example of OODB schema

Country of *CITY*) no longer needs to be restricted to a system-defined type e.g., string. It now is a tuple of a user-defined relation *TEST_INFO* (or *COUNTRY*). Allowing a column of a relation to hold another relation directly leads to a nested relation. That is, the value of a row/column entry of a relation can now be a tuple of another relation, and the value can in turn be a tuple of another relation, and so forth, recursively. This gives a database system the potential to support advanced applications such like multimedia systems (which manage image, audio, text data, and compound documents that comprise of such data), scientific data processing systems (which manipulate vectors, matrices, etc.), engineering and design systems (which deal with complex nested objects), and so forth. This is the basis for bridging the large gulf in data types supported in today's programming languages and database systems.

Second, allowing the users to attach procedures to a relation and to have the procedures operate on the column values in each tuple achieves the combination of data with a program. For example, in **Figure 4.1**, the procedure *WeatherType* summaries the weather information and gives output of weather type. Procedures for reading and updating the value of each column are implicitly available in each relation. A relation now encapsulates the state and behaviour of its tuple: the state is the set of column values and the behaviour is the set of procedures that operate on column values. The user may write any procedure and attach it to a relation to operate on values of any tuple or tuples of relation. There is virtually an unlimited application of procedures in this way.

Third, allowing the users to organise all relations in the database into hierarchy, such that between a pair of relations *P* and *C*, *P* is made the parent of *C*, if *C* takes (inherits) all columns and procedures defined in *P* (besides those defined in *C*), and table *C* may have more than one parent relations from which it may take columns and procedures, the relational model integrates the object-oriented concept of inheritance. The child relation is said to inherit columns and procedures from the parent relations (This is called multiple inheritance) An IS_A (generation and specification) relationship holds between a child relation and its parent relation. In **Figure 4.1,** the relation *CAPITAL* is defined as a child of relation *CITY*. *CAPITAL* automatically inherits the five columns of *CITY*, i.e., *Name,*

Country, Longitude, Latitude, Entertainment, even if they are not specified in its definition. The inheritance hierarchy offers two advantages over the conventional relational model of a simple collection of largely unrelated relations: (1) it makes it possible for a user to create a new relation as a child relation of one or more existing relations; the new relation inherits (reuses) all columns and procedures defined in existing relations and their ancestor relations; (2) it makes it possible for a system to enforce the IS-A relationship between a pair of relations. RDBs require the users to manage and enforce this relationship.

UniSQL/X also makes one more extension to the relational model: allowing the row/column entry of a relation to have a set of values (i.e., any number of values), rather than just a single value; and further allowing the set of values to be of more than one arbitrary data type. For example, the data type of column *Entertainment* of *CITY* is a set of *ENTERTAINMENT*, that is, the value of the column may be a set of tuples of a user-defined relation *ENTERTAINMENT* (e.g., carnival, horse racing, etc., each of which has a set of attributes). This extension is not an object-oriented concept, but it is designed to address a fundamental deficiency in the relational data model [Kim, 1994] that requires the column value to be atomic and therefore limits its modelling capability. It provides an ability to represent the many-to-many relationship between two collections along the aggregation hierarchy. The restriction in RDBs that the row/column entry may hold only a single value forces the users to create an extra relation and/or duplicate tuples in one relation if a column of a relation should hold more than one value. For instance, to model the above example of *CITY* and *ENTERTAINMENT* in a RDB, where a city may have more than one entertainment activity, either each tuple of the relation *CITY* needs to be duplicated for each value of the column *Entertainment*, or an extra relation, say *CITY-ENTERTAINMENT*, need to be created. The relation *CITY* and *CITY-ENTERTAINMENT* need to be joined to retrieve information about cities and entertainment activities.

UniSQL/X thus extends the relational model in four important ways. Although each extension individually may appear to be minor, the consequences of the extensions, individually and collectively, with respect to ease of application data modelling and /or subsequent increase in performance, can be significant. If we make an equivalence between

the post-relational and object-oriented terms, then "relation" equates to "class", "tuple of relation" equates to "instance of a class", "column" equates to "attribute", "procedure" equates to "method", "relation hierarchy" to "class hierarchy", "child relation" to "subclass", "parent relation" to "superclass", and "nested relation" to "aggregation hierarchy". These equivalencies can be expressed in **Table 4.1**. In this thesis, we use two sets of terms interchangeably.

Table 4.1 Equivalencies between post-relational and object-oriented terms

Post-Relational Model Terms	Object-Oriented Model Terms
Relation	Class
Tuple of relation	Instance of a class
Column	Attribute
Procedure	Method
Relation hierarchy	Class hierarchy
Child relation	Subclass
Parent relation	Superclass
Nested relation	Aggregation hierarchy

Compared to the ODMG Object Model described in Section 2.3.2, this model possesses most key features of ODMG Object Model. It is an object-oriented data model! Because it is also extended from relational data model and has thereby its counterparts in relational model, it provides a potential in exploiting relational techniques for the management of objects. This model is adopted as a snapshot model to incorporate time. Additionally, we preserve the basic object concepts such as any real-world entity is modelled as an object, each complex object is associated with a unique identifier, etc., so that *heterogeneity* in the time dimension and the *grouped completeness* of algebra can be maintained (this will be discussed later in the next chapter).

4.3 A Temporal Object Data Model

The temporal data are the record of the evolutionary history of entities. The states of entities may change in different ways, as shown in **Figure 4.2.** Representing temporal data in a database initially uses the interval description. It is quite suitable for step-wise constant data. Because continuous time-varying data can always be represented as discrete time-series in computers, time-point representation is often chosen as it makes it easy to generalise the various situations. We adopt the time-point representation and use temporal sets (temporal elements) as timestamps so that a temporal object can be represented by a time sequence, and the lifespan of an object can be associated at both attribute and tuple level for the unified model.

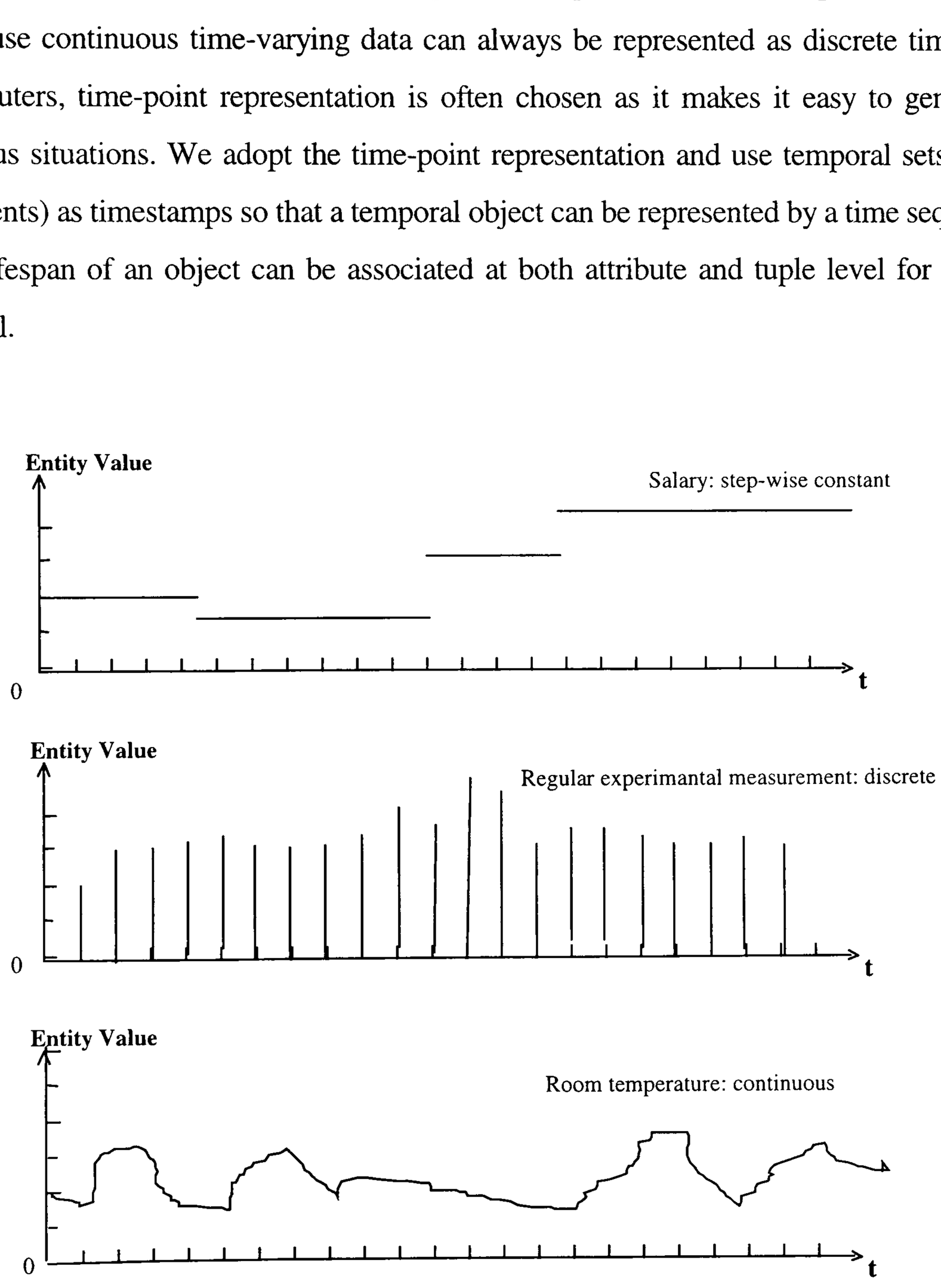


Figure 4.2 Three basic types of temporal data

4.3.1 Time Space and Temporal Set

A time space $T = \{..., t_0, t_1, ...\}$ is a set of times, at most countably infinite, over which is defined the linear (total) order $<^T$, where $t_i <^T t_j$ means t_i occurs before (earlier than) t_j. For the sake of simplicity, it can be assumed that T is isomorphic to the set of natural numbers $\{..., n-1, n, n+1, ...\}$. Any subset of T is called a **temporal set**. A temporal set can be represented as a union of disjoint time intervals. The most basic property of temporal sets is that they are closed under finite unions, intersections, and complementation. That is, if T_1 and T_2 are temporal sets, then so are $T_1 \cup T_2$, $T_1 \cap T_2$, $T_1 - T_2$, *and* $\neg T_1$.

For example, let $T_1 = \{1, 2, 5, 8, 23\}$ and $T_2 = \{2, 7, 9, 11, 23, 34\}$. Then

$T_1 \cup T_2 = \{1, 2, 5, 7, 8, 11, 23, 34\}$

$T_1 \cap T_2 = \{2, 23\}$

$T_1 - T_2 = \{1, 5, 8\}$

Absolute time indicates that a specific valid time at a given timestamp granularity is associated with a fact. For example, 3/4/1990 is an absolute time point. Such a time depends neither on the valid time of another fact nor on the current time, now.

Relative time indicates that a valid time of a fact is related to either the valid time of another fact or the current time, now. For example, seven days after his birth.

Both absolute time and relative time can be represented by a time space.

4.3.2 Chronon, Interval, Span and Lifespan

Definitions

A chronon is the shortest duration of time supported by a temporal databases, i.e., a nondecomposable unit of time.

A time interval is the time between two instants. For example, [1,5] is an interval.

A span is a directed duration of time. A duration is an amount of time with known length, but no specific starting or ending instants. For example, the span of [1,5] is 5 if the time chronon is 1. A span is either positive, denoting forward motion of time, or negative, denoting backwards motion in time.

A timestamp is a time value associated with some object, e.g., an attribute value or tuple.

The lifespan of a database object is the time over which it is defined. If the object (attribute, tuple, relation) has an associated timestamp, then the lifespan of that object is the value of the timestamp.

The above definitions are taken from [Jensen *et al.,* 1994; Tansel *et al.,* 1993].

If an object *o* exists in a certain period of time, which is a subset of *T* (i.e., the temporal set), this period is called the object's *lifespan*, denoted as *L(o)* for the object *o*. If the lifespan is continuous[†], it can be denoted as $L(o)=[t_{start}, t_{end}]$, the duration of time is called a span: $span(o)= t_{end} - t_{start}+1$. In order to support for derived lifespans, it is allowed the usual set-theoretic operations over lifespans. That is, if L_1 and L_2 are lifespans, then so are $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, and $\neg L_1$.

A temporal object is defined as a time sequence (TS for short): $\{t, o(t)\}$, $t \in L(o) \subset T$, denoted as $<L(o), o(t)>$, where $o(t)$ represents object *o*'s value at the time *t*. A temporal object $<L(o), o(t)>$ asserts that the object $o(t)$ is valid for its lifespan $L(o)$ *and* its value changes with time. If a TS contains a value for each time point in the lifespan duration, it is called a regular TS [7]: $<L(o), o(t)> = \{..., t_{i-1}, o_{i-1}; t_i, o_i; t_{i+1}, o_{i+1}; ...\} = \{..., o_{i-1}, o_i, o_{i+1}, ...\} = \{o_i.\}$, where o_i represents object *o*'s value at the time point t_i. If a TS contains values for

---

[†] In principle, the lifespan L(o), a subset of T, is not necessarily required to be continuous, though it is required to be in this thesis.

only subset of time points within the lifespan, it is called an irregular TS: $<L(o), o(t)> =\{..., t_{i-1}, o_{i-1}; t_i, o_i; t_{i+1}, o_{i+1}; ...\}$.

A discrete time event, where the value of the entity is recorded at every single time point, as shown in **Figure 4.2,** can be represented by a regular TS. A discrete time event, where the value of the entity is not recorded at every single time point, can be represented by an irregular TS.

A step-wise constant, as shown in **Figure 4.2,** can be represented by an irregular TS where the value o_i at time t_i, is assumed to retain for $[t_i, t_{i+1})$. We use the term *epoch* from signal processing field to refer to the time at which the object changes its value, e.g., t_i. The interval during which the value o_i persists is decided by the epoch t_i and its succeeded epoch t_{i+1}, i.e., $[t_i, t_{i+1})$. If there are n elements in a TS, it is said that there are n epochs. For example, suppose John has worked for a company from 1975 to 1998, his salary was initially 1500 and has been changed to 1900 in 1978, change to 2300 in 1984, 2700 in 1991, and 2900 in 1996. If the time chronon is assumed as a year, the lifespan of John's salary is [1975, 1998], and

<[1975,1998], John's Salary>
={1975, 1500; 1978, 1900; 1984, 2300; 1991, 2700; 1996, 2900}

where the first salary 1500 retains for [1975, 1978) and the last salary 2900 retains for [1996, 1998]. The epoch number is 5. It can be seen, from the later discussion in Chapter 6, 7 and 8, that epoch represents a transformed time space and will serve as a convenient indicator for cost analysis and query processing.

A continuous time event, as shown in **Figure 4.2,** depending on the recording of the data, can be represented by either a regular TS or an irregular TS. When it is represented by a regular TS, it is treated as a discrete time signal created by sampling the corresponding continuous time signal. As long as the sampling frequency is greater than two times the highest frequency of the signal, the continuous time signal can be recovered from the

discreet time signal [Oppenhein and Schafer, 1975]. If it is represented by an irregular TS, as it is time-varying the value between two recorded time points can be decided by an interpolation function depending on the application, e.g., linear interpolation. This will be further discussed in Chapter 8.

A constant object *o*, may be represented with no timestamp where its time-reference is implied as *L(o)*. It can also be represented with an explicit time-reference as a temporal object: <*L(o)*, *o*>.

As a TS is a set, so a temporal object can be represented by its sub-objects. In practice the lifespan may consist of disjoint, noncontiguous segments. As in [Ginsbury, 1993], we prefer to use null rather than defining multiple segments in the lifespan. For instance, if we know Mary's salary records during the time [1967, 1982] and [1990, 1998] as {1967, 1400; 1977, 1890} and {1990, 2000; 1996, 2100}. Although we do not know her salary between 1982 and 1990, we will have {1967, 1400; 1977, 1890; 1982, null; 1990, 2000; 1996, 2100} where null is persisted from 1982 till 1990 when the value 2000 exists.

4.3.3 Integrating the Temporal Object with the Unified Model of RDB and OODB

In the OODB represented by the unified model of RDB and OODB, every real world entity is uniformly modelled as an object that is grouped into a class (relation) and interrelated to other objects through associations. Now we take a class (relation) *C* (disregarding its associations of aggregation and inheritance hierarchies), as shown in **Table 4.2**.

Table 4.2 Interaction of tuple lifespan and attribute lifespan

Relation	A_1	A_2	...	A_n
$tuple_1$				
$tuple_2$				
...	...	...	...	
$tuple_m$				$value_{m,n}$

If the $value_{m,n}$ is a temporal object with lifespan $l_{m,n}$, and the $tuple_m$ is also a temporal object with the lifespan denoted as $L(t_m)$, we have

$$L(t_m)=l_{m,1} \cup l_{m,2} \cup ... \cup l_{m,n}$$

The lifespan of attribute A_n is

$$L(A_n)=l_{1,n} \cup l_{2,n} \cup ... \cup l_{m,n}$$

The lifespan of relation C is

$$L(C)=L(A_1) \cup L(A_2) \cup ... \cup L(A_n) = L(t_1) \cup L(t_2) \cup ... \cup L(t_m)$$

Thus a 2-dimensional relation (class) "table" becomes a 3-dimensional "cube", as shown in **Figure 4.3,** if objects in the relation have uniformly the same lifespan.

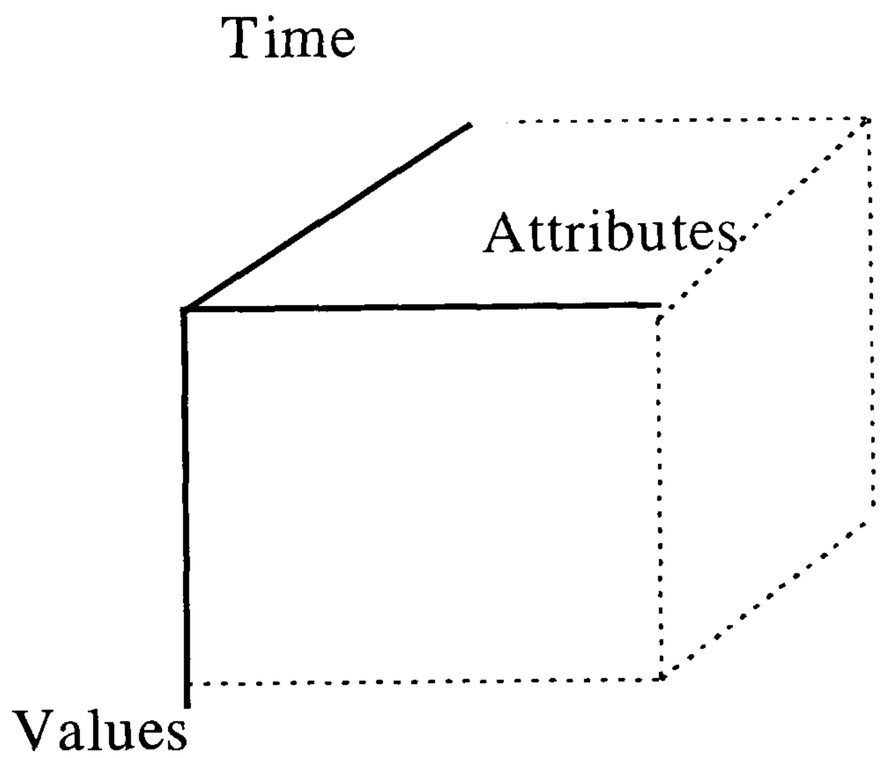


Figure 4.3 A 3-dimensional class

It is obvious that

$$l_{i,j}=L(t_i)\cap L(A_j)$$

This implies that there is no value for an attribute in a tuple for any moment in time outside the intersection of the life spans of the tuple and the attribute. Obviously our temporal object model can support a completely heterogeneous temporal dimension, but at the cost of maintaining a distinct lifespan for each value. This is important because homogeneity is sometimes difficult to maintain, although homogeneity is necessary as no time-slices of a homogenous relation produce null values [Pissinou *et al.*, 1994].

If the domain of attribute A_i of class C is another class C', then implicitly, $L(A_i)=L(C')$. If class C' is the superclass of class C, then $L(C')=L(C)$. If the class C' has more than one subclasses, e.g., C_1 and C_2, then $L(C')=L(C_1)\cup L(C_2)$. Moreover, if a database consists of n classes (relations) $C_1, C_2, ..., C_n$, the lifespan of the database schema is $L= L(C_1)\cup L(C_2)\cup...\cup L(C_n)$.

It is possible to refer to the components of a temporal object. For a temporal object $o=<T,o>$, $o.\upsilon$ and $o.T$ refer to its value and temporal set components, respectively. For the above salary example, let o represent John's salary, $o.\upsilon$ ={1500, 1900, 2300, 2700, 2900} and $o.T$ ={1975, 1978, 1984, 1991, 1996}. Sometimes we omit υ, i.e., $o.\upsilon=o$, (or $o.\upsilon(t)=o(t)$) to refer to the value of the object o without causing a confusion, e.g., o ={1500, 1900, 2300, 2700, 2900} refers to John's salary history. It is especially the case when talking about a regular TS, we use $o(t)=\{,...o_{i-1}, o_i, o_{i+1},...\}$.

Let A represent the name of an attribute that can take a temporal object for its values, then $A.\upsilon$ and $A.T$ represent the value set and temporal set components of the attribute A. Further, the same notation may be applied to class (relation) C. If C is a temporal relation, then $C.\upsilon$ and $C.T$ represent the value set and temporal set components of the class C.

Taking the database schema in **Figure 4.1** as an example, its temporal database schema can be represented as shown in **Figure 4.4**. Note that now the class/relation *TESTINFO* becomes the temporal relation where daily weather changes are recorded for main cities. For simplicity we suppose that the temporal set (time stamps) starts at time 1 and ends up today (*n*), and time chronon is a day. Then the lifespan of relation *TESTINFO* can be uniformly represented as $L=\{1,...,n\}$, i.e., *TESTINFO* is a regular TS. Other relations are constant relations whose lifespans are implied the same as *L*.

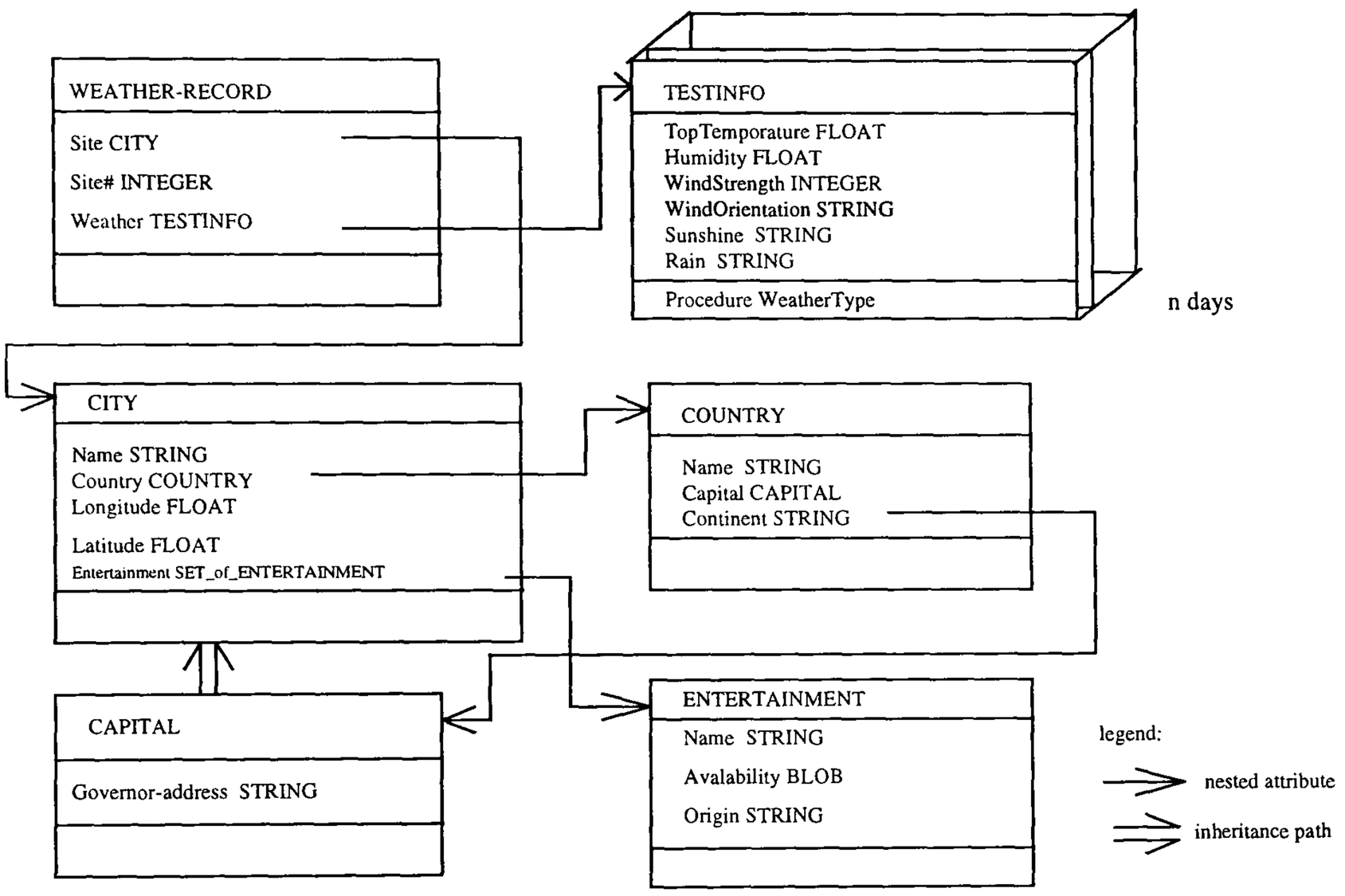


Figure 4.4 Database schema of "International Weather Record Database"

4.4 More Application Examples

To illustrate the applicability of the temporal object data model to real world problems, this section provides two more examples as case studies.

4.4.1 Case Study 1: "The Wood Panel Deformation Measurement Database"

In this case study, the model defined in the previous section is applied to a real world sequential image measurement system. It deals with the problems of data modelling and

management of sequential image database that can not be handled by conventional databases (e.g., RDB or OODB).

"The Wood Panel Deformation Measurement System" [Chen *et al.*, 1994; Robson *et al.*, 1995; Clarke *et al.*, 1995] brings together the results of recent research in art observation, electro-optical image processing and advanced database management in order to increase knowledge and understanding of deformation and cracking of wood panel paintings (which lead to paint loss) caused by changes in ambient conditions. A deformation analysis of movement occurring in wood panel was required by the Hamilton Kerr Institute of the Fitzwilliam Museum, University of Cambridge, where 74 wood panels used for supporting fine art paintings were tested. An automated 3-D measuring system using photogrammetric and machine vision techniques has been developed at City University. The panels to be measured were divided according to wood type: linden; oak; poplar; and Scots pine. Each type was supported by a number of different reinforcement types to give 74 panel reinforcement combinations. An array of retro-reflective targets were placed on each test panel. The number and disposition of the targets on each test panel varied from 175 to 464 according to the pattern of auxiliary supports. The total number of epochs (the number of sequential images of a test panel) was 25 (i.e. 25 humidity levels at different time). The experiment was carried out in a uniform way, i.e., for each panel, there were 25 tests for different humidity levels at different times. **Table 4.3** gives such an example. For each epoch, there were about 400 images in total to be grabbed by 5 cameras at different positions, which occupied about 170M storage. Therefore over 10,000 images were grabbed and processed. The average number of targets on each test panel was 250, resulting in a total of 2,500,000 targets to be processed.

Table 4.3 Sample experiment setting

day	0	10	20	21	31	41	42	52	62	...
rh%	30	30	30	70	70	70	30	30	30	...
epoch	0	1	2	3	4	5	6	7	8	...

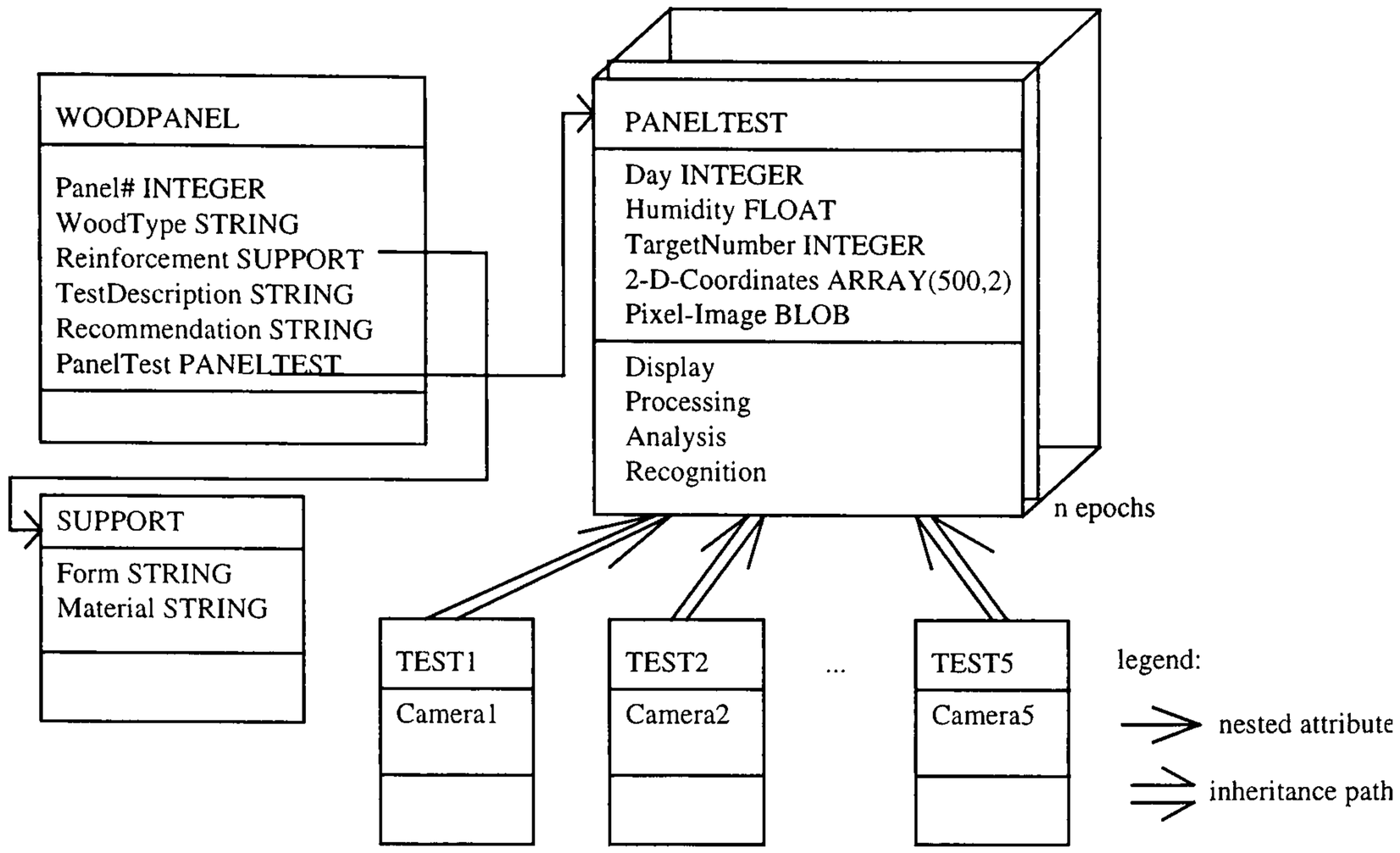


Figure 4.5 Database schema of "Wood Panel Deformation Measurement System"

This database application represents a typical scientific application, collecting results from observation, experiments and simulation, and has features of large amount of data, complex data types (data, text and image), and timestamps (sequential images). It also involves a lot of data processing (e.g. image processing, pattern recognition, 2-D to 3-D image construction, etc.).

Applying our temporal object data model to this case, we have generated a database schema presented in **Figure 4.5.** *WOODPANEL* is a constant relation with a set of attributes. The domain of column *Reinforcement* of *WOODPANEL* is another relation *SUPPORT*. The domain of column *PanelTest* of *WOODPANEL* is another relation *PANELTEST*. *PANELTEST* is a temporal relation with a set of attributes that are of different data types. Note that absolute time (e.g., 12/3/1997) is not important, it is time relative to the fact of the test that is of interest. The time chronon is a day. If each test is completed within 6 months, say, 180 days, the lifespan of *PANALTEST* is [0,180) whist the lifespan of other constant relations implies the same as [0,180). *PANELTEST* is an irregular TS and there are n epochs (e.g., n=25) in this relation. Data/image processing procedures are represented as

methods to attach to the relation *PANELTEST*. *TEST1* to *TEST5* are subsets of *PANELTEST*, and inherit all three dimensional attributes and methods from *PANELTEST*. *<L, TEST1>* represents the sequential test information and images grabbed by *Camera1*.

4.4.2 Case Study 2: "The Neurological Patient Care Database"

This case study concerns the data modelling and management in a health-care information system. From this practical example we will see how the time granularity is determined and different forms of tested data that may come from different sources are represented in a temporal object database.

Physicians need to draw upon many different kinds of information in the course of their work. Health informatics [Pickover, 1995; Silberschatz, 1996] is therefore emerging as a field that concerns itself with the organisation and management of information in support of patient care, education, research, and administration. It draws from disciplines such as cognitive and educational psychology, decision theory, information science, and computer science. The application of health informatics relies on the use of computer and communication technology to translate theory into practice. The database management support plays an essential role to make this become reality [Silberschatz, 1996], as a data model provides concepts and constructs for data modelling/processing required by real-world organisations and a database management system incorporates a data model and provides high-level facilities for storage, retrieve and maintenance of data.

Much of the difficulty in managing health-care information systems comes from the different sources of data that is involved, complex data structures, historical information collection, data processing or dealing probabilities in clinic reasoning; but the organisation of data is also a major problem [French *et al.,* 1990]. Here we look at "The Neurological Patient Care Database" example.

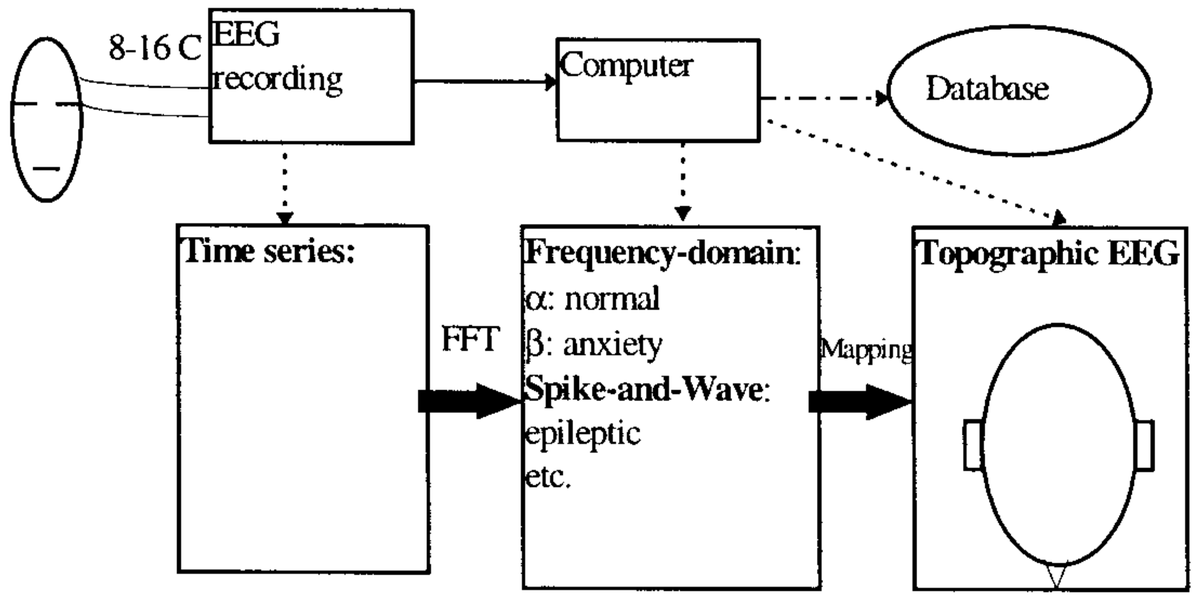


Figure 4.6 Computer-aided EEG system

Electroencephalograph (EEG) [Carpenter, 1996] is a technique of recording the electrical activity of the brain through the intact skull. Electrodes are applied to the scalp and potential changes so recorded are amplified and presented for interpretation as an inked tracing on moving paper. Machines in common use have eight, sixteen or more channels so that the activity from many different areas of the head can be recorded simultaneously. The technique is simple and harmless and may give valuable diagnostic information, particularly in patients with suspected epilepsy, encephalitis, etc. But the time-domain signals are difficult for doctors to read. With the aid of a computer, as shown in **Figure 4.6**, the frequency-domain information such as the rhythms of α, β, δ, θ, etc. which reveal the direct correlation with the state of a patient, is easily provided through FFT transformation. The "topographic EEG" that is thereby created represents a more recent development of quantitative EEG method. However, using this approach, the most valuable information on "spike-and-wave" that is significant to the diagnosis of epilepsy becomes less distinct. The reason is that the measured EEG signal with "spike-and-wave" is a random process. Strictly speaking, there does not exist FFT for such a signal. Therefore looking into the time-domain signal or searching for different data processing methods is sometimes necessary. In addition, sole EEG index is not enough for doctors to make the diagnosis of cerebral diseases such as clinic doctors can not diagnose the structural damage on the brain like a cerebrovascular disease without looking the patient's CT scan image. Therefore sharing information with other systems is unavoidable. In short, clinical decision making, biomedical computing, reuse and

sharing of medical information, support for further research and education all require for data management support. Our temporal object data model provides a good approach to data modelling and management for such a medical information system.

The database schema is designed for "The Neurological Patient Care Database" as shown in a simplified form in **Figure 4.7**. The relation *PATIENT* in concern has a set of attributes. Major EEG examination result is represented as a column of the relation. Some other assistant examination result (e.g., CT) is represented as another column. The domain of each of these two columns is another relation: *EEG* (or *OTHERS*). The chronon of time can be determined as a day that suits for hospital daily routine. Suppose the lifespan of *EEG* is *L(EEG)*. (Instances of *EEG* do not need to be of the same lifespan). There are *n* epochs in relation *EEG*, representing *n* examinations (Instances of *EEG* do not need to be of the same times of examination). In each examination, there recorded a time-domain signal (as an attribute value) where the short time sequence is represented as 'blob'-like data, as the duration of the time sequence is much smaller than the chronon of time that needs to be recorded. There are *m* epochs in relation *OTHERS* with lifespan *L(OTHERS)*. *OTHERS* may be recorded and stored at different site. The lifespan of constant relations is implied as $L(EEG) \cup L(OTHERS)$. *EEG-IMAGE* and *CT-IMAGE* are subclasses of *IMAGE*, therefore they take all properties and methods from *IMAGE*, alongside their own properties and methods.

The system represented by our data model has the following features:

(1) Integration of data, text, and images (that come from different sources) where some metadata [French *et al.*, 1990] such as description of test, etc. are uniformly represented into attributes;

(2) Representation of collection of historical data where the time series of each EEG record is represented as 'blob' in database schema instead of temporal data;

(3) Reuse of programs: the methods supporting image viewing, processing, etc., can be reused by its any subclass images.

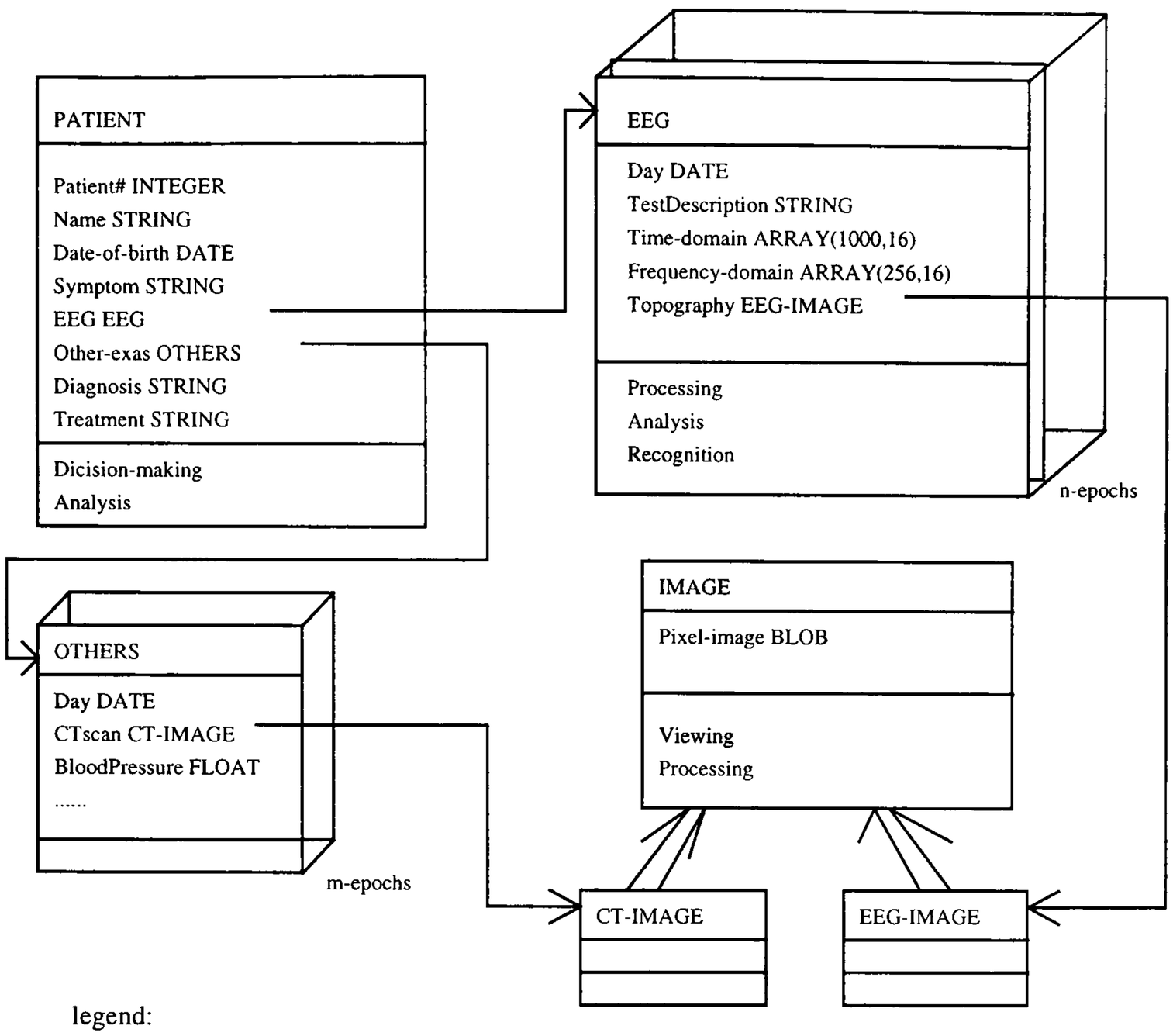


Figure 4.7 Database schema of "The Neurological Patient Care Database"

Further data processing/analysis for research purpose is one of salient features of health-care information systems. The temporal OODB architecture has a generic connection with the clinic data processing/ analysis procedures (system). The connection is based on Object Linking and Embedding (OLE) techniques [Microsoft, 1993], as shown in **Figure 4.8**.

For example, **Figure 4.9** shows different data processing algorithms that require time-domain EEG signal from "The Neurological Patient Care Database". The data required can be supplied by the database management system. The output of data processing and

analysis procedures such as different topographic EEG, etc., can then be added to the databases for future use.

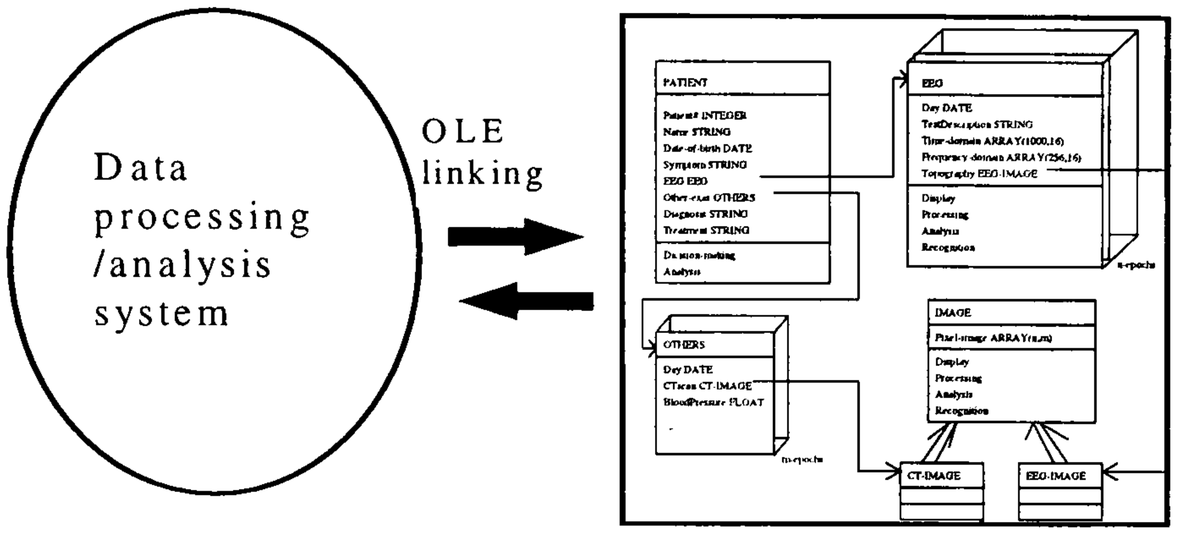


Figure 4.8 OLE link between database and analysis system

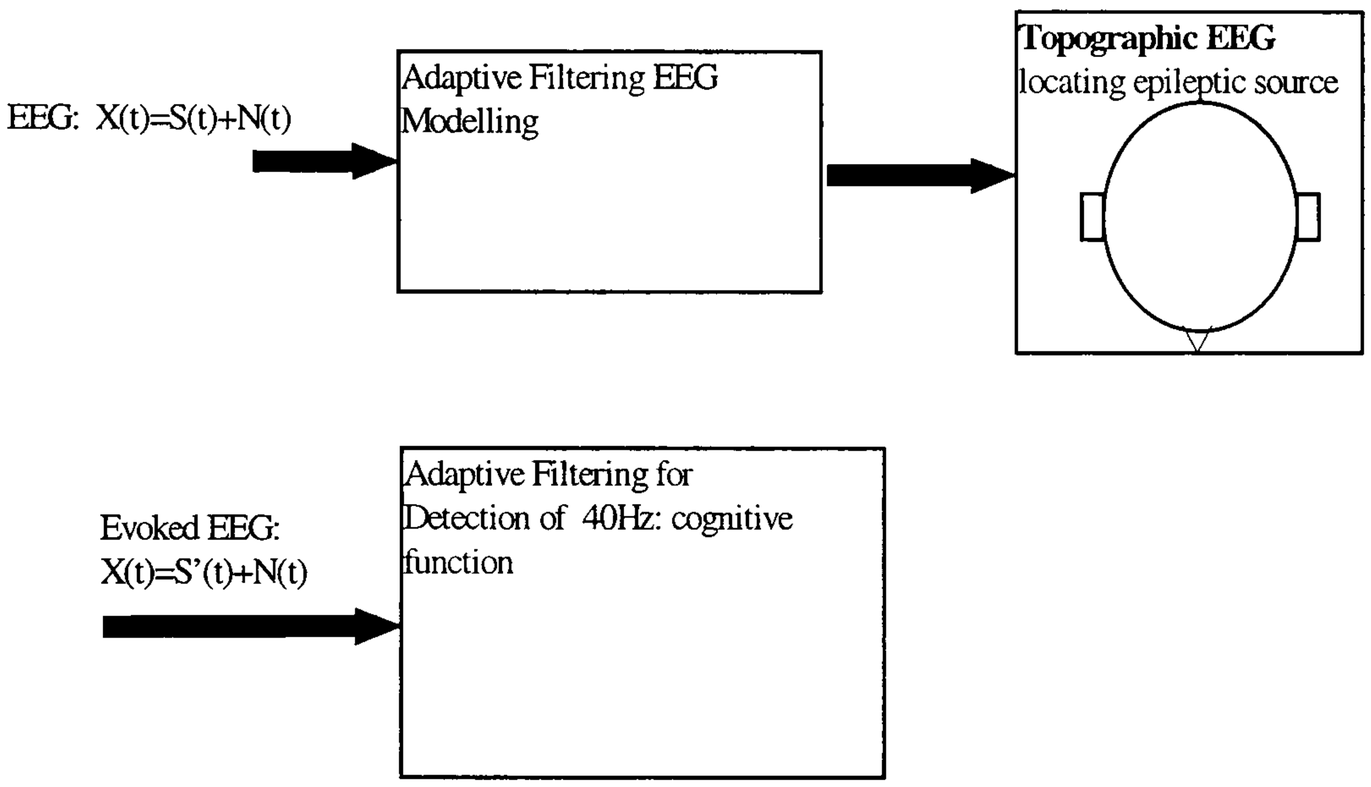


Figure 4.9 Different data processing procedures

It can be seen that the temporal object-oriented database approach provides a data modelling capability for data representation in the health-care information system and direct database support for the corresponding data manipulation. Further data processing/analysis, as well as other research procedures can be easily supported by the database.

In addition, because our data model is adapted from the unified data model RDBs and OODBs, it provides a good basis for interoperating OODBs with RDBs and thereby distributed heterogeneous databases. That is highly required by health-care information systems for information sharing and interoperation between different databases at different sits.

4.5 Features of the Temporal Object-Oriented Data Model

Although temporal databases have been an active area of research for over fifteen years, there is no commonly accepted data model. It is advocated in [Segev *et al.*, 1995] that a very simple conceptual data model is adopted that captures the essential semantics of time varying relations. Our work pursues this theme. The model presented in this chapter possesses the following characteristics.

1) The model is grouped and supports both homogeneity and heterogeneity in the time dimension

A temporal tuple is temporally homogenous if the lifespans of all attribute values within it are identical. A temporal relation is said to be temporally homogeneous if its tuples are temporally homogeneous. A temporal database is said to be temporally homogeneous if its relations are temporally homogenous. Models that employ tuple timestamping rather than attribute-value timestamping are necessarily temporally homogeneous as only temporally homogeneous relations are possible. On the other hand, those models that employ attribute-value timestamping rather than tuple timestamping can be temporally heterogeneous. The motivation for homogeneity arises from the fact that no time-slices of a homogeneous relation produce nulls. Support of homogeneity sometimes could create duplicate attribute values and is therefore difficult to maintain. In such cases, heterogeneity in the time-dimension is important.

Those models which employ tuple-time-stamping are termed *temporally ungrouped* whereas those models that employ complex attribute values bearing the temporal dimension are

termed *temporally grouped* [Pissinou *et al.,* 1994]. The temporally grouped model is commonly accepted to be desirable [Segev, Jensen and Snodgrass, 1995], as it will lead to *grouped completeness* of algebra, that can support the rather strong notion of the "history of attribute" (this will be discussed in the next chapter).

A number of reports defining a temporal relational data model, including Clifford [1993], Tansel [1993; 1997], Gadia *et al.* [1993], Ginsburg [1993] and Kafer *et al.* [1990], etc., assume homogeneity in the time-dimension.

Most object-oriented databases and post-relational products include constructors for complex types like lists and arrays that allow time-stamped entity to be represented as a "blob", both homogeneity and heterogeneity in time-dimension can be supported. But the "blob" is managed by the system, the interpretation is made solely by the application program. A temporal object data model with the *heterogeneity* in time dimension is also a grouped model [Pissinou *et al.,* 1994].

The data model defined in this dissertation is an object-oriented data model. Every real world entity can be represented as an object. In the database schema, an object can represent either a relation (class) tuple or an attribute value. The data model is also a temporal data model. An object can be either time-varying or constant. Therefore our data model is temporally grouped. Heterogeneity in the time dimension can be supported. Of course, homogeneity in the time-dimension can be supported when it is necessary, but at the cost of maintaining a uniform temporal set and lifespan, as it is special case of heterogeneity.

2) The model uses epochs that represent a transformed time-space

We borrowed the term *epoch* from the signal processing discipline to represent the time when an entity changes its value. The ordered epoch numbers constitute a transformed time space (as shown in **Figure 4.10**). From the query processing point of view, at each epoch, a new value of the entity will be created, which of course requires space to store the new value. Also, time is required to retrieve the value of the entity from the storage. Epochs, then, serve

as a convenient token for the analysis of the query processing cost (this will be further discussed in later chapters).

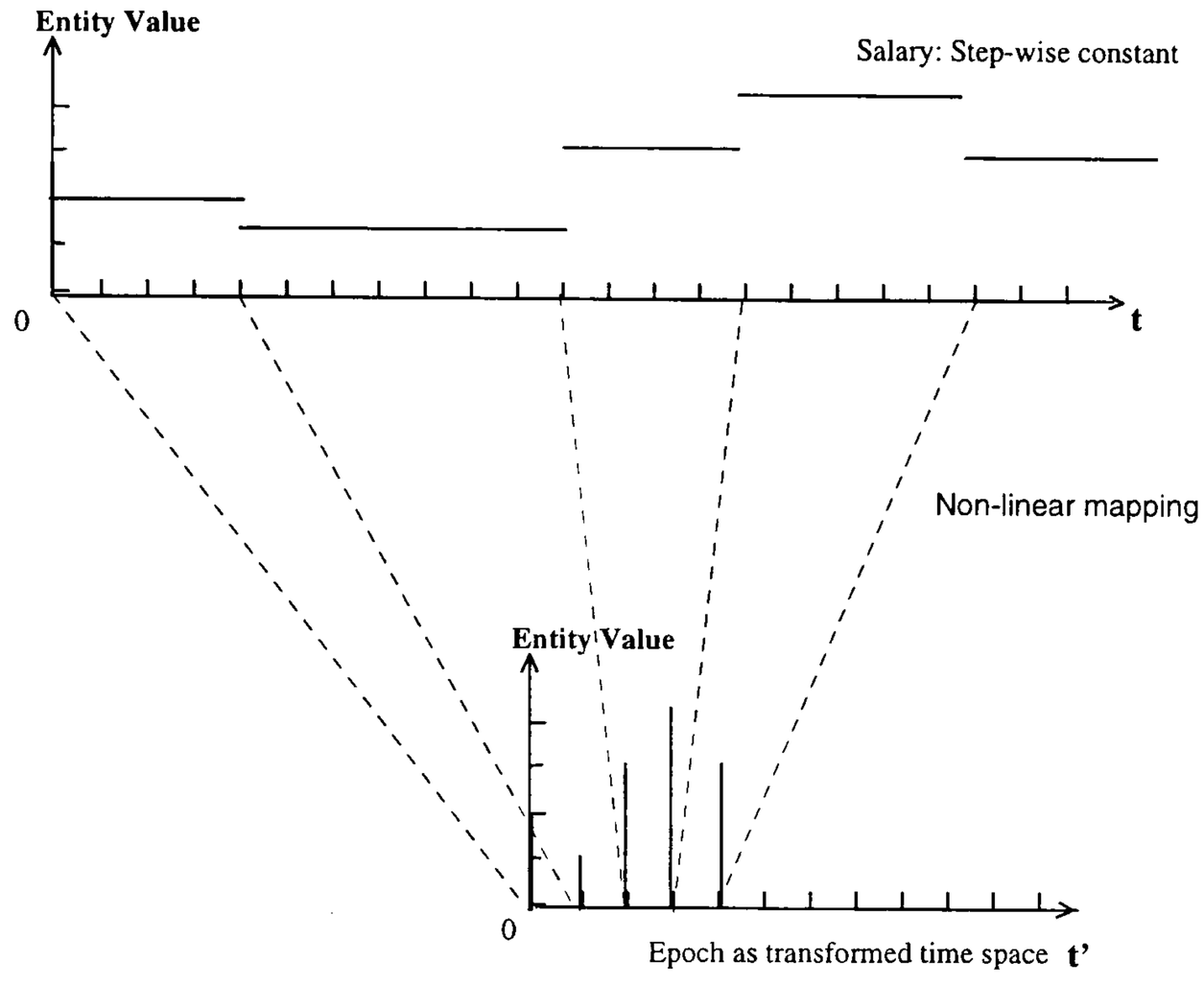


Figure 4.10 Illusion of mapping on time-spaces

3) The model possesses an extensible structure

The object-oriented data model presented in this chapter is adapted from the unified model of OODB and RDB to which a time dimension has been added. The database schema is in the form of relational-like cubes but with aggregation and inheritance hierarchies. So the temporal object-oriented database is a superset of object-oriented database that in turn is a superset of relational database. This provides a basis to extend the well proven query processing techniques of RDBs and TDBs to process temporal object queries.

4.6 Summary

In this chapter, a temporal object data model has been presented, which has been adapted from the unified model of OODB and RDB in UniSQL/X to which a time dimension has

been added to form temporal relational-like cubes and the aggregation and inheritance hierarchies are also retained.

Compared to other temporal relational or object-oriented data models, the temporal object data model defined here possesses following distinguishable characteristics:

- It is a temporally grouped model and supports both homogeneity and heterogeneity in the time dimension;
- It uses *epochs* that represents a transformed time space and can serve as a convenient token for the cost analysis of the query processing.
- The temporal object-oriented database represents a hierarchical structure with three types of associations: aggregation, inheritance and time-reference.

The temporal object-oriented data model determines access primitives and provides a basis for query processing, which will be discussed from the next chapter.

Case studies were also provided to demonstrate the applicability of the model to real world problems.

Chapter 5

An Algebra for the Temporal Object Data Model*

This chapter develops an algebra for the temporal object data model described in the previous chapter. As the temporal object data model is adapted from the unified model of RDB and OODB in UniSQL/X to form temporal relational-like cubes but with aggregation and inheritance hierarchies, a query algebra that provides operations for data accessing and manipulation through the associations of aggregation, inheritance and time-reference, reflects the spirit of both temporal relational algebra and object algebra. Data query examples from the Wood Panel Deformation Measurement Database illustrate algebraic operations, and the properties of the algebra are outlined.

5.1 Introduction

From the algebra point of view, a temporal OODB defined by the data model presented in the previous chapter can be viewed as a collection of temporal objects, grouped together in classes (relations) and interrelated through associations of aggregation, generalisation and time-reference. Each temporal relation can be viewed as a 3-dimensional cube. If the existing structure of "inheritance" hierarchy and "aggregation" hierarchy between classes is not considered, the structure of queries is essentially the same in both the RDB and OODB paradigm. The only effect that the temporal dimension has is to transform some tables (or even only some attributes) to cubes. There already are some reports on algebraic operations in temporal relational databases [Tansel, 1993; Clifford, 1993; Gadia, 1988; Mckenzie and

---

* The work presented in this chapter has been published in the paper 6 listed in *Author's Publications*.

Snodgrass, 1991; Tansel and Tin, 1998]. We therefore have a common base to expand (temporal) relational algebra to temporal object algebra.

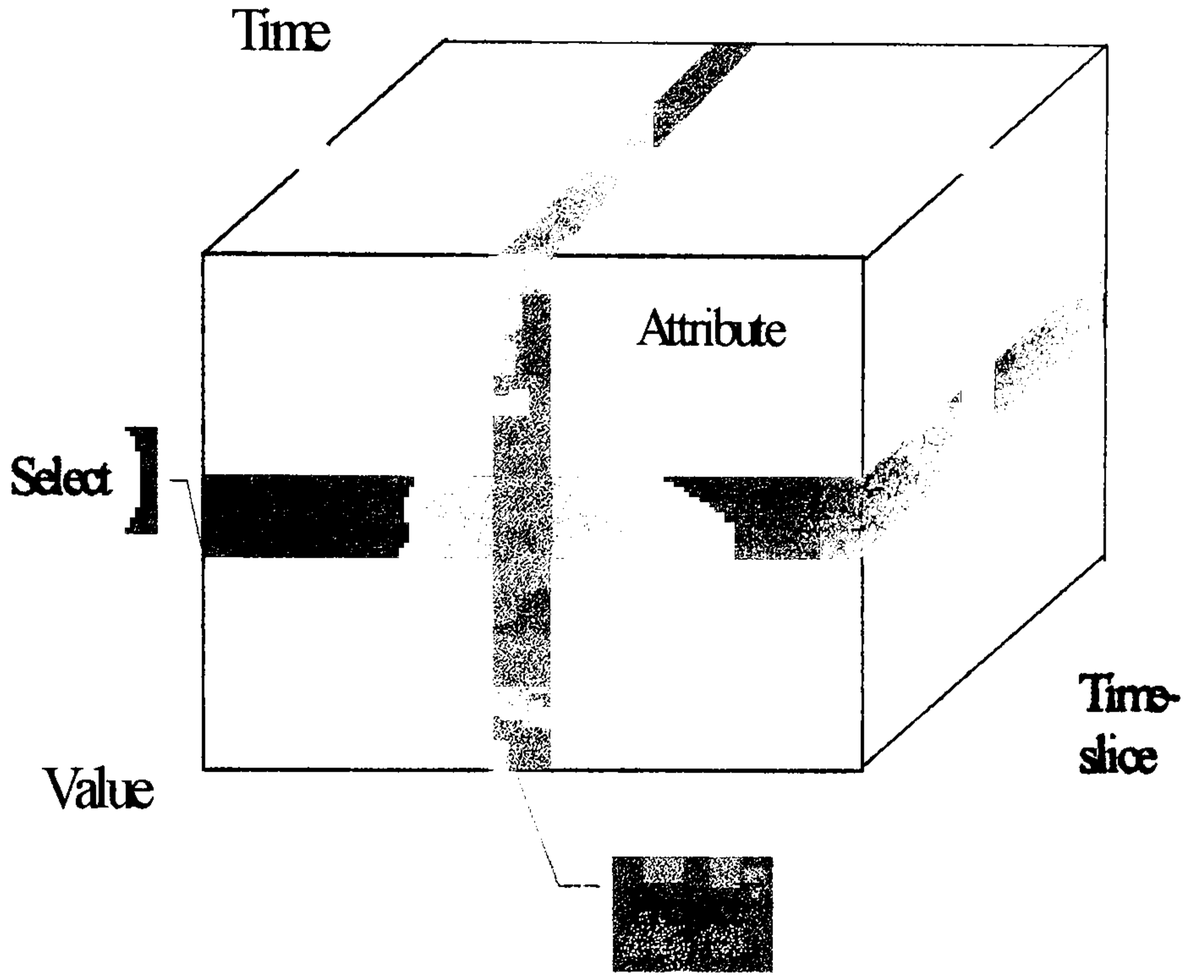


Figure 5.1 Illustration of basic algebra. Time, Attribute and Value are three dimensions of a relation. Select, Project and Time-slice are basic operations on these three dimensions.

Basically, the standard relational algebra provides a unary operator for each of its two dimensions: *select* for the value dimension and *project* for the attribute dimension, as shown in **Figure 5.1**. Temporal relational algebra introduces the operation of time-slice that operates on the time dimension. An object algebra allows the predicate of the *select* operation on a contiguous sequence of attributes along a branch of the class-aggregation hierarchy (which is usually expressed by a path--this concept will be discussed later in this chapter). The algebra which we are going to define for the model will extend the (temporal) relational and object algebra to address the features of the aggregation hierarchy and time dimension. The algebra is defined against a set of objects (which could be regarded as equivalent to class/relation). This concept is preserved so that it can readily take advantage of

inheritance and enable applications to automatically reach any existing objects of interest, without requiring explicit references to those objects [Yu and Osborn, 1991][†].

The remainder of this chapter is organised as follows. Section 2 specifies and classifies the predicates that appear in algebraic operation. Section 3 specifies the properties of identity and equality. A definition of algebraic operations is given in Section 4. Query examples and a brief evaluation are provided in Section 5. Properties of the algebra are presented in Section 6 and Section 7 gives a summary. Through out this chapter, examples are taken from the Wood Panel Deformation Measurement Database presented in Chapter 4.

5.2 Predicates

Predicates play an essential role in query evaluation and processing in any database. There are basically three types of predicate in our temporal object-oriented database: a simple predicate, a nested predicate and a temporal predicate.

A simple predicate is of the form <*attribute-name operator value*>. The value may be an instance of a primitive class (type) (e.g., string, integer, etc.) or an object identifier (OID) of the instance of some class. The latter is important because it may be used for testing the object equality, that is, equality of referenced objects. The operator may be a scalar comparison operator (=, <, >, etc.) or a set comparison ($\in$, $\subset$, $\subseteq$, set-equality, etc.). Examples of simple predicates are: *Panel#*=3, *Reinforcement*=001 where 001 represents the OID of an object in class *SUPPORT*.

A nested predicate is a predicate on a contiguous sequence of attributes along a branch of the class-aggregation hierarchy of a class. Path-expressions [Bertino and Martino, 1993] are defined to express a nested predicate.

[†] In the object-oriented methodology, the subclass will automatically take (inherit) all attributes and methods defined to its super-class, apart from its own attributes. So there is generally no need to define inheritance-like operators in a query algebra.

Definition *Path expression*

Given an aggregation hierarchy *H,* a path *P* is defined as

$C_1.A_1.A_2....A_n(n\geq 1)$

where

C_1 is the class in *H;*

A_1 is an attribute of class C_1;

A_i is an attribute of a class C_i in *H*, such that C_i is the domain of the attribute A_{i-1} of class C_{i-1}, $(1<i\leq n)$.

Path expressions can be compared using the comparators =, ≠, ≥, ≤, <, >, etc. Since path expressions represent sets, these comparators may have to be qualified with the use of *some* or *all*. Path-comparisons can be combined with Boolean connectives *and, or, not*. Path expressions can also be compared using standard set-comparators *contains, containsEq, subset, subsetEq*, etc. For example, *WOODPANEL.Reinforcement.Form* is a path, and *WOODPANEL. Reinforcement.Form = "lattice"* is a nested predicate.

A temporal predicate is a predicate referring to a temporal set in the time dimension. There are two types of temporal predicates: a simple temporal predicate and a nested temporal predicate. A simple temporal predicate can be expressed as <*temporal-set operator value*>. The operator could be <, ≤, =, >, ≥, and the combination of these, representing the semantics of time such as *before, until, while, after, since, during,* etc. For example, '<' in $t<4$ retains the semantic of *before,* representing T_0=[1,2,3] if t starts from 1. t=5 retains the semantic of *while* time is 5, and $5\leq t\leq 10$, represents an interval T_1=[5,10] that applies to *during*. A nested temporal predicate can be expressed by integrating the path-expression into a simple temporal predicate. If *o* is an attribute name or a path-expression or a predicate, we use the function *when* denoted as $\varpi(o)$, to express the temporal domain of *o* (we will give a formal definition later). For example, we use the following expression to refer to time point(s), at

which the image's humidity level is, say, 30%rh: $\varpi(WOODPANEL.Image.Humidity = 30\%rh)=4$ *(say,* $t_0=4$ *).* In this case, the only time point at which it occurs is $t_0=4$.

The temporal predicate can also be embedded into a path expression and we may use an enhanced path expression to refer to the value component of a temporal object along a branch of the class-aggregation hierarchy whose formal definition is given below:

Definition *Enhanced path expression*

Given an aggregation hierarchy H, an enhanced path P is defined as a path expression with the addition of an explicit time-reference:

$$C_1.A_1.A_2....A_n^{TM} \ (n\geq 1)$$

where ^{TM} denotes the path involving the time-reference, such as $C_1.A_1....A_n\ [t\in T_1]$, $C_1.A_1....A_n\ [t_0<t<t_1]$, etc. Obviously, the enhanced path expresses the nested predicate with an explicit time-reference. In other words, the enhanced path can express a predicate that refers to both the aggregation-hierarchy and the time-dimension. Therefore a path expression in a general OODB is a special case of the enhanced path expression and the path comparisons that are generally used in OODBs can also be applied to the enhanced path expression. Taking the above example, an enhanced path expression, which refers to an image's humidity level=30 % rh is *WOODPANEL. Image.Humidity*(t_0=4) = 30 % *rh.* Here the predicate specifies both *Humidity* value and time point value. More generally, we use $o|_{T1}$ to denote the restriction of o on the temporal set T_1. The examples can be given as:

$WOODPANEL.Image.Humidity|_{t0=4}$ = 30%*rh*, and

$WOODPANEL.Image.Humidity\ |_{t=4,5,6}$ = {30 % *rh,* 40% *rh,* 50% *rh}.*

A method may be used for any part of a predicate, that is, as the attribute-name, the operator, or the value. We could think *of* $\varpi(O)$ and $o|_{T1}$ as methods as well.

If P_1 and P_2 are predicates, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $\neg P_1$. These constitute complex predicates.

5.3 Identity and Equality

Identity is a property of an object that distinguishes the object from all others. It is important to distinguish between the following different types of equality.

1) Identity equality of objects: two objects *o* and *o'* are identity equal if they are the same object (i.e., they have the same OID), denoted as "==". That is, *o==o'* if *OID(o)=OID(o')*.

2) Value equality of objects: two temporal objects are equal if the values and the temporal sets of all their attributes are recursively equal, denoted by "=". That is, two temporal objects *o* and *o'* are value equal if $o.T(A_i)=o'.T(A_i)$ and $o.\upsilon(A_i)=o'.\upsilon(A_i)$ (or $o.\upsilon(A_i)(t) = o'.\upsilon(A_i)(t)$ at every *t*). The term value equality is analogous to the snapshot equivalent/weekly equivalent in temporal RDBs that states that two tuples are snapshot equivalent or weekly equivalent if the snapshots of the tuples at all times are identical.

Two identical objects are also equal whereas the reverse is not true. For example, the humidity attribute of panel#1 and panel#2 may take the same value set and temporal set, therefore they are value identical. Of course they are not identical objects as they refer to different objects and have different OID.

3) Shallow-equality: two objects are shallow-equal if their attributes share the same value and the same references, and their corresponding temporal sets are equal although they are not identical, denoted as =I.

Identity is important for a number of reasons. Duplication in set membership is based on object identity, i.e., a set will not contain two objects with the same identifier. In addition,

there are many cases in the algebra in which implicit comparisons are made using identity equality. There are also some cases when the comparison is made by value equality. Finally, shallow equality is required for the join operation. The use of identity will be illustrated in the following sections.

5.4 Closure

Before going to the definition of algebraic operations, let us recall the closure property in relational algebra.

The relational closure property states that the output from each relational operation is another relation [Date, 1996]. Because the output of any relation is the same kind of objects as the input (they are all relations), so the output from one operation can become input to another. It is possible to write nested expressions, i.e., expressions in which the operands are themselves represented by expressions, instead of just by relation names. Although the temporal object data model is more complex than the relational data model, it is essentially a relation but with three additional associations: aggregation, inheritance and time-reference. If we could reserve the closure property in defining basic algebra, then it would be easy to represent a query referring to those associations. As we shall see, the algebra presented in this thesis retains the closure property.

5.5 Query Algebra

5.5.1 Temporal Unary Set Operations

When the value of temporal object changes with time, even if the record of time-varying values is physically fragmented, it represents the same object. Records of time-varying objects may be amended due to availability of late measurement or better estimation for null or unreliable data. Two special temporal unary set operations are of interest here: **Time-insert** and **Time-delete**.

Given two fragmented subsets of a temporal object set O: $O(T_1)$ and $O(T_2)$, we define

$O(T_1)$ ***Time-insert*** $O(T_2)= \{o \mid o \in O\}$ *where* $L(O)=T_1 \cup T_2$

$O(T_1)$ ***Time_delete*** $O(T_2)$ $\{o \mid o \in O\}$ *where* $L(O)=T_1-T_2$

where L represents the life-span of O after *Time-insert* or *Time-delete*. T_1 and T_2 are temporal sets (for example, intervals) where two fragmented sub sets exist. An illustration of these operations is given in **Figure 5.2**.

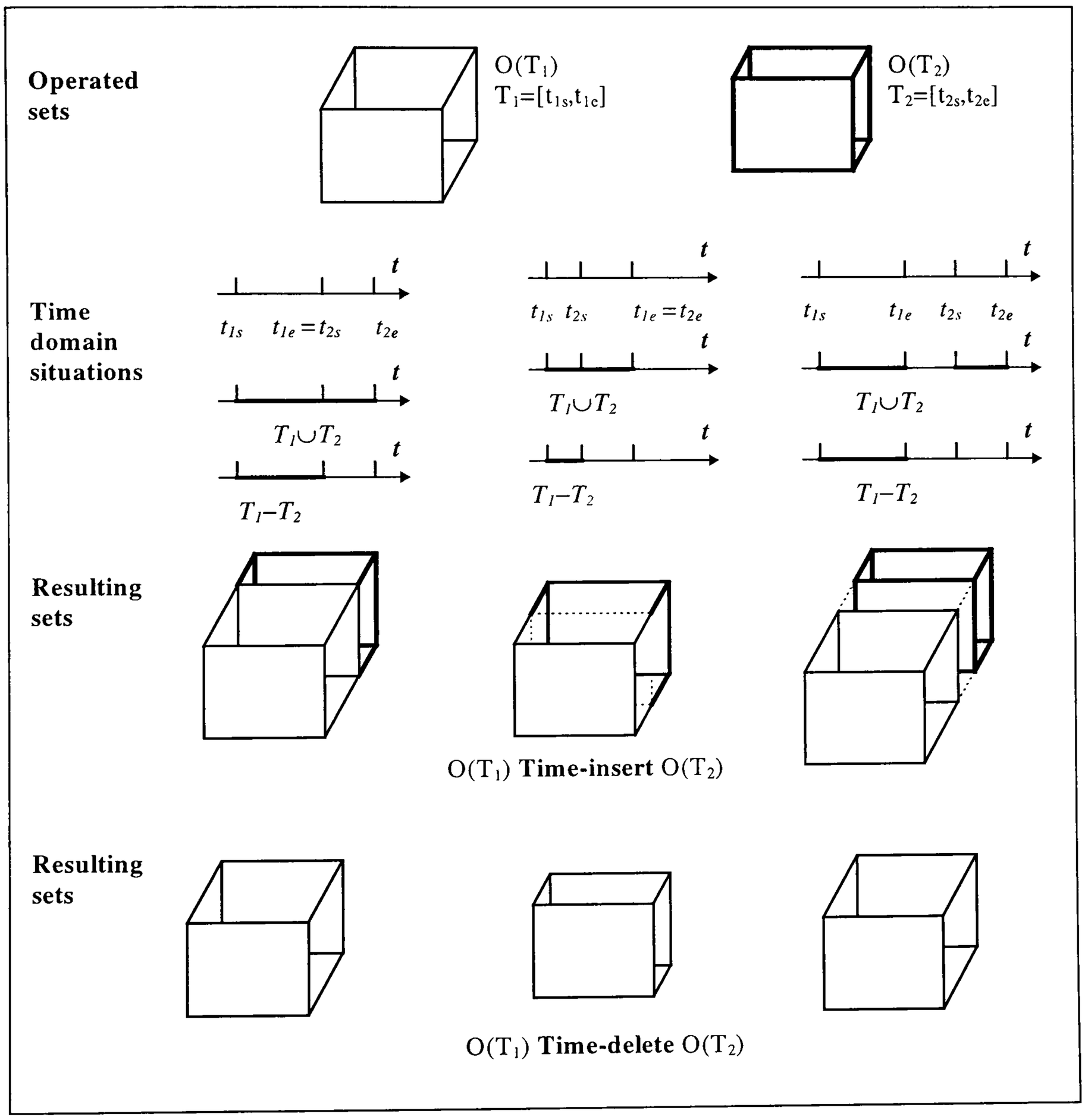


Figure 5.2 Illustration of temporal unary set operations. The first row represents two temporal sets. Below this row there are three cases of $T_1 \cup T_2$ and T_1-T_2, resulting in three cases of *Time-insert* and *Time-delete*.

For ***Time-insert,*** when the temporal sets of T_1 and T_2 are overlapped over the temporal set T_3, i.e., $T_3 \subseteq T_1$ and $T_3 \subseteq T_2$, the value set at T_3 is decided by the following rules:

Suppose:

$ov(t_i) \in O(T_3)$,

$ov_1(t_i) \in O(T_1)$,

$ov_2(t_i) \in O(T_2)$,

and $t_i \in T_3$,

then:

$ov(t_i) = (ov_1(t_i) + ov_2(t_i))/2$, *if no null value exists;.*

$ov(t_i) = ov_1(t_i)$, *if* $ov_2(t_i)$ *is null;*

$ov(t_i) = ov_2(t_i)$, *if* $ov_1(t_i)$ *is null.*

5.5.2 Binary Set Operations

Traditional binary set operations are union, difference and intersection. In mathematics, a set operation, for example, union, is the set of all elements belonging to either or both of the original sets. A relation is a set, very loosely speaking, a set of tuples. It is therefore possible to construct the union of two relations. The possible result could be a set consisting of all tuples appearing in either or both of original relations. This result, although, is a set, it is not a relation as relations can not contain a mixture of different kinds of tuples, they must be tuple-homogeneous. However, we do want the result to be a relation as we want to reserve the closure property. Therefore, the union in our algebra does not conform completely to the generally accepted notation of mathematical union, rather, it is a special form of union, in which the two input relations should be what we might loosely call "the same shape" or "same type", i.e., both must have the same attributes and methods. If the two relations are the same shape in this sense, then we can perform a union, and the result will also be a relation of the same shape. In other words, the closure property will be preserved. The term **type-compatibility** is often used to refer to "same shape" concept [Date, 1996]. Here we define this concept in terms of our temporal object oriented model.

Definition *Type-compatibility*

Two relations are type-compatible if

1) they have the same set of attribute names and method names;
2) the corresponding attributes (i.e., attributes with the same name in the two relations) are defined on the same domain.

Now we are ready to define set operators as below and illustration of set operations is given in **Figure 5.3**.

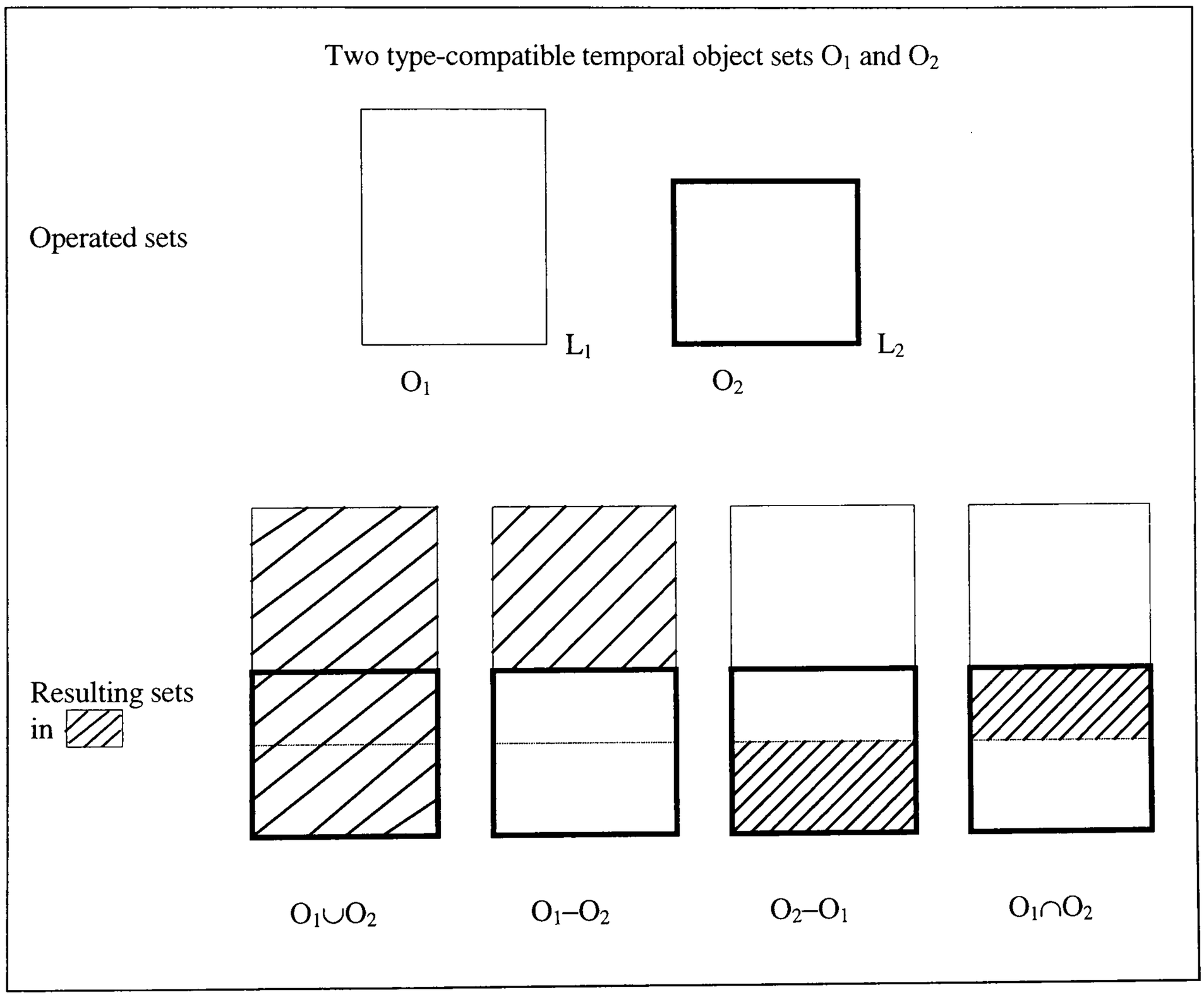


Figure 5.3 Illustration of set operations

If O_1, O_2 are type-compatible temporal object sets, then the sets operators Union, Intersection and Difference are identical to Codd's corresponding relational operators [Codd, 1971; 1972]:

Union $O_3=O_1 \cup O_2=\{o \mid o \in O_1 \vee o \in O_2\}$ *where* $L(O_3)=L(O_1) \cup L(O_2)$

Difference $O_3=O_1-O_2=\{o \mid o \in O_1 \wedge \neg o \in O_2\}$ *where* $L(O_3)=L(O_1)-L(O_2)$

Intersection $O_3=O_1 \cap O_2=\{o \mid o \in O_1 \wedge o \in O_2\}$ *where* $L(O_3)=L(O_1) \cap L(O_2)$

As in relational algebra, the duplication in the resulting set is eliminated following the temporal unary set operation rules.

5.5.3 Special Operations

Select $\sigma_P O$ selects the elements *"o"* of set O such as the predicate $P(o,t)$ holds.

$$\sigma_P O = \{o \mid o \in O \wedge P(o,t)\}$$

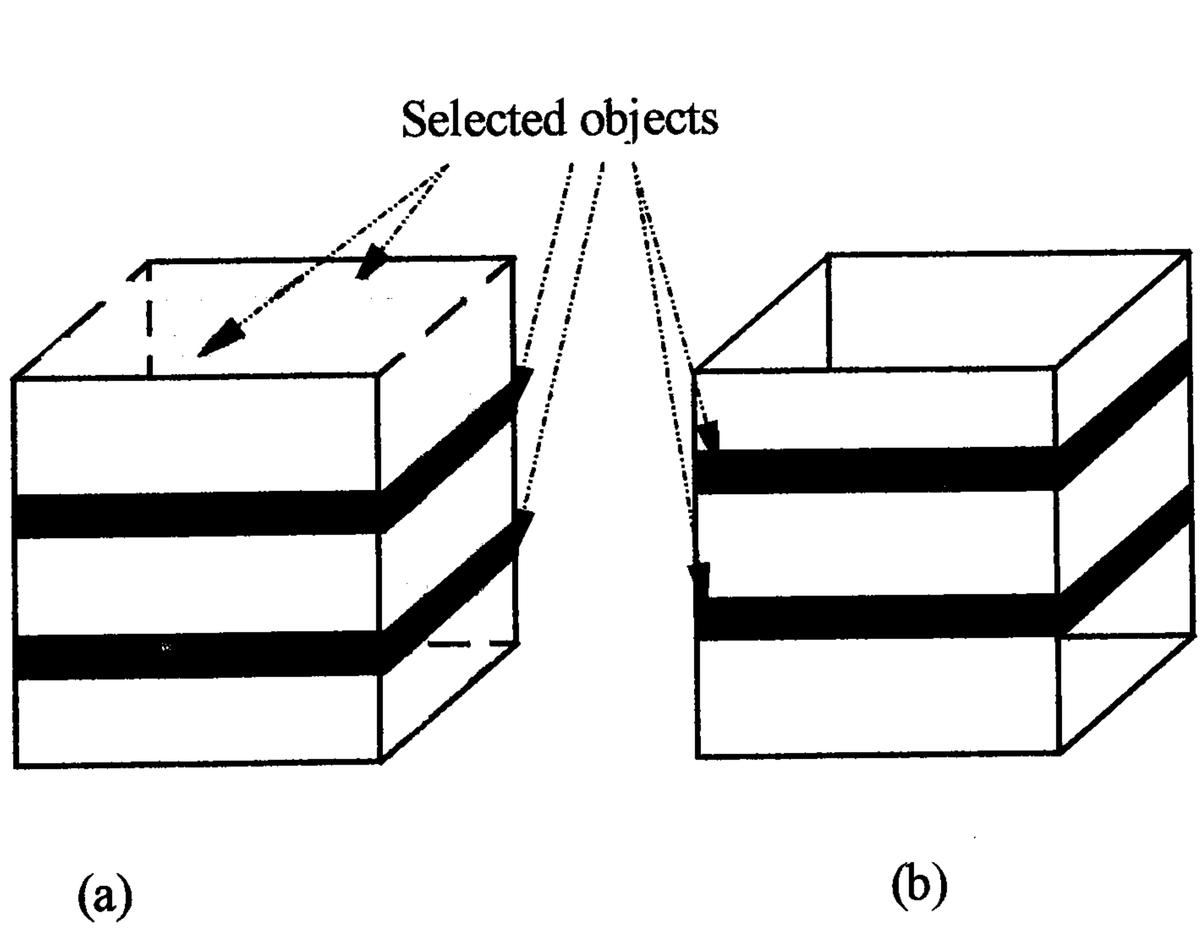


Figure 5.4 Illustration of operator *select*

Select is a hybrid operation, reducing a class (relation) in both the value and the temporal dimension as shown in **Figure 5.4**. If the predicate does not refer to time, it then merely

reduces the class along the value dimension as shown in **Figure 5.4** (b). Note that although the predicate *P(o,t)* may involve the association of aggregation hierarchy, the query target class is only at *O*. For example, for σ_P*WOODPANEL*, the predicate *P* may involve aggregation class *RANALTEST* or *SUPPORT*, but the target of query is *WOODPANEL*, i.e., the query output only gives the objects of this class and the values of attribute *Reinforcement* or *PanelTest* are OIDs instead of objects of relation *PANELTEST* or *SUPPORT*.

Map g: $O_1 \rightarrow O_2$. For the type of objects in O_1 (i.e., $o \in O_1$), *g* returns an object of type of O_2 (i.e., $g(o) \in O_2$).

$$g{:}O_1 \rightarrow O_2 = \{g(o) | o \in O_1\}$$

Map provides a capability of mapping between different types. For example, the column *PanelTes*t of the relation *WOODPANEL* only gives the value of OID of objects in the relation *PANELTEST*, if we want to return the objects of *PANALTEST*, *Map g:PanelTest* → *PANELTEST* will do the work.

Project $\pi_{<A_1,\ldots,A_i>}O$ extends *Map* by allowing the application of many functions to an object, thus supporting the creation and maintenance of selected relationships between objects.

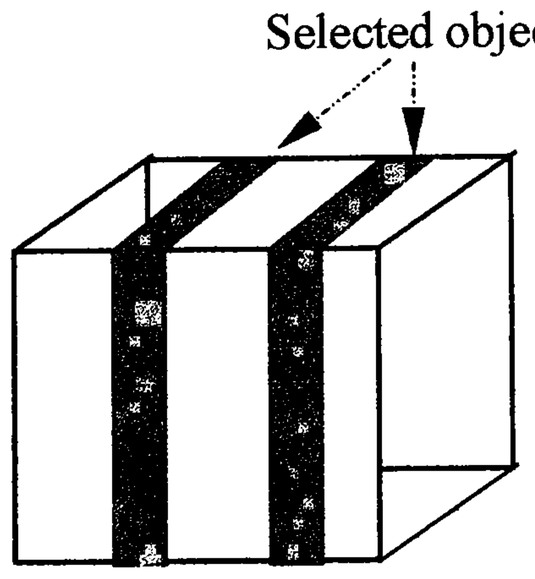


Figure 5.5 Illustration of operator ***project***

$$\pi_{<A_1,\ldots,A_i>} O = \{< A_1 : g_1(o),\ldots,A_i : g_i(o) > | o \in O\}$$

where O is of type *set [T]*, the A_i 's are unique attribute names, and each g_i takes a single input of type T and returns an object of type T_i. $g_1 \ldots g_i$ are similar to g in *map* . If $g_i = 1$, it returns OID of the domain object of A_i unless A_i is atomic. We retain $g_i = 1$ (unless it is specified not) so that we keep our project operator on a set of objects (relation) like the relational project. Therefore the project operator, when applied to class (relation) O, removes from O all but a specified set of attributes. As such it reduces a relation along the attribute dimension. For example, *Project* π<*PanelTest*>*WOODPANEL* will return OIDs of objects of the relation *PANELTEST*. *Project* π<*Humidity*>*PANELTEST* will return a sequence of humidity values with timestamps.

Time-slice ξ_{L1}***(O)*** defines the relation (set of objects) containing those objects derived by restricting each object in the operand relation to those times specified by L_1.

$$\xi_{L1}(O) = \{o | \forall t \in (L_1 \cap L(o))[o(t) \in O]\}$$

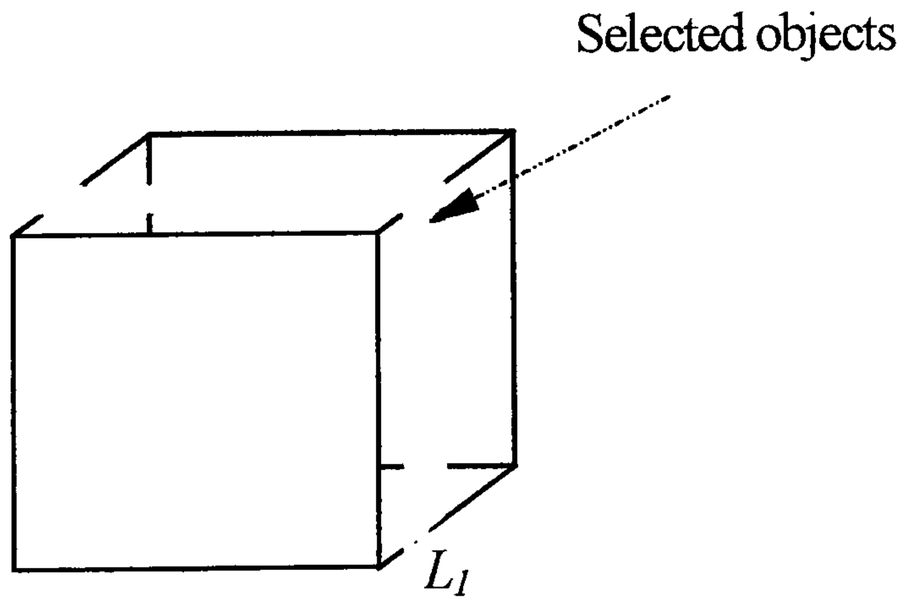


Figure 5.6 Illustration of operator *time-slice*

Obviously the lifespan of $\xi_{L1}(O)$ is $L_1 \cap L(O)$. So the *time-slice* reduces the relation solely along the temporal dimension. If L_1 equals to a time point t_1, i.e., $T_1 = t_1$, then $\xi_{L1}(O)$

represents the event $o(t_1)$ happened at t_1. For example, $\xi_{L1=[20,21]}(Humidity)$ will return a set of attribute values of humidity, for each object there are two values: one at time 20, another at time 21.

Offset $\gamma(O, l)$ "shifts" a snapshot relation at t_1 by the number of positions specified by the offset.

$$\gamma(O(t_1), l) = O(t_1 + l)$$

For example, *offset* $\gamma(\xi_{L1=[20,21]}(Humidity), 20)$ will return a set of attribute values of humidity, for each object there are two values: one at time 40 (=20+20), another at time 41(=21+20).

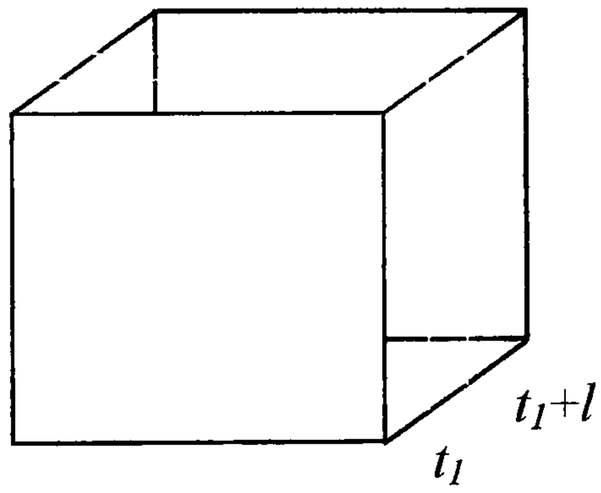


Figure 5.7 Illustration of operator *offset*

When $\varpi(O)$ defines an operator that maps a relation (set of objects) O to its temporal set:

$$\varpi(O) = l(O)$$

The result of *when* is a time value; it can serve as a parameter or a predicate to those operators, like time-slice, etc. An example has been given in Section 5.2, and **Figure 5.8** illustrates the operation effect.

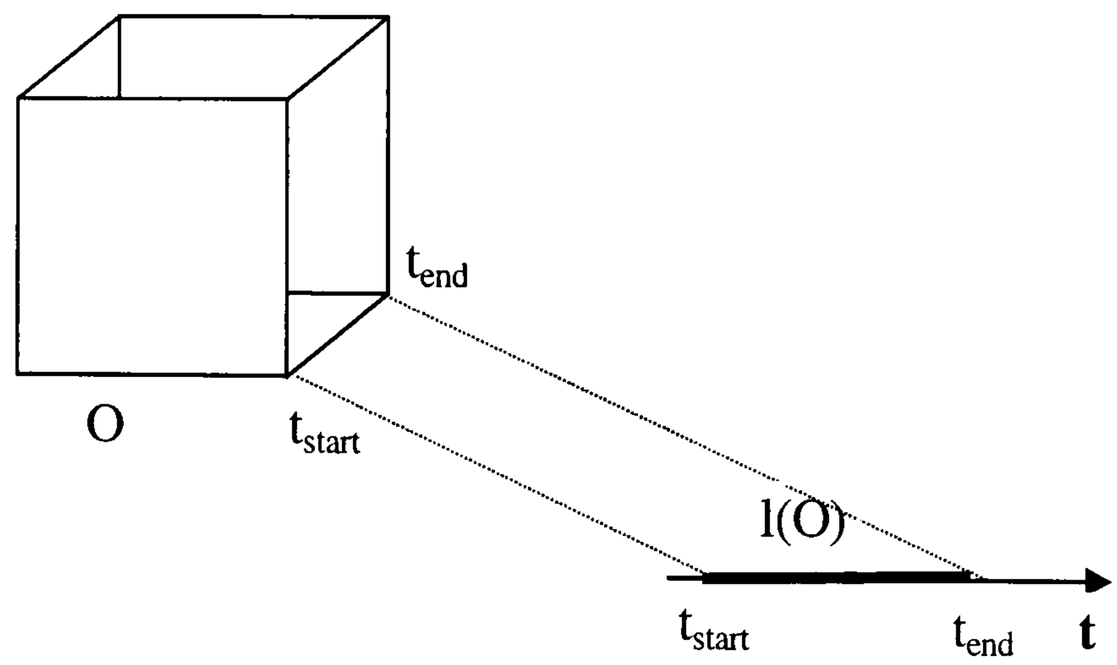


Figure 5.8 Illustration of operator *when*

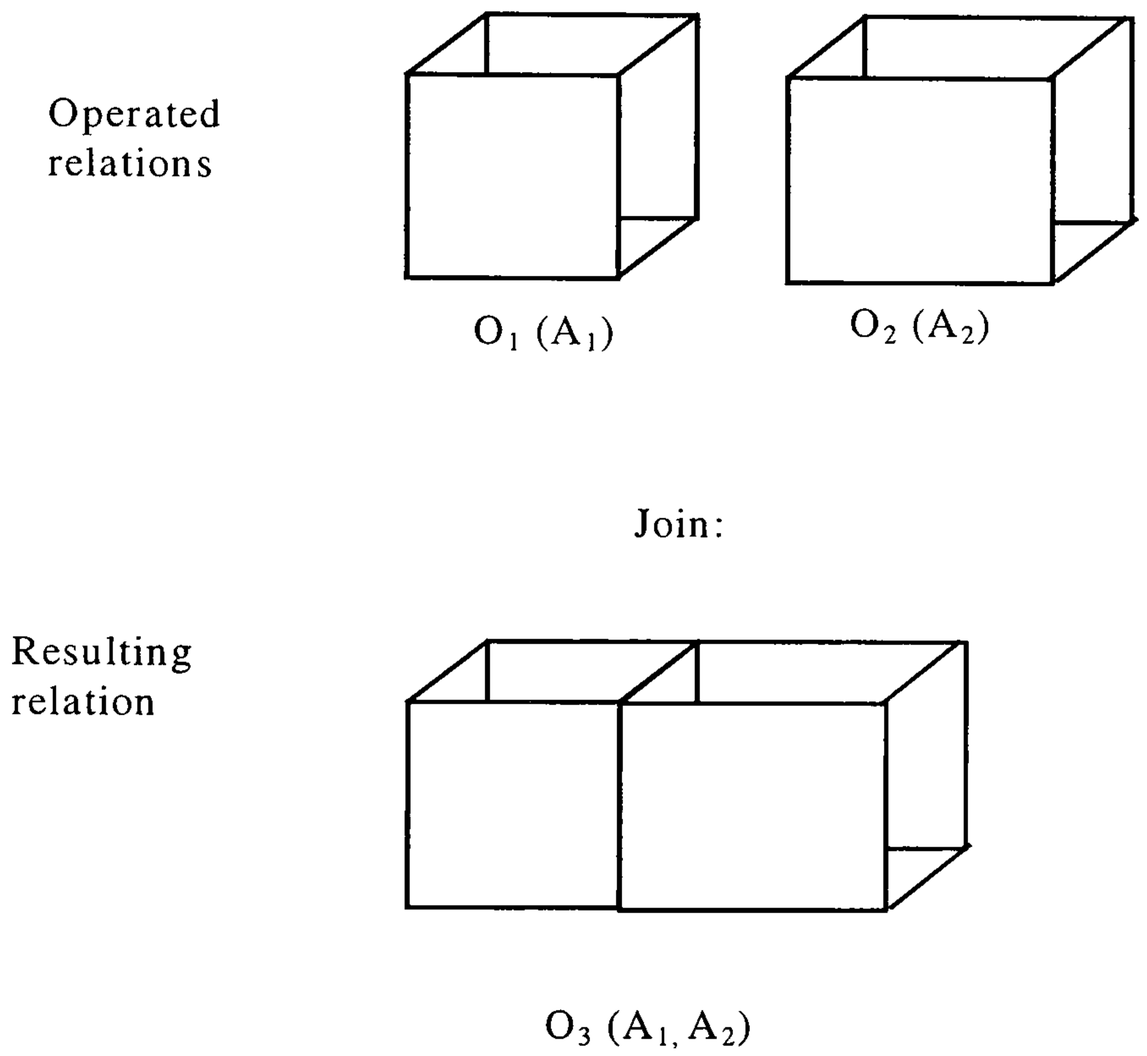


Figure 5.9 Illustration of operator *join*

Join $O_1 \triangleright\!\triangleleft_{P<A\ o1,\ Ao2>} O_2$ is an explicit join operator used to create relationships between objects from two collections in the databases. Unlike relational joins, in which the domains of the join attributes must be identical, we require the join attribute to only be compatible [Kim, 1989]. Two attributes A_i and A_j are compatible, if the value domain and the temporal domain of A_i are identical to those of A_j (or a superclass or subclass of the domain of A_j).

Shallow-equality could express this compatibility. Although join attributes are compatible, they have different OIDs. So the join we defined is essentially a θ-*join* as in relational algebra.

$$O_1 \rhd\lhd_{P<A\,o1,\,Ao2>} O_2 = \{<A_{o1}: o_1, A_{o2}: o_2> \mid o_1 \in O_1 \wedge o_2 \in O_2 \wedge P(o_1, o_2)\}$$

Unnest^T***(***μ_K***O).*** Suppose a relation (class) O_1 has the scheme $< A_1, ..., A_n > = <o_1(1), ..., o_1(n)>$ and the schema of $o_1(k)$ (i.e., $<m,A_k>$) be $<A_k.1,...,A_k.m>, 1 \le k \le n$, then *Unnest*^{T} is defined as

$$\begin{aligned} &O_2 = \mu_K O_1 \\ &= \{o_2 \mid o_2(i)=o_1(i) \text{ for } 1 \le i \le k-1 \wedge o_2(i)=o_1(i+1) \text{ for } k \le i \le n-1 \\ &\wedge o_2(i)=A_k(i-n+1) \text{ for } n \le i \le n+m-1\} \end{aligned}$$

Nest^T***(***υ_y***O).*** Let the relation (class) O_1 have the scheme $< A_1, ..., A_n > = <o_1(1), ..., o_1(n)>$, $y=\{i_1, i_2, ..., i_k\}$ is a subset *of* $\{1, 2, ..., n\}$, and $x=\{1, 2, ..., n-y\}$. *Nest*^{T} has the scheme of $<B_1, ..., B_{n-k+1}> = <o_2(1), ..., o_2(n-k+1)>$, where $o_2(j)=o_l(r)$ for $1 \le j \le n-k$, $r \in x$, and $o_2(n-k+1)$ has the scheme relation: $<B_{n-k+1}.1, ..., B_{n-k+1}.k>$. Similar to the unnest operator, the nested component is placed at the last column of $\upsilon_y O$. So *Nest*^{T} is defined as

$$\begin{aligned} &O_2=\upsilon_y O_1=\{o_2 \mid o_2(j)=o_1(r) \text{ for } 1 \le j \le n-k,\ r \in x \\ &\wedge o_2(n-k+1)=\{z \mid \exists o\ (o \in O_1 \wedge o(r)=o_1(r) \text{ for } r \in x \wedge z(j)=o(ij) \text{ for } 1 \le j \le k)\}\ \} \end{aligned}$$

The effects of *unnest*^{T} *and nest*^{T} are illustrated in **Figure 5.10**. They provide different ways (nest or unnest) to represent a temporal relation. Taking an extreme example, the *Humidity* value is a temporal object, if it is represented in a nest format, it is one element object; if it takes unnested format, it is *n (n=l (Humidity))* elements object. *Unnest*^{T} *and nest*^{T} are not really necessary in temporal object-oriented database systems, as OODB can always represent a 'blob' object. The reservation of these two operators is just for the completeness of algebraic definitions as most temporal relational algebra retains these (e.g., [Clifford *et al.*, 1993]).

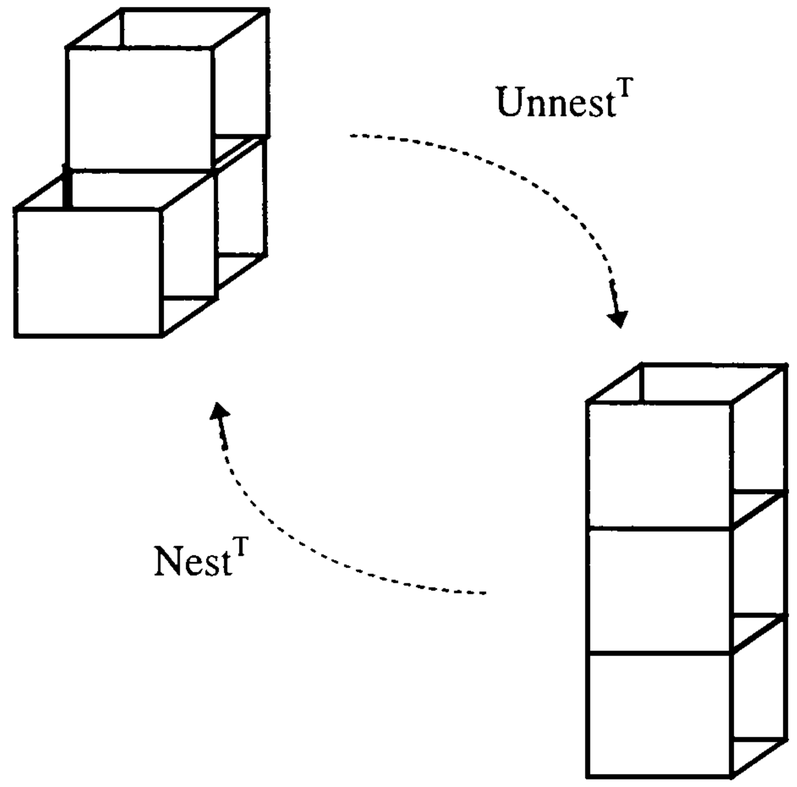


Figure 5.10 Illustration of operators *unnest*^{T} *and nest*^{T}

Besides the above database operators, we can easily define some aggregate operators. Suppose *Agg-func* is one of functions *Avg, Min, Max,* and *Sum*, then *Agg-func*$_{T_1}(O)$ returns the function value over the specified period T_1. Null records in tuples are ignored if there is at least one non-null record otherwise the output is a null record.

5.6 Query Examples

In this section the applicability of our algebra to more complex data queries is illustrated through following query examples.

Query 1 "Find all the wood panels whose type is 'pine' and was reinforced in form of 'lattice' by 'oak, glued' ". This query did not involve any temporal aspect. We can treat it like a constant object query while its lifespan implies the same as the lifespan of temporal objects. We express this query in the following algebra:

$$O_{WI} = \sigma_{P1}\, WOODPANEL$$
$$= \{o \mid o \in WOODPANEL \wedge WOODPANEL.WoodType = 'pine'$$
$$\wedge\, WOODPANEL.Reinforcement.Form = 'lattice'$$

$\wedge$ *WOODPANEL.Reinforcement.Material='oak, glued'}*

For the attributes of *Reinforcement* and *PanelTest*, this query only gives the OIDs instead of all the *SUPPORT* objects and *PANELTEST* objects (with sequential images).

Query 2 "Decide when the humidity level of above selected wood panel is 30% rh". The algebra is as

$$O_t = \varpi(o \in O_{W1} \wedge o.PanelTest.<Humidity.\upsilon> = 30\ \%\ rh)$$

Query 3 "Select the wood panel's PANELTEST whose wood type is 'pine' and reinforced in form of 'lattice' by 'oak, glued', while its humidity level is *30* % rh and grabbed by Camera 1". We have the following algebra:

$$O_{W2} = \pi_{<WOODPANEL.PanelTest>} (\sigma_{P1}\ WOODPANEL)$$

$$= \pi_{<WOODPANEL.PanelTest>}\ O_{W1}$$

$$O_{W3} = Map: O_{W2} \rightarrow \{o \mid o \in TEST1\}$$

$$O_{W4} = \sigma_{P2} (O_{W3}) = \{o \mid o \in TEST1 \wedge o.<Humidity.\upsilon> = 30\ \%\ rh\}$$

Or:

$$O_{W4} = \sigma_{P2} (O_{W3}) = \{o \mid o \in TEST1 \wedge t \in O_t\}$$

Query 4 "Find the humidity level values of above selected *TEST1* which appeared during time O_t". This query involves the temporal reasoning. We apply the following algebra operators to support this reasoning.

$$O_{W5} = \pi_{<TEST1.Humidity>} (O_{W3})$$

$$O_{W6} = \sigma_{P3} (O_{W5}) = \{o \mid o \in Humidity \wedge t \in O_t\}$$

Or:

$$O_{W6} = \pi_{<TEST1.Humidity>} (O_{W4})$$

Query 5 "Get the average humidity level of above selected panel". The corresponding algebraic expression is :

$$Agg\text{-}func_{T1}\, O_{W6} = Avg_{T1=Qt}\, O_{W6}$$

5.7 Properties of Algebra

The algebra defined possesses the following properties:

(1) Closure

The closure property states that *output from one operation can become input to another* [Date, 1995]. Our algebra imposes operators on relations (sets of objects) (except *when* $\varpi(o)$, we already treat it as a method). The output is also a relation. In this sense our algebra is closed.

Compared with other object algebra definitions, such as Shaw and Zdonik [1990], Straube and Ozsu [1990], Cluet and Delobel [1994], and Alhajj and Arkun [1993], the *project* operator integrates *map* operator, the output is a class hierarchy rooted at target class and is therefore not a relation (class) any more. The retention of the closure property in our algebra is through the reservation of $g_i = 1$ in *project* operator so that the output of *project* is also a relation. The closure property is important when a temporal object query processor is to exploit the query processing and optimization techniques that are developed and extended from RDBs techniques.

Note that over two dozen proposals have been made for an object algebra [Ozsu and Blakeley, 1995], no algebra so far defined is based on any unified model of RDBs and OODBs, although it has been claimed that an object algebra should extend relational algebra

consistently [Yu and Osborn, 1991; Shaw and Zdonik, 1990]. Furthermore, none of these object algebras consider the temporal dimension. Our temporal object algebra reflects the spirit of object algebra [Shaw and Zdonik, 1990; Straube and Ozsu, 1990; Cluet and Delobel, 1994; Alhajj and Arkun, 1993], but in addition to supporting access through aggregation and inheritance associations, it also accesses objects through the time dimension. These access mechanisms are embodied in the enhanced nested predicates and (nested) temporal predicates.

(2) Reducibility

The algebra defined possesses the property of reducibility. By the reducibility, we mean that, when the time dimension is not taken into account the temporal object algebra will be reduced to the object algebra and when the object-oriented features of aggregation and inheritance are not taken into consideration, the algebra will be reduced to the relational algebra.

The reducibility of algebra provides good foundation to build up a temporal object query optimizer that is extended from object optimizer and relation optimizer, and to extend the existing query processing strategy and techniques to process temporal object queries, which will be discussed in more detail in the later chapters.

(3) Grouped completeness

As mentioned in previous chapters, temporal data models are classified into two categories: *temporally ungrouped* and *temporally grouped.* Models which employ tuple-time-stamping are termed *temporally ungrouped* whereas models that employ complex attribute values bearing the temporal dimension are termed *temporally grouped* [Pissinou *et al.,* 1994]. While the expressive power of *ungrouped completeness* was generally accepted as a desirable property for TSQL, there were considerable concerns on *grouped complete* [Pissinou *et al.,* 1994]. By *grouped completeness,* we mean that the model supports the rather strong notion of the "history of an attribute". For example, one can talk about "Panel #1's humidity history" as a single object, and ask to see it, or define constraints over it, etc. In

temporal RDBs, as stated in [Clifford *et al.*, 1993; Pissinou *et al.,* 1994], there is no algebra that has been shown to be *grouped complete*. In our temporal object data model, every object is associated with an OID. If every OID is maintained in a database (in some data models, primitive entities such as integers, or characters, are represented by values and have no OID associated with; our temporal object is represented by a time sequence, which is not a primitive data), then our algebra will be *grouped complete*.

5.8 Summary

In this chapter, we have defined an algebra, i.e., a collection of operations, for our temporal object data model. The adaptation of the unified model of RDB and OODB by the addition of a time dimension to form the relational-like cubes that allow aggregation and inheritance associations, provides a basis to develop the temporal object algebra that extends a (temporal) relational and object algebra. The temporal object algebra defined retains the closure property of relational algebra. It also possesses the property of reducibility. Furthermore, the *grouped completeness* of the algebra can also be maintained.

The basic algebraic operators are summarised in **Table 5.1.**

The fundamental intent of the algebra is to allow the writing of expression representing user's queries. In general, algebraic expressions serve as a high-level and symbolic representation of user's intent. Because they are high-level and symbolic, they can be manipulated according to a variety of high-level, symbolic transformation rules. The algebra can then serve as a convenient basis for query processing and optimization. This will be discussed in the next chapters.

Table 5.1 Summary of algebraic operators

Operations	Definition	Notes
$O(T_1)$Time-insert $O(T_2)$	$O_3=\{o \mid o\in O\}$ where $L(O_3)=T_1\cup T_2$	$O(T_1)\subseteq O$, $O(T_2)\subseteq O$
$O(T_1)$Time-delete $O(T_2)$	$O_3=\{o \mid o\in O\}$ where $L(O_3)=T_1-T_2$	$O(T_1)\subseteq O$, $O(T_2)\subseteq O$
Difference O_1-O_2	$O_3=O_1-O_2=\{o \mid o\in O_1 \wedge \neg o\in O_2\}$ where $L(O_3)=L(O_1)-L(O_2)$	O_i is a collection. $L(O_i)$ is the life-span of O_i.
Union $O_1\cup O_2$	$O_3=O_1\cup O_2=\{o \mid o\in O_1 \vee o\in O_2\}$ where $L(O_3)=L(O_1)\cup L(O_2)$	
Intersection $O_1\cap O_2$	$O_3=O_1\cap O_2=\{o \mid o\in O_1 \wedge o\in O_2\}$ where $L(O_3)=L(O_1)\cap L(O_2)$	
Select σ_P O	$\sigma_P O=\{o \mid o\in O \wedge P(o,t)\}$	σ_P O selects the elements "o" of set O such as the predicate P(o, t) holds.
Map $g:O_1\rightarrow O_2$	$g:O_1\rightarrow O_2=\{g(o) \mid o\in O_1\}$	For the type of objects in O_1 (i.e., $o\in O_1$), g returns an object of type of O_2 (i.e., $g(o)\in O_2$).
Project $\pi_{<A1,...,Ai>}O$	$\pi_{<A1,...,Ai>}$ O $=\{<A_1:g_1(o),..., A_i:g_i(o)> \mid o\in O\}$	If $g_i=1$ it returns the OID of the domain object of A_i unless A_i is atomic. We retain $g_i=1$ so that the *project* on a set of objects (relation) likes the relational *project*.
Join $O1 \bowtie p < A_{o1}, A_{o2} > O2$	$O1 \rhd\lhd p < A_{o1}, A_{o2} > O2 = \{< A_{o1}:o1, A_{o2}:o2 > \mid o1 \in O1 \wedge o2 \in O2 \wedge P(o1,o2) \}$	Essentially a θ-join as in relational algebra.
Time-slice $\S_{T1}(O)$	$\S_{T1}(O)=\{o \mid \forall t\in (T_1\cap L(o))\ [o(t)\in O]\}$	The life-span of $\S_{T1}(O)$ is $T_1\cap L(o)$. Time-slice purely reduces the relation along the temporal dimension. If T_1 equals to a time point t_1, i.e., $T_1=t_1$, then $\S_{T1}(O)$ represents an event $o(t_1)$ happened at t_1.
Offset γ (O, l)	$\gamma(O(t_1), l)= O(t_1+l)$	"Shifts" a snapshot relation at t_1 by the number of positions specified by the offset l.
When $\varpi(O)$	$\varpi(O)=L(O)$	Maps a set of objects O to its temporal set.
Aggregation Agg-func $T1(O)$	Agg-func $T1(O)$, where func={Avg, Min, Max, Sum, etc.}	Returns the function value over T_1.

Chapter 6

A Uniform Framework for Processing Temporal Object Queries*

This chapter presents a uniform framework for processing temporal queries. Within the uniform framework a set of transformation rules are specified for the algebraic optimization. Based on these transformation rules, a decomposition strategy is proposed for evaluating the queries that involve a path with time-reference.

6.1 Introduction

Our temporal data model, as shown in **Figure 6.1**, extends the unified model of RDB and OODB by adding a time-dimension, whilst the unified model itself refines the relational model by incorporating three important object-oriented features: nested relation, inheritance and encapsulation. The algebra of the model possesses the property of reducibility. That is, when the time-dimension is not taken into account, the algebra is reduced to the object algebra and when object-oriented features are not taken into consideration, it is reduced to the relational algebra. Further, the algebra is closed, so that the output of one operation can be the input of another. These features provide us with a basis for using existing relational and object-oriented query processing techniques to process temporal object queries. In this chapter we explore an extensible approach to processing temporal queries that exploits the widely adopted existing object query processing techniques and the well established relational query processing techniques. In particular, we will identify a set of query transformation rules for algebraic optimization within this uniform query processing framework. With a view to addressing the central issue of path optimization in object query processing when time is present, a decomposition

* The work in this chapter has been presented in the paper 3 and 4 listed in *Author's Publications.*

strategy is proposed to process the temporal query involves a path with time-reference, based on the identified transformation rules. The detail of processing the decomposed query components will be discussed in the next chapter.

The rest of chapter is organised as follows. The layered structure of query optimizer is presented in Section 6.2. Query transformation is discussed in Section 6.3. A decomposition strategy for query evaluation is described in Section 6.4 and Section 6.5 provides a summary of the chapter.

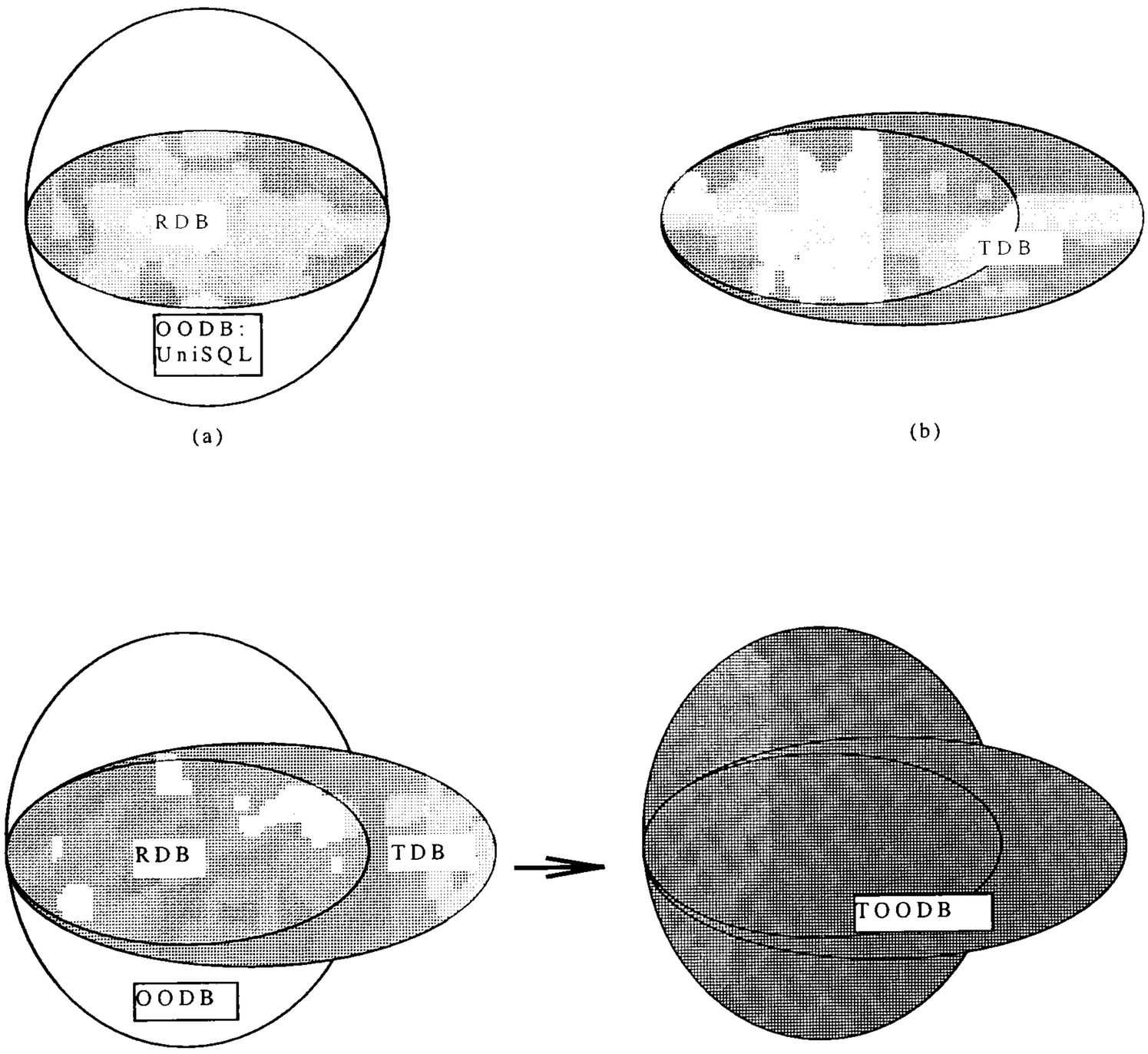


Figure 6.1 Data model extensibility:

(a) UniSQL is extended from RDB model;
(b) Most TDBs are in the context of RDB;
(c) Adding a time-dimension into UniSQL forms a TOODB model

6.2 Optimizer Layering

Our optimizer can be seen to be a layered structure as shown in **Figure 6.2**, where the temporal optimizer is built on the top of an object optimizer that in turn, is founded on a relational optimizer. When the time dimension does not exist, the object optimizer plays the key role. As the object optimizer is extended from the relational optimizer, when the object-oriented features are removed from the data model, the relational optimizer comes into play. The separation of query processor functionality in this way makes it easy to exploit existing query processing techniques at the appropriate layer during both algebraic and non-algebraic optimization stages as can be seen from the discussion hereafter. In summary, temporal queries can be processed and optimized within the existing object-oriented query processing framework through a smooth extension of existing query processing techniques.

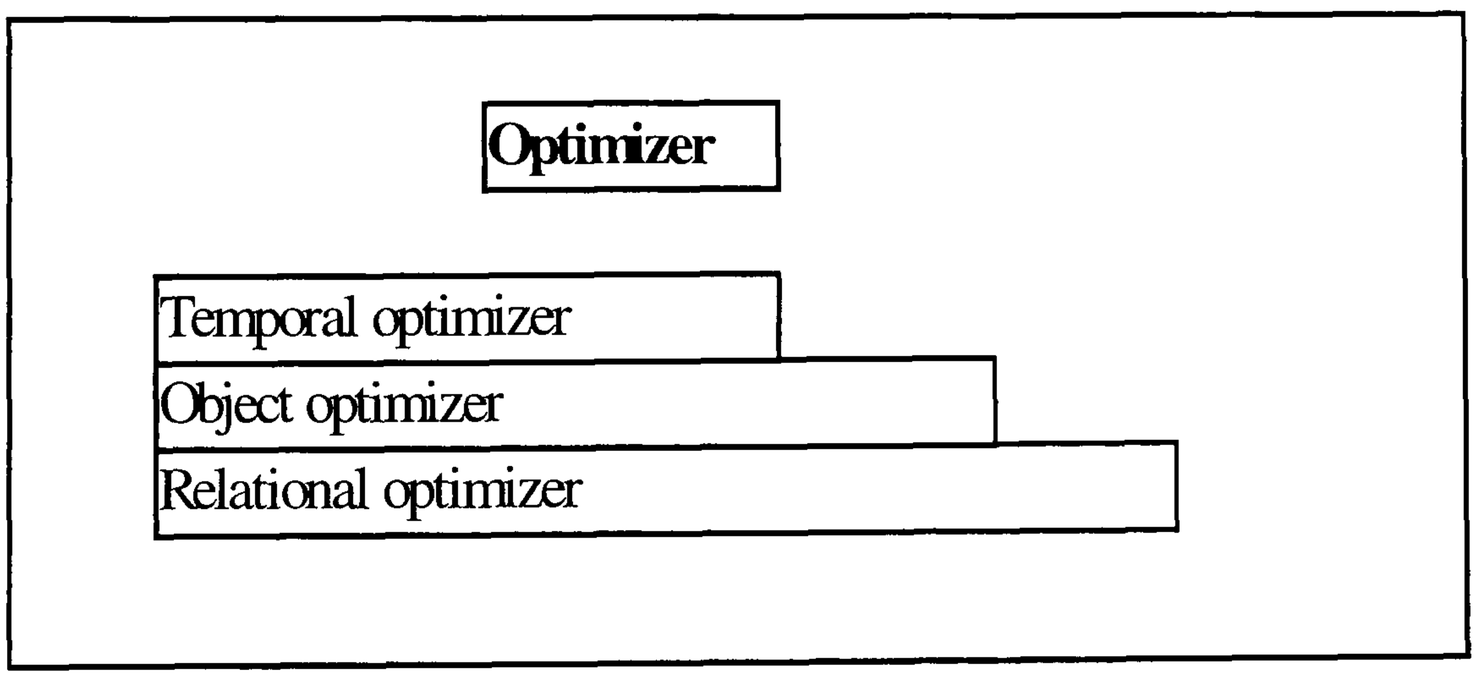


Figure 6.2 Optimizer layering:
separation of query processing functionality

6.3 Query Transformations

Query optimization concerns the problem of selecting an efficient query plan for a query from the set of all its possible query plans. The size of the search space of equivalent query plans for a snapshot query[†] is determined in part by the algebraic equivalence available in the snapshot algebra as each query has a number of equivalent expressions, which make up the search space. These expressions are equivalent in terms of the results they generate but may be quite different in terms of their costs. The query optimizer modifies the query expressions, by means of algebraic transformation rules, in an attempt to obtain one that generates the same result with the lowest possible cost. Therefore the algebraic manipulation for query optimization is the transformation of one query into an equivalent query that might be more efficient to evaluate.

The algebraic manipulation for query optimization in a temporal object-oriented database follows the same principle as above, but the time dimension needs to be taken into consideration where applicable. In principle, as the temporal object is represented as a time sequence that can be thought of as the equivalence of a 'blob' object, the transformation rules in the snapshot object algebraic optimization can be carried over. Also, as our temporal algebra is consistently extended from relational algebra, the relational algebraic transformation rules can also apply. But, the algebraic optimization has to take object-oriented features and the time dimension into account, that results in a set of query transformation rules that are not only relation rules. In this section, we specify the following transformation rules that can be applied during optimization to generate equivalent query expression. Taking into account the object-oriented features, and the time dimension, these rules are characterised as relational rules, temporal rules, inheritance rules and path transformation rules, and are discussed below.

[†] In this thesis, a data model that does not concern the time dimension is referred as a snapshot data model and its corresponding database is referred as a snapshot database. A query to such a snapshot database is called a snapshot query. Similarly the algebra for a snapshot data model is called a snapshot algebra.

6.3.1 Relational Rules

The following rules are called relational rules as they are derived from well-known algebraic optimization techniques in RDBs [Desai, 1990; Ullman, 1989; Jarke and Koch, 1984] and can be applied to the situations where the object-oriented features and time dimension are not taken into consideration. These rules are based on basic algebraic laws of idempotence, commutativity, associativity and distributivity [Desai, 1990], which have been previously presented in Chapter 3. These basic laws can be incorporated in the basic rules discussed below, which can be applied during algebraic optimization to generate equivalent query expressions. The effects of the transformation rules are either to avoid redundant creation and manipulation of intermediate results, or to reduce the size of the intermediate result relations.

In the following, we suppose P_1, ..., P_i ... are predicates, A, A_1, ..., A_i ... are sets of attributes, and C_1, ..., C_i ... are classes /tables.

(1) Combine a cascade of *selects*.

$$\sigma_{P1}(\sigma_{P2}(C_1)) \equiv \sigma_{P2}(\sigma_{P1}(C_1)) \equiv \sigma_{P2 \wedge P1}(C_1) \quad (1)$$

Rule (1) means that if the predicates P_1 and P_2 are only involved in the attributes of C_1, they can be evaluated at the same time.

Example 6.1 Consider the database schema in **Figure 4.5**, for the query "Get the details of WOODPANEL with Panel# =40 where the WoodType is 'pine' ", the algebra can be expressed as:

$$\sigma_{Panel\#=40}\ (\sigma_{WoodType='pine'}(WOODPANEL))$$

which is equivalent to:

$$\sigma_{Panel\#=40 \wedge WoodType='pine'}(WOODPANEL)$$

The latter expression can be evaluated by testing for the predicate

$$Panel\#=40 \wedge WoodType='pine'$$

against each tuple of relation *WOODPANEL*.

(2) Combine a *union* of *selects*.

$$\sigma_{P1}(C_1) \cup \sigma_{P2}(C_1) \equiv \sigma_{P1 \vee P2}(C_1) \quad (2\text{-}1)$$

$$C_1 \cup \sigma_{P1}(C_1) \equiv C_1 \quad (2\text{-}2)$$

Obviously, the right side expressions of equation (2-1) and (2-2) have simplified the left side expressions.

(3) Combine a cascade of *project* into a single *project*.

$$\pi_{Ai}(\pi_{Aj} C_1) \equiv \pi_{Ai} C \text{ where } A_i \subseteq A_j \quad (3)$$

Example 6.2 Consider the query against the database schema in **Figure 4.5**:

$$\pi_{Panel\#}(\pi_{Panel\#, WoodType}(WOODPANEL))$$

This query can be simplified as

$$\pi_{Panel\#}(WOODPANEL)$$

avoiding a redundant *project*.

(4) Commute *select* and *project*.

$$\sigma_P(\pi_A(C)) \equiv \pi_A(\sigma_P(C)) \tag{4-1}$$

and

$$\pi_A(\sigma_P(C)) \equiv \sigma_P(\pi_A(C)) \tag{4-2}$$

This rule provides an equivalent expression. Depending on the applicable situation, either the *project* or the *select* can be performed first. However, if P involved attributes $A_i \not\subset A$, then when commuting *project* with *select* we have to use the following equivalence:

$$\pi_A(\sigma_P(C)) \equiv \pi_A \sigma_P(\pi_{A \cup Ai}(C)) \tag{4-3}$$

Or the *select* has to be performed first.

(5) Use associative and commutative laws for *joins* and *Cartesian products*.

$$\begin{aligned} &R \triangleright\triangleleft S \equiv S \triangleright\triangleleft R \\ &R \triangleright\triangleleft S \triangleright\triangleleft T \equiv R \triangleright\triangleleft (S \triangleright\triangleleft T) \equiv (R \triangleright\triangleleft S) \triangleright\triangleleft T \equiv (T \triangleright\triangleleft S) \triangleright\triangleleft R \equiv\ldots \\ &R * S \equiv S * R \\ &R * S * T \equiv R * (S * T) \equiv (R * S) * T \equiv (R * T) * S =\ldots \end{aligned} \tag{5}$$

The order of the *join* and *product* is very important as it can substantially affect the size of the intermediate relations and, therefore, the total cost of generating the result relation.

(6) Perform *select* before a *join* or *Cartesian product*[‡].

Consider $\sigma_C(R \triangleright\lhd S)$. If the attributes involved in the predicates P are in the scheme of C_1 and not in C_2, that is, $attr(P) \in C_1$ and $attr(P) \notin C_2$, then

$$\sigma_P(C_1 \triangleright\lhd C_2) \equiv \sigma_P(C_1) \triangleright\lhd C_2 \qquad (6\text{-}1)$$

If the attributes involved in the predicate P are in the scheme of C_1 but not in C_2, i.e., $attr(P) \in C_2$ and $attr(P) \notin C_1$, then

$$\sigma_P(C_1 \triangleright\lhd C_2) \equiv C_1 \triangleright\lhd \sigma_P(C_2) \qquad (6\text{-}2)$$

If the attributes involved in the predicate P are in the same scheme of C_1 and C_2, i.e., $attr(P) \in C_1$ and $attr(P) \in C_2$, then

$$\sigma_P(C_1 \triangleright\lhd C_2) \equiv \sigma_{P1}(C_1) \triangleright\lhd \sigma_{P2}(C_2) \qquad (6\text{-}3)$$

If $P=P_1 \wedge P_2$ and the attributes involved in the predicate P_1 are from C_1, i.e., $attr(P_1) \in C_1$, and the attributes involved in the predicate P_2 are from C_2, i.e., $attr(P_2) \in C_2$, then

$$\sigma_P(C_1 \triangleright\lhd C_2) \equiv \sigma_{P1}(C_1) \triangleright\lhd \sigma_{P2}(C_2) \qquad (6\text{-}4)$$

If $P=P_1 \wedge P_2 \wedge P_3$ and the attributes involved in the predicate P_2 are only in C_1, i.e., $attr(P_2) \in C_1 \wedge attr(P_2) \notin C_2$, the attributes involved in the predicate P_3 are only in C_2,

---

[‡] Note that when we talk about relational rules, it is assumed that the object-oriented and temporal features are not taken into account. Otherwise these rules may not apply. For example, if the predicate P involves a path, then it may take longer time to evaluate select than join. In this case, rule (6) may not apply.

i.e., $attr(P_3) \in C_2 \wedge attr(P_3) \notin C_1$, and the attributes involved in the predicate P_1 are in C_1 and C_2, then

$$\sigma_P (C_1 \rhd\lhd C_2) \equiv \sigma_{P1} (\sigma_{P2}(C_1) \rhd\lhd \sigma_{P3} (C_2)) \qquad (6\text{-}4)$$

The above equivalencies also apply when the *Cartesian product* operation is substituted for the *join*.

It is possible to combine *projects* with a binary operation that precedes or follows it. Only the attributes values specified in the *project* need to be retained. The remaining ones can be eliminated as we evaluate the binary operation.

(7) Perform a modified *project* before a *join*.

Note that when a *project* operation is preceded by a *join*, it is possible to push the *project* down before the *join*, but the *project* requires new attributes. This necessitates performing the original *project* after *join*. However, unless the cardinalities of intermediate relations are reduced, which would reduce the cost of the *join* operation and the subsequent size of the joined relation, the usefulness of pushing a *project* before a *join* is questionable.

$$\pi_A (C_1 \rhd\lhd C_2) \equiv \pi_A (\pi_{A1}(C_1) \rhd\lhd \pi_{A2}(C_2)) \qquad (7)$$

where $A_1 = C_1 \cap (A \cup C_2)$ and $A_2 = C_2 \cap (A \cup C_1)$, and C_1, C_2 represent the set of attributes in these relation schemes. When $A \equiv C_1 \cup C_2 - C_1 \cap C_2$, there is no improvement because $A_1 \equiv C_1$ and $A_2 \equiv C_2$.

(8) Commuting *project* with a *Cartesian product*.

Consider the expression $\pi_A (C_1 * C_2)$. This expression can be replaced by the following equivalent one under these conditions: A_1 is the set of attributes in A that is in the scheme of C_1, and A_2 is the set of attributes in A that is in the scheme of C_2.

$$\pi_A (C_1 * C_2) \equiv \pi_{A1}(C_1) * \pi_{A2}(C_2) \qquad (8)$$

(9) Commuting *project* with a *union*.

Consider the expression $\pi_A(C_1 \cup C_2)$. It can be substituted by the equivalent one given below provided the relations C_1 and C_2 are compatible. In other words, they are defined on similar relation schemes. Dissimilarities in the names of attributes could be handled by appropriate renaming.

$$\pi_A(C_1 \cup C_2) \equiv \pi_A (C_1) \cup \pi_A(C_2) \qquad (9)$$

(10) Commute *select* with a *union*.

Again, the relations C_1 and C_2 must be compatible and any difference in names of the attributes could be handled by appropriate renaming.

$$\sigma_P(C_1 \cup C_2) \equiv \sigma_P(C_1) \cup \sigma_P(C_2) \qquad (10)$$

(11) Commute *select* with a *difference*.

As in rules (9) and (10) above, relations C_1 and C_2 must be compatible and renaming would resolve any differences in the names of the attributes.

$$\sigma_P(C_1 - C_2) \equiv \sigma_P(C_1) - \sigma_P(C_2) \qquad (11)$$

We could replace the relations C_1, C_2, etc. in each of the above rules by a relational expression. Note that the *difference* operation is not commutative.

Finally the query processor can use the knowledge of the relation schemes and functional dependencies to find additional equivalent forms for a query expression. The following example illustrates this.

Example 6.3 Given $C_1(A_1, A_2, A_3)$ and $C_2(A_3, A_4, A_5, ...)$, where A_i is a set of attributes, the query $\pi_{A1=a1}(C_1 \triangleright\!\!\triangleleft C_2)$ can be replaced by $(\pi_{A1=a1}\ C_1) \triangleright\!\!\triangleleft C_2$ and the query $\sigma_{A3\ A4}(C_2) \triangleright\!\!\triangleleft \sigma_{A4\ A5}(C_2)$ is equivalent to $\sigma_{A3\ A4\ A5}(C_2)$.

6.3.2 Temporal Transformation Rules

When the time-dimension is taken into consideration, the following transformation rules play roles that are called temporal transformation rules[§].

(12) Perform *time-slice* before *select*.

$$\textit{time-slice}_{T1}\ (\ \sigma_{P1}(C_1)\ ) \equiv \sigma_{P1}\ (\textit{time-slice}_{T1}\ \ C_1\ ) \qquad (12)$$

As is the case for most relational databases, we assume that the data of a relation/class are stored tuple by tuple instead of column by column. For a temporal relation/class, the effect of time on a temporal object is to generate multiple versions of the same tuple fields. For a temporal object in a temporal relation/class, multiple versions of the same tuple fields can be assumed to be stored together in a disk space (physical blocks), unless it is stated that it has been partitioned (that is usually only considered during query evaluation stage). The exact cost of *time-slice* depends on the implementation, but, at least we can assume that only data ranged over T_1 are retrieved for $\textit{time-slice}_{T1}$.

[§] Although the actual physical data structure is not usually considered during algebraic manipulation, we will generally assume that the data of a relation/class are stored tuple by tuple instead of column by

Therefore, doing *time-slice* first will avoid reading out all data during execution stage and therefore saves the corresponding cost. It is especially the case when a temporal object occupies more than one page or block in disk, so only the pages/blocks that contain the requested data are retrieved from the disk (not the whole temporal object). This is one of the features that distinguishes temporal and object-oriented query processing from traditional relational query processing: i.e., the query transformation can not always take place at a logical level; the costs related to physical data storage have to be taken into consideration sometimes. That is why query processing in temporal or object-oriented databases can not completely separate query transformation from query evaluation [Leung and Muntz, 1993; Blakeley *et al.,* 1993; Seshadri *et al.*, 1996; Cluet and Delobel, 1992; Ozkan *et al.,* 1995], and some even merge these two into one, as mentioned in [Ozsu and Blayeley, 1995].

Unless the predicate P is involved in the range outside T_1, the *time-slice* can always be performed before *select*.

Example 6.4 Consider the database schema in **Figure 4.4**, list TESTINFO in July whose maximum top temperature during July is higher than 35^0 C. This query can be expressed as:

$$time\text{-}slice_{T1}\ (\ \sigma_{P1}\ (TESTINFO)\ ) \equiv \sigma_{P1=(TopTemperature>=35)}\ (time\text{-}slice_{T1=July}\ TESTINFO)$$

If the database keeps a one-year record, doing the right side of above equation only needs to retrieve a one-month record for each instance of *TESTINFO* from the disk. The instance of *TESTINFO* is listed, if the predicate is satisfied. Note that the predicate in the example also specifies the range of time that can be incorporated into the range of *time-slice*. There is obviously no need to retrieve a one-year record to check if the predicate is satisfied.

---

column, the former being the case for most relational databases. For a temporal relation/class, the effect of time on a temporal object is to generate multiple versions of the same tuple fields.

(13) Perform *time-slice* before *project*.

$$time\text{-}slice_{TI}\ (\pi_{Ai}(C_1)) \equiv \pi_{Ai}\ (time\text{-}slice_{TI}\ C_1) \qquad (13)$$

Like Rule (12), doing *time-slice* first reduces query cost as a smaller range of each temporal object (time sequence) is accessed.

(14) Perform *offset* before *project*.

$$offset_I\ (\pi_{Ai}(C_1) \equiv \pi_{Ai}\ (offset_I\ C_1) \qquad (14)$$

Depending on the organisation of storage, if executing *offset* first will avoid the need to examine all data or reduce the duplicate operations, it is performed before *project*.

Example 6.5 Consider **Example 6.4.** If the listed TESTINFO in that example is TESTINFO_1, then the query "list TESTINFO_1's TopTemperature in October that is three months later than July" can be expressed as:

$$offset_I\ (\pi_{Ai}(TESTINFO\_1) \equiv \pi_{Ai=TopTemperature}\ (offset_I\ TESTINFO\_1)$$

However, doing *project* first does not answer the query, so when the October's record is retrieved, it has to do *project* again to answer the query. Therefore it is better to perform *offset* before *project*.

(15) Commute *agg-func* and *project*.

$$\pi_{Ai}\ (agg\text{-}func_{TI}\ C_1) \equiv agg\text{-}func_{TI}(\pi_{Ai}(C_1)) \qquad (15)$$

If executing $agg\text{-}func_{TI}$ could greatly reduce the need to examine all data (i.e., T_1 is much smaller than the lifespan of C_1) or the degree of $(\pi_{Ai}(C_1))$ is close to that of C_1, $agg\text{-}func_{TI}$ should be performed before *project*; if T_1 covers almost the range of lifespan

of C_1, performing *project* first will reduce the unnecessary calculation on the results of *agg-func*$_{T1}$.

Rules (12)-(15) can be incorporated into Rules (1)-(11) whenever applicable. It is also a good heuristic to push *select*, *project* and *time-slice* as far down the query graph as possible, especially to perform the time-slice as early as possible.

It has been identified that some transformations are incorrect [Seshadri *et al.*, 1994]:

- A *select* can not be pushed through an *aggregate* (i.e., *agg-func*$_{T1}$) operator or *offset* operator.

Example 6.6 Consider **Example 6.4.** If the listed TESTINFO in that example is TESTINFO_1, the query "select TESTINFO_1 whose average top temperature in July is greater than 30^0" can be expressed as:

$$\sigma_{P1}(TESTINFO\_1) = \sigma_{P1=(agg\text{-}agv\ (T1=July)(\ TopTemperature)=30)}(TESTINFO\_1)$$

For this query, the *select* can not be performed before a*gg-agv* $_{t1=July}$, Instead, *agg-agv* is performed first and the result of this operation then participates in the predicate evaluation for the *select.*

Example 6.7 The query "select TESTINFO_1 whose top temperature on the first of July is the same as that on the last day of July" can be expressed as:

$$\sigma_{P1}(TESTINFO\_1) = \sigma_{P1=(\ TopTemperature(1)=offset\ (31)(TopTemperature(1))}(TESTINFO\_1)$$

For this query, *offset* has to be performed before *select.*

- An *agg-func*$_{T1}$ operator can not be pushed through an *offset* operator and vice versa.

Example 6.8 Consider

$$agg\text{-}avg_{T1=August}\ (offset_{l=one\ month}\ (\ TESTINFO\_1)$$

Obviously, both *offset* and *agg-func* have specified the range of time, and $agg\text{-}avg_{T1}$ returns a value of the function, not a temporal object. Therefore *offset* and *agg-func* cannot be commuted.

6.3.3 Inheritance Rules

One particularly important difference between defining transformation rules for relational systems and for object-oriented systems, is that relational query expressions are defined on flat relations whereas object queries are defined on classes that have inheritance relationships amongst them. The transformation rules that take into account such inheritance relationships are called inheritance rules.

Suppose C_2 is subclass of C_1, i.e., an *is-a* relationship between C_2 and C_1 holds. C_2 is more specific in the sense that it has more attributes than C_1 has. Apart from the attributes inherited from C_1, it has its own attributes. Taking account this relationship, the following rules apply:

$$\pi_{A1}\ (C_1 \cup C_2) \equiv \pi_{A1}\ C_1$$
$$where\ C_2\ is\ subclass\ of\ C_1\ and\ A_1 \in C_1 \qquad (16)$$

$$\pi_{A1}\ (\sigma_P(C_1) \cup \sigma_P\ (C_2)) \equiv \pi_{A1}\ (\sigma_P(C_1))$$
$$where\ C_2\ is\ subclass\ of\ C_1\ and\ A_1 \in C_1\ and\ attr(P) \in C_1 \qquad (17)$$

Rules (16)-(17) can be used to simplify the expression.

Example 6.9 Consider the database schema in **Figure 4.4.** For the query "find the names of cities and capitals whose first letter is 'A' ", we have the following equivalent expressions. Obviously, the expression on the right of the equation eliminates an unnecessary operation.

$$\pi_{Name}(\sigma_{Name='A*'}(CITY) \cup \sigma_{Name='A*'}(CAPITAL)) \equiv \pi_{Name}(\sigma_{P}(CITY))$$

6.3.4 Path Transformation Rules

Object-oriented databases significantly reduce the need for a *join*. Responding to user queries frequently involves the execution of *selects* rather than *joins*. The *select* operation allows its predicate to apply to a contiguous sequence of attributes along a branch of the class-aggregation hierarchy. The path expression is used to represent this sort of predicate. In the definition of our algebra, we have extended the *select* operator by allowing its predicate to apply to a contiguous sequence of attributes along both the aggregation hierarchy and time-dimension, and the enhanced path expression has been defined for this purpose. These have been fully discussed in Chapter 5. As in most object-oriented databases [Demuth *et al.*, 1994], the *select* is the most powerful operator in temporal object-oriented databases.

The class-aggregation hierarchy holds an attribute/domain link between a neighboured pair of classes. The attribute/domain link between this pair of classes, e.g., C_{i-1} and C_i, is effectively the *join* of these classes, in which the attribute A_{i-1} of the class C_i and identifier OID, which is defined by the system and can be considered as an attribute of class C_{i-1}, are join attributes [Bertino and Martino, 1993]. Therefore, an object query with a path expression involving N classes C_1, C_2, ..., C_n, is equivalent to a relational query, which requires joining N relations corresponding to N classes. That is why the *select* operator is often called an ***implicit join***. According to the definition of our algebra, when the predicate of the *select* operator involves a path expression (denoted as $C_1.A_1.A_2....A_n$ ^{TM} *op value),* it is equivalent to a series of joins, i.e.,

$$\sigma_{C1.A1.A2....An^{TM}\ op\ \upsilon}(C1) = \pi_{A(C1)}(C1 \triangleright\triangleleft^{P} \pi_{A2}(C2) \triangleright\triangleleft^{P} ... \triangleright\triangleleft^{P} \sigma_{An\ op\ \upsilon}(Cn)) \qquad (18)$$

Here ^{TM} indicates that the path is an enhanced path involving a time dimension. We use $\triangleright\triangleleft^{P}$ to represent the join that is more restricted than the join defined in our algebra in that the join attributes are the attribute A_{i-1} of the class C_{i-1} and identifier OID of C_i, which is defined by the system and can be considered as an attribute of class C_i, if we join C_{i-1} and C_i. The *project* (i.e., π) in the right side of equation specifies the query target.

If there is a complex predicate involving a single path such as

$$P = C1.A1\ op\ v1\ \wedge\ C1.A1.A2\ op\ v2 ... \wedge\ C1.A1.A2...An\ ^{TM}op\ vn$$
$$= P1\ \wedge P2....\wedge\ Pn$$

then we have a general form

$$\sigma_{P(C1.A1.A2....An^{TM})}(C1) = \pi_{A(C1)}(\sigma_{P1}C1 \bowtie^{P} \pi_{A2}\sigma_{P2}(C2) \bowtie^{P} ... \bowtie^{P} \sigma_{Pn}(Cn)) \qquad (19)$$

where P_i is optional in that it can be omitted if it does not exist, although P_n, which involves both time and value dimensions of the last class in the class-aggregation hierarchy, must be specified. The first *project* specifies the query target.

The purpose of defining path transformation rules is to provide alternative expressions that might be easier to evaluate during the plan generation step.

The way that a path is visited necessitates a path traversal operator, which allows alternative ways to execute the path. Whenever applicable, different methods of visiting the path such as *forward traversal*, *reverse traversal* and m*ixed traversal* may apply.

Queries may involve a single path or multiple paths. If we ignore the methods of visiting classes above, there can be classified two types of path traversal operators: the *linear* path traversal operator is for a single path execution whereas the *star* path traversal operator is for multiple path execution [Tang *et al.*, 1996]. For simplicity in our discussion, only the single path is considered here**.

Definition *Linear path traversal operator*

The linear path traversal operator is a navigational operator that executes the implicit join through a path, denoted as $Navi\_op[C_1.A_2....A_n^{TM}]$. It is equivalent to a set of joins as in equation (19).

$$Navi\_op[C_1.A_1.A_2....A_n^{TM}] = \sigma_{P_1} C_1 \triangleright\triangleleft^P \pi_{A_2} \sigma_{P2}(C_2) \triangleright\triangleleft^P ... \triangleright\triangleleft^P \sigma_{P_n}(C_n) \qquad (20)$$

According to the query equivalencies (associative law) of the previous subsection, the above linear path traversal operator can be further rewritten into the following form:

$$\begin{aligned} Navi\_op[C_1.A_1.A_2....A_n^{TM}] &= Navi\_op[C_1.A_1.....A_{n-1}] \triangleright\triangleleft^P \sigma_{P_n}(C_n) \\ &= Navi\_op[C_1.A_1.A_2....A_{n-1}] \underset{C_{n-1}.A=OID(C_n)\wedge P_n}{\triangleright\triangleleft} C_n \end{aligned} \qquad (21)$$

Thus

$$\sigma_{P(C_1.A_1.A_2....A_n^{TM})}(C_1) = \pi_{A(C_1)}(Navi\_op[C_1.A_1.....A_n] \underset{C_{n-1}.A=OID(C_n)\wedge P_n}{\triangleright\triangleleft} C_n) \qquad (22)$$

Rules (18)-(22) implies that a *select* operator with path expressions (i.e., an implicit join) can be evaluated using different algorithms, such as, translating the query into a sequence of joins, naive pointer chasing, or subdividing the query into sub-paths that can be evaluated separately using different strategies or algorithms. This is desirable as it has previously been

---

** For the situation of multiple paths without a common path, the solution provided here can be applied directly. For the situation of multiple paths with a common path, the common path will be identified first to avoid repeatedly visiting the same classes.

shown that converting implicit joins to explicit joins during the optimization phase may yield better execution plans [Blakeley *et al.,* 1993; Ozsu and Blakeley, 1995; Ozkan *at al.,* 1995] and that object navigation and set-oriented join should co-exist [Gardarin *et al.,* 1996; Tang *et al.*, 1996].

6.4 A Decomposition Strategy for Processing Temporal Object Queries

Path optimization is a central issue in object query processing that distinguishes object-oriented from relational query processing. When time is present, the enhanced path is used to express the query involving the path with time-reference. Processing such a query is more involved than that for a pure object query. In this section, we investigate a strategy to process such a temporal object query, i.e., a strategy to traverse the enhanced path expression, within the object-oriented query processing framework.

For the sake of simplicity, we suppose the time-reference occurs only at the end of the path (even if it does not, then additional accesses to the second storage are required. But this would not significantly add the complexity to the query optimization). Thus predicate P_n involves both time and value dimensions.

Based on the path transformation rule (22), a path can be divided into a series of sub-paths and different strategies or algorithms can be used to evaluate the individual sub-paths. In order to handle the query involving the time-reference and make use of the ordering information of temporal data for optimization, the evaluation of the enhanced path can thus be initially decomposed into two parts: one involves a temporal sub-path (i.e., with time-reference) and another involves an ordinary sub-path (without time-reference) which can be further decomposed into sub-paths.

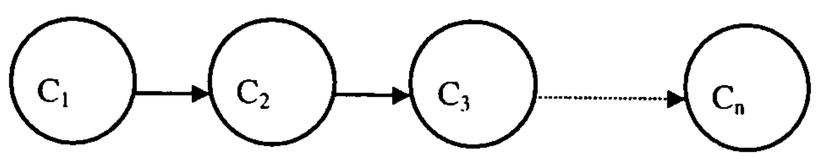


Figure 6.3 A single path

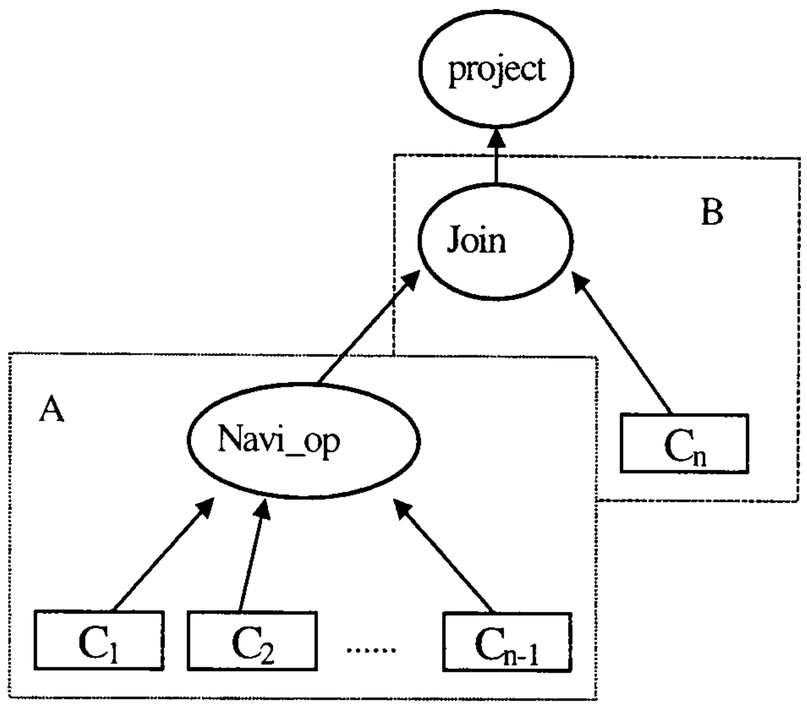


Figure 6.4 An operator graph of the *select* with a single path

Block A is an ordinary sub-path (without time-stamped classes);
Block B is a temporal sub-path (with time stamped class).

Figure 6.4 gives such an operator graph (OG) of the *select* operator in equation (23) that involves a single path as shown in **Figure 6.3**. An OG is a labelled *n*-ary tree where the leaf nodes represent collections of objects, the non-leaf nodes represent operators (e.g., *navigational operator, join, project*, etc.), and the edges represent temporary collections that can be represented by supporting tables. A support table [Gardarin *et al.,* 1996] can be regarded as a collection of tuples of qualified object identifiers and attributes. Two support tables can be joined together if there exists a common supported collection between them. The execution of an OG follows bottom-up order. The navigational operator may have more than two children, and starts from the objects in the left most collection and navigates to the right most collection. Given an OG rooted at node *N*, the cost of evaluating a single path can be expressed as [Gardarin *et al.,* 1996; Tang *et al.,* 1996]:

$$Cost(OG) = Cost(N) + \sum_i Cost(Child_i)$$

The decomposition strategy for processing temporal queries can be further illustrated in **Figure 6.5**. A complex user query with path expressions that involves time-reference is first translated into a set of single path expressions. A single path is then divided into two sub-paths: a sub-path involving time-stamped class that can be optimized by making use of the ordering information of data and an ordinary sub-path (without time-stamped class) that can be further decomposed and traversed using different algorithms. The intermediate results of traversed two sub-paths are then joined together to create the output query.

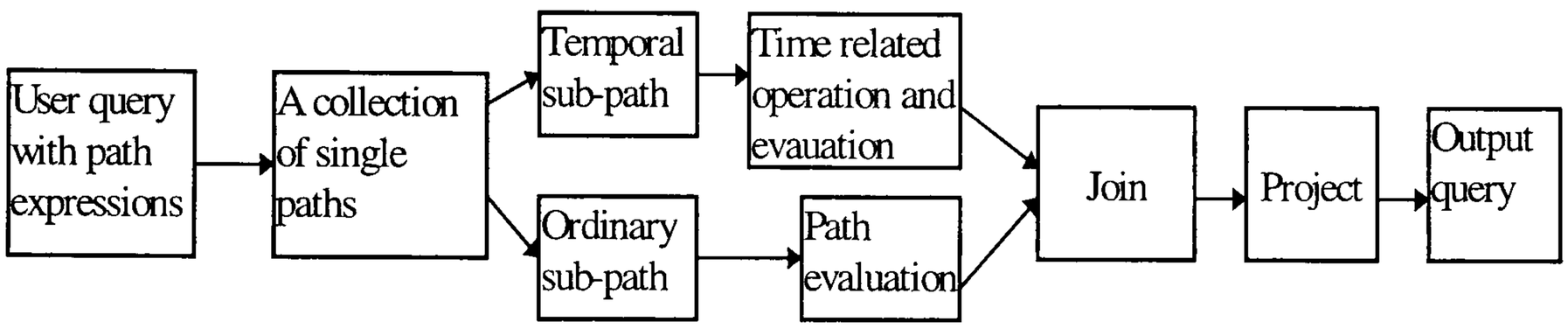


Figure 6.5 Decomposition strategy for processing temporal object queries

For instance, when responding to a user query is represented by the *select* operator with the predicate in the form of $C_1.A_1.A_2....A_n$ ^{TM} *op value*, the execution of the operator can take the form of the operator graph shown in **Figure 6.4**, i.e., first, splitting the single path into two: P1: $C_1.C_2.....C_{n-1}$ and P2: C_n; second, using the navigational operator (or other algorithms) to traverse P1, and applying time-related operations or evaluating the temporal predicates while evaluating P2 (the intermediate result of each traversing creates a support table or a derived class); third, joining two derived classes; finally, projecting the join outputs to C_1 to create the output query.

It will be shown in the following chapters that the decomposition strategy provides a convenient means to exploit the existing query evaluation algorithms to process temporal object queries and to analyse the effects of time on query processing algorithms. It provides an opportunity for optimization that makes use of the ordering information of temporal data.

6.5 Summary

Query processing techniques are dependant upon the data model and query algebra/language. In this chapter, a uniform query processing framework has been presented for processing temporal object queries based on our data model and query algebra defined in Chapters 4 and 5. Within the uniform framework, the query processor can be constructed as a layered structure and its functionality can be separated between the temporal optimizer, object optimizer and relational optimzer. Thereby an extensible approach can be explored, i.e., the query processing techniques and strategies of RDBs and OODBs, as well as sequence processing can be applied or extended at an appropriate layer of the optimizer.

Algebraic optimization for query transformation is carried out within the framework. Taking into account the object-oriented features and time-reference, various sets of query transformation rules are specified for algebraic optimization. These are relational rules, temporal transformation rules, inheritance rules and path transformation rules. Effects of these transformation rules are either to avoid redundant operations, or to reduce the size of intermediate results, or to provide an alternative that might be easier to execute. As the range of time would affect the efficiency of reading the data from the secondary storage, *time-slice, offset* should be performed as early as possible. The query transformation rules are applied to generate an equivalent expression but with the lowest possible cost.

In order to address the central issue of path otimization in object-oriented query processing when time is present, a strategy of decomposition is proposed for processing temporal queries that involve the enhanced paths, based on the path transformation rules. An enhanced path (defined as an extended path with time-references), can be initially divided into two

sub-paths: one involving time-stamped classes and the other an ordinary sub-path (with no time-stamped class) which can itself be further decomposed. The intermediate results following traversal of the two sub-paths are then joined together to create the query output. Execution of the decomposed sub-query components can be optimized by making use of well-known relational join algorithms, sequence processing and stream processing techniques which will be discussed in detail in the next chapter.

It has been shown that temporal queries can be processed within an existing object query processing framework that in turn is extended from a relational query processing framework, through the smooth extension of existing techniques. The extensible structure of the temporal data model and the properties of reducibility and closure in our query algebra provide a basis for this extensible approach. Separation of functionality between the temporal optimizer, the object optimizer and relational optimizer requires less modification to the lower level optimizer. Existing query processing techniques in both relational and object-oriented databases as well as sequence processing can be easily applied and extended.

Chapter 7

Algorithms for Processing Decomposed Query Components*

The last chapter proposed a decomposition strategy to process a temporal object query involving a path with time-reference. This chapter presents algorithms for processing the decomposed query components. These include the time-related operation algorithms and basic join algorithms. These algorithms are implemented with stream processing techniques and described with cost analysis. Simulation results are also provided.

7.1 Introduction

Path optimization is a central issue in object-oriented databases. When time is present, an enhanced path is defined to express a query involving a path with time-references. In order to exploit existing query processing techniques and to make use of the ordering information of time varying data, a decomposition strategy has been proposed for processing such a temporal object query involving the enhanced path in Chapter 6. This chapter considers the corresponding query evaluation with the provision of appropriate algorithms.

Block B in **Figure 6.4** in Chapter 6 consists of temporal predicate evaluation (as well as time-series processing) and a join, and is re-presented in **Figure 7.1**. When optimizing such a query, the object optimizer processes the outer query block, and the temporal optimizer operates on the nested query block. Each optimizer is responsible for its own query block. The temporal data provides more opportunities for optimization as discussed in [Pissinou *et al.*, 1994; Seshedri *et al.*, 1996]. The temporal optimizer is responsible for time-related

*The work in this chapter has been presented in the paper 1, 2 and 5 listed in *Author's Publications.*

operation and optimization. Let *C* represent the supporting tables or the intermediate results of block A of **Figure 6.4** (that is a derived ordinary relation), and *D* represent the intermediate results of block D of **Figure 7.1** (that is a derived temporal relation), as shown in **Figure 7.2**. The object optimizer joins *C* and *D* together.

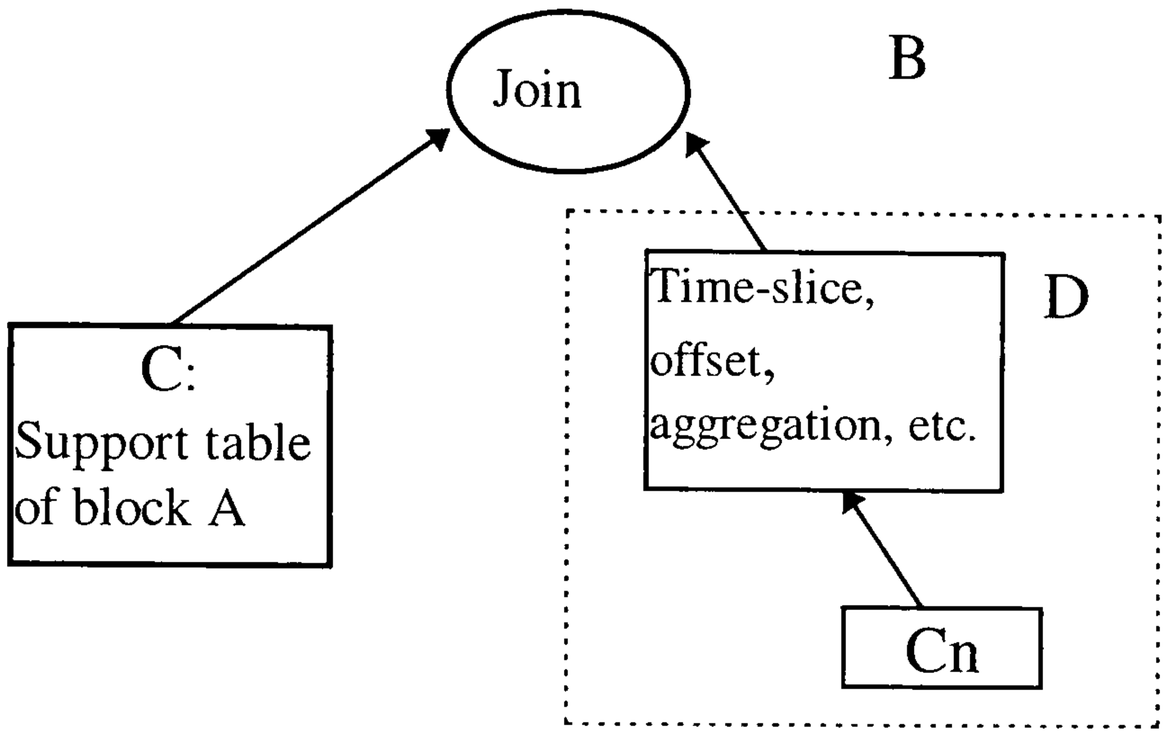


Figure 7.1 Further expression of temporal sub-path

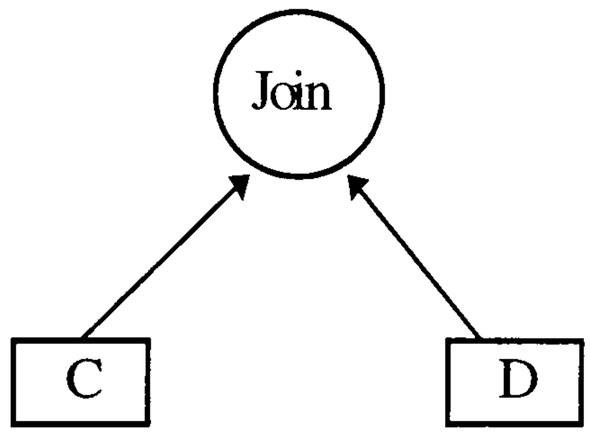


Figure 7.2 Join between C and D

This chapter provides algorithms for both the outer and nested blocks. The algorithms are implemented with stream processing techniques and described with cost analysis in terms of major operations such as block accesses, plus, move, comparison, etc., (among these, block access will dominate others). The actual cost in seconds will be used in simulation.

The remainder of chapter is organised as follows. Section 7.2 describes storage structures for query processing. Section 7.3 presents the algorithms for time-related operations. Join

algorithms are described in Section 7.4 and modifications of these are included in Section 7.5. Section 7.6 provides algorithm simulation results. Heuristics that makes use of time information for optimization will be outlined in Section 7.7 and Section 7.8 offers a summary.

7.2 Assumptions

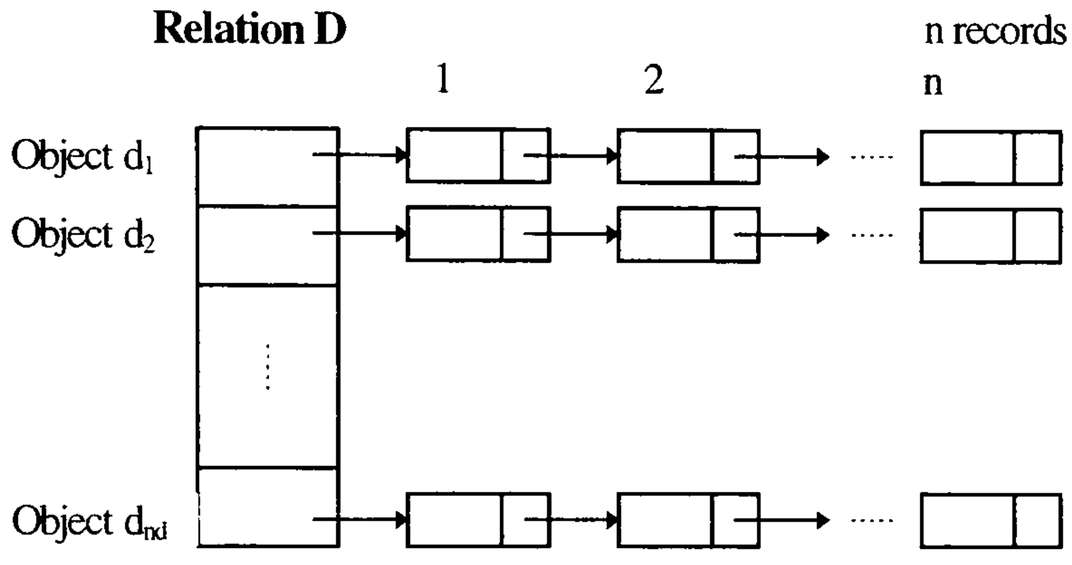


Figure 7.3 Data structure for a relation:
A temporal relation consists of n_d temporal objects;
A temporal object comprises n records representing n versions of same tuple field.

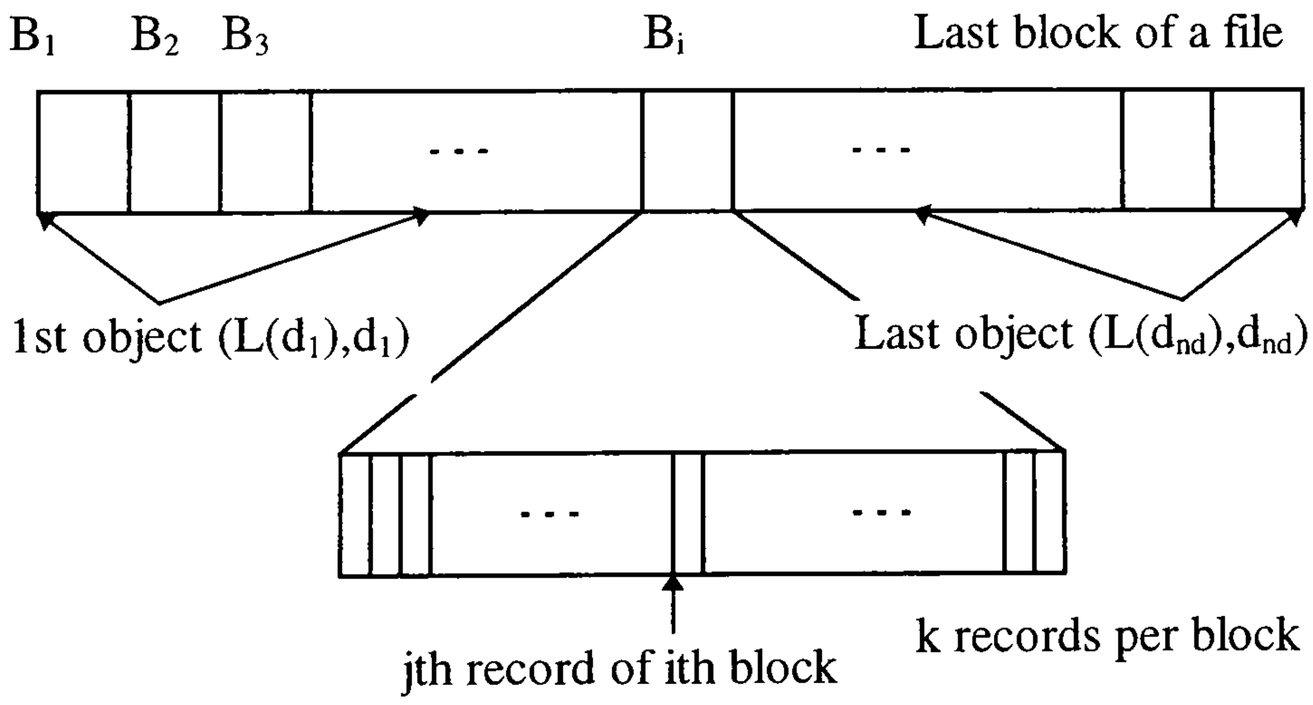


Figure 7.4 A file partitioned into blocks:
n records (history versions) of a temporal object
are stored together on a set of blocks

A temporal relation D (it could be C_n in **Figure 7.1** or D in **Figure 7.2,** depending on the situation) is stored in a file on disk. The data structure of D is as shown in **Figure 7.3,** where D is populated with temporal objects. A temporal object d is viewed as a linked list, comprising a number of records representing a number of versions of the same tuple field in a time ascending or descending order. For simplicity, the life-span is uniformly represented as $L(d)=[1,n_r]$, i.e., $L(d)=L(D)$ where 1 is the start time point and n_r is the end time point of the relation. The timestamps (temporal set) for all objects are the same and the number of epochs (the number of records/versions) in the temporal object d is $|d|=n$. If $n = n_r$ then d represents a regular TS. A temporal object can be supposed to be clustered (storing history versions together on a set of blocks), as shown in **Figure 7.4.**

We make further assumptions. Collections C and D are stored as separate files on disk. There is a many-to-one relationship from C to D. The number of *objects* in C (or D) is represented as $|C|=n_c$ (or $|D|=n_d$). n_c (or n_d) objects are blocked as b_c (or b_d) objects/block. Further, n records of a temporal object d of D are blocked as b_n records/block. Obviously, $b_n = n * b_d$. Let $fan(C, D)$ represent the average number of objects of D that are referenced by an object of C through attribute Ac. No relation is sorted or clustered. The OID is represented by a physical address of the object. Selectivity of the predicate $P()$ on the temporal collection D is treated as the same as that on an ordinary relation, denoted as *sel* (the complexity of selectivity of temporal relations is ignored).

7.3 Stream Processing Algorithms for Time-related Operations

Predicate evaluation in **Figure 7.1** involves the time-related operations and value evaluation. Temporal operations such as *time-slice, offset, agg-func* can be treated as methods and their outputs can then participate in the value evaluation. The temporal optimizer must be sure to 'plan' the evocation of function and to make use of the ordering information for optimization [Pissinou *et al.*, 1994].

We employ the stream processing techniques [Leung and Muntz, 1993; Carrano, 1995]. Stream processing is a paradigm that has been widely studied and used in languages such as C++, Lisp, etc. Abstractly, a stream is defined as an ordered sequence of data objects. As temporal data often implies ordering by time, stream processing approach is a strategy of choice [Leung and Muntz, 1993], so that tuples in a data stream can be efficiently accessed one at a time and in the order of successive time-stamp values, using the data stream pointer.

7.3.1 Stream Processing Algorithm for *Time-slice*

Time-slice $\S_{T1}(O)$ presented in Chapter 5, performs the operation: for every object d in D, select its records ds whose positions fall in $T_1=[n_l, n_m]$, $T_1 \subset L(D)=[1,n_r]$, supposing that there are n records in an object of relation D. To minimise the number of accesses to data, only records satisfying the above condition are retrieved. Employing the C++ stream processing technique to implement this operation can fulfil this task. The file stream in C++ allows a user to treat a file as a stream of input or output. Given a data object, its size can be decided by using the C++ function *sizeof( )*. When the exact position (i.e., the exact address), from which the data object is stored, is determined by C++ function *seekg*, the data object can retrieved and the file pointer moves to the next data object. Given a temporal object d, to retrieve its records d_s from time n_l and n_m, we need to find out its exact positions corresponding to n_l and n_m in the file. We can obtain these by either sequential or binary search within the scope of the object d to decide the epoch number n_{nl} that is corresponding to time point n_l, and the epoch number n_{nm} that is corresponding to time point n_m. This can be shown in the following pseudo code of C++:

search (n_l, n_m, n_{nl}, n_{nm});

/* sequential or binary search to find out the epoch number n_{nl} corresponding to time n_l, and the epoch number n_{nm} corresponding to time n_m */

for (int i=0; i<n_d*; i++)* /*for each object d_{i+1}, $i+1 \in [1, n_d]$, do the following */

{

*fileD.seekg(i*sizeof(d)+ (*n_{nl} *-1)*sizeof(ds));*

/*seek the address of the first d_s, whose time point is n_l, that is in T_1. */

for (int j=0; j<= n_{nm} -n_{nl}; j++)

{fileD.read((char)&Struc_Buf[j], sizeof (ds));*

Buf<<Struc-Buf[j];}

/*sequentially read (n_{nm} -n_{nl}+1) records d_S of the object d_{i+1} from the file,

keep them in *Struc_Buf[j],* and output results to *Buf* that could be a screen, a printer, a buffer, etc. */

}

file2.write((char)&Buf, sizeof (Buf));* /*if we want to write the output results to a file */

Although C++ provides stream access that allows access one record/object at a time, the system actually performs I/O at the block level and perhaps hides this fact from the program [Carrano, 1995]. We follow the assumption [Carrano, 1995] that when the system provides an access for one record/object, it assesses the entire block that contains the record/object. If the next record/object is already in the stream accessed, it does not need to access the block again. Therefore we still can measure I/O access by blocks or pages. For the above algorithm, the number of block accesses is estimated as:

$$n_d*(n_{nm} - n_{nl} + 1)/b_n \leq n_d*(n_m - n_l + 1)/b_n,$$

plus searching block access cost:

$\leq n/b_n$ in the case of sequential search [Carrano, 1995];

or $\leq 2*(log_2\ n)$ in the case of binary search [Carrano, 1995].

As with other database executive algorithm analysis [Shekita and Carey, 1990], the output cost is ignored here because it does not make a contribution to the algorithm efficiency. (This will be assumed in the algorithms analysis hereafter too.)

7.3.2 Stream Processing Aggregation Algorithms

The operator *agg-func* T_1 used to perform the aggregation (such as *avg, sum, max*, etc.) can be implemented with stream processing techniques as shown in following pseudo-code of C++:

search (n_l, n_m, n_{nl}, n_{nm});

/* sequential or binary search to find out the epoch number n_{nl} corresponding to time n_l, and the epoch number n_{nm} corresponding to time n_m */

for (int i=0; i<n_d; i++) /* for each object d_{i+1}, $i+1 \in [1, n_d]$, do the following */

{

*fileD.seekg(i*sizeof(d)+ (n_{nl} -1)*sizeof(ds));*

/* seek the address of the first d_s, whose time point is n_l, that is in T_1. */

for (int j=0; j<= n_{nm}-n_{nl}; j++) fileD.read((char)&Struc_Buf[j], sizeof (ds));*

/*sequentially read (n_{nm} -n_{nl}+1) records d_S of the object d_{i+1} from the file and keep them in *Struc_Buf[j]* */

agg_func(Struc_Buf.itemi, func, value);

/*perform an aggregation function *func* for a specified attribute (i.e., itermi) */

Buf<<value; /* output results to *Buf* that could be a screen, a printer, a buffer, etc. */

}

void agg_func(item, char func, float value);*

int m= n_{nm}-n_{nl}+1;

switch (func)

{

case sum:

for (value=0; int i=0; i<m; i++)

for (int i2=0;i2<=ϖ(item(i+1))-ϖ(item(i)); i2++) value+=item[i];

/* insert data for missed time points and add them to the value, where ϖ(*item(i))* is our algebraic operator *when* that maps *item(i)* to its time point. For simplicity here we assume that missed data is of stepwise constant. */

break;

```
case avg:
for (value=0; int i=0; i<m; i++)
for (int i2=0; i2<=ϖ( item(i+1) )-ϖ( item(i) ); i2++) value+=item[i];
value=value/m;
break;

case max:
for (value=-10^35 ; int i=0; i<m; i++)
if (item[i]>value)
value=item[i];
break;
.....
}

file2.write((char*)&Buf, sizeof (Buf)); /*if we want to write the output results to a file */
```

In addition to the number of block accesses as in the *time-slice* algorithm, the following major operations are needed for *Agg-func*, if we ignore the time for assignment and *when:*

n_m-n_l+*1* plus for *sum*;

n_m-n_l+*1* plus and *1* division for *avg*;

n_m-n_l+*1* comparisons for *max*;

.....

7.4 Join Algorithms

This section discusses the algorithms to join *C* and *D* together. The advantage to represent them as explicit joins is that we can use well-established join algorithm strategies to perform optimization. Here a temporal object stands as a "blob" object that can be treated as an ordinary object in a snapshot OODBs. There are two types of joins to join *C* and *D* together: **forward join** and **reverse join**. According to the access methods of traversing the path, i.e.,

nested-loop method and **sort-merge** method, there are four basic join algorithms: **nested-loop forward join**, **sort-merge forward join**, **nested-loop reverse join and sort-merge reverse join** [Bertino and Martino, 1993]. The following presents these algorithms that are implemented employing stream processing techniques.

7.4.1 Nested-Loop Forward Join (NLFJ)

NLFJ, sometimes called the pointer-based nested-loop algorithm [Shekita and Carey, 1990], is the algorithm that uses naive pointer traversal to compute the join. An instance *c* of *C* is retrieved and the value of *Ac* is determined. Given this identifier, the address of the object *d* of *D* is determined. The object *d* of *D* is retrieved and the predicate is evaluated. If true, *c* and *d* are joined. This process is repeated until all instances of *D* are visited. NLFJ can be expressed using the following pseudo-code C++:

for (int i=0; i<n_C; i++) /* for each object *c*, do the following */
{fileC.read((char)&BufC, sizeof(c));* /*read the object *c* of *C*/
fileD.seekg(c.Ac);
/* according to the value of the attribute Ac, i.e., the OID of *d*, locate the address of *d* of *D* */
fileD.read((char)&BufD, sizeof (d));* /* read the object *d* of *D* */
if (predicate) Buf<<(join c and d);
/*if the predicate satisfied, join *c* and *d,* and then output results to *Buf* that could be a screen, a printer, a buffer, etc. */
}
file2.write((char)&Buf, sizeof (Buf));*/*if we want to write the output results to a file.*/

For the above algorithm, the number of block access is estimated as

read C: n_c/b_c;
read D: $fan(C,D) * n_c /b_d + = fan(C,D) * n_c * n/b_n +$.

There are

fan(C,D) * n_c comparisons for predicate evaluation;

sel fan(C,D)* * n_c moves (for join).

One of the problems with the pointer-based nested-loop algorithm is that it makes no attempt to optimize disk reads [Shekita and Carey, 1990]. As a result, a particular disk block in *D* can end up being read more than once. For example, suppose that two objects c_1 and c_2 reference the same object *d* in *D*. Depending on how *C* is organised, c_1 and c_2 may not be physically clustered together in *C*. If that is the case, then between the time when c_1 is joined to *d* and the time when c_2 is joined to *d*, the block containing *d* may be paged out of memory by buffer replacement algorithm. In that event, that block would have to be read twice, once to join c_1 with *d* and a second time to join c_2 with *d*.

7.4.2 Sort-Merge Forward Join (SMFJ)

SMFJ, sometimes called the pointer-based sort-merge algorithm [Shekita and Carey, 1990], avoids the aforementioned problem by first sorting all of the objects in *C* by the value of *Ac* (i.e., the OID of *d* in *D*). The effect of sorting *C* in this manner is to group all of the objects in *C* that reference the same page in *D*. Doing so guarantees that each page in *D* will be read only once. The algorithm is executed as follows. All the objects of *C* are read into memory and sorted as in the standard sort-merge algorithm, except that here the output runs are sorted by OID values rather than by the join attribute. According the value of Ac, the address (i.e., the OID) of an object *d* of *D* is determined so that the object *d* is retrieved, the predicate is evaluated and if true, *c* and *d* are joined. Repeat this process till all addresses are visited. The pseudo-code of C++ is

```
fileC.read((char*)&BufC, sizeof(C) ); /* read the whole collection C */
sort C according to Ac;
for (int i=0; i<n_C; i++) /*for each object c of C, do the following */
{
fileD.seekg(c.Ac);
/*according to the value of the attribute Ac of c, i.e., the OID of d, locate the address of d of D*/
fileD.read((char*)&BufD, sizeof (d)); /* read the object d of D */
```

if (predicate) Buf<<(join c and d);

/*if the predicate is satisfied, join *c* and *d,* and then output results to *Buf* that could be a screen, a printer, a buffer, etc. */

}

file2.write((char)&Buf, sizeof (Buf));* /*if we want to write the output results to a file */

For the above algorithm, the number of block access is estimated as

read C: n_c/b_c;

read D: $fan(C,D) * n_c /b_d = fan(C,D) * n_c * n/(b_n)$.

There are:

$fan(C,D) * n_c$ comparisons for predicate evaluation;

$sel * fan(C,D) * n_c$ moves (for join);

sorting cost: $sorting(n_c)$.

When the time-dimension *n* is big enough such that a temporal object occupies more than one block, SMFJ will not obviously be better than NLFJ (but at the price of sorting *C*, and a bigger memory to hold the whole *C*).

7.4.3 Nested-Loop Reverse Join (NLRJ)

This strategy is similar to NLFJ, except that *D* is the first class visited. An object *d* of *D* is read into memory and predicate is evaluated. If the predicate is verified, then a search on the instance *c* of *C* is executed to determine which instance has object *d* as the value of the attribute *Ac*. *c* and *d* are then joined. This process is repeated until all instances of *D* are visited. The pseudo-code for the algorithm is:

for (int i=0; i<n_d; i++) /* for each object d_{i+1}, $i+1 \in [1, n_d]$, do the following */

{fileD.read((char)&BufD, sizeof (d));* /* read an object d_{i+1} of *D* */

if (predicate) /* if the predicate is satisfied */

for (int j=0; j<n_C; j++) /* for each object c_{j+1}, $j+1 \in [1, n_C]$, do the following */

{fileC.read((char) &BufC, sizeof (c));* /* read an object c_{j+1} of *C* */

if ((fileD.tellg()-sizeof(d))==c.Ac)

Buf<<(join c and d); }

/* verify if the address (the OID) of d_{i+1} is equal to the value of $c_{j+1}.Ac$, if so, join c_{j+1} and d_{i+1}, and output results to *Buf* that could be a screen, a printer, a buffer, etc. */

}

file2.write((char)&Buf, sizeof (Buf));* /*if we want to write the output results to a file. */

The number of block access is estimated as

read C: $sel * n_d * n_c / b_c$;

read D: $n_d / b_d = n_d * n / b_n$.

There are:

n_d comparisons operations for predicate evaluation;

$sel * n_d * n_c$ comparisons for value evaluation;

$sel * n_d * (fan(C,D) * n_c / n_d) = sel * fan(C,D) * n_c$ moves (for join).

Clearly objects in *C* have to be read many times, resulting in high I/O cost. If there are reverse references from the instances of *D* to the instances of *C*, the instances of *C* do not needed to be examined. Instead, the objects are accessed directly by the following these references.

7.4.4 Sort-Merge Reverse Join (SMRJ)

In SMRJ, all the instances of *D* are accessed, the predicate is evaluated and a list of OIDs of instances qualifying the predicate is generated. *C* is read into memory and sorted according

to *Ac*. The instances of *C* are then selected to determine which instances have the identifier as the value of attribute *Ac*. If so, *c* and *d* are joined. The pseudo-code for the algorithm is:

for (int j=0; int i=0; i<n_d; i++) /* for each object *d* of *D*, do the following */

{fileD.read((char)&d, sizeof(d));* /* read an object *d* of *D* */

if (predicate) {D'[j]=d; jd[j]=fileD.tellg()-sizeof(d); j++;}

}

/* if the predicate is satisfied, keep the object *d* in *D'[ ]*, and its address (the OID) in *jd[ ]*.

This is equivalent to perform select first, the resulting relation is *D'[ ]*, its cardinality is *j*. */

fileC.read((char)&BufC, sizeof (C));* /* read the relation *C* */

sort C according to Ac;

j2=0;

for (int i=0; i<j-1; i++) /* for each object d_i in *D'[ ]*, $i \in [0, j-1)$, do the following */

for (int i2=j2; i2<n_C; i2++)

/* for each object c_{i2} in *C*, $i2 \in [j2, n_C)$, do the following

(where *j2* starts from 0 and increases by 1 after a join is made) */

{if (C[i2].Ac==jd[i]) {Buf<<(join C[i2] and D'[i]); j2=i2+1; };

/* if the value of c_{i2}.*Ac* is equal to the address (the OID) of d_i, join c_{i2} and d_i,

and output results to *Buf* that could be a screen , a printer, etc. */

else if (C[i2].Ac>jd[i]) break;

/* if the value of c_{i2}.*Ac* is greater than the address (the OID) of d_i, stop looping of *i2*. */

}

file2.write((char)&Buf, sizeof (Buf));* /* if we want to write the output results to a file.

*/

The number of block access can be estimated as

read C: n_c / b_c;

read D: $n_d / b_d = n_d * n/(b_n)$.

There are:

n_d comparisons for predicate evaluation;

$sel*n_d*(fan(C,D)*n_c/n_d+1)$ comparisons for value evaluation;

$sel*fan(C,D)*n_c$ moves (for join); and

sorting cost: $sorting(n_c)$.

7.4.5 Sorting

If SelectSort algorithm [Carrano, 1995] is used to sort items in relation *C* in the ascending order by the attribute *A* it is $O(n_c^2)$ in terms of major operations.

The pseudo code of C++ that applies SelectSort algorithm [Carrano, 1995] to sort the relation *C* in the ascending order by attribute *Ac* can be represented in the following:

```
void Sortfunction(C, n_c, Ac)
/* sort the items in relation C of size n_c in the ascending order by the attribute Ac */
for(int Last= n_c -1; Last>=1;--Last)
{int L=IndexOfLargest(C.Ac, Last+1); /* select largest item in C.Ac[0..Last] */
Swap (C[L),C[Last]; /*swap largest item C[L] with C[Last] */
} /*end for */
```

The function calls other two functions:

```
int IndexOfLargest (const dataType Ac, int Size) /*find the largest item in the column Ac */
{int IndexSoFar=0; /*index of largest item found so far */
for (int CurrentIndex=1; CurrentIndex<Size;++ CurrentIndex)
{if (Ac[CurrentIndex] >=Ac[IndexSoFar]) IndexSoFar= CurrentIndex;
} /*end for */
return IndexSoFar; /* index of largest item */
} /* end IndexOfLargest */

void Swap (dataType & X, dataType &Y) /*function for swapping X and Y */
{
```

```
dataType Temp=X;
X=Y;
Y=Temp;
} /*end Swap */
```

One important divide-and-conquer sorting algorithm, MergeSort [Carrano, 1995], has an elegant recursive formulation and is highly efficient. MergeSort is a recursive sorting algorithm that always gives the same performance, regardless of the initial order of the collection items. The order of MergeSort algorithm is $O(n_c log\ n_c)$.

7.4.6 Summary

Table 7.1 Summary of join algorithm costs

Algorithm	**Number of block access**	**Other operations**
NLFJ	read C: n_c/b_c; read D: $fan(C,D) * n_c /b_d$+ $= fan(C,D) * n_c * n/b_n$+	$fan(C,D) * n_c$ comparisons; $sel * fan(C,D) * n_c$ moves.
SMFJ	read C: n_c/b_c; read D: $fan(C,D) * n_c /b_d$ $=fan(C,D) * n_c * n/(b_n)$	$fan(C,D) * n_c$ comparisons; $sel * fan(C,D) * n_c$ moves; sorting cost: $sorting(n_c)$.
NLRJ	read C: $sel * n_d * n_c /b_c$; read D: $n_d /b_d = n_d * n/b_n$.	$n_d + sel * n_d * n_c$ comparisons; $sel * n_d * (fan(C,D) * n_c/n_d)$ $=sel * fan(C,D) * n_c$ moves.
SMRJ	read C: n_c /b_c; read D: $n_d /b_d = n_d * n/(b_n)$.	$N_d + sel * n_d * (fan(C,D) * n_c/n_d + 1)$ comparisons; $sel * fan(C,D) * n_c$ moves; sorting cost: $sorting(n_c)$.

Table 7.1 gives a summary of join algorithm costs in terms of major operations. As the time required for block accesses typically dominates other factors [Carrano, 1995], it can be concluded that the order of above four basic join algorithms are all *O(n),* in terms of block access. That means the join time cost linearly increases with the expansion in the number of time epochs (or time dimension, in the case of a regular TS).

The advantage of sort-merge method over the nested-loop method is that the storage pages containing class instances of the class are never accessed more than once, resulting in considerable saving in terms of response time. The disadvantage is that the algorithms are restricted by available memory. If the memory is relatively small or the number of the objects in a relation is too big, all objects of the class cannot be read into the memory. The algorithms need to be modified in order to be more practical. This will be discussed in the following section.

The disadvantage of reverse join algorithms is that as there is no direct link from *D* collection to *C*, a value-based join must be used to check the OID membership condition, i.e., it performs value-based comparisons of OIDs, which is generally inefficient in CPU usage terms [Gardarin *et al.*, 1996]. This algorithm is efficient when the predicate in the last collection is selective [Ozsu and Blakeley, 1995; Tang *et al.*, 1996].

We did not discuss hybrid-hash join here, as when the epoch number *n* is big enough such that a temporal object occupies more than one block, implementation of the algorithms with stream processing techniques will not provide an obvious advantage over NLFJ.

7.5 Modification of Join Algorithms

As mentioned in previous section, sort-merge algorithms require a relatively large memory. If the memory is relatively small, all instances of the class can not be read into the memory and the algorithms are not useful. In this case, the algorithms need to be modified. This section presents a solution to amend this problem.

Sorting the collection *C* in the external storage

When the file of *C* is far too large to fit into internal memory all at once, it presents some restriction on sort-merge join algorithms because the sorting algorithms presented earlier in previous section assume that all the data to be sorted are available at one time in internal memory. To solve this problem, alternative is to sort *C* using the techniques of sorting data in an external file [Carrano, 1995]. Using this technique, SMFJ will be modified as below:

sort C according to Ac;
/* sort the collection *C* by using an external MergeSort or SelectSort. */
for (int i=0; i<n_C; i++)
{ fileC.read((char)&BufC, sizeof(c));* /*read one instance *c* of collection *C* */
fileD.seekg(c.Ac); /* seek the address of *d* by OID of (*d*) (i.e., the value of *Ac*) */
fileD.read((char)&BufD, sizeof (d));*
if (predicate) Buf<<(join c and d); /*output results */
}

file2.write((char)&Buf, sizeof (Buf));* /*write output to a file */

It is essentially an NLFJ, except *C* is sorted before the execution. Therefore the join cost is that of NLFJ plus the cost of sorting using external sorting techniques that require at least additional n_c/b_c block access to read *C*.

For SMRJ, in addition to sorting *C* using external techniques, there is no need to read all objects of *D* before *C* is read because the addresses of instances imply the ascending order of OIDs. This means that the memory that SMRJ requires to hold the derived *D* whose objects have satisfied the predicate is not necessary. The modified SMRJ is as follows:

```
sort C according to Ac;
/* sort the collection C by using an external MergeSort  or SelectSort*/
j2=0;
for (int i=0; i<n_d; i++)
{fileD.read((char*)&d, sizeof (d)); /* read an object d of D */
if (predicate)
{
for (int j=j2; j<n_c; j++)
{
fileC.seekg(j*sizeof (c) )
fileC.read((char*) &c, sizeof (c) ); /* read an object c of C */
if ((c. Ac == fileD.tellg()-sizeof(d)))
 /* verify if the address of d is equal to the value of c.Ac */
{Buf<<(join c and d); j2++;}
else if (c.Ac>(fileD.tellg()-sizeof(d))) break;
}}}

file2.write((char*)&Buf, sizeof (Buf)); /*output results */
```

Compared with SMRJ, the modified SMRJ exchange the cost *sorting*(n_c) to the external file sorting cost that requires at least additional n_c / b_c block access to read C. As a trade-off, it saves the memory to hold the whole C and the derived D.

7.6 Simulation

This section provides simulation results to evaluate the join algorithms presented in the previous chapter.

7.6.1 Experimental Database: "The International Weather Record Database"

The experimental database is taken from the database example presented in Chapter 4: "The International Weather Record Database" as shown in **Figure 7.5.** Daily weather changes are recorded for major cities world-wide. The time chronon is a day. For simplicity, suppose that time starts from 1 and ends at today *(n)*. The life-span can uniformly be *L(TESTINFO)=[1,n]*. The number of records in a temporal object of relation *TESTINFO* is *n*, representing a regular TS. The relation *CITY*, analogous to a supporting table that is described in Chapter 6, is relatively small: the cardinality of *CITY* is n_c=100, as our intention is to show the relationship of the join response time with respect to *n*, i.e., the number of epochs (records) in a temporal object of relation *TESTINFO*. In this example, |*TESTINFO*|, i.e., n_d, is also 100. That means *fan(CITY, TESTINFO)* is 1.

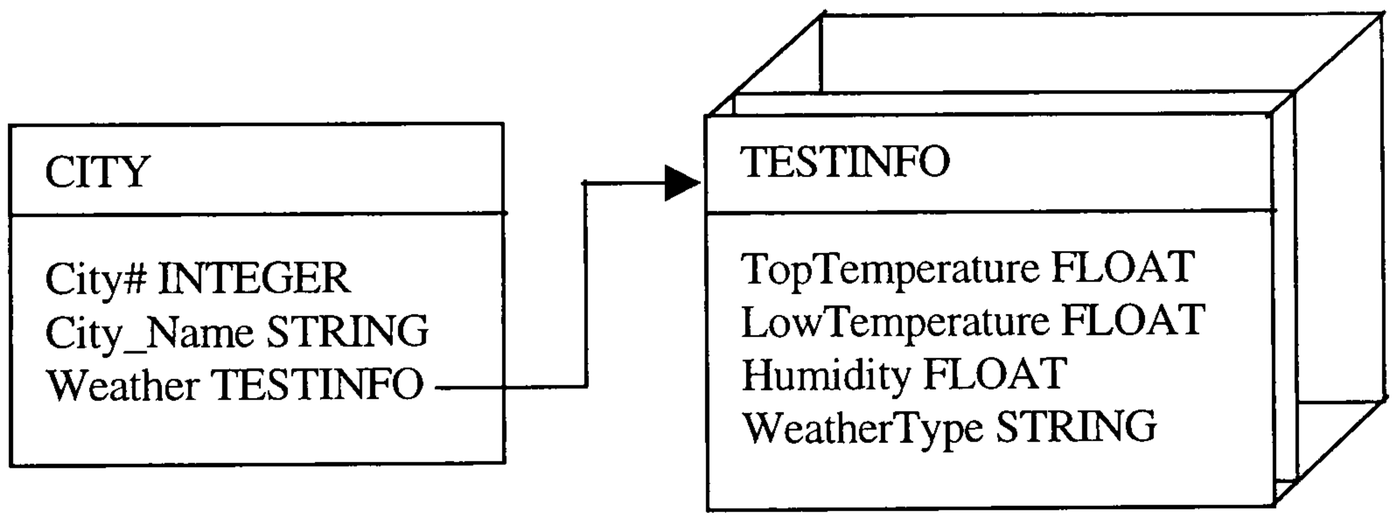


Figure 7.5 Extracted from simplified International Weather Record Database

7.6.2 Simulation Programs

Simulation environment

The simulation environment is Borland C++ Version 4 on a PC.

Simulation programs

The simulation consists of a set of programs:

- The head file that defines the data structures;
- The data generating program that creates two collections *CITY* and *TESTINFO* and stores them in separate files on disk;
- Sort program that use SelectSort[Carrano, 1995] to sort items in a relation in the ascending order by an attribute .
- Unsorting program that unsorts the collection *CITY* and saves the results. The OIDs of *TESTINFO*, that are the values of attribute weather of *CITY,* are the physical addresses of *TESTINFO* instances. As the addresses are generated in an ascending order whilst our algorithms suppose both join collections are unsorted, so we need to unsort them by sorting the *CITY* by different attribute such as *City_Name*.
- NLFJ program that implements the nested-loop forward join algorithm;
- SMFJ program that implements the sort-merge forward join algorithm;
- NLRJ program that implements the nested-loop reverse join algorithm;
- SMRJ program that implements the sort-merge reverse join algorithm;
- The modified SMFJ program that implements the modified algorithm for the sort-merge forward join;
- The modified SMRJ program that implements the modified algorithm for the sort-merge forward join.

The definition of classes is as follows:

```
struct testinfo
{
int top_temperature;
```

```
int low_temperature;
float humidity;
char weather_type[10];
};

const int max_info=n;
typedef testinfo groupinfo[max_info];

struct city
 {
 int number;
 char name[10];
 groupinfo gi;
 };

struct city2
 {
 int number;
 char name[10];
 long g;
};
```

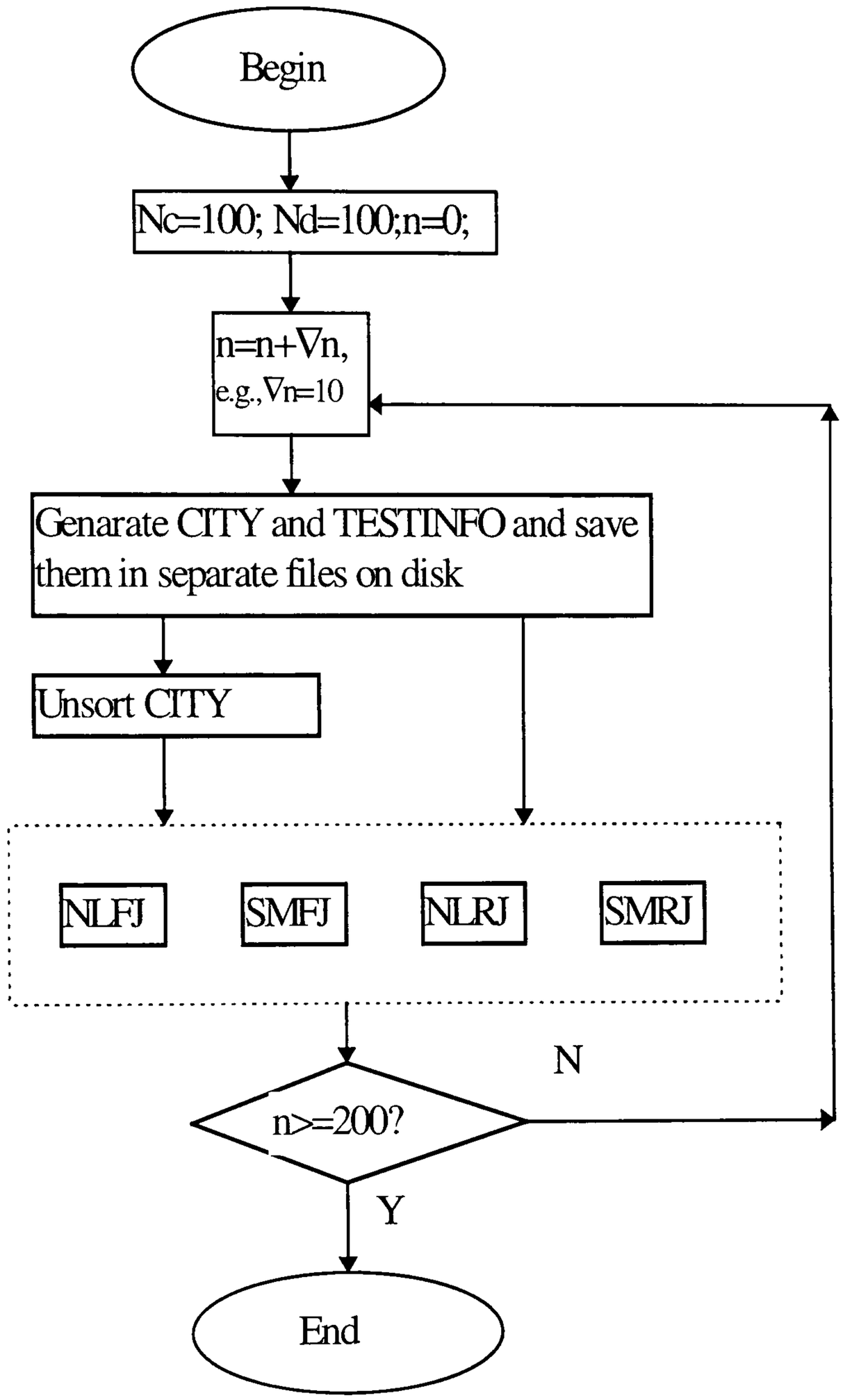


Figure 7.6 Execution of simulation programs of basic join algorithms

Here *groupinfo* extends *testinfo* with *n* elements to represent a temporal class, and *city2 is a* copy of *city* but the third attribute is used to keep the address of *testinfo*, (i.e., the OID of *testinfo*). Once the database schema is defined, the database needs to be populated with a large number of objects by the data generating program. Query processing algorithms such as NLFJ, etc. provide the means to retrieve the data on request. The execution of the basic join algorithm programs is as shown in **Figure 7.6**. Simulations of these algorithms applying different time-slice intervals as well as the modified SMFJ and the modified SMRJ are also provided.

7.6.3 Simulation Results and Discussion

The four basic join algorithms and modified sort-merge algorithms have been implemented on PC using Borland C++ Version 4 where SelectSort algorithm is employed.

Figure 7.7, 7.8, 7.9 and 7.10 present the performance of four join algorithms, where the vertical axis represents join time costs in second and the horizontal axis represents the number of epochs in the TS, i.e., *n*. Different join algorithms are denoted with different types of lines as shown in the figures. Selectivity is set at 10%, 33%, 50% and 100% respectively. Obviously the join cost increases linearly with *n*. Performance of NLRJ is worst, because it reads the relation *CITY* many times. Sort-merge join algorithms are generally good when the relations are relatively small and *n* is small. But they are limited by the memory of the computer and the algorithms are terminated when *n* is greater than 100, because the algorithms can not work in the given hardware environment when $n \geq 100$. There is not too much difference between NLFJ and sort-merge join algorithms, which is because the experimental example possesses a one-to-one relationship.

Figure 7.11 7.12, 7.13 and 7.14 show the performance of join algorithms with respect to selectivity *sel*. It can be seen that the join time cost increases when the selectivity increases.

Figure 7.15, 7.16, 7.17 and **7.18** provide a comparison of the performance of NLFJ, the modified SMFJ, NLRJ and the modified SMRJ when n is expanded to 5000.

All the above simulation results conform to the cost analysis presented in previous sections. That is, the join time cost is linearly increased with the expansion of the time-epochs (the time-dimension, in the case of a regular TS).

Figure 7.19 and **7.20 /Figure 7.21** and **7.22 /Figure 7.23** and **7.24 /Figure 7.25** and **7.26** provide a comparison of the performance of NLFJ /SMFJ /NLRJ /SMRJ with and without time-slice intervals. The performance of join algorithms without time-slice is analogous to that of OODBs which allow users to represent temporal data as a 'blob' object but with no support for time varying query whilst the performance of join algorithms with time-slice is analogous to that of OODBs which support for time-varying data and utilise the heuristics for optimization such as that presented in previous section. The span of time-slice $T_l = [n_l, n_m]$ is denoted as $T_m = (n_m - n, +1)$. When $T_m << n$, there is a significant saving. The bigger the value of $(n - T_m)$, the greater the cost saving. When T_m is close to n and n is close to b_n, there is no significant cost saving.

Discussion

It is generally recognised [e.g., in Ozsoyoglu and Snodgrass, 1995, Leung and Muntz, 1993] that optimization of temporal queries is substantially more involved than that for conventional queries on the one hand, and there is a greater opportunity for query optimization when time is present on the other hand. As is shown in our simulation, the rapid performance degradation is due to ever-growing overflow chains, and if the query is unoptimized, it takes longer and longer to execute. This justifies trying harder to optimize the queries and spending more execution time to perform the optimization.

Most object-oriented database proposals include constructors for complex types like lists and arrays that allow time-stamped entity to be represented as a "blob", which is managed by the system, but interpreted solely by the application program; no facilities for temporal

queries are provided [Seshadri *et al.*, 1996]. It has been pointed out by Kim *et al.* [1997] and Kim [1995; 1994; 1993] that one necessary topic of research in historical data management is to quantitatively establish the performance (and even productivity) differences between using a database system that directly supports temporal attributes and using a conventional database system that does not support either the set-valued attributes or temporal attributes. If it can be convincingly established that the benefits of a database system that supports temporal attributes are substantial, database vendors will strongly motivated to augment their systems with historical data management.

The simulation results as well as cost analysis imply that solely treating a temporal object as a 'blob' object that is managed by the system, but interpreted by user is not a strategy for temporal support in OODBs. It also suggests that for OODBs that support for time-varying data, when the number of epochs is big enough, i.e., $n>>b_n$, there is certainly a need to provide facilities that support temporal queries.

7.7 Heuristics for Optimization

The last section shows the effects of time on query processing algorithms. As pointed out by Leung and Muntz [1993] and Ozsoyoglu and Snodgrass [1995], that because adding time creates multiple tuple versions of the same object, reorganisation does not help to shorten overflow chains, the objective of work in temporal query evaluation, then is to avoid looking at all of the data [Leung and Muntz, 1993; Seshadri *et al.*, 1996]. Good heuristics that take advantage of data ordering and restrict the scope of a sequence can attain this objective.

Here we simply outline the following heuristics that make use of the ordering information of data and can be exploited for optimization. These ideas have been introduced in earlier in chapters.

1) Transform the time-related predicate into *time-slice*.

For example, select the city whose top temperature at t_0 is higher than 30^0 C. If the life-span of *TESTINFO* is [1,10000], i.e., n=10000, and the record/block is b_n = 100, retrieving an object needs 100 block accesses. Retrieving the record at t_0, that specifies the range of access, only needs 1 block access. That is, given $p()$ = $CITY.Weather.TopTemperature|_{t=t0} \geq 30^0$, the algebra for the query is

$$\sigma_{p()}(CITY = \pi_{<City\#,\ City\_Name,\ Weather>}(CITY \overset{P}{\triangleright\!\triangleleft}_{\text{CITY. Weather = OID of TESTINFO and P()}} TESTINFO)$$

It is equivalent to

$$\sigma_{p()}(CITY) =$$

$$\pi_{<City\#,City\_Name,Weather>}(CITY \overset{P}{\triangleright\!\triangleleft}_{\text{CITY. Weather = OID of TESTINFO and P()}} \xi_{T1=t0}(TESTINFO))$$

The block access of *TESTINFO* for the first expression is $n_d*n/b_n=10000*n_d/b_n$ =100 n_d whilst the block access for the second expression is n_d. The second expression is obviously much more economic. i.e., the block access to *TESTINFO* using the first expression costs 100 times more than that in the second expression..

2) Perform *time-slice* as early as possible.

The use of this heuristic is to avoid looking at all data. For example, for the following expression:

$$\pi_{<TopTemperature>}(\xi_{T1=[nl,nm]}(TESTINFO)) = \xi_{T1=[nl,\ nm]}(\pi_{<TopTemperature>}(TESTINFO))$$

The block access of *TESTINFO* for the left expression is $n_d*(n_m-n_l)/b_n$ whilst the block access for the right expression is n_d*n/b_n. As long as $n>>(n_m-n_l)$, a considerable saving will be attained from the left hand expression (e.g., n= 10000 whereas (n_m-n_l) =10,

choosing the left hand expression will be 1000 times cheaper than the right hand expression.

3) Combine sequences of unary operations

A cascade of unary operations such as time-slice, offset, select and project can be combined by applying them in a group as we scan each object. Similarly, we can combine these unary operations with a prior binary operation such as join, if we apply the unary operations to the result of the binary operations as we construct it.

4) Making use of temporal constraints

7.8 Summary

In this chapter, we have presented a set of algorithms for processing the decomposed query components which have resulted from employing the decomposition strategy presented in the previous chapter. These algorithms can be employed to process a temporal object query that involves an enhanced path (a path with an explicit time-reference). The algorithms include time-related operations and four basic *join* algorithms, which are described with cost analyses and implemented using stream processing techniques. The order of all four *join* algorithms is *O(n)*. Of them, NLRJ has the worst performance because it tends to perform an excessive number of disk accesses. Sort-merge algorithms (i.e., SMFJ, SMRJ) are generally good, but limited by their heavy demands on memory. When all instances of class *C* cannot be read into the memory, the modified *join* algorithms, i.e., the modified SMFJ and the modified SMRJ, can be used.

Simulation results that evaluate the four basic join algorithms (NLFJ, SMFJ, NLRJ and SMRJ) and the modified algorithms (the modified SMFJ and the modified SMRJ) are also provided. The simulated results conform with the cost analysis presented. That is, the join time costs of four basic join algorithms are linearly increased with the expansion in the number of the time-epochs (or the time-dimension, in the case of a regular TS). As

predicated the performance of NLRJ is worst, as it reads through the relation *CITY* many times. The simulation shows that as expected, sort-merge join algorithms are generally good when the relations are relatively small and *n* is small, but they are limited by memory capability. The modified sort-merge algorithms amend this limitation.

Simulation results that compare join algorithms of NLFJ, SMFJ, NLRJ and SMRJ both without *time-slice,* and with different *time-slice* intervals justify investigation into temporal processing and optimization. Utilising heuristics that make use of the ordering information of time varying data could lead to considerable cost savings.

It has been demonstrated that the decomposition strategy provides a convenient way of evaluating the performance of algorithms that take account of the time-dimension, and provides an opportunity for optimization that makes use of the ordering information of temporal data. It is also shown that the number of *epochs* is a significant token and plays an important role in the cost analysis.

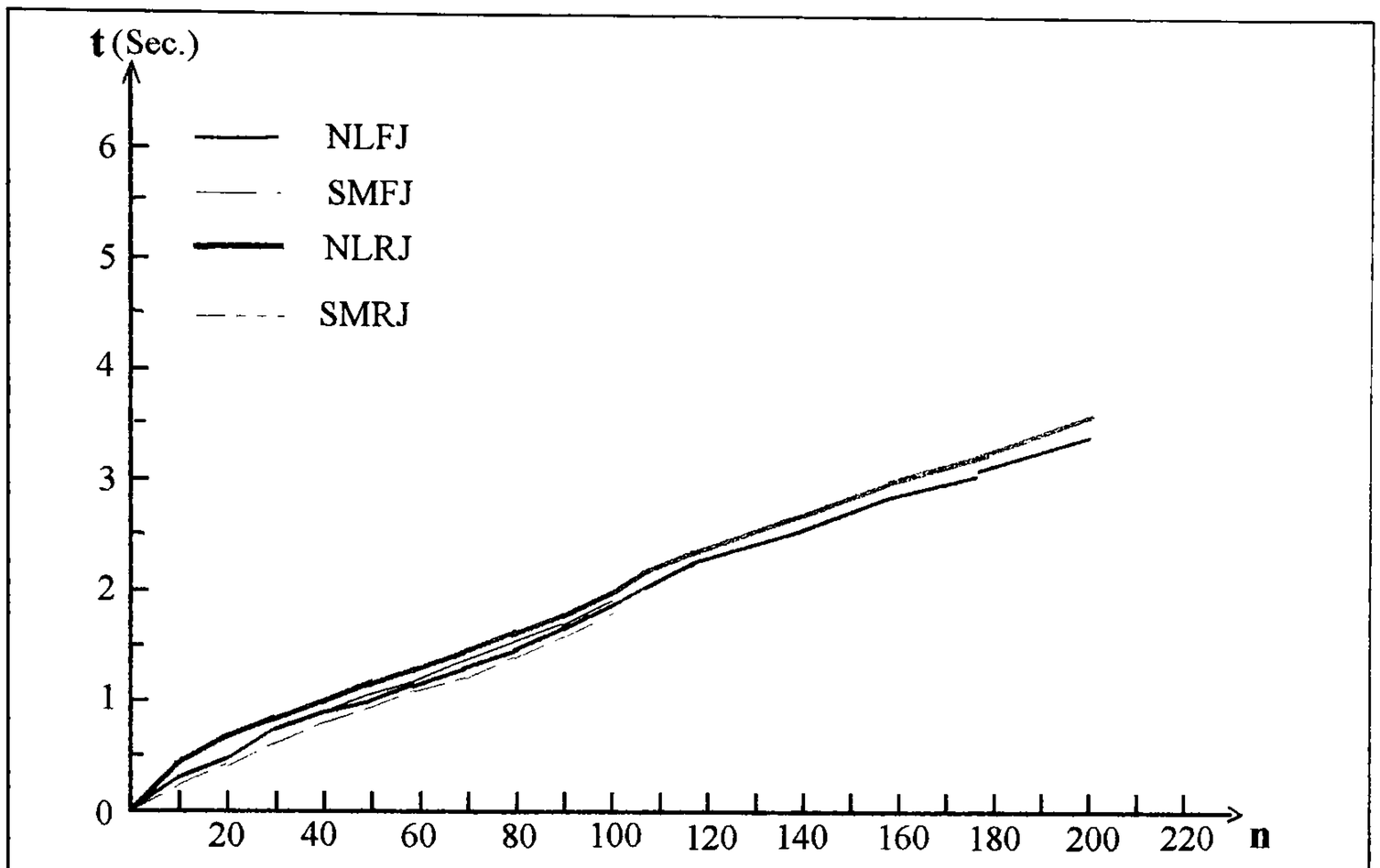


Figure 7.7 Join time cost with respect to n (sel=10%)

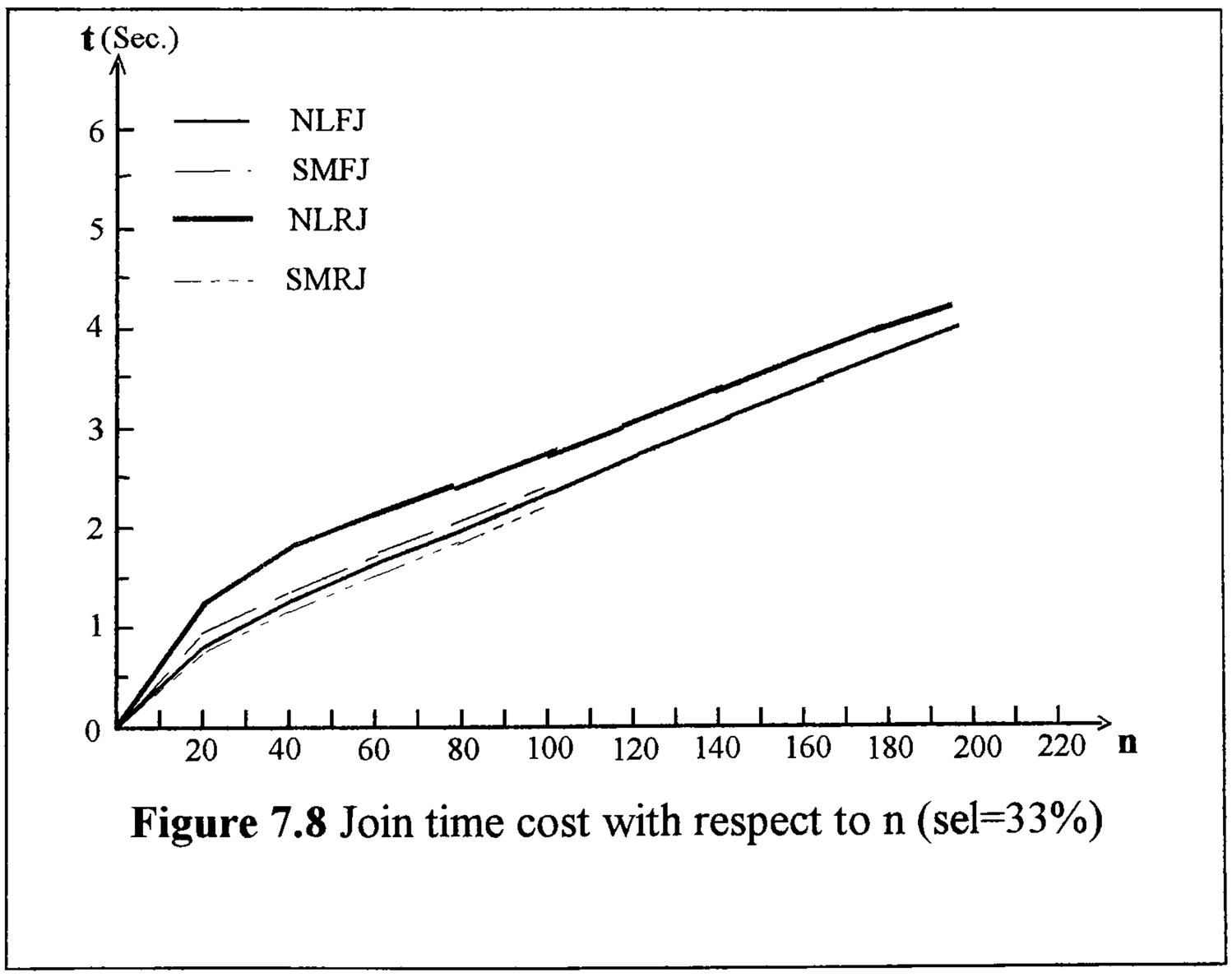


Figure 7.8 Join time cost with respect to n (sel=33%)

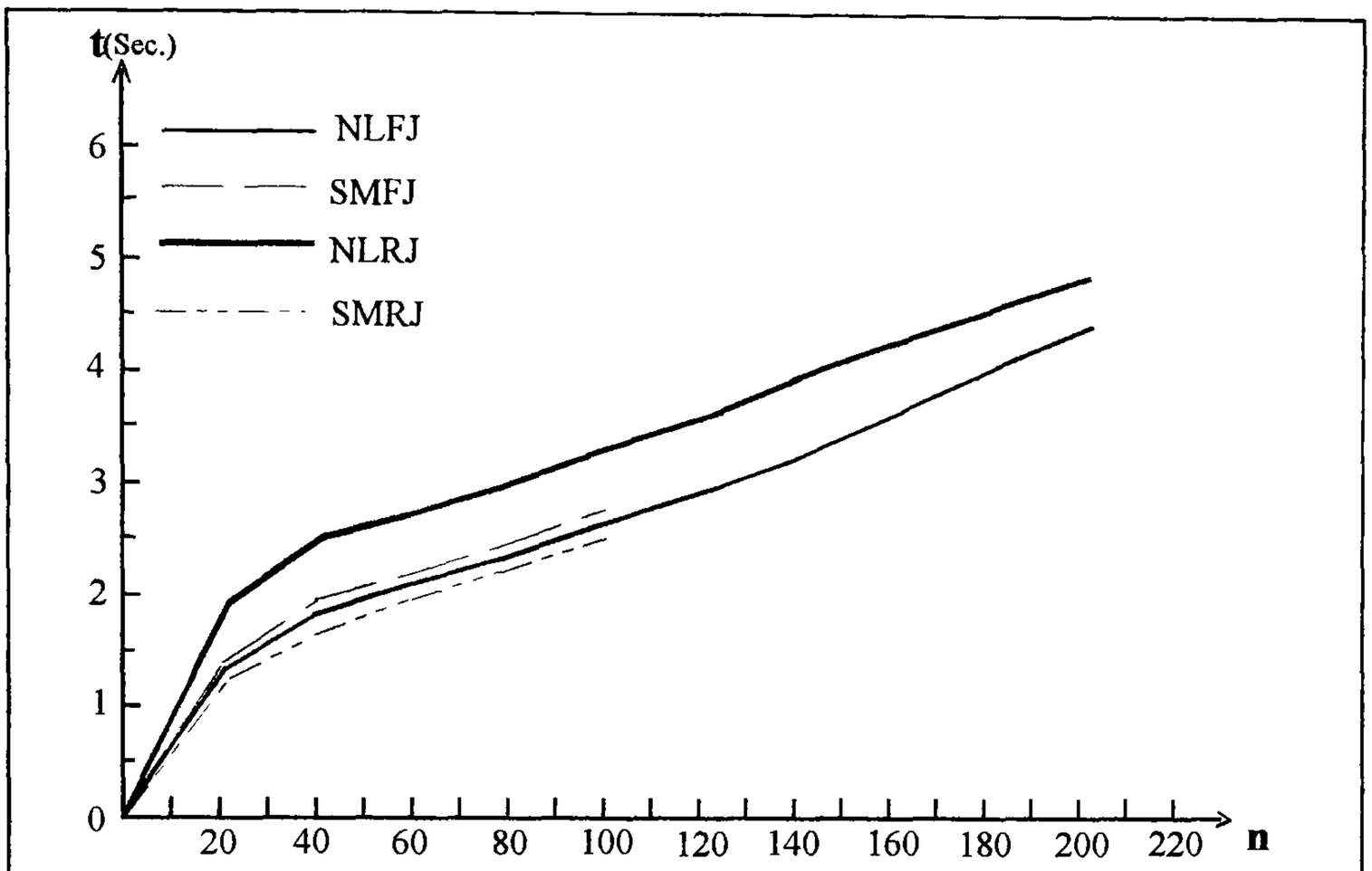


Figure 7.9 Join time cost with respect to n (sel=50%)

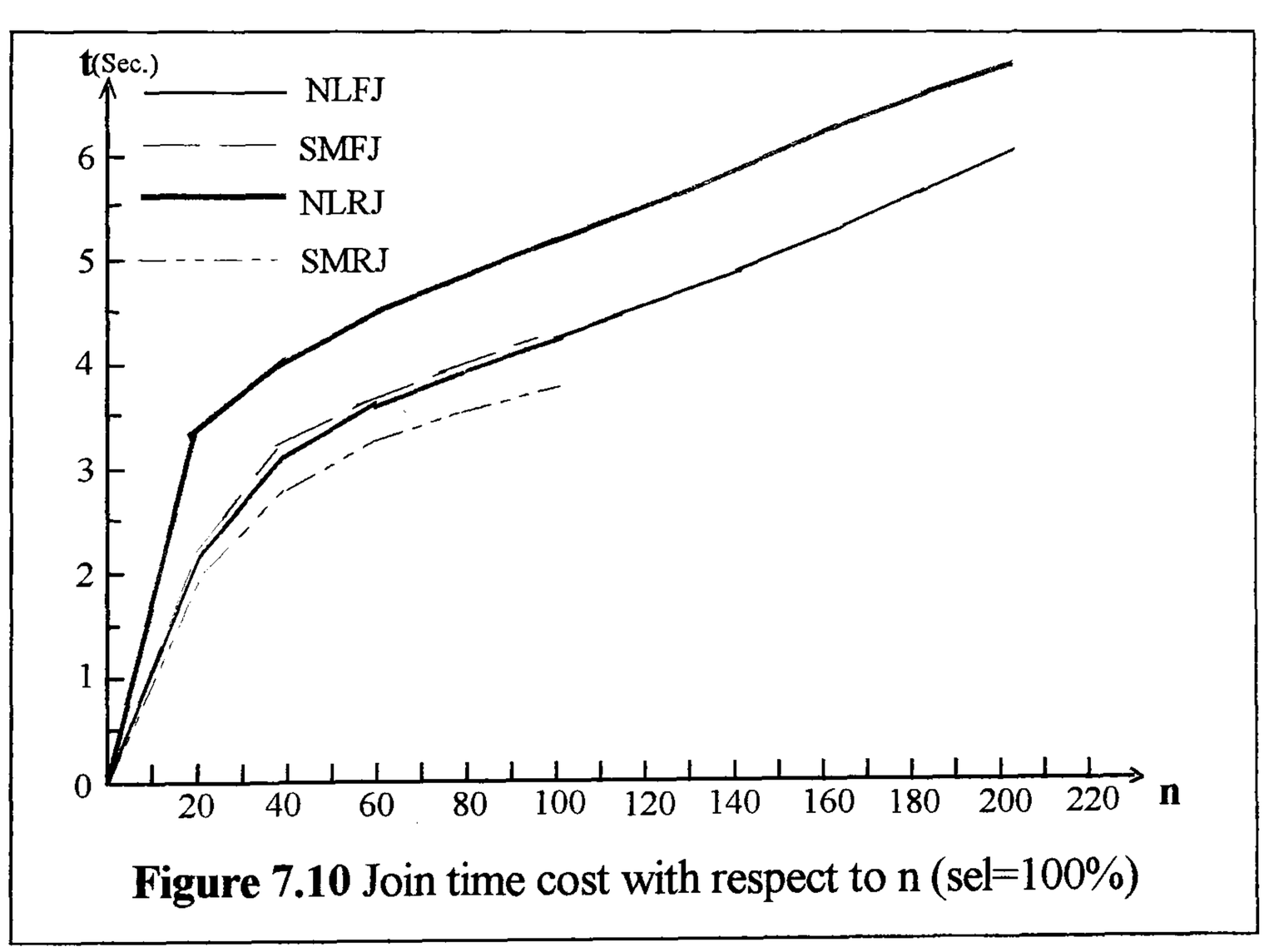


Figure 7.10 Join time cost with respect to n (sel=100%)

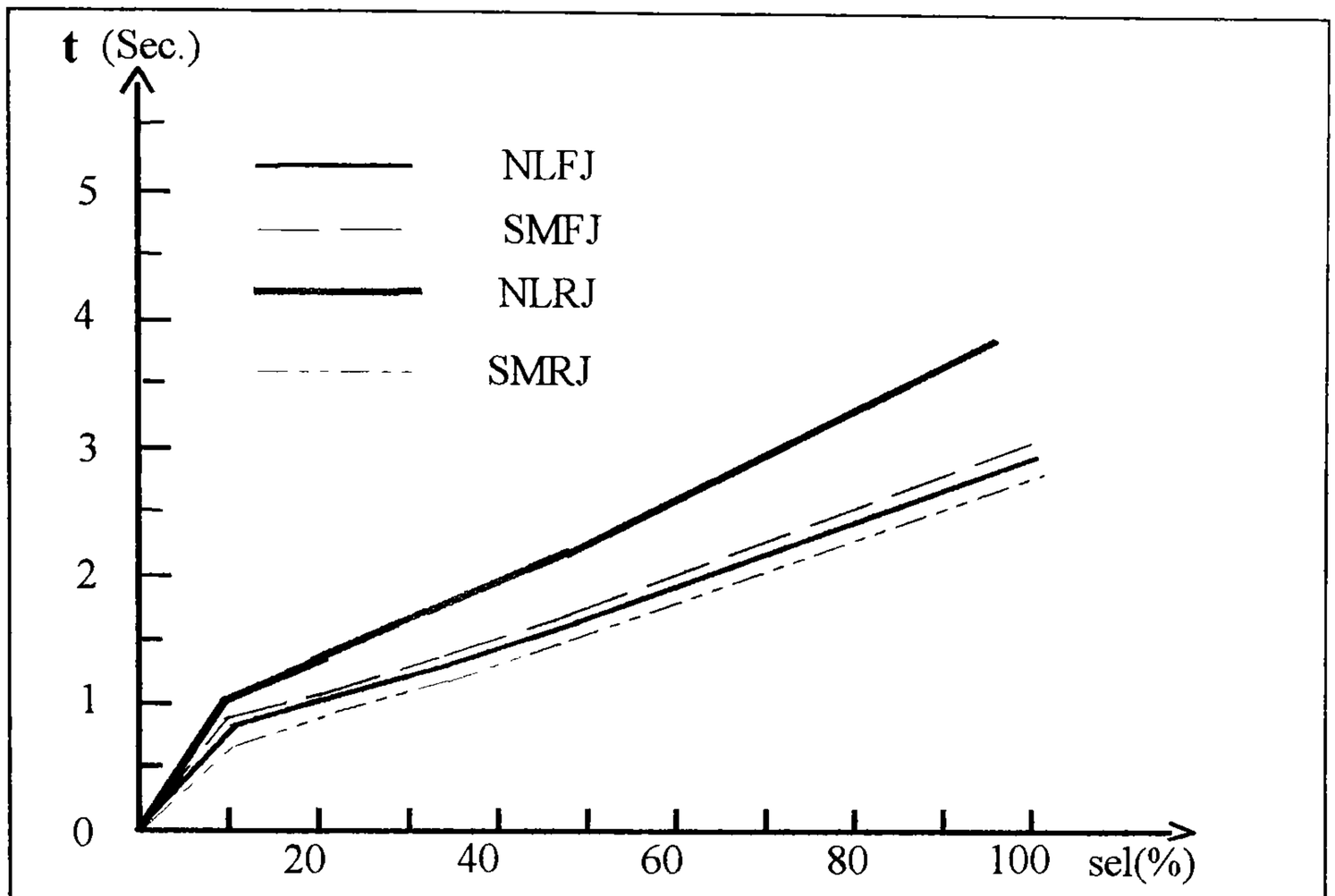


Figure 7.11 Join time cost with respect to sel% (n=40)

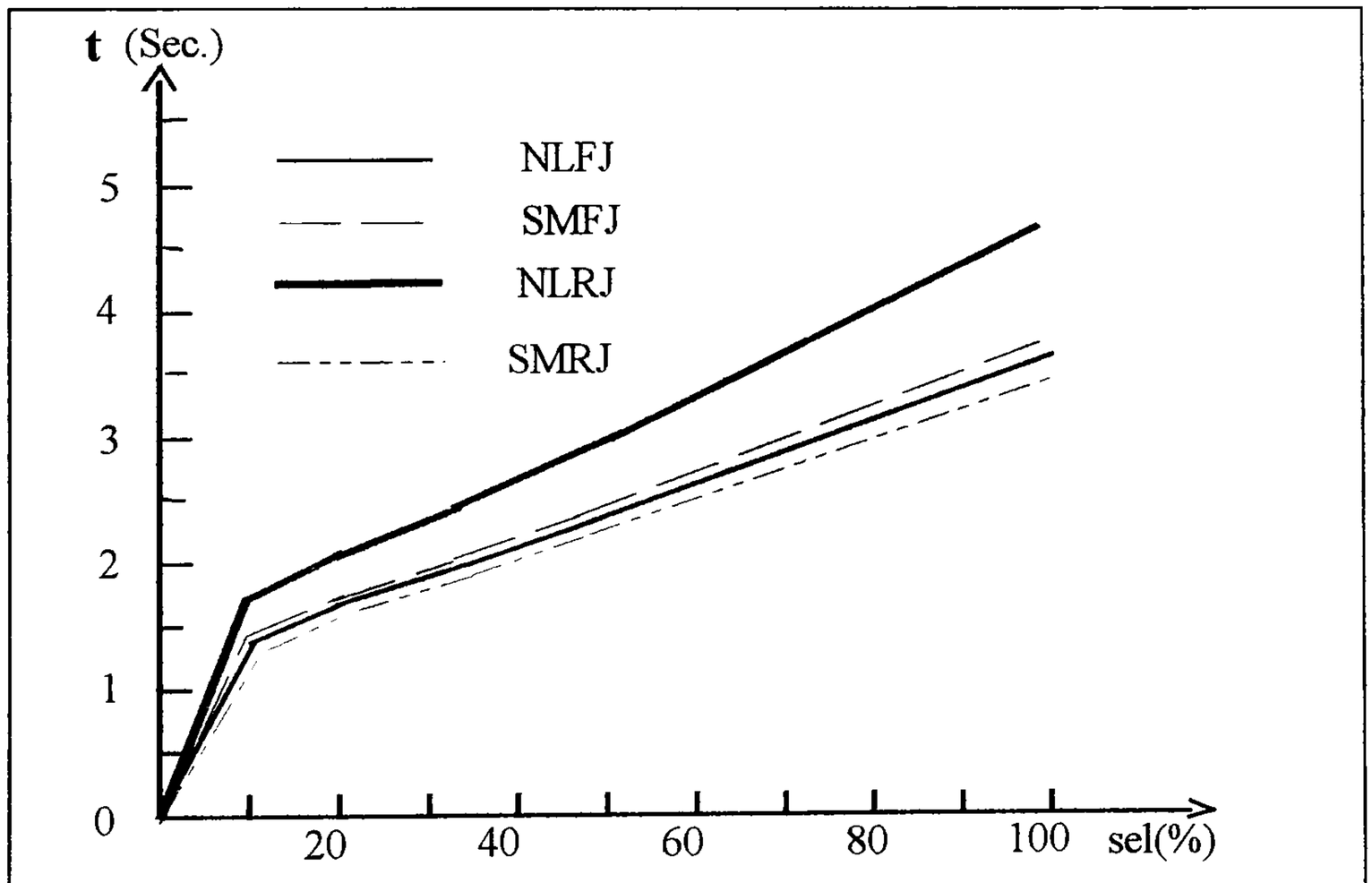


Figure 7.12 Join time cost with respect to sel% (n=80)

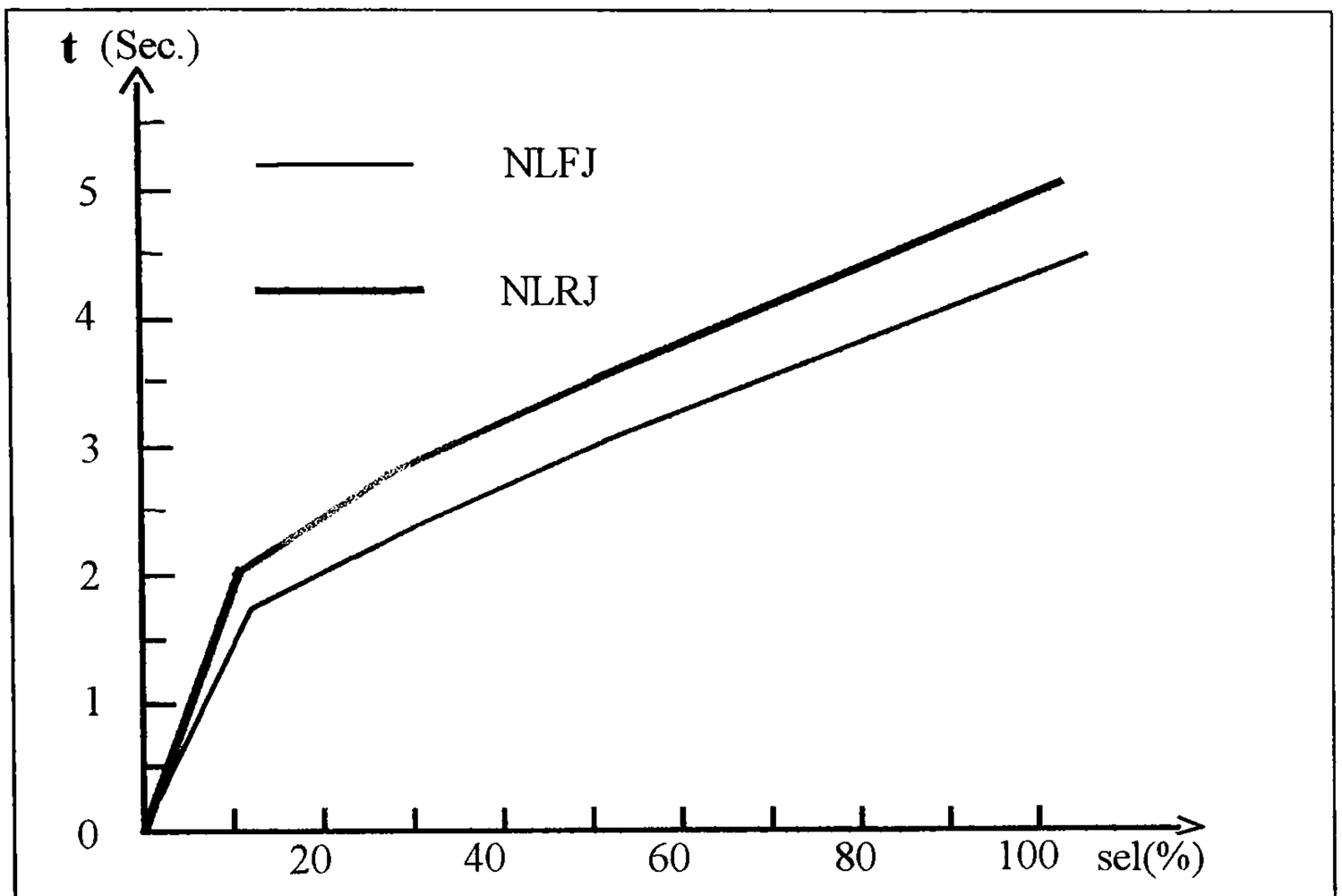


Figure 7.13 Join time cost with respect to sel% (n=100)

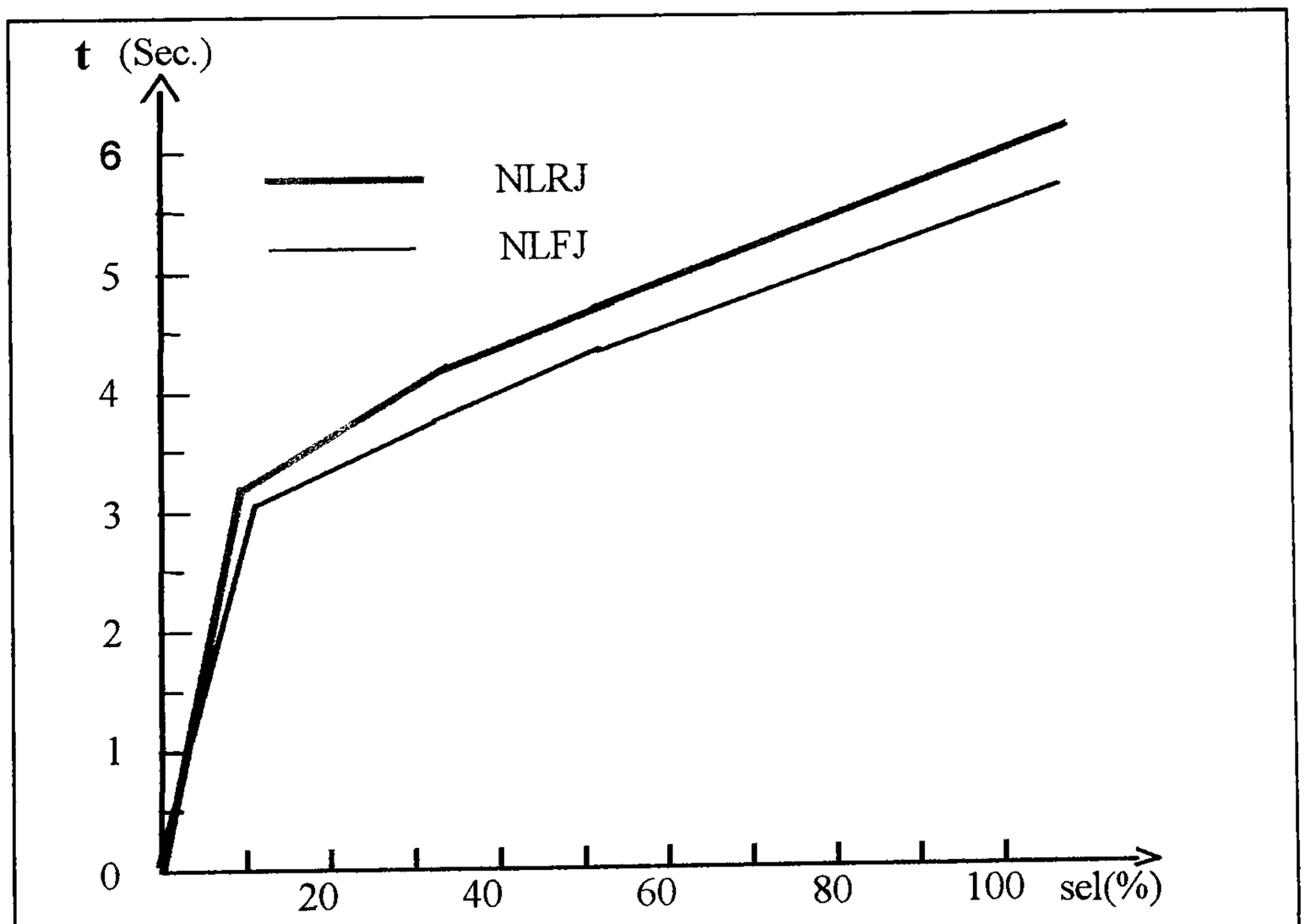


Figure 7.14 Join time cost with respect to sel% (n=180)

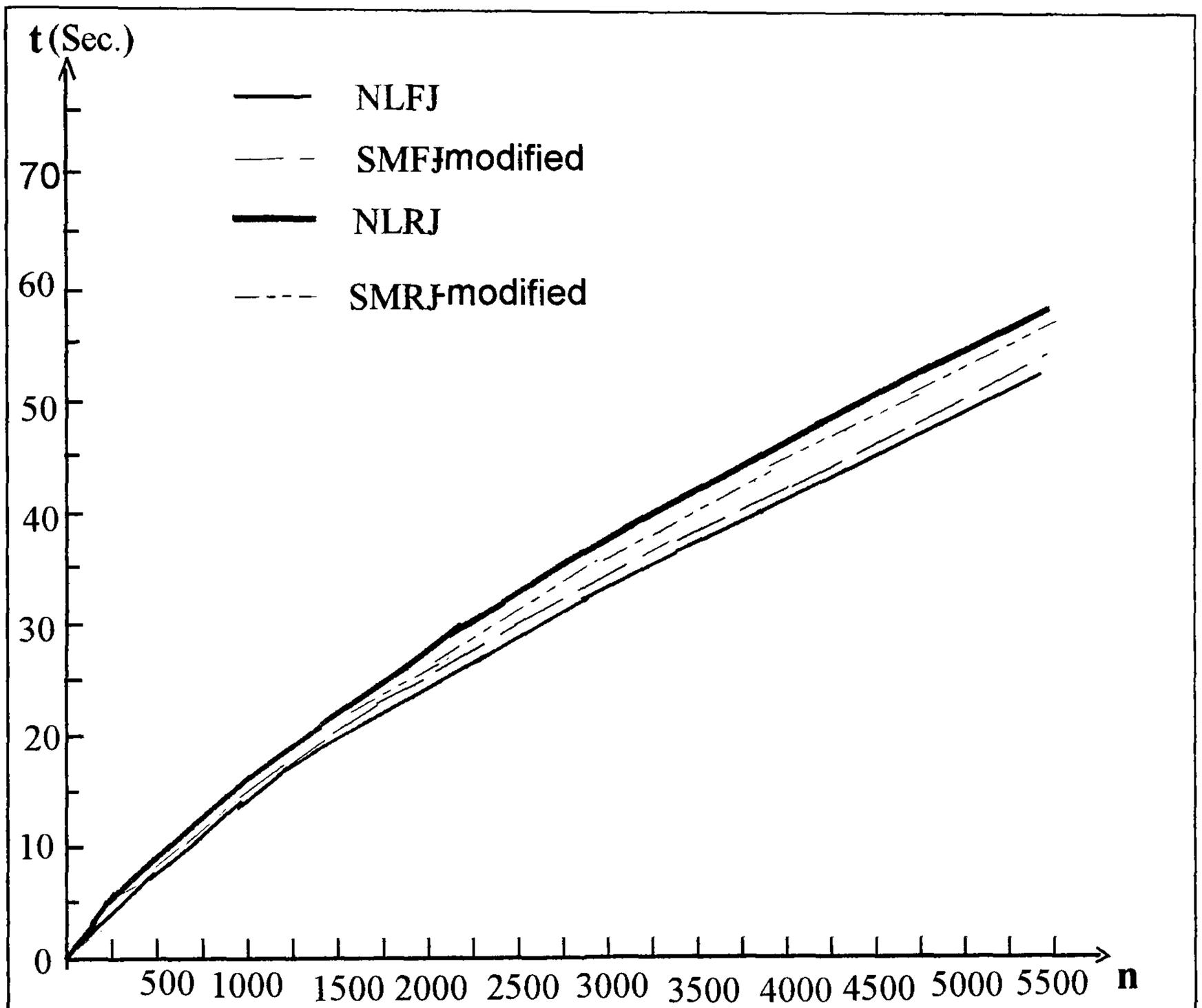


Figure 7.15 Join time cost with respect to n (sel=10%)

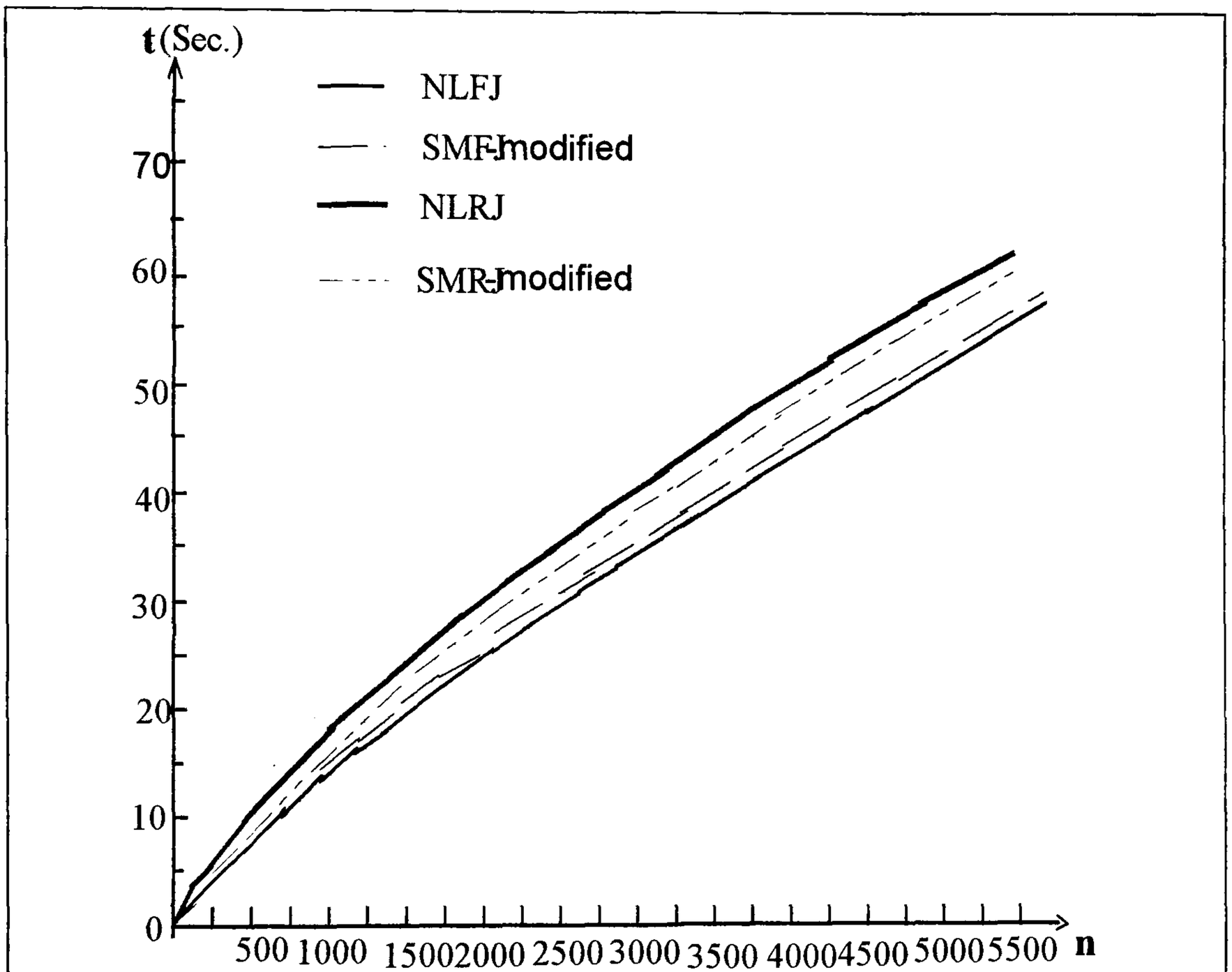


Figure 7.16 Join time cost with respect to n (sel=33%)

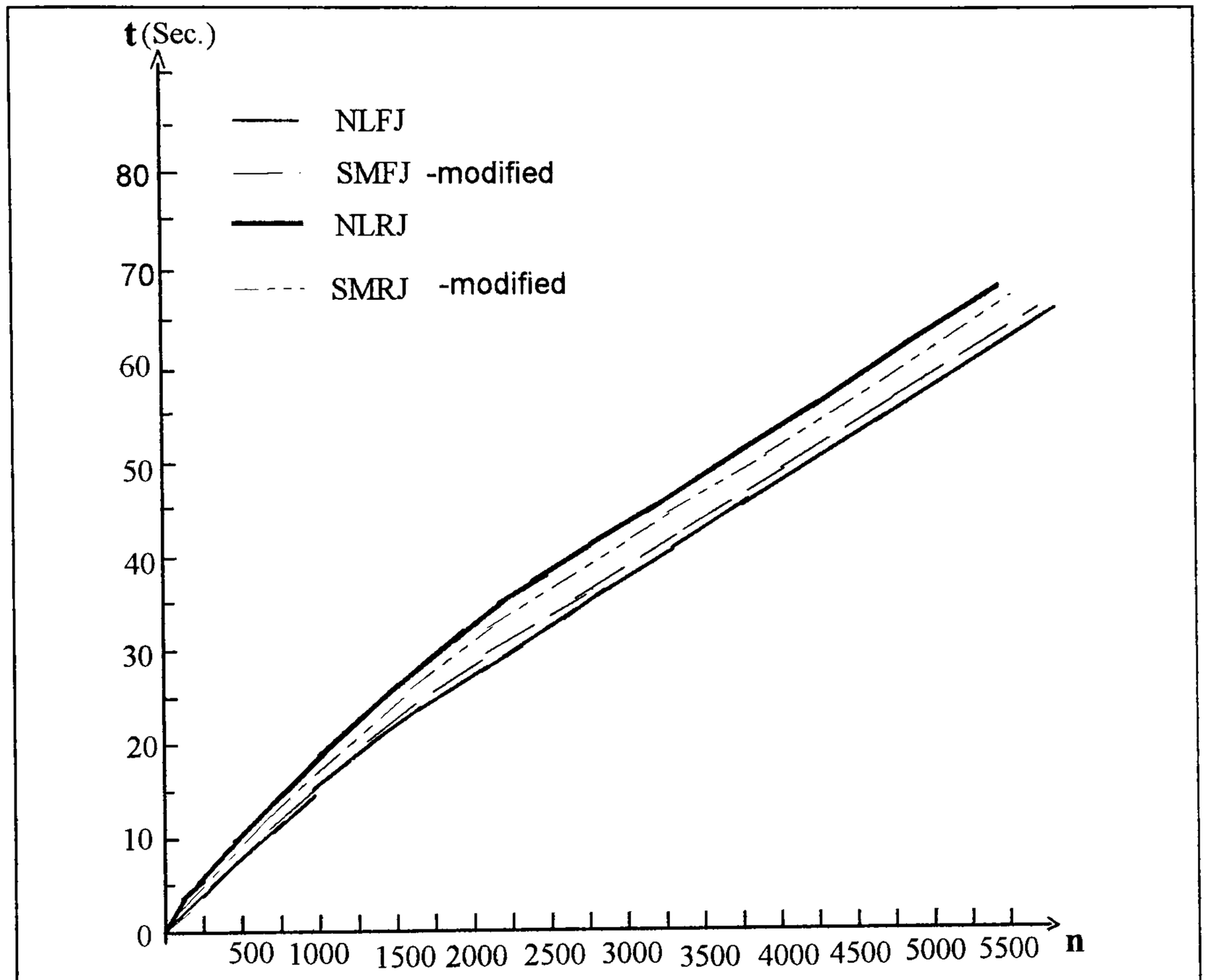


Figure 7.17 Join time cost with respect to n (sel=50%)

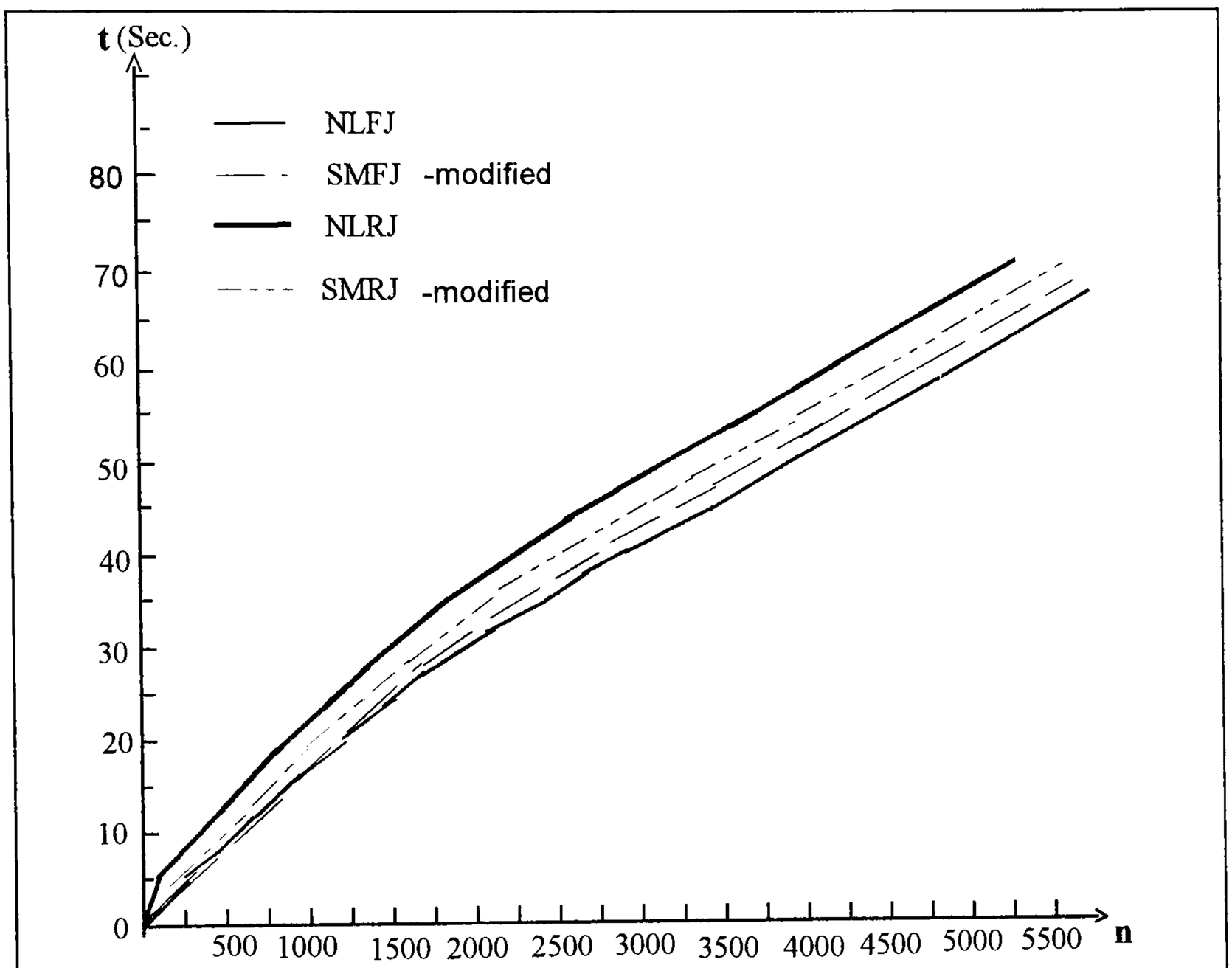


Figure 7.18 Join time cost with respect to n (sel=100%)

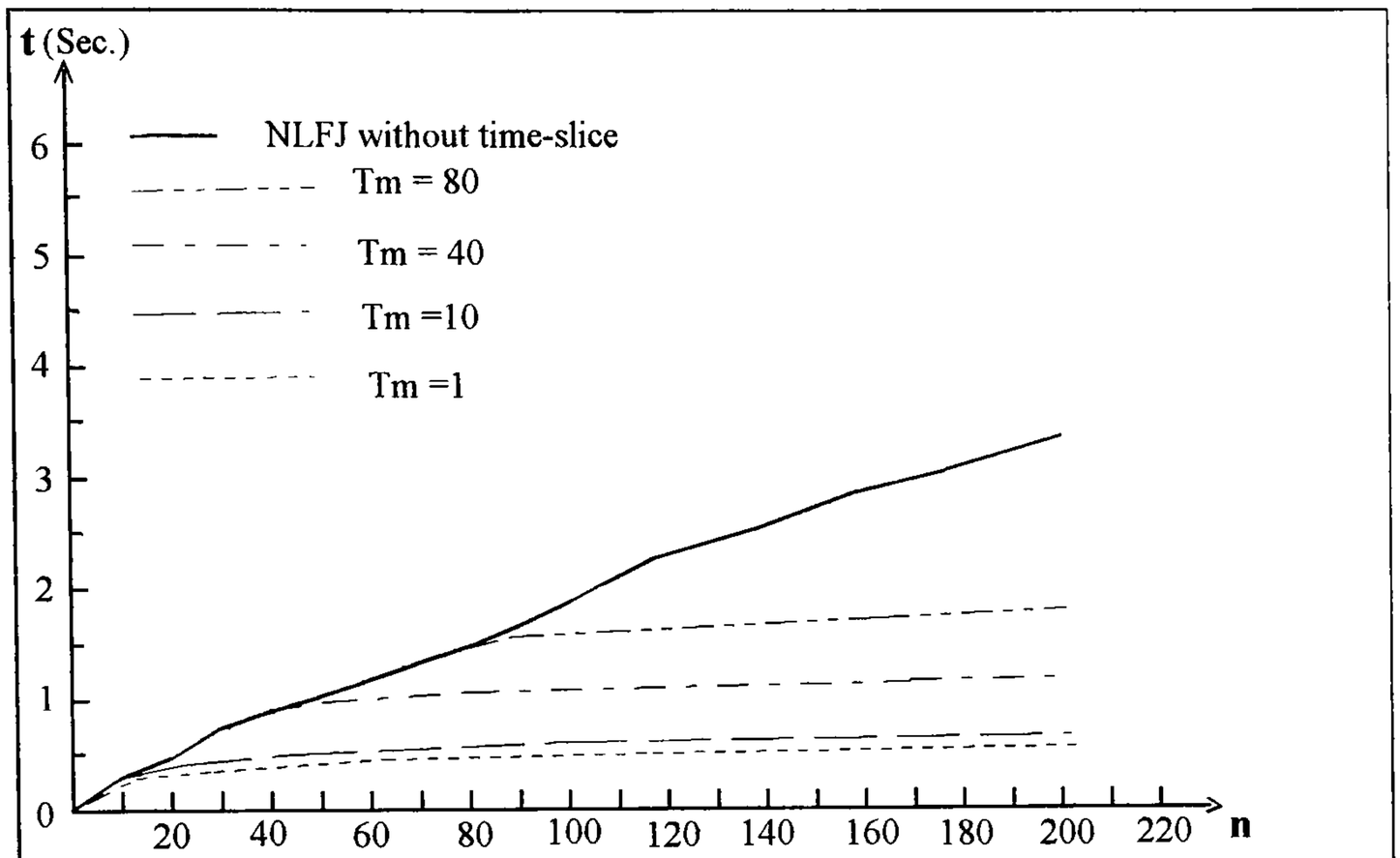


Figure 7.19 NLFJ time cost with respect to n (sel=10%)

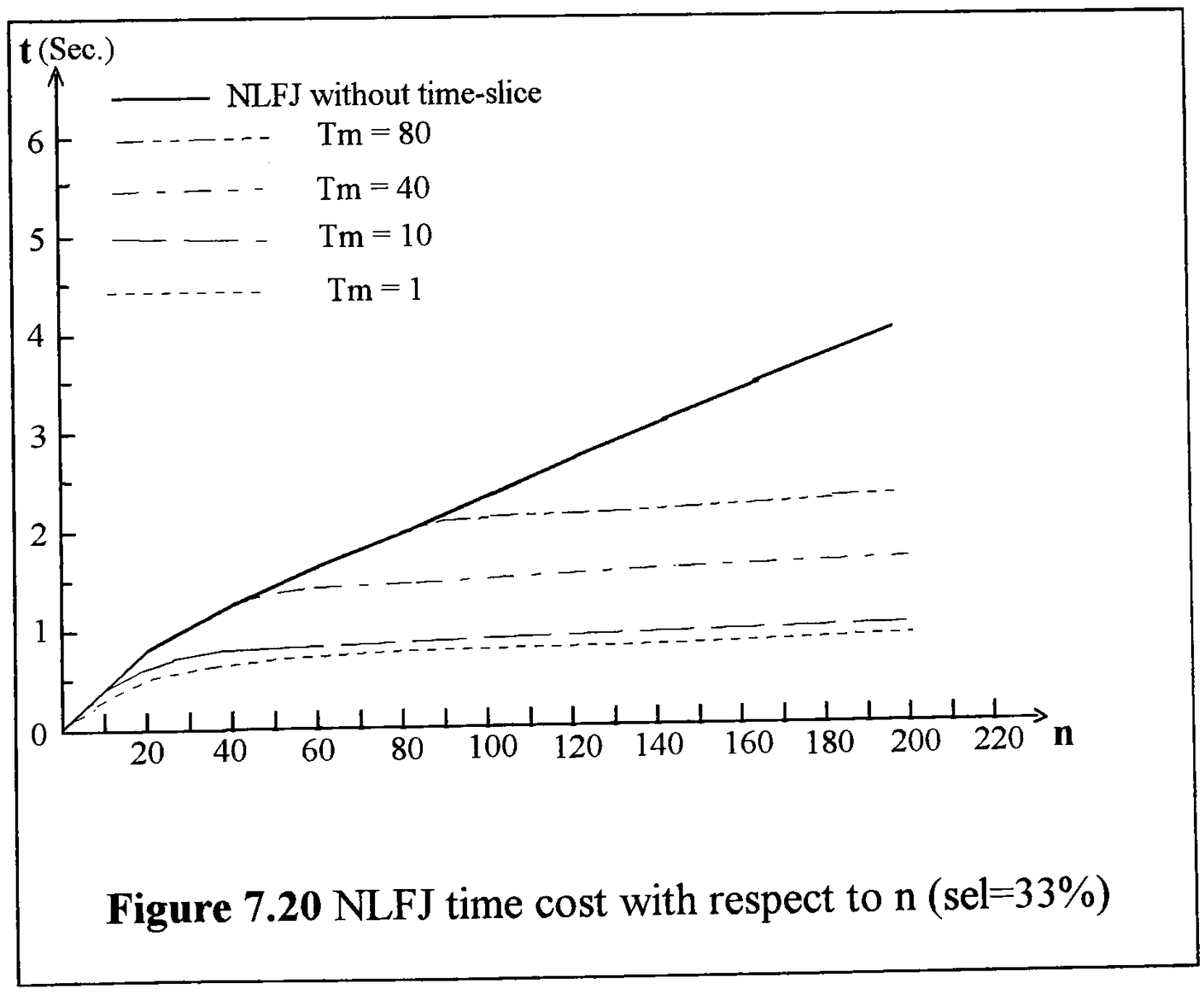


Figure 7.20 NLFJ time cost with respect to n (sel=33%)

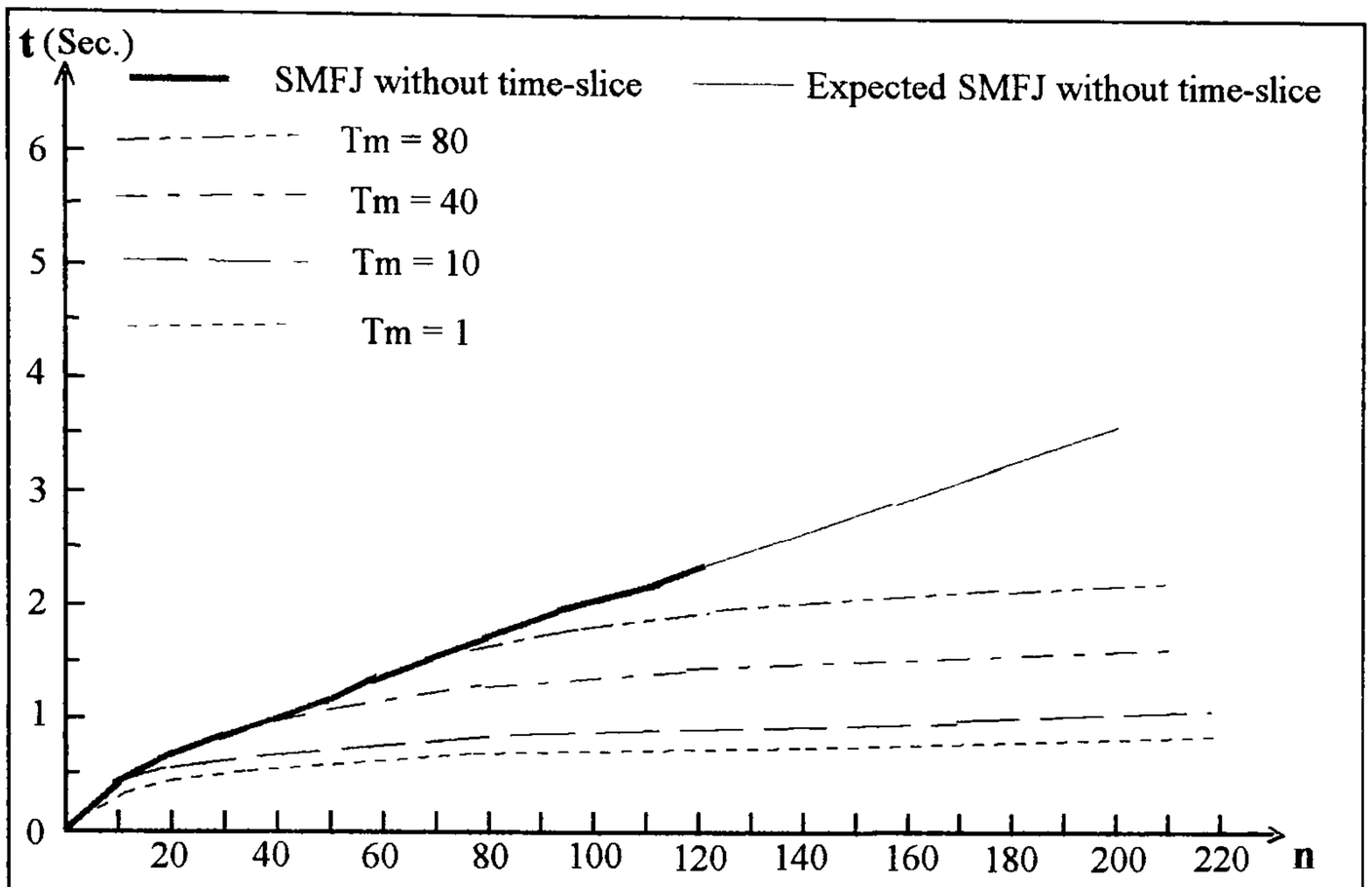


Figure 7.21 SMFJ time cost with respect to n (sel=10%)

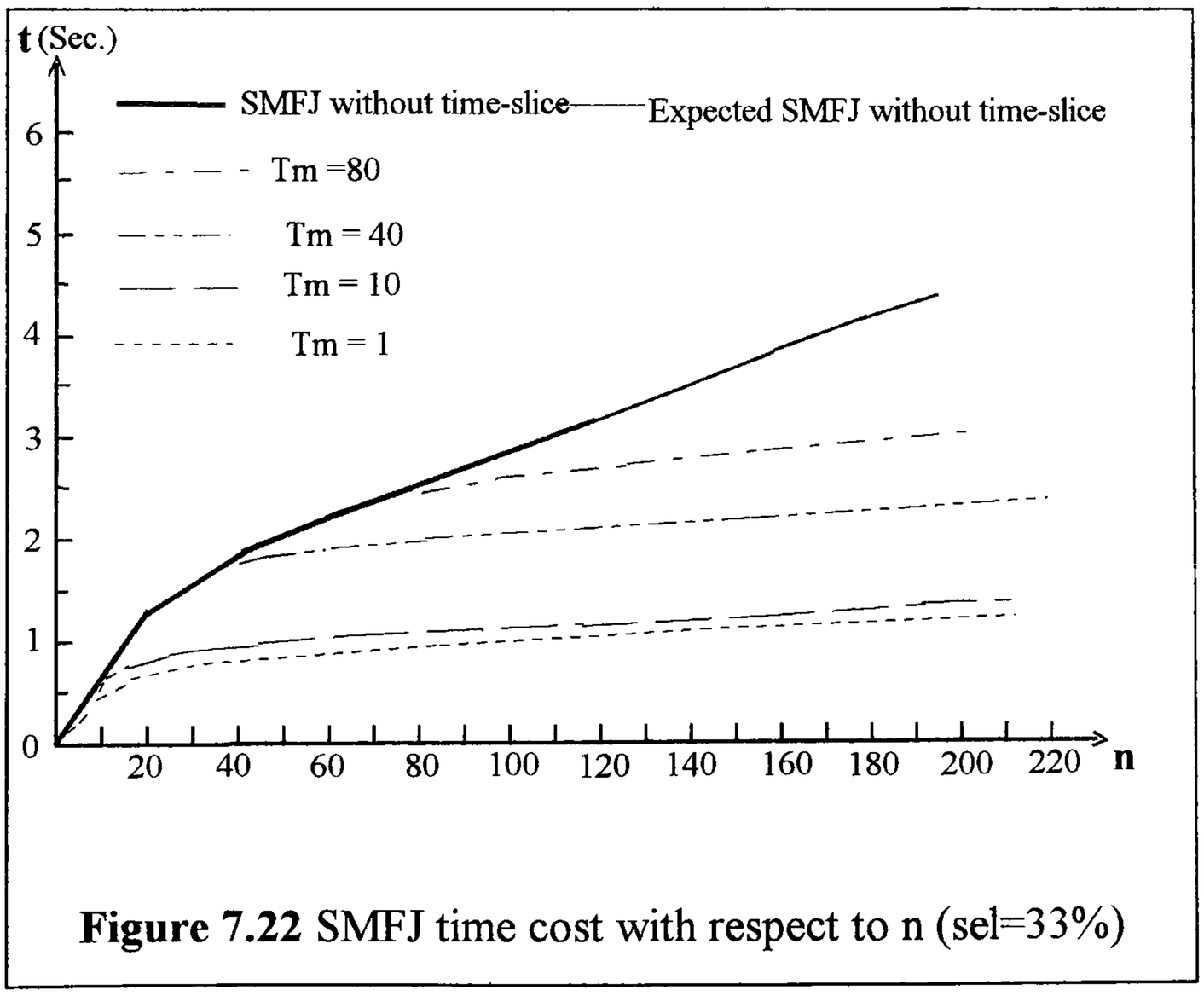


Figure 7.22 SMFJ time cost with respect to n (sel=33%)

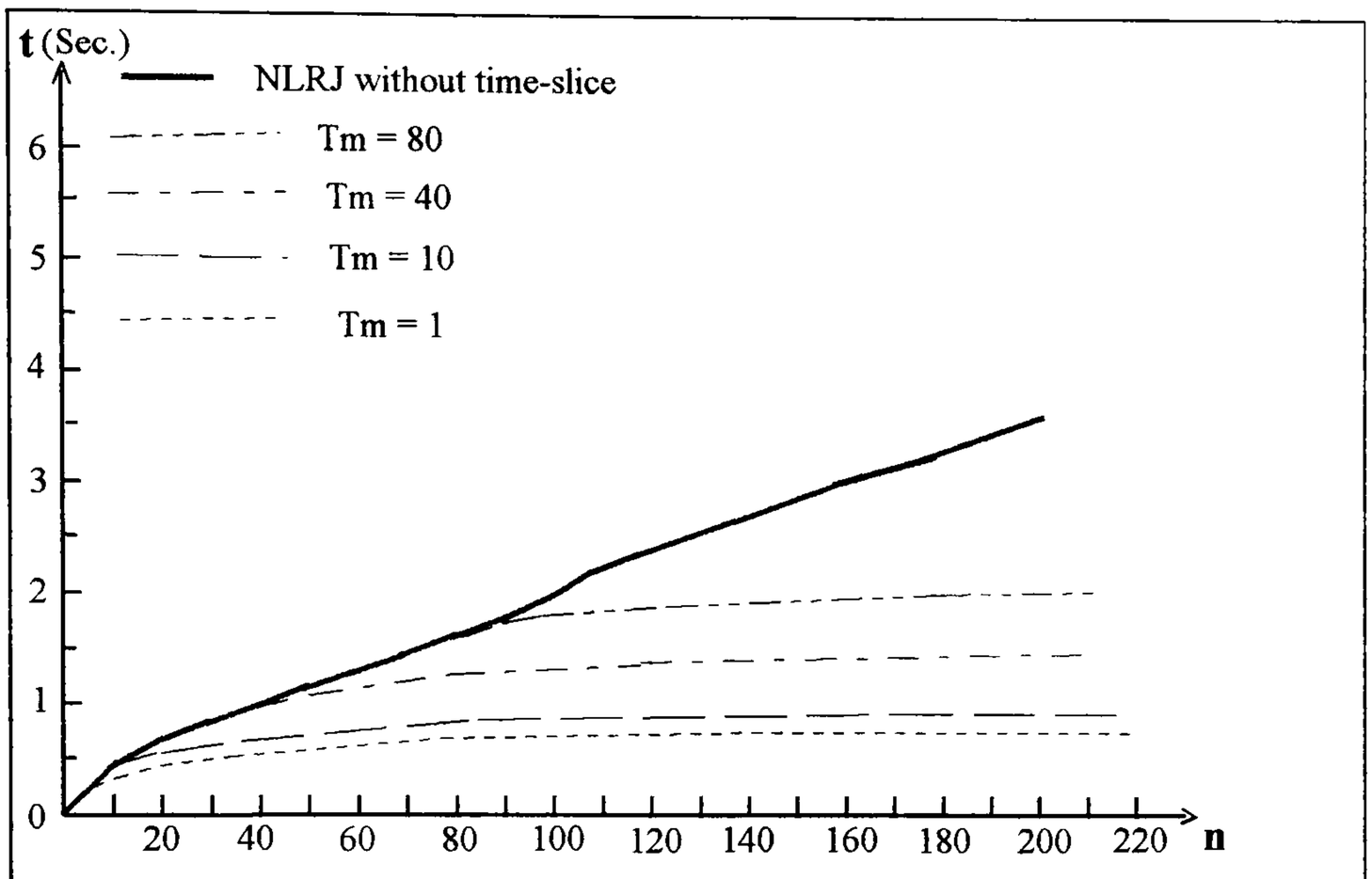


Figure 7.23 NLRJ time cost with respect to n (sel=10%)

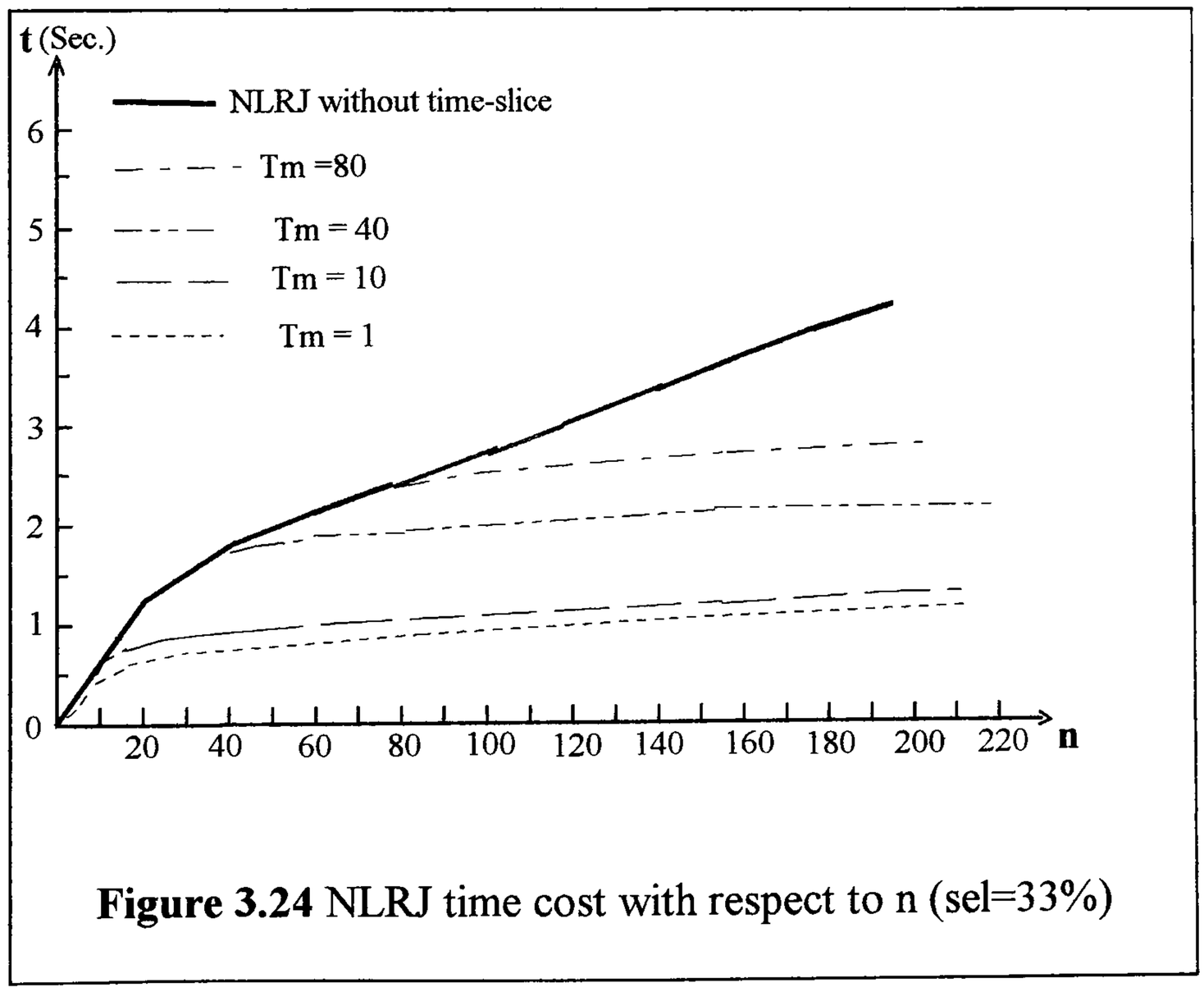


Figure 3.24 NLRJ time cost with respect to n (sel=33%)

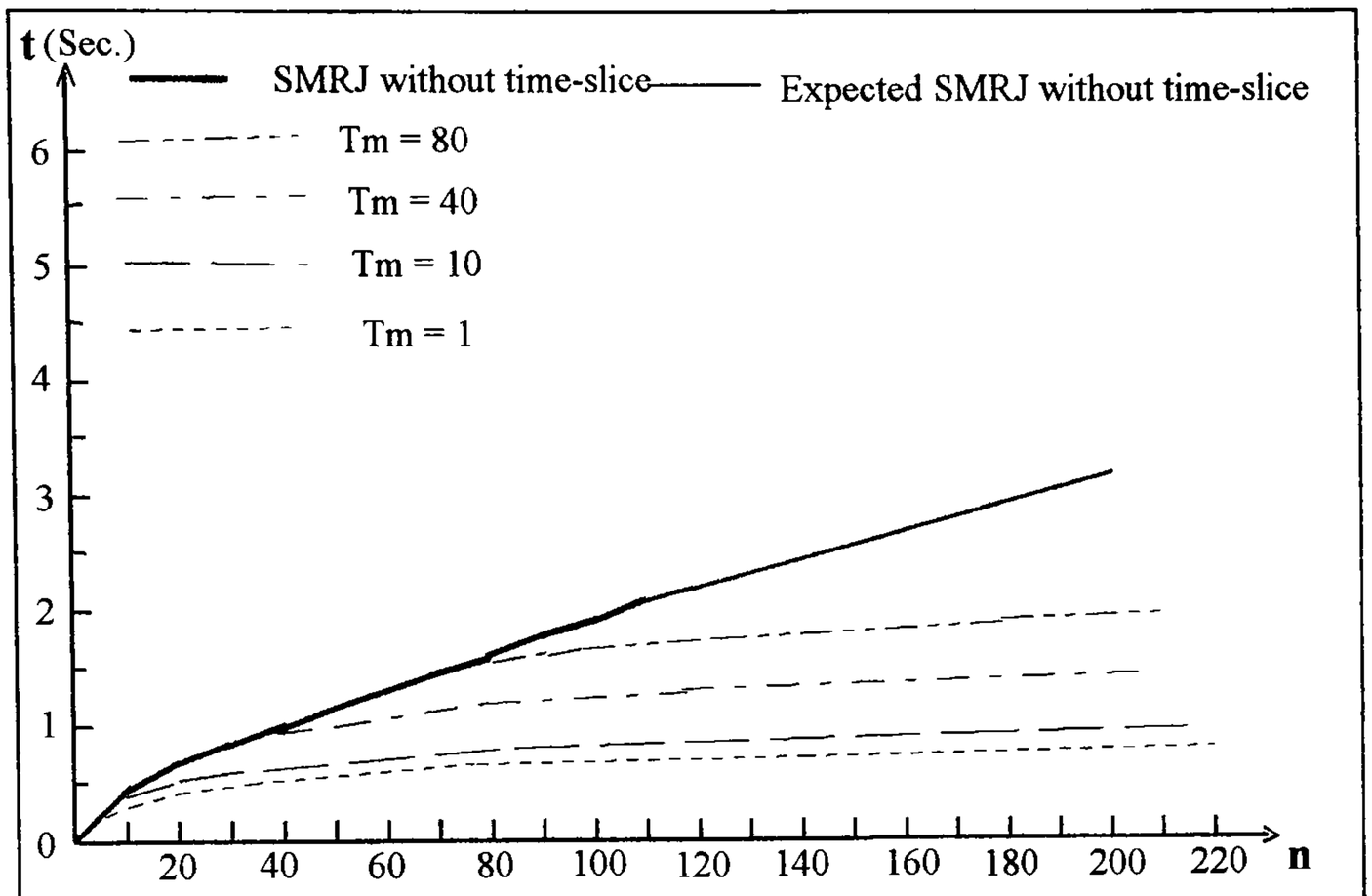


Figure 7.25 SMRJ time cost with respect to n (sel=10%)

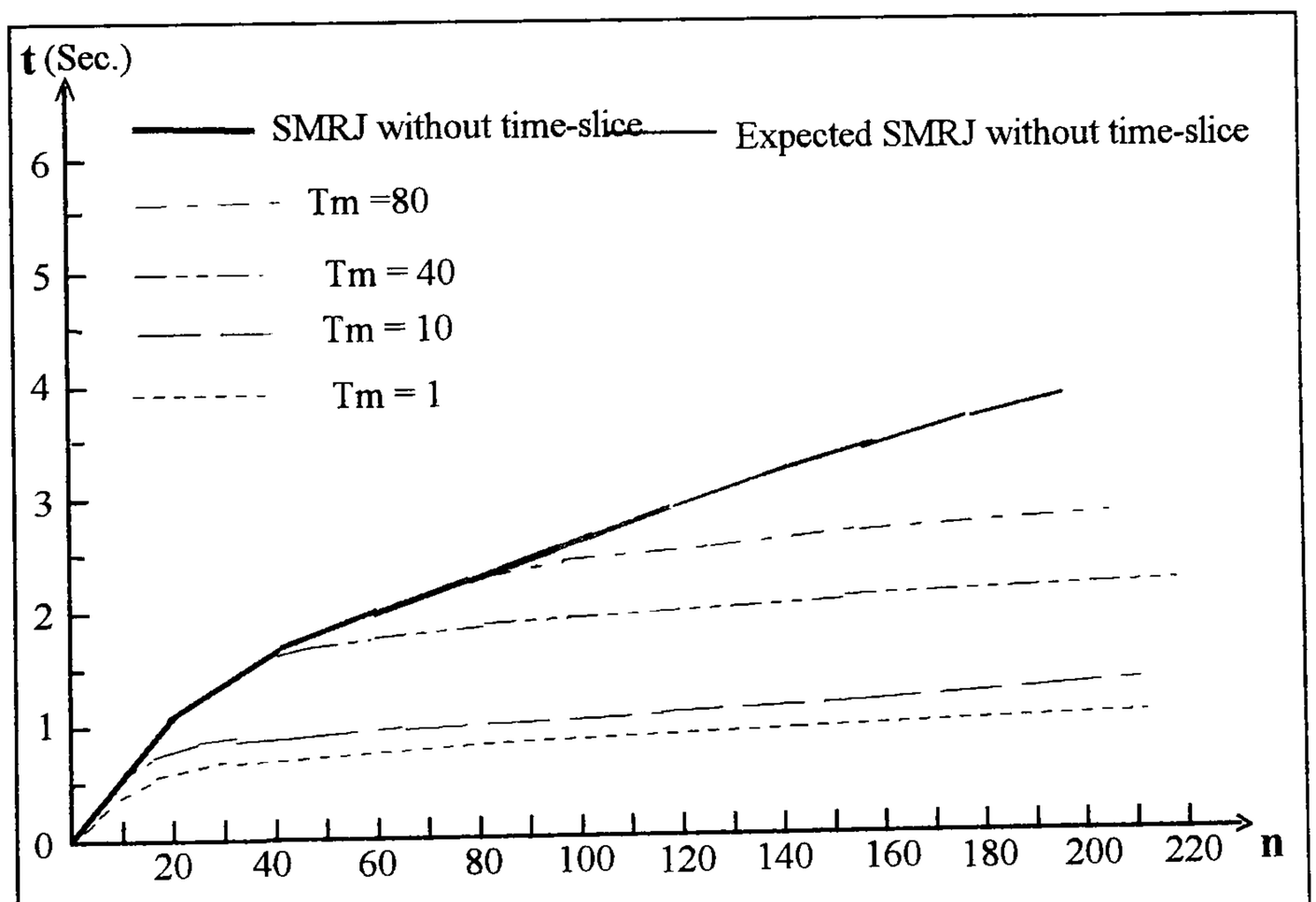


Figure 7.26 SMRJ time cost with respect to n (sel=33%)

Chapter 8

Query Processing with Incomplete Temporal Data

Previous chapters have presented an extensible approach to processing temporal object queries within the uniform query processing framework. In particular, a decomposition strategy is employed to process a query involving a path when time is present and algorithms have been presented to process the decomposed query components. This chapter will discuss the situation where user's queries require data that have not been explicitly recorded in the database. Temporal data allows substantial information to be exploited for query processing in such a situation.

8.1 Introduction

Our definition of the temporal data model describes a TOODB that is populated with temporal objects that are grouped into classes/relations and interrelated through associations of aggregation, generalisation and time-reference. A temporal object is represented as a time sequence, which can be either regular or irregular.

Time advances in one direction: the time domain is continuously expanding, and the most recent time point is the largest value in the domain. This implies that there is a lot of information associated with temporal data, which can be exploited during query processing. It has been shown from the previous chapter that exploiting the natural clustering or sort order will be beneficial to query optimization, and evaluations that use stream processing techniques and heuristics will reduce the scope of sequence scan.

Unlike query processing in a snapshot database where the user query always asks for data that have been recorded in the database, in temporal databases, a user query may require the value of an event at a specific time for which the database has no entry (no record) at that time point. In such a case, drawing implications from existing temporal data would help to answer user queries.

This chapter will first look at implications of temporal data and then discuss how to exploit these implications in query processing with incomplete temporal data. The rest of the chapter is organised as follows. Implications of temporal data will be briefly described in Section 8.2. Exploiting these implications to respond to a query requiring data that have not been explicitly recorded in the database is discussed under the title of interpolation in Section 8.3. Section 8.4 outlines a method for query processing in such a situation. Finally, Section 8.5 offers a summary.

8.2 Implications of Temporal Data

Time advances in one direction. This implies temporal completeness, temporal succession, temporal density, and temporal constraint [Roddick and Patrick, 1992], which can be exploited during query processing.

Temporal completeness

Temporal completeness can be compared with Robinson's definition of data completeness [1979]:

A thing playing a role of some relevance in the object system must be represented by a single thing playing a corresponding role in the model. That is, a thing in the model must have the same characteristics with respect to the model as the thing in the object system has with respect to that system.

The same applies to the temporal dimension. If an event of relevance occurs in the object system and the time at which it occurs is of interest, then it must be possible to model the

time of occurrence, and the temporal context pertaining to that event must be preserved regardless of future events. For example, if the absolute time of occurrence of an event is required then recording of absolute time should be supported.

According to Robinson:

Any thing in the model must be the sole representation of something playing an appropriate role in the object system. A corollary to this is that if there is something we do not want to say about the object system we should not say anything in the model. We should not have an elaborate way of saying something that is unimportant or absent in the object system.

Robinson's comments have a corollary within temporal data capture with the degree of description of temporal events.

Temporal succession

If a relevant sequence of events occurs in a modelled environment then the model should be able to record faithfully the sequence of the events.

The difference between this feature and that of temporal completeness is that completeness permits the model to fully describe the events in their temporal context, whereas temporal succession facilitates the accurate recording of the sequence of events or changes in the object system.

Temporal density

If the facts in a database model are, in some way, time-stamped, then what is the value of an attribute at a time where there is no specific entry? To be more specific, if a variable has a value at time t_1, then at time t_2 where $t_1 \neq t_2$, even if there is no information recorded for the variable at t_2, an assumption (or the application of one of a set of rules) should be possible to estimate the attribute's value. Obviously a useful modelling technique will have to allow for such assumption routines to cater for these cases, or at least state the assumption rules being used. These assumption rules generally fall into three categories:

- Step-wise constant

  A fact true (or a variable with a value) at time t_1 should be assumed to be true (or have the same value) at time t_2 where $t_1 < t_2$, unless it is explicitly recorded different at a time t_3 where $t_1 < t_3 < t_2$.

- Discrete events

  A fact true (or a variable with a value) at time t_1 should be assumed to be false (or have the value zero or null according to the context) at time t_2 where $t_1 \neq t_2$, unless it is explicitly recorded otherwise.

- Continuous change

  It should be possible for a variable with a value v_1 at time t_1 to be approximated at time t_2 even if its value v_1 at time t_1 is not explicitly recorded.

  For the time sequence that represents continuous change event in time, interpolation algorithms or temporal modelling techniques may be required to estimate the such values that will be discussed in the next section.

Handling of error terms and incomplete data

The handling of fuzzy data has been dealt with extensively in the literature in relation to static, non-temporal databases, however, handling of error terms in temporal databases requires special consideration. Where there is a measure of uncertainty about the accuracy of a piece of temporal data it may become difficult to provide accurate response to database queries. These uncertainties can be categorised as follows:

- Granularity errors-- the difference between time points is too coarse;
- Recording inaccuracies--the time data is erroneous or inaccurate;
- Relational uncertainties--the relationship between events and intervals is incompletely known.

As the time sequence is continuous in the time dimension, stochastic estimation approaches can be applied for curve fitting or smoothing, which can be exploited in interpolation algorithms.

Temporal constraints

The ability of a model to enforce data to adhere to integrity constrains is fundamental to the functionality of any database system, and has been covered well in the literature in relation to snapshot databases. Many constraints have a time element associated with them which is frequently ignored, even by those designing temporal modelling systems. Examples of temporal constraints are [Roddick and Patrick, 1992]:

- An event (or the beginning or end of a temporal set) that is constrained to fall at or within a given set of absolute time intervals.

- An event may be considered to occur within a given set of absolute time intervals relative to another event.

- An event may be considered to occur within a given set of relative constraints with respect to another event.

Temporal constraints can be exploited for interpolation and semantic optimization.

8.3 Interpolation

As discussed in the last section, temporal data implies substantial information that can be exploited for query processing and optimization. In this section, we will focus on how to make use of this information to respond to a user's query requiring data that is not explicitly recorded by introducing some techniques of interpolation.

Recall that a temporal object is represented as a time sequence (TS) in Chapter 4. For a continuous change event, it can be represented by either a regular TS or an irregular TS depending on the recording of the data. When it is represented by a regular TS, there is a corresponding value for every time point. However, if the granularity is too coarse, there would be a need for interpolation in order to answer a user's query that does not correspond to an explicit record. When it is represented by an irregular TS, interpolation algorithms or temporal modelling techniques may be required to decide the immediate value of an object between the values at various times recorded. (Note that for discrete and step-wise constants, the aforementioned assumptions would allow us to decide a value at a time point that has not been recorded. Even for discrete events, stochastic approaches can also be applied to estimate the missing data).

Therefore, interpolation is often required in processing temporal queries, because:

- granularity is too coarse in the case of both irregular and regular TS;
- missing values (null) in the case of both irregular and regular TS; and
- unequal spacing in the case of irregular TS.

The implications of temporal completeness, temporal succession and temporal density assure the use of interpolating.

The algorithms of interpolation themselves are outside of the scope of query processing. But employing interpolation is useful within a temporal database when a user's query requires data not explicitly recorded. To clarify the use of interpolation, this section briefly introduces some interpolation techniques, especially in situations where the data are not linear. Problems directly related to query processing would then be made clear.

Given values of an unknown function of time corresponding to certain values of time t, to answer the question "What is the function" is always impossible with a limited amount of data. Determining the behaviour of a function $f(t)$, as evidenced by the sample of data pairs $[t, f(t)]$ by approximation is the task of interpolation (or extrapolation) [Gerald and Wheatley, 1994]. The approach to be used is to fit a polynomial curve to the points. This approach is appropriate to many applications.

The strategy discussed here used in approximating unknown values of a function is straightforward. We will find a polynomial that fits a selected set of points $(t_i, f(t_i))$ and assume that the polynomial and function behave nearly the same over the interval in question. Values of the polynomial then should be reasonable estimates of the values of the unknown function. When the polynomial is of the first degree, this leads to linear interpolation. If the polynomial is of degree higher than the first, it can approximate a function that is non-linear.

However, there are problems with interpolating polynomials when the data are not "smooth" (i.e., that there are local irregularities). In such cases, a polynomial of high degree would be required to follow the irregularities, but it can be found that such polynomials, while fitting irregularities, deviate widely at other regions where the function is smooth. One solution is to fit subregions of data with different polynomials, but this method too is problematic in that the joins of the different polynomials are not continuous in their slope. To remedy this problem, special types of polynomials, called splines, are useful [Gerald and Wheatley, 1994].

The study of splines leads to some other special forms of polynomials (Bezier curves and B-spline curves) that do not interpolate (they do not pass exactly through all of the points) but they are useful for sketching smooth curves.

We do not always want to find a polynomial that fits exactly to the data. Often the values we wish to fit are not exact, or they may come from a set of experimental measurements that are subject to error. Fitting exactly a polynomial in this instance would also fit the errors in the data and this is undesirable.

A technique called least squares is normally used in such cases [Gerald and Wheatley, 1994]. Based on statistical theory, this method finds a polynomial (or some other kind of approximating function) that is more likely to approximate the true values.

To illustrate the idea of interpolation, the following gives two examples of popular methods for interpolation from [Gerald and Wheatley, 1994], i.e., Lagrangian polynomials and least squares approximation.

1) **Lagrangian polynomials**

Assume that the given data are exact and represent values of unknown function. If we want to find a polynomial that passes through the same points as our unknown function, we need to set up a system of equations involving the coefficients of the polynomial.

For example, suppose we want to fit a cubic to these data:

T	*f(t)*
3.2	22.0
2.7	17.8
1.0	14.2
4.8	38.3
5.6	51.7

First, we need to select four points to determine the polynomial (The maximum degree of the polynomials is always one less than the number of points). Suppose we choose the first four points. If the cubic is $at^3 + bt^2 + ct + d$, we can write four equations involving the unknown coefficients *a, b, c* and *d*:

when t=3.2: $a(3.2)^3 + b(3.2)^2 + c(3.2) + d$

if t=2.7: $a(2.7)^3 + b(2.7)^2 + c(2.7) + d$

if t=1.0: $a(1.0)^3 + b(1.0)^2 + c(1.0) + d$

if t=4.8: $a(4.8)^3 + b(4.8)^2 + c(4.8) + d$

The set of equations gives:

a=-0.5275; b=6.4952; c=-16.1177; d=24.3499

and the polynomial is

$$f(t)=-0.5275\,t^3+6.4952\,t^2-16.1177\,t+24.3499 \qquad (8\text{-}1)$$

We can then estimate the values of the function at some value of t, say t=3.0, by substituting 3.0 for t in the polynomial. At t=3.0 the estimated value is 20.21.

This procedure is awkward and this technique leads to an ill-conditioned system of equations [Gerald and Wheatley, 1994].

The Lagrangian polynomial is perhaps the simplest way to obtain a polynomial for interpolation with unevenly spaced data (i.e., irregular TS). Data where t-values are not equally spaced often occurs as the result of experimental observations or where historic data are examined.

Suppose we have a table of data with four pairs of t- and $f(t)$-values, with t_i indexed by variable t:

T	$f(t)$
T_0	f_0
T_1	f_1
T_2	f_2
T_3	f_3

Through these four pairs we can pass a cubic. The Lagrangian form for this is

$$P3(t)=\frac{(t-t_1)(t-t_2)(t-t_3)}{(t_0-t_1)(t_0-t_2)(t_0-t_3)}f_0+\frac{(t-t_0)(t-t_2)(t-t_3)}{(t_1-t_0)(t_1-t_2)(t_1-t_3)}f_1 + \frac{(t-t_0)(t-t_1)(t-t_3)}{(t_2-t_0)(t_2-t_1)(t_2-t_3)}f_2+\frac{(t-t_0)(t-t_1)(t-t_2)}{(t_3-t_0)(t_3-t_1)(t_3-t_2)}f_3 \qquad (8\text{-}2)$$

We can write the algorithm for interpolation with a Lagrangian polynomial of degree *m* in pseudo-code of C++:

To interpolate for *f(t)*, given *t* and a set of *m+1* data pairs, (t_i, f_i), i=0,1,...*m*. It is not assumed the uniform spacing between the *t* values, nor does it is needed that values arranged in a particular order. The *t*-values must all be distinct, however.

```
sum=0;  // set sum=0
for (int i=0; i=m; i++);
real P=1;
for (int j=0; j=m; j++);
if (j≠i) P=P*(t-t(j) )/(t(i)-t(j));
//end of j
sum=sum+p*f(i);
} //end of i
//sum is the interpolated value f(t).
```

The trouble with the standard Lagrangian polynomial technique is that we do not know which degree of polynomial to use. If the degree is to low, the interpolating polynomial does not give good estimates of *f(t)*. If the degree it too high, undesirable oscillations in polynomial values can occur. **Nevill's method** can overcome this difficulty [Gerald and Wheatley, 1994]. It essentially computes the interpolated value with polynomials of successively higher degree, stopping when the successive values are close together. The successive approximations are actually computed by linear interpolation from the intermediate values.

There are two disadvantages in using the Lagrangian polynomial method for interpolation. First, it involves more arithmetic operations than does the divided-difference method [Gerald and Wheatley, 1994]. Second, and more important, if we want to add or subtract a point from the set used to construct the polynomial, we essentially have to start over the computations. Both the Lagrangian polynomial and Nevill's method also must repeat all of the calculations if we must interpolate at a new

t-value. **The divided-difference methods** avoid all of this computation [Gerald and Wheatley, 1994]. The problem of interpolating from tabulated data is considerably simplified if the values of the function are given at **evenly spaced intervals** of the independent variable. It is necessary to arrange the data with *t*-values in ascending order.

2) Least-squares approximation

When the values we wish to fit are not exact, or they may come from a set of experimental measurements that are subject to error, a technique called least squares is normally used in such cases. Based on statistical theory, this method finds a polynomial that is more likely to approximate the true values.

We use m as the degree of the polynomial and N as the number of data pairs. Obviously if $N=m+1$, the polynomial passes exactly through each point and the methods in previous sub-section apply. Here, we consider the situation $N>m+1$.

We assume the functional relationship

$$y=a_0+a_1 t+ a_2 t^2+...+a_m t^m \qquad (8\text{-}3)$$

with errors defined by

$$e_i=Y_i-y_i=Y_i- a_0-a_1 t_i- a_2 t_i^2-....-a_m t_i^m \qquad (8\text{-}4)$$

where Y_i represents the observed or experimental value corresponding to t_i, $i=1,2,\ldots, N$.

We minimise the sum of squares

$$S=\sum_{i=1}^{N} e_i{}^2=\sum_{i=1}^{N}(Y_i - a_0 - a_1 t_i^1 - a_2 t_i^2 - \ldots - a_m t_i^m)^2 \qquad (8\text{-}5)$$

At the minimum, all the partial derivatives vanish, giving $m+1$ equations:

$$\frac{\partial S}{\partial a_0}=0=\sum_{i=1}^{N} 2(Y_i - a_0 - a_1 t_i^1 - a_2 t_i^2 - \ldots - a_m t_i^m)(-1),$$

$$\frac{\partial S}{\partial a_1} = 0 = \sum_{i=1}^{N} 2(Y_i - a_0 - a_1 t_i^{\,1} - a_2 t_i^{\,2} - \ldots - a_m t_i^{\,m})(-t_i), \qquad (8\text{-}6)$$

...

$$\frac{\partial S}{\partial a_m} = 0 = \sum_{i=1}^{N} 2(Y_i - a_0 - a_1 t_i^{\,1} - a_2 t_i^{\,2} - \ldots - a_m t_i^{\,m})(-t_i^{\,m}).$$

Dividing each by -2 and rearranging the $m+1$ normal equations to be solved simultaneously:

$$a_0 N + a_1 \sum t_i + a_2 \sum t_i^{\,2} + \ldots + a_m \sum t_i^{\,m} = \sum Y_i$$

$$a_0 \sum t_i + a_1 \sum t_i^{\,2} + a_2 \sum t_i^{\,3} + \ldots + a_m \sum t_i^{\,m+1} = \sum t_i Y_i$$

$$a_0 \sum t_i^{\,2} + a_1 \sum t_i^{\,3} + a_2 \sum t_i^{\,4} + \ldots + a_m \sum t_i^{\,m+2} = \sum t_i^{\,2} Y_i \qquad (8\text{-}7)$$

.....

$$a_0 \sum t_i^{\,m} + a_1 \sum t_i^{\,m+1} + a_2 \sum t_i^{\,m+2} + \ldots + a_m \sum t_i^{\,2m} = \sum t_i^{\,m} Y_i$$

where the summations run from 1 to N.

Putting these equations in matrix form shows an interesting pattern in the coefficient matrix:

$$\begin{bmatrix} N & \sum t_i & \ldots & \sum t_i^{\,m} \\ \sum t_i & \sum t_i^{\,2} & \ldots & \sum t_i^{\,m+1} \\ \ldots & \ldots & \ldots & \ldots \\ \sum t_i^{\,m} & \sum t_i^{\,m+1} & \ldots & \sum t_i^{\,2m} \end{bmatrix} \mathrm{a} = \begin{bmatrix} \sum Y_i \\ \sum t_i Y_i \\ \ldots \\ \sum t_i^{\,m} Y_i \end{bmatrix} \qquad (8\text{-}8)$$

where $a=[a_0, a_1, a_2,..., a_m]$. The matrix of above equation is called the normal matrix for the least-squares problem. There is another matrix that corresponds to this, called the design matrix. It is of the form:

$$A = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ t_1 & t_2 & \ldots & t_N \\ \ldots & \ldots & \ldots & \ldots \\ t_1^{\,m} & t_2^{\,m} & \ldots & t_N^{\,m} \end{bmatrix}$$

It is easy to show that AA^T is just the coefficient matrix of equation (8-8). We can rewrite the above equation in matrix form as:

$$AA^T a = Ba = AY \tag{8-9}$$

where $B= AA^T$ and $Y=[Y_1, Y_2,..., Y_N]$.

There are various methods to solve equation (8-9). The following method avoids ill-conditioning [Gerald and Wheatley, 1994].

The matrix $B=AA^T$ is symmetric and positive semidefinite [Gerald and Wheatley, 1994]. In linear algebra, it is shown that B can be diagonalized by an orthogonal matrix P:

$$PBP^T=PAA^TP^T=D$$

where the diagonal elements of D are the eigenvalues of B. Note that orthogonality implies that $PP^T=I$, the identity matrix.

Since B is positive semidefinite, all of its eigenvalues are nonnegative. This means that we can define a matrix R as

$$R = \sqrt{D}, \quad \text{or} \quad R^2 = D$$

The diagonal elements of R are called the singular value of A. The equation and its solution can be rewritten as:

$$AA^Ta=P^TDPa=PR(PR)^Ta=AY \tag{8-10}$$

$$a=PD^{-1}P^TAY \tag{8-11}$$

This eliminates having to multiply out AA^T and by extending this approach, leads to an important methods for solving the equation (8-8) called singular-value decomposition [Gerald and Wheatley, 1994].

8.4 Query Processing with Incomplete Temporal Data

From the discussion in last section, it is clear that the main problem concerned with query optimization, when interpolation is used, is to efficiently retrieve *N* epochs of data required for interpolation, in order to produce an efficient interpolation algorithm.

Any smooth function can, under very general conditions, be represented locally by a polynomial, to any desired degree of accuracy [Kendall and Ord, 1990]. The implications of temporal completeness, temporal succession, temporal density and temporal constraints imply that a continuous time event is, at least locally, a smooth function. It can then be required that the *N* epochs (*N* records) of data are close or around the estimated time point *t*. Suppose that the temporal object has *n* records (i.e., the total number of epochs is *n*) that are naturally clustered together in time ascending or descending order. We can summaries the method for query processing that corresponds to returning an object value at time *t* below:

1) According to the application, determine the degree *m* of the polynomial to be applied. For example, for a temporal time event, which changes dramatically, we can choose a higher degree. Otherwise we choose a lower degree.

2) Determine the method for interpolation. In case of the stochastic method, for example, the least-squares approximation, *N* is determined, otherwise $N=m+1$.

3) Retrieve *N* records of the temporal object from the database.

a) $N=n$

Retrieve the whole historical records (*n* records) of the object from the database.

b) $N<n$

Perform a binary search or sequential search (as discussed in Chapter 7) to determine the epoch number *i* whose time value is t_i and its successive epoch number $i+1$ whose time value is t_{i+1}, and there exists $t_i<t<t_{i+1}$.

If $i < N/2 < n/2$, retrieve the first N records of the object from the database.

If $i > n-N/2$, retrieve the last N records of the object from the database.

If $N/2 < i < n-N/2$, retrieve $N/2$ records before and including i and $N/2$ records after and including $i+1$, i.e., retrieve N records of the object whose time epoch numbered as:

$$i-N/2, \ldots, i, i+1, \ldots, i+N/2$$

Note that for simplicity, we retrieve $N/2$ data before the estimated time t and $N/2$ data after the estimated time t. In some (extremely) uneven time space situations, unevenly retrieval of data can be applied.

4) Execute the chosen interpolation algorithm.

5) Return the object value at time t.

For some discrete time events with data missing at some time points, we can apply the above method to estimate the value at those time points. If missing data is not the case, we can simply return null for an unrecorded time point.

For the step-wise constant situation, perform a binary search or sequential search to determine the epoch number i whose time value is t_i and its successive epoch number $i+1$ whose time value is t_{i+1}, where $t_i < t < t_{i+1}$. Return the object value at point i.

Note that if a user query requires data whose time point is outside the object's lifespan, i.e., at a future time point, we have the problem of predication. Predication is required by many temporal database applications such as sales forecasting, weather forecasting, etc. The purpose of maintaining such a temporal database is not only to record the history of time-varying events, but also to predicate the trend or future change of the events. Predication algorithms are often based on a regular TS. In the case of irregular

TS, the interpolation is applied first to construct a regular TS. Once a TS is modelled by some function, the predication can then be easily carried out based on that function. The subject of predication is outside of the scope of query processing.

8.5 Summary

This chapter has been dealing with query processing with incomplete temporal data. When a user query requests the data that is not explicitly recorded in the database, the techniques of interpolation or assumption rules that make use of the implication of the time would correspond to the user's query. The implications of temporal completeness, temporal succession, temporal density and temporal constraints assure the interpolation approach and assumption rules. Again, making use of the ordering information of the data and utilising the heuristics that reduces the scope of scanning a sequence would improve the efficiency of the algorithms.

Chapter 9

Conclusions and Future Work

This chapter concludes the thesis. The main contributions of the work are highlighted and conclusions have been drawn. Query processing in temporal object-oriented databases is a broad subject posing a major challenge, and this leads to a need to specify the limitations of the work carried out and to propose further work.

9.1 Major Contributions of the Work

This thesis mainly consists of three parts. Part I (Chapters 2 and 3) was an overview of background and research status in this area. Part II (Chapter 4 and 5) constructed a fundamental part for query processing, i.e., the temporal data model and its algebra. The techniques and algorithms for processing temporal object queries were presented in Part III (Chapter 6 to 8). The main work done in the thesis has been expounded through out these chapters and has been summarised in Chapter 1. Here we highlight the major contributions of the work as follows.

The major contributions of this work are concerned with employing an extensible approach to processing temporal object queries within a uniform query processing framework. Firstly, a temporal object data model has been defined, in which a temporal object represented by a time sequence can model various time varying entities in the real world. The model possesses the extensive features of object orientation, and the temporal aspects. Secondly, an algebra with the properties of closure and reducibility

has been developed to provide access to objects that are interrelated through associations of aggregation, inheritance and time-reference. Thirdly, this thesis presents a layered structure for optimizer design. The importance of this structure is that it separates the functionality between the temporal optimizer, the object optimzer and relation optimizer and allows the exploitation or extension of the existing query processing techniques at different layers. Fourthly, a set of query transformation rules are specified for the algebraic manipulation. Fifthly, the enhanced path has been defined to refer to a path with a time reference and a decomposition strategy is proposed for processing temporal object queries involving such an enhanced path. Sixthly, query processing algorithms are implemented using stream processing techniques that makes use of ordering information of time varying data. Seventhly, this thesis presents valuable cost analysis and simulation results: the join time cost is linearly increased with the expansion in the number of time-epochs (it is linearly increased with the expansion of time in the case of a regular TS). The term epoch taken from the signal processing field has been served as an important token for cost analysis. It is shown that utilising heuristics would result in a considerable cost saving. Finally, it has been shown that the implications of temporal data can be exploited to respond to the user query requiring the data that are not explicitly stored in the database.

It has been demonstrated that an object query processor can enhance its query processing capability by utilising relational query processing techniques, and that temporal object queries can be processed within the existing object query processing framework, through smoothly extending existing techniques. Techniques that take advantage of the semantic richness of temporal data, including stream processing techniques, natural clustering or sort order, heuristics for reducing the scope of sequence scanning, interpolation, etc., are beneficial to query processing and optimization.

Note that the project initially started in 1993. During the time we were writing up publications from our research results (papers 1-5 listed in author's publications), ODMG2.0 [Cattell, 1997] was released. ODMG Object Model introduced the concepts of lifetime of an object and structure of timestamp, though there is no construct has been specified for defining temporal object. OQL uses the relational standard SQL as a basis

but supports more powerful capabilities, i.e., it includes object extensions for object identity, complex objects, path expressions, etc. The temporal object data model defined in this thesis, via incorporating time elements into the unified model of RDB and OODB from UniSQL/X, possesses the main key object features defined in ODMG Object Model. The temporal object that is represented as a time sequence, complies with the ODMG Object Model concepts: the values of an object's properties can change over time; even though it changes over time it has the same OID. The algebra presented in the thesis defines operations that support OQL defined primitives such as implicit join, explicit join, path traversal, etc. Therefore, the idea of the extensible approach proposed in this thesis to processing temporal object queries, and its techniques and strategies, though they are based on the unified model of RDB and OODB from UniSQL, can be applied to both object-oriented databases (based on ODMG) and object-relational databases (based on SQL-3).

9.2 Conclusions

Here we draw the following conclusions or lessons learnt from the work, which further make clear why our contributions are significant.

1) The temporal object data model defines access primitives and provides a starting point for query processing

The temporal object data model is defined by incorporating a time dimension into the unified model of OODBs and RDBs from UniSQL/X [Kim, 1993; 1994; 1995; D'Andrea and Janus, 1996], which is extended from the relational data model.

- The model presents an extensible structure, i.e., it forms temporal relational-like cubes but with aggregation and inheritance hierarchies so that the temporal object-oriented database defined by the model is a superset of object-oriented database (i.e., retaining snapshot reducibility to an OODB) that in turn is a superset of relational database.

- A temporal object is represented by a time sequence that can model various practical time varying entities. Both homogeneity and heterogeneity in the time dimension can be supported.
- The *epoch,* which refers to the time when temporal object changes its value, represents a transformed time space and can then serve as a convenient token for the analysis of the query processing cost.

The model with these features defines the access primitives that at least partially determine the power of algebra. It provides a start point towards query processing.

2) A query algebra with the properties of reducibility and closure allows extending and using of existing query processing techniques for processing temporal object queries

A query algebra, that provides access to objects through aggregation, inheritance and time-reference, is then defined as a general query model /language. This algebra has the following desirable properties.

- The algebra is closed in the sense that the output from one operation can become input to another. This ensures that the use of relational techniques is possible.

- The algebra possesses the property of reducibility. By reducibility, we mean that the temporal object algebra can be reduced to the object algebra when the time dimension is not taken into account and the object algebra can be further reduced to the relational algebra when the aggregation hierarchy and inheritance hierarchy are not taken into account. This enhances the abilities to utilise and extend the existing query processing techniques for temporal queries.

- The algebra can be *grouped complete* so that it supports a rather strong notion of the "history of an attribute". This satisfies the needs required by many temporal database applications.

The algebra provides a basis for query processing and optimization. The features of the algebra allow the extended approach to be exploited.

3) Separation of the functionality between temporal optimizer, object optimizer and relational optimizer ensures existing techniques can be used or extended

Due to the extensible structure of our data model and reducibility of the algebra, the query optimizer can be designed as a layered structure. That is, the temporal optimizer is built on the top of the object optimizer that is on the top of the relational optimizer. This separates the functionality between different optimizers, and the existing query processing techniques can be exploited or extended (when necessary) at different levels.

The layered structure of the query optimizer lays out a uniform framework for processing temporal object queries.

4) Algebraic manipulation can be performed by specifying a set of transformation rules that expands relational rules by taking into account of object-oriented features and time dimension

We have identified a set of query transformation rules that comprise:

- relational rules;
- temporal transformation rules;
- inheritance rules; and
- path transformation rules.

The relational rules derived from well-known algebraic optimization techniques in RDBs play an essential role in query optimization. When the time-dimension is taken into account, the temporal transformation rules come into effect. Inheritance rules are object specific and simplify queries. The path transformation rules in OODBs have been extended to address the features of the time dimension, that conceive query processing techniques and strategies for processing temporal object-oriented queries (e.g., the decomposition strategy and join algorithms see below).

5) A decomposition strategy provides a convenient means for temporal object query processing and cost analysis, and an opportunity for optimization

A temporal object query can be represented by an enhanced path (defined as an extended path with time-reference). A decomposition strategy is devised for such a query.

The decomposition strategy for processing temporal queries can be formerly stated as follows. A complex user query with path expressions that involves a time-reference is first translated into a set of single path expressions. A single path is then divided into two sub-paths: a sub-path involving time-stamped class that can be optimized by making use of the ordering information of data and an ordinary sub-path (without a time-stamped class) that can be further decomposed and traversed using different algorithms. The intermediate results of traversed two sub-paths are then joined together to create the output query.

The advantages of decomposing the temporal query into sub-query components are that it provides a convenient means for evaluating query processing algorithms and analysing the effects of the time on query processing costs. It also provides an opportunity for optimization, e.g., well-known join algorithms can be used to optimize the query.

6) Stream processing technique is a good choice of implementing temporal query processing algorithms

As temporal data often ordered by time, the stream processing approach is a strategy of choice to implement relevant algorithms.

With the stream processing techniques, the following algorithms have been implemented:

- Stream processing time-slice algorithms and stream processing aggregation algorithms for the time-related operations;

- Four basic join algorithms (i.e., nested-loop forward join, sort-merge forward join, nested-loop reverse join and sort-merge reverse join) and their modifications.

These algorithms are presented with the corresponding cost analysis and implemented on a PC using Borland C++ Version 4.

7) The join time cost is linearly increased with the expansion in the number of time-epochs (it is linearly increased with the expansion of time in the case of a regular TS)

Adding time creates multiple versions for the same object that in turn affects query efficiency. Both cost analysis and simulation results presented in this thesis reveal the effects of time on query processing algorithms, i.e., the join time cost is linearly increased with the expansion in the number of time-epochs (it is linearly increased with the expansion of time dimension in the case of regular TS).

This justifies the effort involved in trying harder to optimize queries and performance comparison between join algorithms with and without time-related operation support.

8) Utilising heuristics, that take advantage of data ordering and restrict the scope of search space, will result in a considerable cost saving

There is a great opportunity to exploit temporal semantics in query optimization. To further reduce the time cost, the following heuristics that take advantage of data ordering and restrict the scope of search space have been presented as optimization strategies:

- Transform the time-related predicate into time-slice operations;
- Perform time-slice as early as possible;
- Transform temporal constraints into time-slice operations.

The principle of these heuristics is to look only at the data that is relevant to the query. It has been shown in the thesis by both analysis and simulation that utilising the heuristics could lead to a significant time cost saving.

9) **Implications of temporal data can be used to derive data that is not explicitly stored in the database.**

When a user query requiring the data that is not explicitly recorded in the database, the techniques of interpolation or assumption rules that exploit the implication of temporal data would help to answer the user's query. The implications of temporal completeness, temporal succession and temporal density assure the use of interpolation and assumption rules. Again, making use of the ordering information of the data and utilising the heuristics that reduces the scope of scanning a sequence can improve the efficiency of interpolation algorithms.

10) **Solely treating a temporal object as a 'blob' object that is managed by the system, but interpreted by the user is not a strategy for temporal support in OODBs**

Because of the rapid performance degradation due to ever-growing overflow chains, for OODBs that support for time-varying data, when the number of epochs is large, i.e., $n>>b_n$, there is certainly a need to provide facilities for temporal queries.

9.3 Future Work

Query processing in temporal object-oriented databases is a broad subject and poses many difficult challenges. The work done in this thesis is still limited. We would envisage the future work that might include:

1) **Extending the join algorithms to operate on the classes with many-to-many relationships**

Like many path executing algorithms used OODBs, the join algorithms presented in the thesis operate under the assumption that the two join classes have a many-to-one

relationship. Future work could be to extend the join algorithms to operate on classes with many-to-many relationships.

2) Multiple path optimization

For simplification, we have discussed the query processing with a single path. When queries involve multiple paths, the techniques proposed in the thesis can be extended and applied to such cases. But, further study on cost analysis is needed to demonstrate the performance of global optimization in this case.

3) Making use of temporal information for further optimization

Time implies a lot of information that can be used for optimization. This has been demonstrated in this thesis in some extent. Building on work in this thesis, utilising temporal constraints for semantic optimization would be a good topic of further research. Detailed temporal predicate optimization would also be worth investigating.

4) Dealing with generalisation and specification

This thesis has focused on the central issue of object query processing, i.e., path optimization. As is the case for most OODBs, it is assumed that once a class is specified as a subclass of a class, it will automatically have the attributes defined by its superclass. Support for generalisation and specification is quite system-implementation dependent (most RDBs and post-relational databases require users to manage and enforce this relationship). If it is supported by the system, query processing algorithms like what have been presented in this thesis will be affected, and it should be taken into consideration.

List of Figures

Figure 1.1 Features in OODBs and RDBs 3

Figure 2.1 An example relation: employee 19

Figure 2.2 Temporal database papers 34

Figure 3.1 Query processing strategies 49

Figure 3.2 Object query processing methodology 68

Figure 4.1 An example of OODB schema 81

Figure 4.2 Three basic type of temporal data 85

Figure 4.3 A 3-dimensional class 90

Figure 4.4 Database schema of "International Weather Record Database" 92

Figure 4.5 Database schema of "Wood Panel Deformation Measurement System" 94

Figure 4.6 Computer-aided EEG system 96

Figure 4.7 Database schema of " The Neurological Patient Care Database" 98

Figure 4.8 OLE link between database and analysis system 99

Figure 4.9 Different data processing procedures 99

Figure 4.10 Illustration of mapping on time-spaces 102

Figure 5.1 Illustration of basic algebra 105

Figure 5.2 Illustration of temporal unary set operations 111

Figure 5.3 Illustration of set operations 113

Figure 5.4 Illustration of operator *select* 114

Figure 5.5 Illustration of operator *project* 115

Figure 5.6 Illustration of operator *time-slice* 116

Figure 5.7 Illustration of operator *offset* 117

Figure 5.8 Illustration of operator *when* 118

Figure 5.9 Illustration of operator *join* 118

Figure 5.10 Illustration of operators $unnest^T$ and $nest^T$ 120

Figure 6.1 Data model extensibility 127

Figure 6.2 Optimizer layering 128

Figure 6.3 A sample path 145

Figure 6.4	An operator graph	145
Figure 6.5	Decomposition strategy for processing temporal object queries	146
Figure 7.1	Further decomposition of temporal sub-path	150
Figure 7.2	Join between block A and Block B	150
Figure 7.3	Data structure for a temporal relation	151
Figure 7.4	A file partitioned into blocks	151
Figure 7.5	Extracted from a simplified weather record database	167
Figure 7.6	Execution of simulation programs	170
Figure 7.7	Join time cost with respect to n (sel=10%) (N<200)	177
Figure 7.8	Join time cost with respect to n (sel=33%) (N<200)	177
Figure 7.9	Join time cost with respect to n (sel=50%) (N<200)	178
Figure 7.10	Join time cost with respect to n (sel=100%) (N<200)	178
Figure 7.11	Join time cost with respect to sel % (n=40)	179
Figure 7.12	Join time cost with respect to sel % (n=80)	179
Figure 7.13	Join time cost with respect to sel % (n=100)	180
Figure 7.14	Join time cost with respect to sel % (n=180)	180
Figure 7.15	Join time cost with respect to n (sel=10%) (N<5000)	181
Figure 7.16	Join time cost with respect to n (sel=33%) (N<5000)	182
Figure 7.17	Join time cost with respect to n (sel=50%) (N<5000)	183
Figure 7.18	Join time cost with respect to n (sel=100%) (N<5000)	184
Figure 7.19	NLFJ time cost with respect to n (sel=10%) without and with different time-slice intervals	185
Figure 7.20	NLFJ time cost with respect to n (sel=33%) without and with different time-slice intervals	185
Figure 7.21	SMFJ time cost with respect to n (sel=10%) without and with different time-slice intervals	186
Figure 7.22	SMFJ time cost with respect to n (sel=33%) without and with different time-slice intervals	186
Figure 7.23	NLRJ time cost with respect to n (sel=10%) without and with different time-slice intervals	187
Figure 7.24	NLRJ time cost with respect to n (sel=33%) without and with different time-slice intervals	187

Figure 7.25 SMRJ time cost with respect to n (sel=10%) without and with different time-slice intervals 188
Figure 7.26 SMRJ time cost with respect to n (sel=33%) without and with different time-slice intervals 188

List of Tables

Table 2.1 Temporal relational data models 38
Table 2.2 Temporal object-oriented data models 39
Table 2.3 Temporal relational query languages 40
Table 2.4 Temporal object-oriented query languages 41
Table 4.1 Equivalencies between post-relational and object-oriented terms 84
Table 4.2 Interaction of tuple lifespan and attribute lifespan 89
Table 4.3 Sample experiment setting 93
Table 5.1 Summary of algebraic operations 125
Table 7.1 Summary of join algorithm costs 163

List of Author's Publications Relevant to the Thesis

1 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, Processing temporal queries in the context of object-oriented databases, *Information and Software Technology,* Vol 41, No 5, p283-295, 1999, Elsevier Science, ISSN 0950-5849.

2 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, Effects of time on temporal object query processing algorithms, *Chinese Journal of Advanced Software Research,* Vol 6, No 2, 1999, Allerton Press, New York, USA, ISSN 1074 7443.

3 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, A uniform framework for processing temporal object queries, *Technology of Object-Oriented Languages and Systems: TOOLS 24*, Proceedings, p28-37, 1998, IEEE Press, ISBN 0-8186-8551-4.

4 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, Decomposition: a strategy for query processing in temporal object-oriented databases, *New technologies on Computer Software*, p124-133, International Academic Publishers, 1997, ISBN 7-80003-408-9/TP.15 (awarded the "BEST PAPER" by NTCS/W-97 Committee).

5 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, Query processing and optimization in temporal object-oriented databases, *Proc. of Int. Workshop on Database and Expert Systems Applications (DEXA* 97), p474-481, Sept. 1997, IEEE Press, ISBN 0-8186-7662-0.

6 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, An algebra for a temporal object data model, *LNCS 1134, Database and Expert Systems Applications (DEXA 96), Proceedings,* p667-677, Zurich, Switzerland, 9-13 Sept. 1996, Springer, ISSN 0302-9743.

7 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, Query processing in object-oriented databases, *Cybernetics and Systems'96, Proceedings,* p803-808, Vienna, Austria, 8-12 April 1996 (won the "BEST PAPER AWARD" on the conference), ISBN 3 85206 133 4.

8 **Lichun Wang**, Michael Wing, Colin Davis, Norman Revell and Jin Chen, Data modelling and management in sequential image databases: a temporal object-oriented approach, *IEE Colloquium Digest on Intelligent Image Databases,* p1/1-1/6, 22 May 1996, 1996/119, IEE Press, ISSN 9063-3308.

9 **Lichun Wang**, Michael Wing, Colin Davis and Normal Revell, A temporal object-oriented model for health-care information systems, *Proc. of 1996 Chinese Automation Conf. in UK,* p193-198, Oxford, UK, 21-22 Sept. 1996.

10 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, Query processing in object-oriented databases, *TR6.96*, p1-6, ISSN 1362-2285.

11 **Lichun Wang**, Michael Wing, Colin Davis and Norman Revell, An algebra for a temporal object data model, *TR6.96*, p7-19, ISSN 1362-2285.

12 **Lichun Wang**, Jin Chen, Michael Wing, Colin Davis and Normal Revell, A temporal object-oriented model for sequential image measurement and analysis systems, *TR6.96*, p20-29, ISSN 1362-2285.

13 **Lichun Wang**, A framework for unifying RDB and OODB models, *Proc. of MUCORT'95*, Middlesex University, UK, 1995, *TR1.96*, p49-54, ISSN 1362-2285.

References

ADB, (1992). Matisse Technology Overview, *Technical Report*, ADB/Intellitic.

Alhajj, R. and Arkun, M. E. (1993). A query model for object-oriented databases. *Proc. of 9th Int. Conf. on Data Engineering*, p163-172.

Ariav, G. (1986). A temporal oriented model, *ACM Trans. Database Systems*, 11(4): 499-527.

Atkinson, M., *et al.* (1990). The object-oriented database system manifesto. *Deductive and Object-Oriented Databases,* Kim, W., Nicholas K.M. and Nishio, S. (editors), Elsvier Science Publishers.

Bassiouni, M. A. and Lewellyn, (1992). A relational-calculus query language for historical databases, *J. Computer Languages*, 17(3): 185-197.

Bancilhon, F. and Ramakrishnan, R. (1986). An amateur's introduction to recursive query processing strategies. *Proc ACM-SIGMOD Int. Conf. on Management of Data,* p16-52.

Banerjee, J., Kim, W. and Kim, K. C. (1988). Queries in object-oriented databases. *Proc. 4th Int. Conf. on Data Engineering,* Los Angeles, Calif.

Beech, D. and Mahbod, (1988). Generalized version control in an object-oriented database, *Proc. Int. Conf. Very Large Databases*, p14-22.

Bell, D. and Grimson, J. (1992). *Distributed Database Systems*, Addison Wesley.

Ben-Zvi, J. (1982). The time relational model, *PhD thesis*, Computer Science Dept., UCLA, 1982.

Bemuth, B., *et al.* (1994). Algebraic query optimization in the CoOMS structurally object-oriented database system. *Query Processing for Advanced Database Systems,* p121-143, Morgan Kaufmann Publishers.

Bernstein, *et al.* (1998). The Asilomar Report on Database Research, *SIGMOD Record,* Vol., 27, No.4, Dec.1998.

Bertino, E., Ferrari, E. and Guerrini, G. (1998). An approach to model and query event-based temporal data, *Proc. of 5th International Workshop on Temporal Representation and Reasoning, p122-31, IEEE Press, 1998.*

Bertino, E. and Guglielmina, C. (1993). Path-index: an approach to efficient execution of object-oriented queries. *Data and Knowledge Engineering,* 10:1-27.

Bertino, E. and Foscoli, P. (1995). Index organization for object-oriented database systems. *IEEE Trans. on Knowledge and Data Engineering,* 7(2):193-209.

Bertino, E. and Martino, L. (1993). *Object-Oriented Database Systems: Concepts and Architectures.* Addison-Wesley Publishers Ltd.

Bertino, E. (1994). A survey of indexing techniques for object-oriented databases. *Query Processing for Advanced Database Systems*, p383-417, Morgan Kaufmann Publishers.

Bertino, E., Ferrari, E., and Guerrini, G. (1997). T Chimera: a temporal object-oriented data model, *Theory and Practice of Object Systems,* 3(2):103-125.

Bettini, C., Wang, X. S. and Jajiodia, S. (1998). An archetecture for supporting interoperability among temporal databases, *Temporal Databases: Research and Practice*, Springer-verlag, Berlin, 1998, p36-55.

Bettini, C., Wang, X. S., Bertino, E. and Jajodia, S. (1995). A semantic assumptions and query evaluation in temporal databases, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, San Jose, CA, USA, p257-268.

Bhargava, G. and Gadia, S. K. (1993). Relational database systems with zero information loss, *IEEE Trans. Knowledge and Data Engineering,* 5(7): 76-87, Feb.

Blakely, J. A., *et al.* (1989). Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems,* 14(3): 369-401.

Blakely, J.A., (1993). Experiences building the open OODB query optimizer, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Washington DC, USA, 287-296.

Bolour, A., Anderson, T. L., Dekeyser, L. J. and Wong, H. K. T. (1983). The role of time in information processing: a survey, *ACM SIGMOD Record,* 12(3): 27-50.

Bohlen, M. H. (1995). Temporal database system implementation. *SIGMOD RECORD,* 24(4): 53-60.

Cardenas, A., *et al.* (1993). The knowledge-based object-oriented PICQUERY + language, *IEEE Trans. Knowledge and Data Engineer*ing, 5(4): 644-657, Aug.

Carey, M. J., *et al.* (1990). The EXODUS extensible DBMS project: an overview. *Readings on Object-Oriented Database Systems* (Maier, D. and Zdonik, S., editors.). Morgan Kaufmann, San Mateo, CA.

Carery, M.J., DeWitt, D. J. and Vandenburg, (1988). A data model and query language for EXODUS, *Proc. ACM Int. Conf. Management Data, Chicago, p413-423. June.*

Carpenter, R. H. S. (1996). *Neurophysiology*. London: Edward Arnold.

Carrano, F. M. (1995). *Data Abstraction and Problem Solving With C++.* Benjamin/ Cummings.

Caruso, M. and Sciore, E. (1988). Meta-functions and contexts in an object-oriented database language, *Proc. ACM Int. Conf. Management Data, Chicago, p56-65, June.*

Cattell, R. G. G. (1994). *Object Data Management: Object Oriented and Extended Database Relational Systems*, Addison Wesley.

Cattell, R. G. G. (editor) (1996). *The Object Database Standard: ODMG-93 Release1.2.* Morgan Kaufmann

Cattell, R. G. G. *et al.* (editors) (1997). *The Object Database Standard: ODMG2.0.* Morgan Kaufmann.

Chakravarthy, U.S., *et al.* (1990). Logic-based approach to semantic query optimization. *ACM Trans. on Database Systems.* 15(2):162-207.

Chen, J., Clarke, T. A. and Robson, S. (1994). Optimised target matching based on a 3-D space intersection and a constrained search for multiple camera views. *Videometrics III, SPIE Proc.*, Vol 2350, p 324-335.

Chu, W. W., *et al.* (1992). A temporal evolutionary object-oriented data model and its query language for medical image management, *Proc. Int. Conf. Very Large Databases*, Aug.

Clarke, A., *et al.* (1995). Automated three dimensional measurement using multiple CCD camera views. *The Photogrammetric Record*, 15(85): 27-42.

Clifford, J. and Croker, A. (1987). The historical relational data model (hrdm) and algebra based on lifespans, *Proc. Int. Conf. Data Engineering*, p528-537, Los Angeles, Feb.

Clifford, J. and Warren, D. S. (1983). Formal semantics for time databases, *ACM Trans. Databases Systems*, 8(2): 214-254, June.

Clifford, J., *et al.* (1993). The historical relational data model (HRDM) revisited. *Temporal Databases: Theory, Design, and Implementation* (edited by Tansel, A.U. *et al.*), p6-27. Benjamin/ Cummings.

Cluet S. and Delobel, C. (1994). Towards a unification of rewrite-based optimization techniques for object-oriented queries. *Query Processing for Advanced Database Systems (edited by* J. Freytag, *et al.).* Morgan Kaufmann.

Codd, E. F. (1970). A relational model of data for large shared data banks, *Communication of ACM,* Vol 13, No 6, June.

Codd, E. F. (1971). A database sublanguage founded on the relational calculus, *Proc. of the ACM-SIGFIDET workshop, Data Description, Access and Control*, San Diego, Calif., Nov 11-12, ACM, New York p35-68.

Codd, E. F. (1972). Relational completeness of data base sublanguages. *Courant Computer Science Symposia No 6: Data Base Systems.* Pretice-Hall, New York, p67-101.

D'Andrea, A. and Janus, P. (1996). UNISQL's next-generation object-relational database management system. *SIGMOD RECORD*, 25(3):70-76, Sept.

Date, C. J. (1995). *An Introduction to Database Systems*, 6th Edition. Addison-Wesley.

Date, C.J. (1994). Marrying objects and relational (interview). *Database Research Group: Data Base Newsletter* 22, No3, May/June 1994 (part I); *Data Base Newsletter* 22, No 4 July/August 19994 (Part II).

Date, C. J. and Darwen, H. (1998). *Foundation for Object/Relational Databases: The Third Manifesto*, Addison Wesley, 1998.

Darwen, H. and Date, C. J. (1995). The Third Manifesto, *ACM SIGMOD Record* 24, No 1.

Dayal, U., *et al.* (1985a). PROBE- A research project in knowledge-oriented database systems: preliminary analysis, *Technical Report*, Computer Corporation of America, CCA-85-03, July.

Dayal, U., *et al.* (1985b). PROBE: a knowledge-oriented database management system. *Proc. Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, Feb.

Dayal, U. and Wuu, G. T. J. (1992). A uniform approach to processing temporal queries, *Proc. of the 18th Int. Conf. on VLDB*, p407-417, Canada.

Demuth, B., Geppert, A. and Gorchs, T. (1994). Algebraic query optimization in the CoOMS structurally object-oriented database system, *Query Processing for Advanced Database Systems (edited by* J. Freytag, *et al.).* Morgan Kaufmann.

Desai, B. C. (1990). *An Introduction to Databases Systems (Chapter 10 Query Processing),* West Publishing Company.

Dreyer, W., Dittrich, A. K. and Schmidt, D. (1994). Research respectives for time series management systems, *SIGMOD RECORD,* 23(1): 10-15, March.

Eisenberg, A. and Melton, J. (1999). SQL: 1999, formerly known as SQL3. *SIGMOD Record,* Vol 28, No 1. March 1999.

French, J., *et al.*, (1990). Summary of the final report of the NSF Workshop on Scientific Database Management, *SIGMOD RECORD,* 19(4):32-40.

Freytag, J. C., Maier, D. and Vossen, G. (editors). (1994). *Query Processing for Advanced Database Systems*, Introduction, Morgan Kaufmann Publishers.

Gadia, S. K. (1986). Towards a multihomogeneous model for a temporal databases, *Proc. Int. Conf. Data Engineering*, p390-397, Los Angeles, Feb.

Gadia, S. K. (1988). A homogeneous relational model and query languages for temporal databases. *ACM Trans. on Database Systems,* 13(4): 418-448.

Gadia, S. K., and Yeung, C. S. (1992). A generalized model for a relational temporal databases, *Proc. of Conf. Very Large Databases*, Vancouver, Canada, Aug.

Gadia, S. K. and Nair, S. S. (1993). Temporal databases: a preclude to parametric data. *Temporal Databases: Theory, Design, and Implementation* (edited by A. U. Tansel, *et al.*), p28-66. Benjamin/ Cummings Publishing.

Gardarin, G., *et al.* (1996). Cost-based selection of path expression processing algorithms in object-oriented databases. *Proc. of the 22th Int. Conf. on VLDB*, p390-401, Mumbai (Bombay), India.

Gerald, C. F. and Wheatley, P. O. (1994). *Applied Numerical Analysis,* Addison-Wesley Publishing Company.

Ginsburg, S. (1993). A temporal data model based on time sequences, *Temporal Databases: Theory, Design, and Implementation* (edited by A. U. Tansel, *et al.*), p248-270. Benjamin/ Cummings Publishing.

Goralwalla, I. A., Ozsu, M. T. and Szafron, D. (1998). An object-oriented framework for temporal data models. *Temporal Databases: Research and Practice*, Springer-Verlag, Berlin, Germany, 1998, p1-35.

Goralwalla, I. And Ozsu, M. T. (1993), Temporal extensions to a uniform behavioural object model, *Proc. Int. Conf. Entity-Relationship Approach*, Dallas, June.

Graefe, G. and Maier, D. (1988). Query optimization in object-oriented database systems: a prospectus, *2nd Int. Workshop on Object-Oriented Database Systems,* Springer-Verlag.

Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), p73-170.

Graefe, G., *et al.* (1994). Extensible query optimization and parallel execution in Volcano. *Query Processing for Advanced Database Systems,* p305-336, Morgan Kaufmann Publishers.

Graham, I. (1994), *Object-Oriented Methods*, Addison-Wesley, 1994.

Griffiths, A. and Theodoulidis, B. (1996). SQL+i: Adding Temporal Indeterminacy to the Database Language SQL. LNCS, *Proc. of 15th British National Conference on Databases*, p204-221, UK.

Gunadhi, H. and Segev, A. (1990). A framework for query optimization in temporal databases. *Proc. of 5th Int. Conf. on Statistical & Scientific Database Management,* p131-147.

Gyssens, M., *et al.* (1994). Tagging as an alternative to object creation. *Query Processing for Advanced Database Systems*, p201- 242, Morgan Kaufmann Publishers.

Jarke, M. (1984). External semantic query simplification: a graph-theoretic approach and its implementation in Prolog. *Proc. 1st Int. Workshop Expert Database Systems,* p 467-482.

Jarke, M. and Koch, J. (1984). Query optimization in database systems, *Computing Surveys*, 16(2):111-152.

Jarke, M. and Koch, J. (1985). Introduction to query processing, *Query Processing in Database Systems* (edited by Kim, W., Reiner, D.S. and Batory, D.S.), Springer-Verlay.

Jenq, P., Woelk, D., Kim, W. and Lee, W. (1990). Query processing in distributed ORION. *Proc. on Extending Database Technology*, Venice: Springer-Verlag.

Jensen, C. S., Mark, L. and Roussopoulos, N. (1991). Incremental implementation model for relational databases with transaction time, *IEEE Trans. Knowledge and Data Engineering*, 3(4): 461-473, Dec.

Jensen, C. S., and Mark, L. (1992). Queries on changes in an extended relational model, *IEEE Trans. Knowledge and Data Engineering*, 4(2): 192-200, April.

Jensen, C. S., Mark, L. and Roussopoulos, N. (1993). Using differential techniques to efficiently support transaction time, *VLDB J.*, 2(1): 75-111, Jan.

Jensen, C. S., Soo, M. D., and Snodgrass, R. T. (1994). Unifying temporal models via a conceptual model, *Information Systems*, 19(7): 513-547, Dec.

Jensen, C. S., *et al.* (1994). A consensus glossary of temporal database concepts, *SIGMOD RECORD,* 23(1): 52-64. March.

Jeusfeld, M. and Staudt, M. (1994). Query optimization in deductive object bases. *Query Processing for Advanced Database Systems*, p145-176, Morgan Kaufmann Publishers.

Jones, S., Mason, P. and Stamper, R. (1979). Legol 2.0: a relational specification languages for complex rules, *Information Systems*, 4(4): 293-305, Nov.

Kafer, W., Ritter, N. and Schoning, H. (1990). Support for temporal data by complex objects, *Proc. of 16th VLDB Conf.*, Brisbane, Australia, P24-35.

Kafer, W., Ritter, N. and Schoning, H. (1992a). Mapping a version model to a complex object data model, *Proc. Int. Conf. Data Eng.*, p348-357.

Kafer, W., Ritter, N. and Schoning, H. (1992b). Realizing a temporal complex-object data model, *Proc. ACM Int. Conf. Management Data,* p266-275.

Keller, T., Graefe, G. and Maier, D. (1991). Efficient assembly of complex objects, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, p148-157.

Kendall, M. and Ord, J. K. (1990). *Time Series.* Edward Arnold.

Kemper, A. and Moerkotte, G. (1994). Query optimization in object bases: Exploiting Relational Techniques. *Query Processing for Advanced Database Systems*, p 63-97, Morgan Kaufmann Publishers.

Khoshafian, S. (1993). *Object-Oriented Databases.* John Wiley &Sons, Inc.

Kifer, M., Kim, W. and Sagiv, Y. (1992). Querying object-oriented databases. *Proc. of ACM SIGMOD Conf. on Management of Data,* 393-402.

Kim, W., Reiner, D.S., and Batony, A.S. (editors) (1985). *Query Processing in Database Systems*. Springer-Verlay, 1985.

Kim, W. (1989). A model of queries for object-oriented databases. *Proc. of the 5th Int. Conf. on VLDB*, p423-432, Amsterdam, 1989.

Kim W., *et al.,* (1990), Architecture of the orion next-generation database system, *IEEE Trans. Knowledge and Data Eng.*, 2(1): 109-124, March.

Kim, W. (1990). Object-oriented approach to managing statistical and scientific databases, *Proc. of 5th Int. Conf. on Statistic & Scientific Database Management.* p1-13.

Kim, W. (1993). Object-oriented databases systems: promises, reality, and future. *Proc. of the 19th Int. Conf. on VLDB*, p 676-687, Dublin, Ireland.

Kim, W. (1994). Next-generation database systems: objects and beyond. *Proc. of IISF/ACM Japan Int. Symposium, Computers as our Better Partners,* p188-196, Tokyo, Japan, March 1994. World Scientific Publishing, Singapore.

Kim, W. (editor) (1995). *Modern Database Systems: the Object Model, Interoperability, and Beyond,*. ACM Press.

Kim, W., Garza, J. F. and Graham, B. (1997). Database technology beyond object-relational, Advances in Databases and Information Systems, *Proc. of International Workshop in Databases and Information Systems,* Springer-Verlag, London, 1997.

Kim, K. C., Kim W. and Dale, A. (1989). Cyclic querying processing in object-oriented databases. *Proc. 5th Int. Conf. on Data Engineering,* Los Angeles, Calif, Feb.

Kimball, K. A., (1978). *The DATA system*, master thesis, University of Pennsylvania.

Kline, N. (1993). An update of temporal databases bibliography, *ACM SIGMOD Record,* 22(4): 66-80.

Kotz-Dittrich A. and Dittrich, K. R. (1995). Where object-oriented DBMSs should do better: a critique based on early experiences. *Modern Database Systems: the Object Model, Interoperability, and Beyond,* (edited by W. Kim), p238-253. ACM Press.

Kulkarni, K., Bauer, J., *et al.* (1994). ADT-based type system for SQL. *Query Processing for Advanced Database Systems*, p6-33, Morgan Kaufmann Publishers.

Lausen, G. and Marx, B. (1994). Evaluation aspects of an object-oriented deductive database language. *Query Processing for Advanced Database Systems*, p177-198, Morgan Kaufmann Publishers.

Lehner, W., Ruf, T. and Teschke, M. (1995). Optimizing database access performance in scientific applications without compromising logical data independence. *Proc. of the 2nd Int. Conf. on Applications of Databases*, USA, 13-15 Dec.

Leung, T. Y. C. and Muntz, R. R. (1990). Query processing for temporal databases. *Proc. of IEEE Int. Conf. on Data Engineering*, p200-207.

Leung, T. Y. C. and Muntz, R. R. (1992). Temporal query processing and optimization in multiprocessor database machines, *Proc. of the 18th VLDB Conf.,* p383-394, British Columbia, Canada.

Leung, T. Y. C. and Muntz, R. R. (1993). Stream processing: temporal query processing and optimization. *Temporal Databases: Theory, Design, and Implementation* (edited by Tansel, A. U., *et al.*), p329-355. Benjamin/ Cummings.

Lomet, D. and Salzberg, B. (1989). Access methods for multiversion data, *Proc. ACM Int. Conf. Management Data,* p315-324, June.

Lorentzos, N. A. and Johnson, R. G. (1998). Extending relational algebra to manipulate temporal data, *Information Systems*, 13(3): 289-296.

Lorentzos, N. A. and Mitsopoulos, Y. G. (1997). SQL extension for interval data, *IEEE Trans. on Knowledge and Data Engingeering*, 6(3):480-499.

Lum, V., *et al.* (1984). Designing DBMS support for the temporal dimension, *Proc. ACM Int. Conf. Management Data*, p115-130, Boston, June.

Maier, D., Daniels, S., Keller, T., *et al.* (1994). Challenges for query processing in object-oriented databases. *Query Processing for Advanced Database Systems*, p337-380, Morgan Kaufmann Publishers.

Manola, F. and Dayal, U. (1986). PDM: an object-oriented data model. *Proc. Int. Workshop on Object-Oriented Database Systems,* Sept.

McKenzie, L. E. and Snodgrass, R. T. (1991). Evaluation of relational algebras incorporating the time dimension in databases, *ACM Computing Surveys*, 23(4): 501-543, Dec.

McKenzie, L. E., and Snodgrass, R. T. (1991). Supporting valid time in an historical relational algebra: proofs and extensions, *Technical Report TR-91-15,* Dept of Computer Science, Univ. of Arizona, Tucson, Aug.

McKenzie, L. E. (1986). Bibliography: temporal databases, *ACM SIGMOD Record,* 15(4): 40-52.

Melton, J. and Simon, A. R. (1993). *Understanding the New SQL: A Complete Guide,* Morgan Kaufmann, San Mateo, Cal.

Microsoft Cooperation (1993). *OLE2 Classes--For Microsoft Foundation Class Libraries*, Microsoft Press.

Mitchang, B. and Pirahesh, H. (1994). Integration of composite objects into relational query processing: the SQL/XNF approach. *Query Processing for Advanced Database Systems*, p35-62, Morgan Kaufmann Publishers.

Navathe, S. B. and Ahmed, R. (1989). A temporal relational model and a query language, *Information Sciences,* Vol. 49, p147-175.

Nicolas, J-M. (1982). Logic for improving integrity checking in relational database, *In Acta Informatika*, 18: p227-253.

OMG and Xopen. (1992). *The Common Object Request Broker: Architecture and Specification*, Object Management Group and X/Open, Framingham, Mass. and Reading Berkshire, UK.

Oppenhein, A. V. and Schafer, R. W. (1975). *Digital Signal Processing,* Prentice-Hall.

Orenstein, J. A., *et al.* (1986). The architecture of the PROBE database system. *Probe Project Workshop*.

Osborn, S. L. (1988). Identity, equality and query optimization, *Advances in Object-Oriented Database Systems. 2nd Int. Workshop on Object-Oriented Databases Systems* (edited by Dittrich, K. R.), p346-351, Springer-Verlag, Lecture Notes in Computer Science, 334, Sept.

Osborn, S. L. (1989a). Algebraic query optimization for an object algebra, *Technical Report,* No 251, Univ. of Western Ontario.

Osborn, S. L. (1989b). The role of polymorphism in schema evolution in an object-oriented database. *IEEE Trans. on Knowledge Engineering*, Sept.

Ozkan, C., *et al.* (1995). A heuristic approach for optimization of path expressions. *Proc. of Database and Expert Systems Applications*, p523-534, London, Sept.

Ozsoyoglu, G. and Snodgrass, R. T. (1995). Temporal and real-time databases: a survey, *IEEE Trans. on Knowledge and Data Engineering*, 7(4): 513-532, August.

Ozsu, M. T. (1991). Query processing issues in object-oriented database systems-preliminary ideas. *Symposium on Applied Computing*, Kansas City, Missouri, April.

Ozsu, M.T., *et al.* (1995). TIGUKAT: a uniform behavioural object base management system, *VLDB J.*

Ozsu, M. T. and Blakeley, J. A. (1995). Query processing in object-oriented database systems. *Modern Database Systems: the Object Model, Interoperability, and Beyond,* (edited by W. Kim), p146-174. ACM Press.

Ozsu, M. T. and Straube, D. D. (1991). Issues in query model design in object-oriented database systems, Computer Standards & Interfaces, 13(1991):157-167.

Pang, H., Lu, H. and Ooi, B. (1991). Query processing in OODB. *Database Systems for Advanced Applications' 91*, World Scientific Publishing Co.

Pickover, C. A. (1995). *Future Health: Computers and Telecommunication in Medicine in 21st Century.* New York: St. Martin's Press.

Pissinou, N., *et al.* (1993). On temporal modelling in the context of object databases. *SIGMOD RECORD,* 22(3): 8-15, Sept.

Pissinou, N, *et al.* (1994). Towards an infrastructure for temporal databases: report of an invitational ARPA/NSF workshop. *SIGMOD RECORD,* 23(1): 35-51, March.

Rumbaugh, J., Blaha, M., *et al.* (1991). *Object-Oriented Modelling and Design*, Englewood Cliffs, NJ:Pretice-Hall.

Robinson, K. A. (1979). An entity/event data modelling method, *Comput. J.,* 22(3): 270-281.

Robson, S., *et al.* (1995). Seeing the wood from the trees-an example of optimised digital photogrammetric deformation detection. *ISPRS Intercommission Workshop: From Pixels to Sequences-Sensors, Algorithms, and Systems*, Vol 30/5W1, p379-384.

Roddick, J. F. and Patrick, J. D. (1992). Temporal semnatics in information systems-a survey. *Information Systems,* 17(3): 249-267.

Rose, E. and Segev, A. (1991). TOOM-A temporal object-oriented data model with temporal constraints, *Proc. Int. Conf. Entity-Relationship Approach,* Oct.

Rose, E. and Segev, A. (1993a). TOOA: a temporal object-oriented algebra, *Proc. European Conf. Object-Oriented Programming*, July.

Rose, E. and Segev, A. (1993b). TOOSQL- A temporal object-oriented query language, *Proc. Int. Conf. Entity-Relatioship Approach*, Dallas.

Sadeghi, R. (1987). A database query language for operations on historical data, *PhD thesis*, Dundee College of Technology, Dundee, Scotland, Sept.

Sadeghi, R., *et al.* (1987). HQL—A historical query language, *Technical Report,* Dundee College of Technology, Dundee, Scotland, Dec.

Sarda, N. (1990a). Algebra and query language for a historical data model, *The Computer J.*, 33(1): 11-18, Feb.

Sarda, N. (1990b). Extensions to SQL for historical databases, *IEEE Trans. on Knowledge and Data Engineering,* 2(2): 220-230, June.

Sciore, E. (1991). Using annotations to support multiple kinds of versioning in an object-oriented database system, *ACM Trans. Database Systems*, 16(3): 417-438, Sept.

Sciore, E. (1995). Versioning and configuration management in an object-oriented data model, *VLDB J.*

Segev, A. and Shoshani, A. (1987). Logical modelling of temporal data, *Proc. SIGMOD Int. Conf. Management Data,* p454-466, San Francisco, May.

Segev, A. and Shoshani, A. (1993). A temporal data model based on time sequences, *Temporal Databases: Theory, Design, and Implementation* (edited by Tansel, A. U., *et al.*), p249-270.

Segev, A. (1993). Join processing and optimization in temporal relational databases, *Temporal Databases: Theory, Design, and Implementation* (edited by Tansel, A. U., *et al.*), p356-387.

Segev, A., Jensen C.S. and Snodgrass, R. (1995). Report on the 1995 International Workshop on temporal databases. *SIGMOD RECORD,* 24(4): 46-52, Dec.

Seshadri, P., *et al.* (1994). Sequence query processing, *Proc. of ACM SIGMOD Conf. on Management of Data,* p430-441, May.

Seshadri, P., *et al.* (1996). The design and implementation of a sequence database system. *Proc. of the 22th Int. Conf. on VLDB*, p 99-110, India.

Seshadri, P., *et al.* (1996). Cost-based optimization for magic: algebra and implementation, *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, p 435-446.

Shaw, G. M. and Zdonik, S. B. (1990). A query algebra for object-oriented databases. *Proc. of 6th Int. Conf. on Data Engineering*, p 154-162, IEEE.

Shaw, G. M. and Zdonik, S. B. (1989). Object-oriented queries: equivalence and optimization. *1st Int. Conf. on Deductive and Object-Oriented Databases.*

Shaw, G. M. and Zdonik, S. B. (1989). An object-oriented algebra. *Bulletin of IEEE Technical Committee on Database Engineering*, 12, 3 (Sept. 89), 29-36.

Shekita E. J. and Carey, M. J. (1990). A performance evaluation of pointer-based joins. *Proc. of ACM SIGMOD Conf.*, p 300-311, Atlantic, NJ, May.

Silberschatz, A., Stonebraker M. and Ullman, J. (1996). Database research: achievements and opportunities into the 21st century. *SIGMOD RECORD*, 25(1) 52-63, March.

Simon, A. R. (1995). *Strategic Database Technology: Management for the Year 2000.* Morgan Kaufmann Publishers, Inc.

Smith, J. M. and Chang, P. Y. T. (1975). Optimizing the performance of a relational algebra database interface. *Commun. ACM,* 18(10): 568-579. Oct.

Snodgrass, R. T. (1987). The temporal query language Tquel, *ACM Trans. Database Systems,* 12(2): 247-298, June.

Snodgrass, R. (1990). Temporal databases status and research directions, *SIGMOD RECORD*, 19(4): 83-89. Dec.

Snodgrass, R. T. *et al.* (1994). TSQL2 language specification, *ACM SIGMOD Record*, 23(1): 65-86, March.

Snodgrass, R. (editor) (1995). *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers.

Snodgrass, R. (1995). Temporal object-oriented databases: a critical comparison. *Modern Database Systems: the Object Model, Interoperability, and Beyond,* (edited by W. Kim), p 386-408. ACM Press.

Snodgrass, R. (1995). An overview of Tquel, *Temporal Databases: Theory, Design, and Implementation* (edited by Tansel, A. U., *et al.*), p141-179.

Snodgrass, R. T. (1987). The temporal query language Tquel, *ACM Trans. on Database Systems,* 12(2): 247-298, June.

Snodgrass, R. T. and Ahn, I. (1986). Temporal databases, *Computer,* 19(9):35-42, Sept.

Soo, M.D. (1991). Bibliography on Temporal Databases, *ACM SIGMOD Record,* 20(1):14-23.

Soo, M. D., Jensen, C. S. and Snodgrass, R. T. (1994). An algebra for TSQL2, *TSQL2 Commentary*, Sept.

Stem, and Snograss, R. T. (1988). A Bibliography on Temporal Databases, *IEEE Database Enginerring,* 7(4): 231-239.

Stonebraker, M., *et al.* (1990). Third-generation data base system manifesto. *ACM SIGMOD RECORD,* 19(3).

Stonebraker, M., Rowe, L. and Hirohama, M. (1990). The implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering*, 2(1): 125-142.

Stonebraker, M. (1996). Object-Relational DBMSs: The Next Great Wave, Morgan Kaufmann.

Stonebraker, M. (1998). Object-Relational DBMSs: Tracking the Next Great Wave, Morgan Kaufmann.

Straube, D. D. and Ozsu, M. T. (1990). Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems*, 8(4): 387-430, 1990.

Straube, D. D. (1991). Queries and query processing in object-oriented database systems. *PhD Thesis*, Univ. of Alberta, Spring.

Straube, D. D. and Ozsu, M. T. (1995). Queries optimization and execution plan generation in object-oriented database management systems. *IEEE Trans. on Knowledge and Data Engineering,* 7(2): 210-227, April.

Su, S. Y. W. and Chen, H. M. (1991), A temporal knowledge representation model OSAM*/T and its query language OQL/T, *Proc. Int. Conf. Very Large Databases.*

Sung J. and Park, J. (1991). Semantic query processing in object-oriented database systems, *Database Systems for Advanced Applications' 91*, World Scientific Publishing Co.

Tang, Z., *et al.* (1996). Optimizing path expressions using navigational algebraic operators. *Proc of Database and Expert Systems Applications*, p574-583, Zurich, Switzerland, Sept. 9-13.

Tansel, A. U. and Tin, E. (1998). Expressive power of temporal relational query languages and temporal completeness. *Temporal Databases: Research and Practice*, Springer-Verlag, Berlin, Germany, 1998, p129-49.

Tansel, A. U. (1986). Adding time dimension to relational model and extending relational algebra, *Information Systems*, 11(4): 343-355.

Tansel, A. U., *et al.* (1989). Time-by-example query language for historical databases, *IEEE Trans. Software Eng.*, 15(4): 464-478, April.

Tansel, A. U. (1993). A generalized relational framework for modelling temporal data. *Temporal Databases: Theory, Design, and Implementation* (edited by A. U. Tansel, *et al.*), p183-201. Benjamin/ Cummings Publishing.

Tansel, A.U. (1991). A historical query language, *Information Sciences,* Vol. 53, p1-1-133.

Tansel, A. U., *et al.* (editors). (1993). *Temporal Databases: Theory, Design, and Implementation.* Benjamin/ Cummings Publishing Company.

Thompson, P. A. (1991). A temporal model based on accounting principles, *PhD thesis,* Dept of Computer Science, University of Calgary, Calgary, Alta., Canada, Mar.

Toman, D. (1998). Point-based temporal extensions of SQL and their efficient implemntation, *Temporal Databases: Research and Practice*, Springer-Verlag, Berlin, Germany, p211-37.

Tsotras, V. J. and Kumar, A. (1996). Temporal database bibliography update, *SIGMOD RECORD,* 25(1): 41-51.

Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems*, Vol 2: The New Technologies, Chapter 11 Query optimization for database systems, p633-675. Computer Science Press.

Unland, R., *et al.* (1992). Object-oriented database systems: state of art and research problems. *Expert Database Systems* (edited by Keith Jeffery), Academic Press Limited.

Vandenberg, S. L. and DeWitt, D. J. (1991). Algebraic Support for complex objects with arrays, identity, and inheritance, *Proc. ACM-SIGMOD Int. Conf. on Management of Data,* 1991, p158-167.

Wang, L., Wing, M., Davis, C. and Revell, N. (1996a). An algebra for a temporal object data model. *LNCS 1134, Database and Expert Systems Applications*, Proceedings, p667-677, Zurich, Switzerland, Sept. 9-13.

Wang, L., Wing, M., Davis, C. and Revell, N. (1996b). Query processing in object-oriented databases. *Proc. of 13th European Meeting on Cybernetics and Systems Research, p803-808,* April 9-12, 1996, Vienna, Austria.

Wang, L., Wing, M., Davis, C., Revell, N. and J. Chen. (1996c). Data modeling and management in sequential image databases: a temporal object-oriented approach. *IEE Colloquium Digest on Intelligent Image Databases*, p1/1-6.

Wang, L., Wing, M., Davis, C. and Revell, N. (1997a). Query processing and optimization in temporal object-oriented databases. *Proc. of the 8th Int. Workshop on Database and Expert Systems Applications*, Sept. 1-2 1997, Toulouse, France, IEEE press.

Wang, L., Wing, M., Davis, C. and Revell, N. (1998). A uniform framework for processing temporal object queries, *Technology of Object-Oriented Languages and Systems: TOOLS 24*, Proceedings, p28-37, 1998, IEEE press.

Wang, L., Wing, M., Davis, C. and Revell, N. (1999). Processing Temporal Queries in the Context of Object-Oriented Databases, *Information and Software Technology,* 41(1999): 283-295, Elsevier Science, ISSN 0950-5849.

Wang, L., Wing, M., Davis, C. and Revell, N. (1999). Effects of time on temporal object query processing algorithms, to appear in *Chinese Journal of Advanced Software Research,* N0. 2, 1999, Allerton Press, Inc., New York, USA. ISSN 1074 7443.

Wiederhold, G., Fries, J. F. and Weyl, S. (1995). Structured organization of clinic data bases, *Proc. National Computing Conf.* , p479-485.

Wolniewicz R. H. and Graefe, G. (1992). Automatic optimization and parallelization of computations in scientific databases.

Wuu, G. T. J. and Dayal, U. (1992). A uniform model for temporal data retrieval, *Proc. Int. Conf. Data Eng.,* 584-593, Tempe, Ariz, Feb.

Yannakakis, M. (1995). *Perspectives on database theory,* IEEE.

Yao, S. B. (1979). Optimization of query evaluation algorithms. *ACM Trans. on Database Systems*, 4 (2): 133-55.

Yau, C. and Chat, G. S. W. (1991). TempSQL- a language interface to a temporal relational model, *Information Science and Technology*, p40-60. Oct.

Yu, C. T. and Meng, W. (1998). *Principles of Database Query Processing for Advanced Applications*, Morgan Kaufmann Publishers, Inc.

Yu, L. and Osborn, S. L. (1991). An evaluation framework for algebraic object-oriented query models. *Proc. of 7th Int. Conf. on Data Engineering,* p 670-677. IEEE.

Zand, M. Collins, V. and Caviness, D. (1995). A survey of current object-oriented databases, *Data Base Advances*, 26(1): 14-29, Feb.

Zdonic, S. B. (1989). Query optimization in object-oriented databases. *Proc. 22nd Annual Hawaii Int. Conf on System Sciences*, p19-25.

Zdonic, S. B. and Maier, D. (editors). (1990). *Readings in object-Oriented Database Systems*. Morgan Kaufmann Publishers.

Zurek, T. (1998). Parallel processing of temporal joins. *Informatica,* 22(2): 153-66, May.

APPENDIX

Selected Published Papers

A.1 Processing Temporal Queries in the Context of Object-Oriented Databases

A.2 An Algebra for a Temporal Object Data Model

Reprinted from

INFORMATION AND SOFTWARE TECHNOLOGY

Information and Software Technology 41 (1999) 283–295

Processing temporal queries in the context of object-oriented databases

L. Wang[a,*], M. Wing[b], C. Davis[b], N. Revell[b]

[a]*Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK*
[b]*School of Computing Science, Middlesex University, Bounds Green Road, London N11 2NQ UK*

Received 12 January 1998; received in revised form 14 December 1998; accepted 15 December 1998

ELSEVIER

Information and Software Technology is an international technical journal which covers all aspects of software development and information processing, from state-of-the-art research, through software development and implementation, to information systems management. The journal gives equal emphasis to the theories of software engineering and the application of information technology within organizations.

Papers published in the journal are drawn from current developments in areas such as: empirical and experimental analyses, software metrics, software processes and development methods, project management, quality control and standards, object orientation, concurrency, human factors, testing, implementation techniques, database design and information systems, to provide a total view of informations systems technology.

Contributions Those wishing to submit full-length papers, tutorials or review papers should send four copies to either of the Co-Editors. Contributors should refer to the Notes for Authors printed in this issue of the journal. These are also available from the publishers.

ELSEVIER

Information and Software Technology 41 (1999) 283–295

INFORMATION AND SOFTWARE TECHNOLOGY

Processing temporal queries in the context of object-oriented databases

L. Wang[a,*], M. Wing[b], C. Davis[b], N. Revell[b]

[a]*Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK*
[b]*School of Computing Science, Middlesex University, Bounds Green Road, London N11 2NQ UK*

Received 12 January 1998; received in revised form 14 December 1998; accepted 15 December 1998

Abstract

This article investigates an extensible approach to processing temporal queries in the context of object-oriented databases. Within the uniform query processing framework, a strategy of decomposition is proposed for processing temporal queries that involve paths based on the defined temporal object data model and query algebra. Algorithms for processing the decomposed query components have been implemented using stream processing techniques and are presented with cost analyses. Heuristics that optimize the temporal queries are also presented. Both cost analysis and simulations show that join time cost is linearly increased with the expansion in the number of time-epochs and that utilising the heuristics presented in this article can lead to a significant time cost saving.

Keywords: Object-oriented databases; Temporal data; Query processing

1. Introduction

Temporal properties play an essential role in many real world applications. Most object-oriented database (OODB) proposals and post-relational products include constructors for complex types such as lists and arrays that allow time-stamped entities to be represented as a 'blob' object (which is managed by the system, but interpreted solely by the application program); no facilities for temporal queries are provided [15,17]. Research on temporal object-oriented databases (TOODBs) is mainly focused on defining temporal data models by extending existing models [12,13,14,17,19]. Not much work has been done on temporal query processing and optimization [6].

In the context of databases, two time dimensions are of general interest [17]: valid time and transaction time. Valid time concerns the time a fact was true in reality whilst transaction time concerns the time the fact was stored in the database. This article is concerned with valid time data management, and investigates temporal query processing within the formal object-oriented query processing framework, based on our previously defined object data model and algebra [20,21]. As a result of the extensible features of our temporal data model and the reducibility of query algebra, a decomposition strategy is proposed for processing temporal queries that involve a path expression. Algorithms are provided to process the query components that result from this decomposition. Cost analysis and simulation results are given to illustrate how the time that is present in the query affects the query costs and how the heuristics could reduce the cost.

As relational databases (RDBs) always require the users to explicitly join two relations, most attention to temporal query processing in the context of RDBs has focused on specific temporal join algorithms [4,13,19], following the bottom up approach. However, OODBs significantly reduce the need for joins of classes and replace this explicit join with an implicit join (select operator). The path optimization that deals with this implicit join is a difficult and central issue in object query processing, and distinguishes object-oriented from relational query processing [1,10,11,18]. The exploration of temporal query processing with path optimization is difficult and under investigation. This article has made the first effort towards this investigation.

The remainder of this article is organised as follows. Section 2 briefly reviews the temporal object data model and algebra. The decomposition strategy for processing temporal queries is proposed in Section 3. Section 4 provides algorithms to process the decomposed components and heuristics for optimization. Simulation and evaluation are given in Section 5. Section 6 includes concluding remarks and future work.

* Corresponding author. Tel.: +44-171-419-3687; fax:+44-181-364-7069.

E-mail addresses: lichun.wang@cs.ucl.ac.uk (L. Wang), m.wing@mdx.ac.uk (M. Wing), c.davis@mdx.ac.uk (C. Davis), n.revell@mdx.ac.uk (N. Revell)

0950-5849/99/$ - see front matter
PII: S0950-5849(98)00128-1

2. A temporal object data model and algebra

Query processing techniques are dependent upon a data model and algebra/language. Our previous work [20,21] has presented a general temporal object data model and an algebra for this model which are briefly described as follows.

2.1. A temporal object data model

The temporal data model extends the unified data model of RDB and OODB from UniSQL/X [5,9,10] by adding a time dimension. The unified data model of RDB and OODB from UniSQL/X [5,9,10] extends the relational data model in three important ways, each reflecting a key object-oriented concept: (1) nested relations: the value of an attribute entry of a relation can be a tuple of another relation; (2) inheritance: a relation may be specialised through inheritance; (3) encapsulation: a relation can have procedures associated with it. UniSQL/X actually makes one more extension: collections: the tuple/attribute entry of a relation may have a set of values (i.e. any number of values) that can further be of more than one arbitrary data type. When changing the relational terms to object terms such as ''relation'' to ''class'', ''tuple of a relation'' to ''instance/object of a class'', ''procedure'' to ''method'', etc. (see [5,9,10]), the model is an object-oriented model. It is adopted as a snapshot model to incorporate time, and the use of terminology between relations and objects will be exchangeable in this article.

Let $T = \{\ldots, t_0, t_1, \ldots\}$ be a set of times, which is countably infinite, over which is defined the linear (total) order $<^T$, where $t_i <^T t_j$ means t_i occurs before (earlier than) t_j. For the sake of simplicity, we can assume that T is isomorphic to the set of natural numbers $\{\ldots, n-1, n, n+1, \ldots\}$ where the time chronon, that is a non-decomposable unit of time, is 1. Any subset of T is called a temporal set.

If an object o, that is any real world entity, exists for a certain period of time which is a subset of T (i.e. a temporal set), this period is called the object's lifespan, denoted as $L(o)$ for the object o. If the lifespan $L(o) = [t_{start}, t_{end}]$, the duration of time is called a span: $\text{span}(L(o)) = t_{end} - t_{start} + 1$.

A temporal object is defined as a time sequence (TS for short): $\{t, o(t)\}, t \in L(o) \subset T$, denoted as $\langle L(o), o(t)\rangle$, where $o(t)$ represents object o's value at the time t. A temporal object $\langle L(o), o(t)\rangle$ asserts that the object $o(t)$ is valid for its lifespan $L(o)$ and that its value changes with time. If a TS contains a value for each time point in the lifespan duration, it is called a regular TS [7]: $\langle L(o), o(t)\rangle = \{\ldots; t_{i-1}, o_{i-1}; t_i, o_i; t_{i+1}, o_{i+1}; \ldots\} = \{\ldots, o_{i-1}, o_i, o_{i+1}, \ldots\} = \{o_i\}$, where o_i represents object o's value at the time point t_i. If a TS contains values for only a subset of time points within the lifespan, it is called an irregular TS: $\langle L(o), o(t)\rangle = \{\ldots; t_{i-1}, o_{i-1}; t_i, o_i; t_{i+1}, o_{i+1}; \ldots\}$. For a discrete time event where the value of the entity is recorded at every single time point, it can be represented by a regular TS. For the discrete time event where the value of the entity is not recorded at every single time point, it can be represented by an irregular TS. For the step-wise constant, it can be represented by an irregular TS where the value o_i is assumed to retain for $[t_i, t_{i+1})$. We use the term epoch from signal processing field to refer to the time at which the object changes its value, e.g. t_i. The interval during which the value o_i persists is decided by the epoch t_i and its succeeding epoch t_{i+1}, i.e. $[t_i, t_{i+1})$. If there are n elements in a TS, it is said that there are n epochs. For example, suppose that John has been working for a company from 1975 to 1998, and that his salary was initially 1500 and had been changed to 1900 in 1978, changed to 2300 in 1984, to 2700 in 1991, and to 2900 in 1996. If the time chronon is assumed as a year and the lifespan of John's salary is [1975, 1998], then the temporal object ⟨[1975, 1998], *John's Salary*⟩ is equal to

{1975, 1500; 1978, 1900; 1984, 2300; 1991, 2700; 1996, 2900}.

Here, the temporal set is {1975, 1978, 1984, 1991, 1996}, and the value set is {1500, 1900, 2300, 2700, 2900} where the first salary 1500 retains for [1975, 1978) and the last salary 2900 retains for [1996, 1998]. The epoch number is five and the span is $(1998 - 1975 + 1) = 24$. From this discussion it can be seen that the epoch represents a transformed time space and will serve as a convenient indicator for cost analysis. For a continuous time event, depending on the recording of the data, it can be represented by either a regular TS or an irregular TS. When it is represented by a regular TS, it is treated as a discrete time signal created by sampling the corresponding continuous time signal. As long as the sampling frequency is greater than two times the highest frequency of the signal, the continuous time signal can be recovered from the discrete time signal. If it is represented by an irregular TS, as it is time varying the value between two recorded time points can be decided by an interpolation function depending on the application, e.g. linear interpolation. For a constant object o, it may be represented with no timestamp where its time-reference is implied as $L(o)$. (It can also be represented with an explicit time-reference as a temporal object: $\langle L(o), o\rangle$).

As a TS is a set, a temporal object can be represented by its sub-objects. In practice the lifespan may consist of disjoint, non-contiguous segments, as in [7] we prefer to use null rather than defining multiple segments in the lifespan. For instance, if we know Mary's salary records during the time [1967, 1982] and [1990, 1998] as {1967, 1400; 1977, 1890} and {1990, 2000; 1996, 2100}. However we do not know her salary between 1982 and 1990. If the lifespan is assumed as [1967, 1996], the object is defined as {1967, 1400; 1977, 1890; 1982, null; 1990, 2000; 1996, 2100} where null persists from 1982 till 1990 when the value 2000 exists.

In OODBs, every real world entity is uniformly modelled as an object that is grouped into a class/relation. Two ways in which objects are interrelated are the associations of

Relation	A_1	A_2	...	A_n
$tuple_1$				
$tuple_2$				
...	...	...	...	
$tuple_m$				$value_{m,n}$

Fig. 1. Interaction of tuple lifespan and attribute lifespan.

aggregation and inheritance. If we ignore these associations, a class/relation C can be seen as in Fig. 1 where A_n represents the ith attribute of the relation. If $value_{m,n}$ is a temporal object with lifespan $l_{m,n}$, and $tuple_m$ is also a temporal object with the lifespan denoted as $L(t_m)$, we have

$$L(t_m) = l_{m,1} \cup l_{m,2} \cup \cdots \cup l_{m,n}.$$

The lifespan of attribute A_n is

$$L(A_n) = l_{1,n} \cup l_{2,n} \cup \cdots \cup l_{m,n}.$$

The lifespan of relation C is

$$L(C) = L(A_1) \cup L(A_2) \cup \cdots \cup L(A_n)$$

$$= L(t_1) \cup L(t_2) \cup \cdots \cup L(t_m).$$

A temporal relation can thus be represented by a three-dimensional "cube", as shown in Fig. 2, if objects in the relation have uniformly the same lifespan.

It is obvious that

$$l_{i,j} = L(t_i) \cap L(A_j).$$

Clearly our temporal object model can also support a completely heterogeneous temporal dimension.

If the domain of attribute A_i of class C is another class D, then implicitly, $L(A_i) = L(D)$. If class C is a sub-class of class C', then $L(C') = L(C)$. If the class C' has more than one sub-classes, e.g. C_1 and C_2, then $L(C') = L(C_1) \cup L(C_2)$. Moreover, if a database consists of n classes (relations) $C_1, C_2, \cdots C_n$, the lifespan of the database schema is $L = L(C_1) \cup L(C_2) \cup \cdots \cup L(C_n)$.

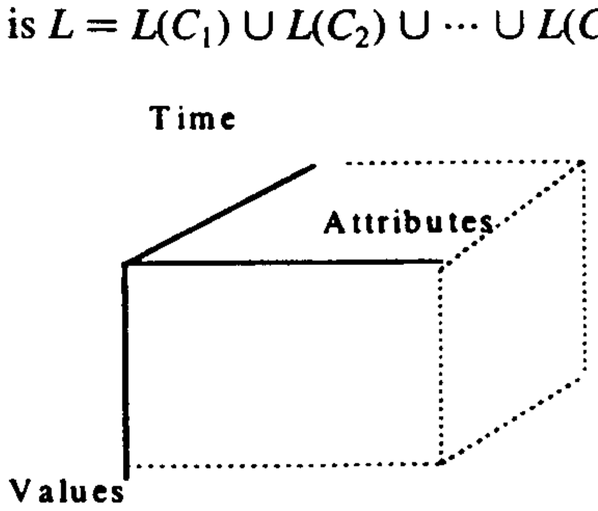


Fig. 2. A 3-dimensional class.

2.2. Algebra

From the algebraic point of view, a TOODB can be viewed as a collection of temporal objects, grouped together in classes (relations) and interrelated through three associations: aggregation, generalisation and time-reference. Each temporal relation can be viewed as a three-dimensional structure (e.g. a cube).

Basically, the standard relational algebra provides a unary operator for each of its two dimensions: *select* for the value dimension and *project* for the attribute dimension. A temporal algebra has the operation *time-slice* for the third, i.e. time dimension. As we explain further in 3.2, an object algebra allows the predicate of the *select* operation to apply to a contiguous sequence of attributes along a branch of class-aggregation hierarchy. This sort of query is usually represented by a path [1]. We have defined the enhanced path [20] that extends the path expression with time-reference so that the *select* provides an access of data with associations of both aggregation hierarchy and time-reference.

The basic algebraic operators are listed in Table 1 (detailed definitions are given in [20]), among which the operator *select* will be the focus of our discussion on query processing as it is the most powerful operator as that in any OODB. In Table 1, $P(\ )$ is a predicate. There are three basic types of predicate: a simple predicate, a nested predicate and a temporal predicate. A simple predicate is of the form ⟨*attribute-name op value*⟩. A nested predicate is a predicate on a contiguous sequence of attributes along a branch of the class-aggregation hierarchy of a class, which is represented by path-expression [1], i.e. *path op value*. A temporal predicate is a predicate referred to the temporal set along the time dimension where comparators $<$, $>$, $=$, etc. can represent the semantics of time referring to such as occurred *before*, *after*, etc. The temporal predicate can be embedded into path expressions. The enhanced path as defined refers to the value component of a temporal object along a branch of the class-aggregation hierarchy. A complex predicate is a combination of these predicates.

3. A decomposition strategy for processing temporal object queries

Our temporal data model, as shown in Fig. 3, extends the unified model of RDB and OODB by adding a time dimension, whilst the unified model itself refines the relational model by incorporating three important object-oriented features: nested relations, inheritance and encapsulation. The algebra of the model possesses the property of reducibility. That is, when the time dimension is not taken into account, the algebra is reduced to the object algebra and when object-oriented features are not taken into consideration, it is reduced to the relational algebra. Further, the algebra is closed, so that the output of operation can be the input of another. These characteristics provide us with

a basis for applying existing relational and object-oriented query processing techniques to process temporal object queries.

3.1. Optimizer layering

Our optimizer can be of a layered structure where the temporal optimizer is built on the top of an object optimizer that in turn, is on the top of a relational optimizer. When the time dimension does not exist, the object optimizer plays a role. As the object optimizer is extended from the relational optimizer, when the object-oriented features are removed from the data model, the relational optimizer comes into play. Separation of query processor functionality makes it easy to exploit and extend the existing query processing techniques at different layers that can be seen from the discussion thereafter. Therefore, the temporal queries can be processed and optimized within the existing object-oriented query processing framework through a smooth extension of the existing query processing techniques [22,23].

3.2. Query transformation involving a path expression

Path optimization is a central issue in object query processing. When the time is present, the path involving in a temporal predicate is represented by the enhanced path expression. In this article, we will focus on temporal object queries that are represented by the *select* operator with the enhanced path expression. For simplicity we suppose that the time-stamped class occurs at the end of the path (if it also occurs at other points along the path, then additional accesses to the second storage would be required, although these would not significantly add the complexity of the query).

The class-aggregation hierarchy holds an attribute/domain link between a neighbouring pair of classes in the hierarchy. The attribute/domain link between the pair of classes e.g. C_{i-1} and C_i is effectively the join of these classes, in which attributes A_{i-1} of C_{i-1} and the object identifier (OID), which is defined by the system and which can be considered as an attribute of class C_i, are join attributes [1]. Therefore, an object query with a path expression involving N classes C_1, C_2,...C_n is equivalent to a relational query, which requires the join in N relations corresponding to N classes. That is why the select operator is often called *an implicit join*. According to the definition of our algebra, when the predicate of *select* involving a path expression is $C_1.A_1.A_2.....A_n^{TM}$ *op* v (where ^{TM} represents that the path is an enhanced path), the equivalence between an implicit join and an explicit join is as

$$\sigma_{C_1.A_1.A_2.....A_n^{TM} op\ v}(C_1)$$

$$= \pi_{A(C_1)}(C_1 \bowtie^p \pi_{A_2}(C_2) \bowtie^p \cdots \bowtie^p \sigma_{A_n op\ v}(C_n)) \quad (1)$$

We use $\bowtie^p$ to represent the join that is more constrained than the join defined in our algebra in that the join attributes are the attribute A_{i-1} of the class C_{i-1} and the OID of C_i, if we join C_{i-1} and C_i. Project specifies the query target.

If there is a complex predicate involving a single path such as

$$P = C_1.A_1.op\ v_1\ and\ C_1.A_1.A_2.op\ v_2..... and$$

$$C_1.A_1.A_2.....A_n^{TM} op\ v_n$$

$$= P_1\ and\ P_2...and\ P_n,$$

then we have a general form

$$\sigma_{P(C_1.A_1.A_2.....A_n^{TM})}(C_1) \quad (2)$$

$$= \pi_{A(C_1)}(\sigma_{p_1} C_1 \bowtie^p \pi_{A_2}\sigma_{p_2}(C_2) \bowtie^p \cdots \bowtie^p \sigma_{p_n}(C_n))$$

where P_i is optional and can be omitted if it does not exist, P_n involves both time and value dimensions, and the first project specifies the query target.

The way to visit the path introduces a path traversal operator [1,8]. The *linear path traversal operator* is a navigational operator to execute the implicit join along a path, denoted as $Navi\_op[C_1, A_2, \ldots, A_n^{TM}]$. It is equivalent to a set of joins as in Eq. (2):

$$Navi\_op[C_1.A_1.A_2.....A_n^{TM}]$$

$$= \sigma_{P_1} C_1 \bowtie^P \pi_{A_2}\sigma_{P_2}(C_2) \bowtie^P \ldots \bowtie^P \sigma_{P_n}(C_n) \quad (3)$$

The aforementioned linear path traversal operator can be further rewritten into the following form:

$$Navi\_op[C_1.A_1.zA_2.....A_n^{TM}]$$

$$= Navi\_op[C_1.A_1.....A_{n-1}] \bowtie^P \sigma_{P_n}(C_n)$$

$$= Navi\_op[C_1.A_1.A_2.....A_{n-1}] \underset{C_{n-1}.A=OID(C_n) \vee P_n}{\bowtie} C_n \quad (4)$$

Thus

$$\sigma_{P(C_1.A_1.A_2.....A_n^{TM})}(C_1) \quad (5)$$

$$= \pi_{A(C_1)}(Navi\_op[C_1.A_1....A_n] \underset{C_{n-1}.A=OID(C_n) \vee P_n}{\bowtie} C_n)$$

The aforementioned transformation equivalencies imply that a *select* operator with path expressions can be evaluated using different algorithms such as translating the query into a sequence of joins, naive pointer chasing, or dividing the query into sub-paths that can be evaluated separately using different strategies or algorithms. It has been previously shown that converting implicit joins to explicit joins during the optimization phase may yield better execution plans [2] and that object navigation and set-oriented join should co-exist [8].

3.3. A decomposition strategy for processing temporal queries

Fig. 4 gives an operator graph (OG) for Eq. (5) that

Table 1
Algebraic operators

Operations	Definition	Notes
$O(T_1)$ Time-insert $O(T_2)$	$O_3 = \{o \mid o \in O\}$ where $L(O_3) = T_1 \cup T_2$	$O(T_1) \subseteq O$, $O(T_2) \subseteq O$ T_i is a temporal set and O is a set/collection
$O(T_1)$ time-delete $O(T_2)$	$O_3 = \{o \mid o \in O\}$ where $L(O_3) = T_1 - T_2$	
Difference $O_1 - O_2$	$O_3 = O_1 - O_2 = \{o \mid o \in O_1 \wedge -o \in O_2\}$ where $L(O_3) = L(O_1) - L(O_2)$	O_i is a collection/set
Union $O_1 \cup O_2$	$O_3 = O_1 \cup O_2 = \{o \mid o \in O_1 \vee o \in O_2\}$ where $L(O_3) = L(O_1) \cup L(O_2)$	$L(O_i)$ is the life-span of O_i
Intersection $O_1 \cap O_2$	$O_3 = O_1 \cap O_2 = \{o \mid o \in O_1 \wedge o \in O_2\}$ where $L(O_3) = L(O_1) \cap L(O_2)$	
Select $\sigma p\ O$	$\sigma p\ O = \{o \mid o \in O \wedge P(o,t)\}$	$\sigma p\ O$ selects the elements "o" of set O such as the predicate $P(o,t)$ holds
Map $g{:}O_1 \rightarrow O_2$	$g{:}O_1 \rightarrow O_2 = \{g(o)(o \in O_1\}$	For the type of objects in O_1 (i.e. $o \in O_1$), g returns an object of type of O_2 (i.e. $g(o) \in O_2$)
Project $\pi\langle A_1 \ldots p; A_l\rangle O$	$\pi\langle A_1,\ldots,A_l\rangle O = \{\langle A_1{:}g_1(o),\ldots,A_i{:}g_i(o)\rangle \mid o \in O\}$	If $g_i = 1$ it returns the OID of the domain object of A_i unless A_i is atomic. We retain $g_i = 1$ so that project on a set of objects (relation) likes the relational project
Join $O_1 \bowtie p\langle A_{o1}, A_{o2}\rangle O_2$	$O_1 \bowtie p\langle A_{o1}, A_{o2}\rangle O_2 = (\langle A_{o1:o1}, A_{o2:o2}\rangle \mid o_1 \in O_1 \vee o_2 \in O_2 \vee P(o_1, o_2))$	Essentially a θ-join as in relational algebra
Time-slice $\S_{T1}(O)$	$\S_{T_1}(O) = \{o \mid \forall t \in)T_1 \cap L(o))[o(t)(O]\}$	The life-span of $_{T_1}(O)$ is $T_1 \cap L(o)$. Time-slice purely reduces the relation along the temporal dimension. If T_1 equals to a time point t_1, i.e. $T_1 = t_1$, then $_{T_1}(O)$ represents an event $o(t_1)$ happened at t_1
Offset $\gamma(O, 1)$	$\gamma(O(t_1), 1) = O(t_1 + 1)$	"Shifts" a snapshot relation at t_1 by the number of positions specified by the offset 1
When $\varpi(O)$	$\varpi(O) = L(O)$	Maps a set of objects O to its temporal set

represents the *select* operator involving a single path with n classes: C_1, C_2,...,C_n. An OG is a labelled n-ary tree where the leaf nodes represent a collection of objects, the non-leaf nodes represent operators (e.g. navigational operator, join, etc.), and the edges represent temporary collections that can be represented by support tables [8]. A support table can be regarded as a collection of tuples of qualified object identifiers and attributes. Two support tables can be joined together if there exists a commonly supported collection between them. The execution of an OG follows a bottom-up order.

The query is more involved when time is present and provides more opportunity for optimization [4,15,17,19]. As we have assumed a temporal class is at the end of a path, in order to process temporal queries with a clear cost analysis and to make use of the ordering information for optimization, we propose the following decomposition strategy that can be further illustrated in Fig. 5.

A complex user query with path expressions that involves time-reference is first translated into a set of single path expressions. A single path is then divided into two sub-paths: a sub-path containing a time-stamped class that can be optimized by making use of the ordering information of temporal data and an ordinary sub-path (without time-stamped class) that can be further decomposed and traversed using different algorithms. The intermediate results of traversed two sub-paths are joined together to create the output query.

It will be shown in the next Section 4 that the decomposition strategy provides a convenience to exploit the existing

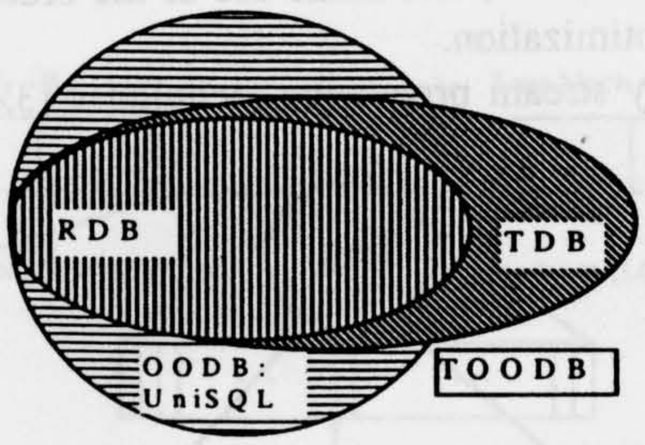


Fig. 3. Extensible features of the data model: UniSQL is extended from RDB model; (Most TDBs are in the context of RDB); Adding a time dimension into UniSQL forms a TOODB model.

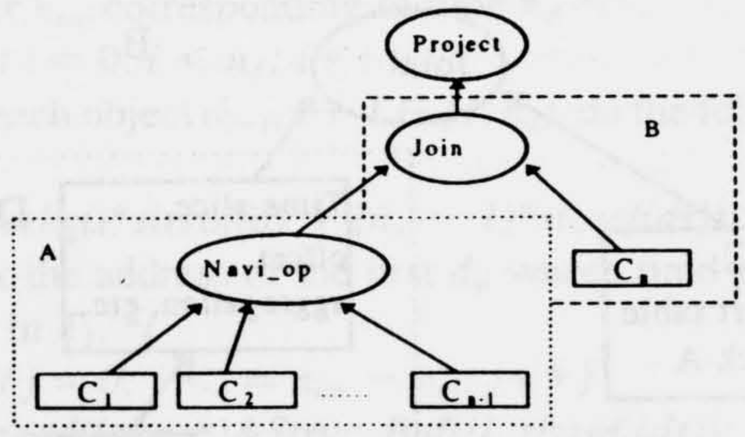


Fig. 4. An operator graph of the *select* with a single path: Block A is an ordinary sub-path (without time-stamped class); Block B is a temporal sub-path (with a time-stamped class).

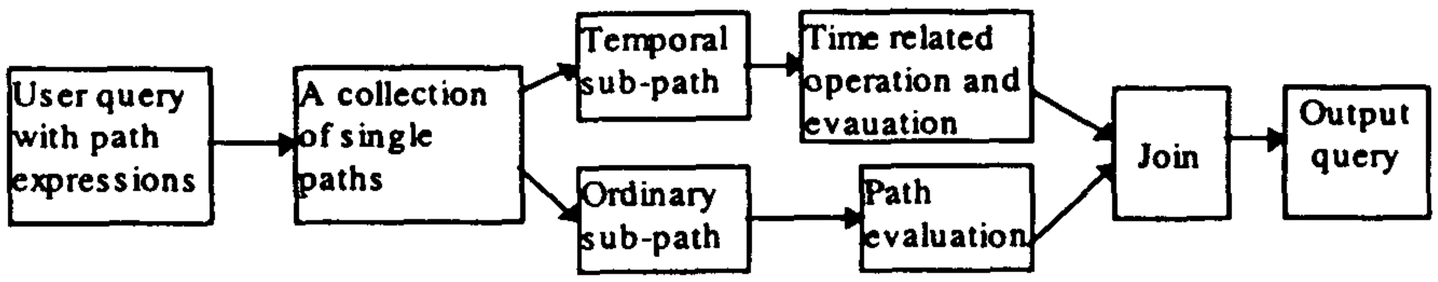


Fig. 5. A decomposition strategy.

query evaluation algorithms to process temporal object queries and provides an opportunity for optimization that makes use of the order information of temporal data.

4. Algorithms for processing the decomposed temporal query components

The block B in Fig. 4 comprises the temporal predicate evaluation (as well as some time related processing) as shown in Eqs. (4) and (5), and a join, which can be further expressed in Fig. 6. When optimizing such a query, the object optimizer works on the outer query block, and the temporal optimizer operates on the nested query block. Each optimizer is responsible for its own query block. The temporal optimizer is responsible for time-related operations and optimization. Let C represent the support table or the intermediate results of block A in Fig. 4 (that is a derived non-time-stamped relation) and D represent the intermediate results of block D in Fig. 6 (that is a derived temporal relation), as shown in Fig. 7. The object optimizer joins C and D together. This section provides algorithms for both the outer and nested blocks. The algorithms are implemented using stream processing techniques and described with cost analysis in terms of major operations such as block accesses to the second storage, and computational plus, move, comparison, etc. (among these, block access will dominate the others). The actual cost in seconds will be used in simulation.

4.1. Assumptions

We assume that a temporal relation D (it could be C_n in Fig. 6 or D in Fig. 7, depending on the situation) is stored in a file on disk. The data structure of D is as shown in Fig. 8, where D is populated with temporal objects. A temporal object d is viewed as a linked list, comprising a number of records representing a number of history versions of the same tuple field in a time ascending or descending order. For simplicity, the lifespan is uniformly represented as $L(d) = [1,n_r]$, i.e. $L(d) = L(D)$ where 1 is the starting time and n_r is the ending time of the relation. The timestamps (temporal set) for all objects are the same and the number of epochs (the number of records) in the temporal object d is $|d| = n$. If $n = n_r$, then d represents a regular TS. A temporal object is clustered (i.e. historical versions are stored together on a set of blocks), as shown in Fig. 9.

We make further assumptions. Collections (relations) C (it is C in Fig. 7) and D are stored as separate files on disk. There is a many-to-one relationship from C to D. The number of objects in C (or D) is represented as $|C| = n_c$ (or $|D| = n_d$). n_c (or n_d) objects are blocked as b_c (or b_d) instances/block. Further, n records of a temporal object d of D are blocked as b_n records/block. Obviously, $b_n = n^* b_d$. Let $fan(C,D)$ represent the average number of objects of D that are referenced by an object of C through attribute A_c. No relation is sorted or clustered. The OID is represented by the physical address of an object. Selectivity of the predicate $P(\ )$ on the temporal relation D is treated as the same as that on an ordinary relation, denoted as *sel* (we ignore the complexity of selectivity of a temporal relation here).

4.2. Time-related operators and optimization

The predicate evaluation in Fig. 6 involves the time-related operators and value evaluation. Temporal operators such as *time-slice*, *offset*, *agg-func* can be treated as methods and their outputs can then participate in the value evaluation. The temporal optimizer must be sure to 'plan' the invocation of function and make use of the ordering information for optimization.

We employ stream processing techniques [3,4]. Stream

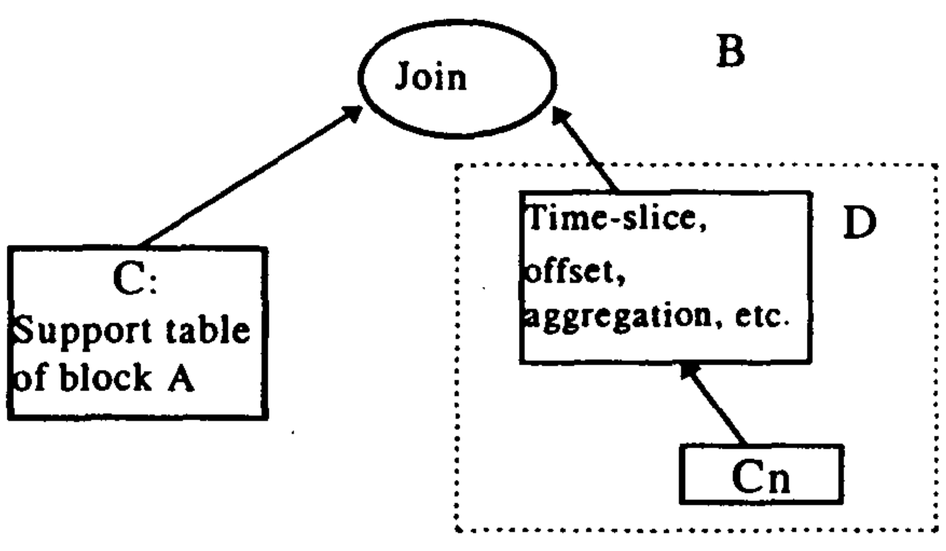


Fig. 6. Further expression of temporal sub-path.

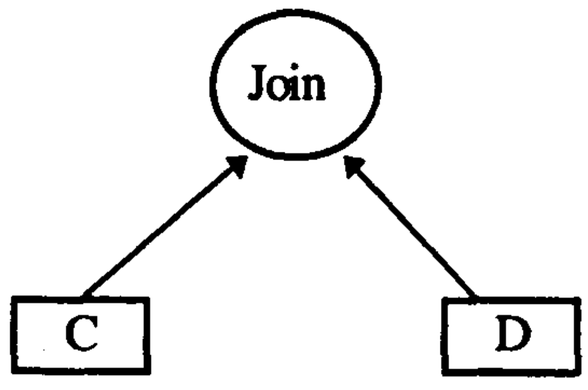


Fig. 7. Join between C and D.

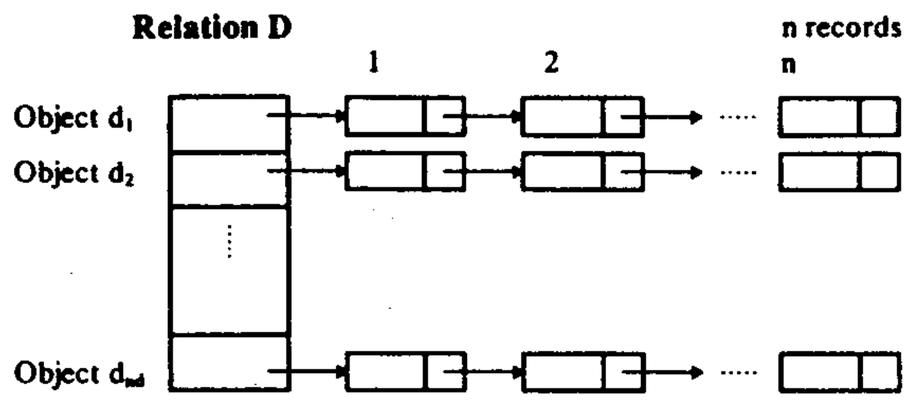


Fig. 8. Data structure for temporal relation D: A temporal relation consists of n_d temporal objects; A temporal object comprises n records representing n versions of same tuple field.

processing is a paradigm that has been widely studied and used in languages such as C++, LISP, etc. Abstractly, a stream is defined as an ordered sequence of data objects. As temporal data are often ordered by time, stream processing approach is a strategy of choice [4] so that tuples in a data stream can be efficiently accessed one at a time and in the order of successive time-stamp values, using the data stream pointer.

4.2.1. Stream processing algorithm for time-slice

Time-slice § $_{T1}(O)$ [20] performs the following operation: for every instance d in D, select its record d_s whose time point falls in $T_1 = [n_l, n_m]$, $T_1 \subset L(D) = [1, n_r]$. To minimise the accesses of data, only records satisfying the aforementioned condition are retrieved. Employing C++ stream processing techniques to implement this operation can fulfil this task. The file stream in C++ allows a user to treat a file as a stream of input or output. Given a data object, its size can be decided by using the C++ function *sizeof*( ). When the exact position (i.e. the exact address), from which a data object is stored, is decided by using C++ function *seekg*, the data object can be retrieved and the file pointer moves to the next data object. Given a temporal object d, to retrieve its records d_s from time n_l to n_m, we need to find out the exact positions corresponding to n_l and n_m in the file. We can obtain these by employing either sequential or binary search [3] within the scope of the object d to decide the epoch number n_{nl} that is corresponding to the time point n_l, and the epoch number n_{nm} that is corresponding to the time point n_m. This can be shown in the following pseudo-code of C++:

```
search(n_l,n_m,n_nl,n_nm);
/* sequential or binary search to find out the epoch
number n_nl corresponding to time n_l, and the epoch
number n_nm corresponding to time n_m */
for (int i = 0; i < n_d; i++)
/* for each object d_{i+1}, i + 1 ∈ [1,n_d], do the following */
{
fileD.seekg(i* sizeof(d) + (n_nl − 1)*sizeof(ds));
/* seek the address of the first d_s, whose time point is n_l
that is in T_1 */
for (int j = 0; j < = n_nm − n_nl; j++)
{fileD.read((char*)&Struc_Buf[j], sizeof (ds));
Buf ≪ Struc_Buf[j]};
/* sequentially read (n_nm − n_nl + 1) records d_s of the
object d_{i+1} from the file and keep them in Struc_Buf[j],
output results to Buf that could be a screen, a printer, a
buffer, etc. */
}
```

Although C++ provides stream access that allows to access one record/object at a time, the system actually performs I/O at the block level and perhaps hides this fact from the program [3]. We follow the assumption [3] that when the system provides an access for one record/object, it accesses the entire block that contains the record/object. If the next record/object is already in the stream accessed, it does not need to access the block again. Therefore we still can measure I/O access by blocks or pages. For the aforementioned algorithm, the number of block accesses can be estimated as:

$$n_d*(n_{nm} - n_{n1} + 1)/b_n \leq n_d*(n_m - n_1 + 1)/b_n$$

plus searching block access cost:

$\leq n/b_n$ in the case of sequential search,
or $\leq 2*(\log_2 n)$ in the case of binary search [3].

4.2.2. Stream processing aggregation algorithms

The operator a*gg-func* T_1 [20] that is used to perform the aggregation function (such as *Avg*, *Sum*, *Max*, etc.) can be implemented using the following pseudo-code of C++:

```
search(n_l,n_m,n_nl,n_nm);
/* sequential or binary search to find out the epoch
number n_nl corresponding to time n_l, and the epoch
number n_nm corresponding to time n_m */
for (int i = 0; i < n_d; i++)
/* for each object d_{i+1}, i + 1 ∈ [1, n_d], do the following */
{
fileD.seekg(i*sizeof(d) + (n_nl − 1)*sizeof(ds));
/* seek the address of the first d_s, whose time point is n_l,
that is in T_1. */
for (int j = 0; j < = n_nm − n_nl; j++)
fileD. read((char*)&Struc_Buf[j], sizeof (ds));
/* sequentially read (n_nm − n_nl + 1) records d_s of the
object d_{i+1} from the file and keep them in Struc_Buf[j] */
agg_func(Struc_Buf.itemi, func, value);
```

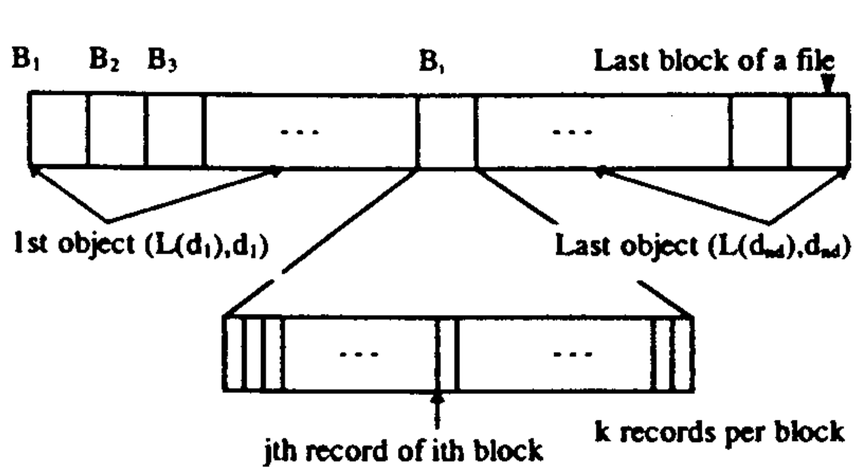


Fig. 9. A file partitioned into blocks: n records (historical versions) of a temporal object are stored together on a set of blocks.

```
/* perform an aggregation function func for a specified
attribute (i.e. itemi) */
Buf ≪ value;
/* output results to Buf that could be a screen, a printer, a
buffer, etc. */
}
void agg_func(item, char* func, float value);
int m = n_nm − n_nl + 1;
switch (func)
{
case sum:
for (value = 0; int i = 0;i < m; i++)
for (int i2 = 0; i2 < = ω̄(item(i + 1)) − ω̄(item(i));
i2++) value + = item[i];
/* insert data for missed time points and add them to
value, where ω̄(item(i)) is our algebraic operator when
that maps item(i) to its time point for simplicity here
assumed missed data is of stepwise constant */
break;
case avg:
for (value = 0; int i = 0;i < m; i ++)
for (int i2 = 0; i2 < = ω̄(item(i + 1)) − ω̄(item(i));
i2++) value + = item[i];
value = value/m;
break;
case max:
for(value = −10^35; int i = 0; i < m; i ++)
if (item[i] > value)
value = item[i];
break;
...
}
```

In addition to the same number of block access as that in time-slice algorithms, the following major operations are needed for *Agg-func*, if we ignore the time for assignment and *when*:

sum needs $n_m - n_l + 1$ plus;
avg needs $n_m - n_l + 1$ plus and 1 division;
max needs $n_{nm} - n_{nl} + 1$ comparisons.

4.3. Join algorithms

This section offers the algorithms to join C and D together. The advantage in representing them as explicit joins is that we can use well-established join algorithm strategies to perform optimization. Here a temporal object is regarded as a 'blob' object and can be treated as an ordinary object in a snapshot OODB. There are two types of joins: forward join and reverse join. According to the access methods employed to traverse the path, i.e. nested-loop method and sort–merge method, there are four basic join algorithms: nested-loop forward join, sort–merge forward join, nested-loop reverse join and sort–merge reverse join [1]. The algorithms that are implemented using stream processing techniques are presented below with cost analysis.

4.3.1. Nested-loop forward join

Nested-loop forward join (NLFJ), sometimes called the pointer-based nested-loop algorithm [16], is the algorithm that uses naive pointer traversal to compute a join. An object c of C is retrieved and the value of the attribute Ac is determined. Given this identifier, the address of the object d of D is determined. The object d of D is retrieved and the predicate is evaluated. If true, join c and d. Repeat this process till all objects of D are visited. NLFJ can be expressed using the following pseudo-code C++:

```
for (int i = 0; i < n_c; i++)
/* for each object c, do the following */
{fileC. read((char*)&BufC, sizeof(c));
/* read the object c of C */
fileD. seekg(c. Ac);
/* according to the value of the attribute Ac, i.e. the OID
of d, locate the address of d of D */
fileD. read((char*)&BufD, sizeof (d));
/* read the object d of D */
if ( predicate) Buf ≪ (join c and d);
/* if the predicate is satisfied, join c and d, and then output
results to Buf that could be a screen, a printer, a buffer,
etc. */
}
```

For the aforementioned algorithm, the number of block access can be estimated as

read C: n_c/b_c;
read D: $fan(C,D)^* n_c/b_d + = fan(C,D)^* n_c * n/b_n + .$

There are:

$fan(C,D)^*n_c$ comparisons for predicate evaluation;
$sel^* fan(C,D)^*n_c$ moves (for join).

One problem with NLFJ is that it makes no attempt to optimize disk reads [16]. As a result, a particular disk block of D can end up being read more than once. For example, suppose that two objects c_1 and c_2 reference the same object d in D. Depending on how C is organised, c_1 and c_2 may not be physically clustered together in C. If that is the case, then between the time when c_1 is joined to d and the time when c_2 is joined to d, the block containing d may be paged out of memory by buffer replacement algorithm. In that event, that block would have to be read twice, once to join c_1 with d and a second time to join c_2 with d.

4.3.2. Sort–merge forward join

Sort–merge forward join (SMFJ), sometimes called the pointer-based sort–merge algorithm [16], avoids the aforementioned problem by first sorting all of the objects in C by the value of the attribute Ac (i.e. the OID of d in D). The effect of sorting C in this manner is to group all of the objects in C that reference the same page in D. Doing so guarantees that each page in D will be read only once. The algorithm is executed as follows. All the objects of C are read into memory and sorted as in the standard sort–merge

algorithm, except that here the output runs are sorted by OID values rather than by the join attribute. According the value of *Ac*, the address (i.e. the OID) of an object *d* of *D* is determined so that the object *d* is retrieved, the predicate is evaluated and if true, *c* and *d* are joined. Repeat this process till all addresses are visited. The pseudo-code of C++ is

```
fileC.read((char*)&BufC, sizeof(C));
/* read the whole collection C */
sort C according to Ac;
for (int i = 0; i < n_c; i++)
/* for each object c of C, do the following */
{
fileD. seekg(c. Ac); /* according to the value of the attribute Ac of c, i.e., the OID of d, locate the address of d of D */
fileD. read((char*)&BufD, sizeof (d));
/* read the object d of D */
if (predicate) Buf ≪ (join c and d);
/* if the predicate is satisfied, join c and d, and then output results to Buf that could be a screen, a printer, a buffer, etc. */
}
```

For this algorithm, the number of block access can be estimated as

read *C*: n_c/b_c;
read *D*: $fan(C,D) * n_c/b_d = fan(C,D) * n_c * n/(b_n)$.

There are:

$fan(C,D)*n_c$ comparisons for predicate evaluation;
$sel*fan(C,D)*n_c$ moves (for join);
sorting cost: $sorting(n_c)$.

When the epoch number *n* is big enough such that a temporal object occupies more than one block, SMFJ will obviously not be better than NLFJ (but at the price of sorting *C*, and a bigger memory to hold the whole *C*).

4.3.3. Nested-loop reverse join

In the nested-loop reverse join (NLRJ), the strategy is similar to that of NLFJ, except that *D* is the first class visited. An object *d* of *D* is read into memory and predicate is evaluated. If the predicate is verified, then a search on the object *c* of *C* is executed to determine which instance has object *d* as the value of the attribute *Ac*. *c* and *d* are then joined. This process is repeated until all instances of *D* are visited. The pseudo-code C++ for the algorithm is:

```
for (int i = 0; i < n_d; i++)
/* for each object d_{i+1}, i + 1 ∈ [1,nd], do the following */
{fileD.read((char*)&BufD, sizeof (d));
/* read an object d_{i+1} of D */
if (predicate) /* if the predicate is satisfied */
for (int j = 0; j < n_c; j++)
/* for each object c_{j+1}, j + 1 ∈ [1,n_c], do the following */
{fileC. read((char*) &BufC, sizeof (c));
/* read an object c_{j+1} of C */
if ((fileD.tellg( )-sizeof(d)) = = c. Ac)
/* verify if the address (the OID) of d_{i+1} is equal to the value of c_{j+1}.Ac */
Buf ≪ (join c and d);
/* if so, join c_{j+1} and d_{i+1}, and output results to Buf that could be a screen, a printer, a buffer, etc. */
}
```

The number of block access can be estimated as

read *C*: $sel * n_d * n_c/b_c$;
read *D*: $n_d/b_d = n_d * n/b_n$.

There are:

n_d comparisons for predicate evaluation;
$sel*n_d*n_c$ comparisons for value evaluation;
$sel*n_d*(fan(C,D)*n_c/n_d) = sel*fan(C,D)*n_c$ moves (for join).
Clearly objects in *C* have been read many times, resulting in high I/O cost.

4.3.4. Sort–merge reverse join

In sort–merge reverse join (SMRJ), all the instances of *D* are accessed, the predicate is evaluated and a list of OIDs of instances qualifying the predicate is generated. *C* is read into memory and sorted according to *Ac*. The instances of *C* are then selected to determine which instances have an identifier in the generated list as the value of attribute *Ac*. If so, *c* and *d* are joined. The pseudo-code C++ for the algorithm is:

```
for (int j = 0; int i = 0; i < n_d; i++)
/* for each object d of D, do the following */
{fileD. read((char*)&d, sizeof(d)); /* read an object d of D */
if (predicate)
{D'[j] = d; jd[j] = fileD.tellg ( )-sizeof(d); j++;}
/* if the predicate is satisfied, keep the object d in D'[ ], and its address (the OID) in jd[ ] */.
/* This is equivalent to perform select first, the resulting relation is D'[ ], its cardinality is j */.
fileC.read((char *)&BufC, sizeof(C)); /* read the relation C */
sort C according to Ac;
j2 = 0;
for (int i = 0; i < j-1; i++)
/* for each object d_i in D'[ ], i ∈ [0, j-1], do the following */
for (int i2 = j2; i2 < n_c; i2++)
/* for each object c_{i2} in C, i2 ∈ [j2, nc) (where j2 starts from 0 and increases by 1 after a join is made), do the following */
{if (C[i2]. Ac = = jd[i]) {Buf<<(join C[i2] and D'[i]);
j2 = i2 + 1};
/* if the value of c_{i2}.Ac is equal to the address (the OID) of
```

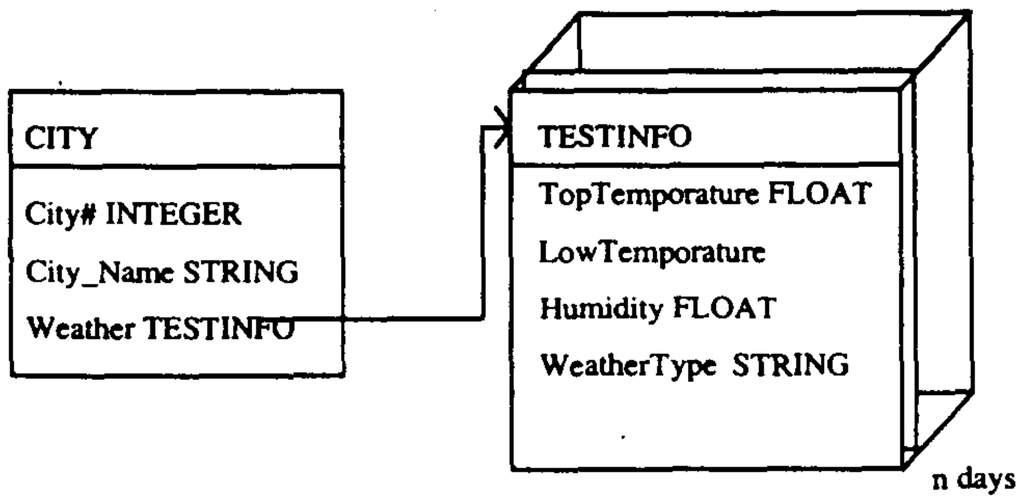


Fig. 10. International weather record database.

d_i, join c_{i2} and d_i, and output results to *Buf* that could be a screen, a printer, etc. */
else if (C[i2]. Ac > jd[i]) break;
/* if the value of $c_{i2}.Ac$ is greater than the address (the OID) of d_i, stop looping of *i*2 */.
}

The number of block access can be estimated as

read C: n_c/b_c;
read D: $n_d/b_d = n_d*n/(b_n)$.

There are:

n_d comparisons for predicate evaluation;
$sel*n_d*(fan(C,D)*n_c/n_d + 1)$ comparisons for value evaluation;
$sel*$ $fan(C,D)*n_c$ moves (for join);
sorting cost: $sorting(n_c)$.

4.3.5. Sorting

If we use the SortMerge algorithm [3] to sort items in relation C in ascending order according to the value of attribute Ac, then the SortMerge algorithm is $O(n_c \log n_c)$, in terms of major operations. If we use the SelectSort algorithm [3], it is $O(n_c^2)$.

4.3.6. Summary

As the time required for block accesses typically dominates other factors [3], we can conclude that the order of above four join algorithms are all $O(n)$, in terms of block access. That means the join time cost linearly increases with the expansion in the number of time epochs (or the time dimension, in the case of a regular TS).

The advantage of the sort–merge method over the nested-loop method is that the storage pages containing class instances of the class are never accessed more than once, resulting in considerable saving in terms of response time. The disadvantage is that the algorithms are restricted by available memory because of reading in whole class C. If all objects of the class cannot be read into the memory, the algorithms need to be modified.

The disadvantage of reverse join algorithms is that as there is no direct link from D collection to C, a value-based join must be used to check the OID membership condition, i.e. it performs value-based comparisons of OIDs, which is generally inefficient in CPU usage terms [8]. This algorithm is efficient when the predicate in the last collection is selective [8,11].

We did not discuss hybrid-hash join here, as when the epoch number n is big enough such that a temporal object occupies more than one block, implementation of the algorithms with stream processing techniques will not provide an obvious advantage over NLFJ.

4.4. Heuristics for optimization

Adding time creates multiple tuple versions with the same object. The aforementioned cost analysis shows the join algorithm performance degradation caused by ever-growing overflow chains. As reorganisation does not help to shorten overflow chains [4], the objective of work in temporal query evaluation then, is to avoid looking at all of the data [4,15]. Based on this principle, we present the following heuristics for the optimization:

- Transform the temporal predicate into *time-slice*;
- Perform *time-slice* as early as possible.

5. Simulation

To illustrate the efficiency of the join algorithms when time is present, a simulation of an international weather record database is presented, as shown in Fig. 10. Daily weather changes are recorded for major cities world-wide. The granularity of a time chronon is a day. For simplicity, suppose the database starts at 1 and ends at *today* (n). The life-span can uniformly be represented as $L(TESTINFO) = [1,n]$. The number of records in a temporal object of relation TESTINFO is also n, representing a regular TS. The relation CITY, analogous to the support table of Fig. 6, is relatively small: the cardinality of CITY is $n_c = 100$, as our intention is to show the relationship of the join response time with respect to n, i.e. the number of epochs (records) in a temporal object of relation TESTINFO. In this example, |TESTINFO|, i.e. n_d, is also 100. That means $fan(CITY,TESTINFO)$ is 1.

The four join algorithms have been implemented on PC using Borland C++ Version 4 where the SelectSort algorithm [3] is employed. Fig. 11(a)–(d) present the performance of four join algorithms, drawn in different lines, where the vertical axis represents join time costs in second and the horizontal axis represents the number of epochs in TS:n. Selectivity is set as 10, 33, 50 and 100%, respectively. It can be seen that the join cost is linearly increased with n. The performance of NLRJ is the worst, because it reads the relation CITY many times. Sort–merge join algorithms are generally good when the relations are relatively small and n is small. However they are limited by the memory of the

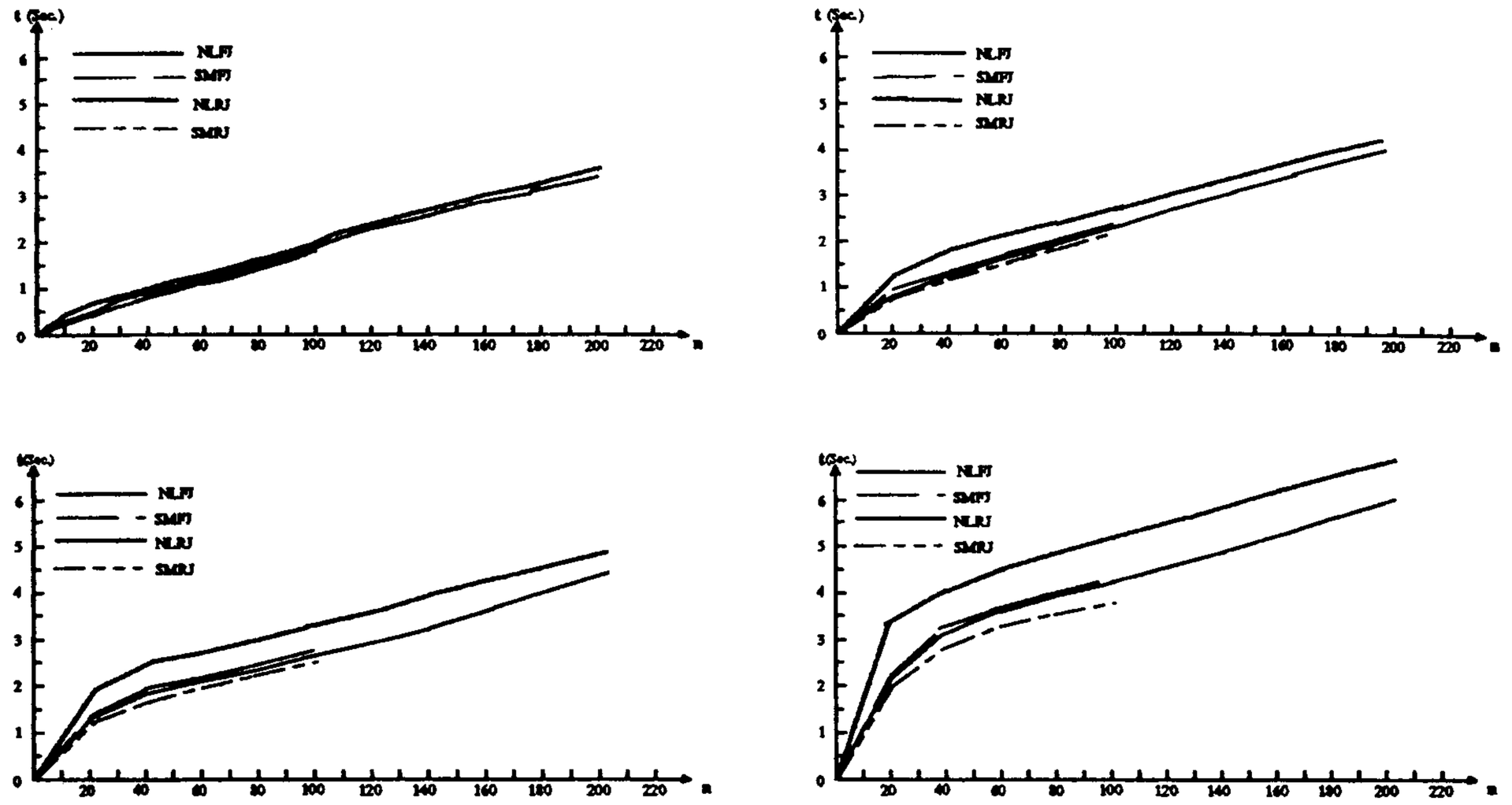


Fig. 11. (a) Time cost with respect to n ($sel = 10\%$), (b) Time cost with respect to n ($sel = 33\%$), (c) Join time cost with respect to n ($sel = 50\%$), (d) Join time cost with respect to n ($sel = 100\%$).

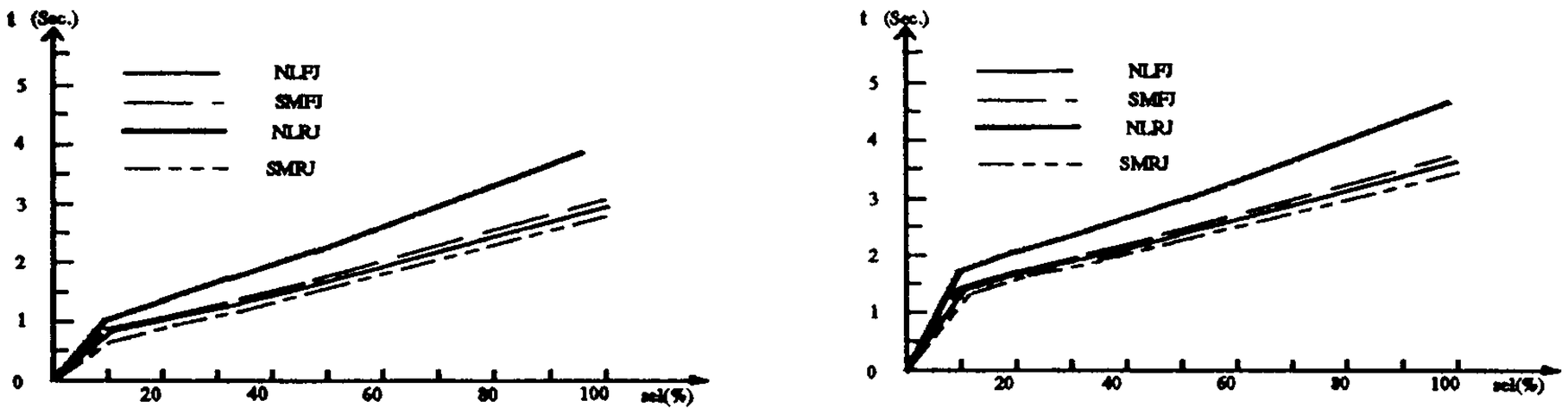


Fig. 12. (a) Join time cost with respect to $sel\%$ ($n = 40$), (b) Join time cost with respect to $sel\%$ ($n = 80$).

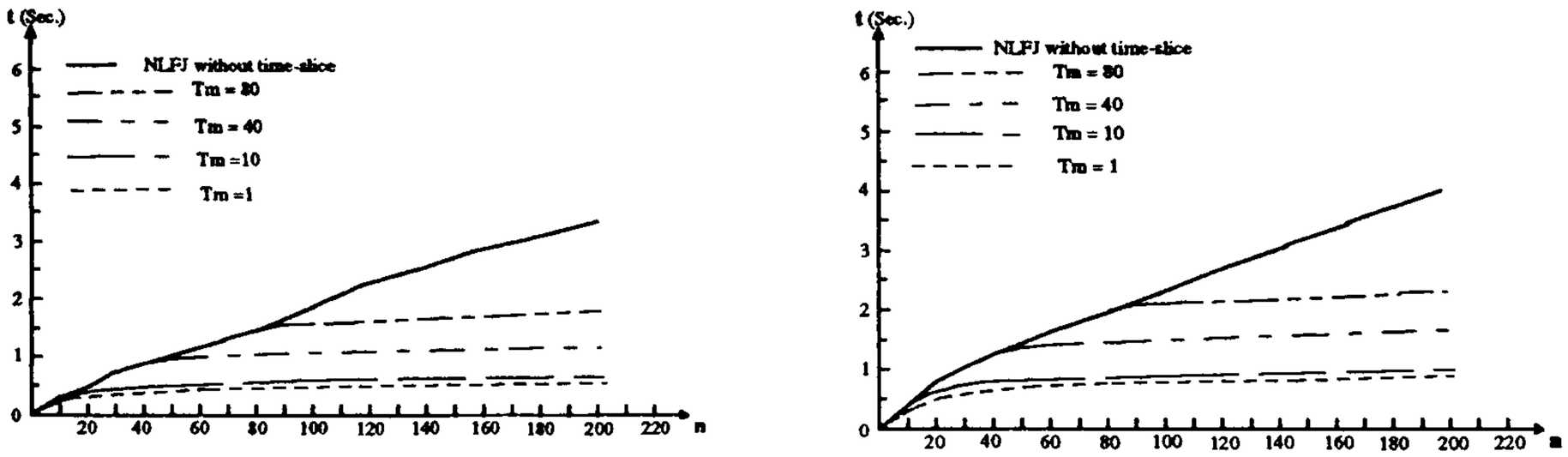


Fig. 13. (a) NLFJ time cost with respect to n ($sel = 10\%$), (b) NLFJ time cost with respect to n ($sel = 33\%$).

computer as the algorithms are terminated when n is greater than 100.

Fig. 12(a) and (b) shows the performance of join algorithms with respect to selectivity *sel*.

Fig. 13(a) and (b) (Fig. 14(a) and (b)) provides a comparison of the performance of NLFJ (NLRJ) with and without time-slice intervals. The performance of join algorithms without time-slice is analogous to that of OODBs which

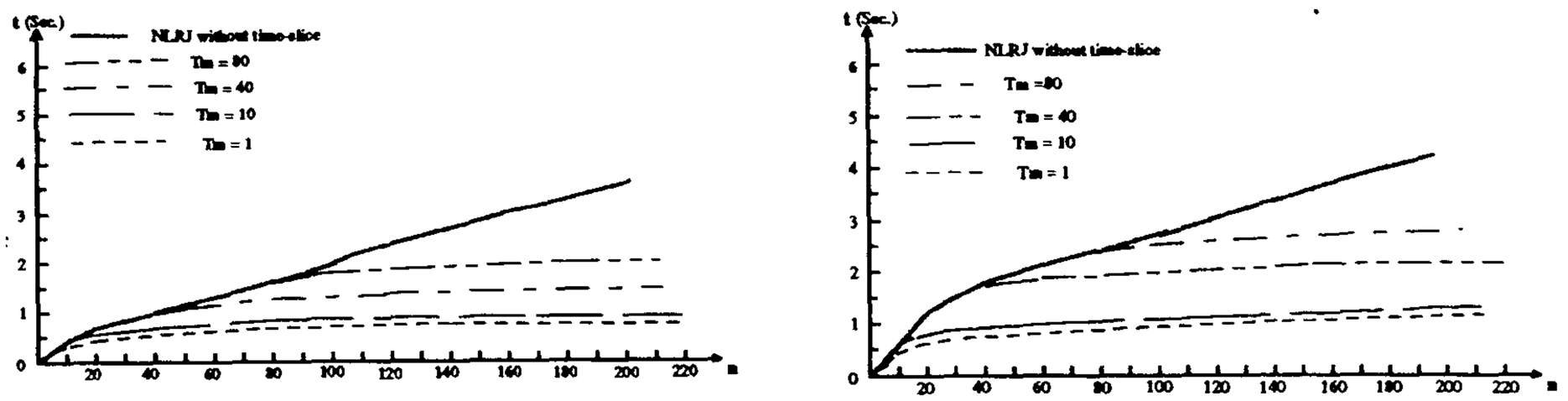


Fig. 14. (a) NLRJ time cost with respect to n ($sel = 10\%$), (b) NLRJ time cost with respect to n ($sel = 33\%$).

allows a user to represent temporal data as 'blob' objects but with no support for time varying query whilst the performance of join algorithms with time-slice is analogous to that of OODBs which support for time-varying data and utilise the heuristics for optimization such as those presented in the previous section. The span of time-slice $T_l = [nl,nm]$ is denoted as $T_m = (n_m - n_l + 1)$. When $T_m \ll n$, there is a significant saving. The bigger the value of $(n - T_m)$, the greater the cost saving. When T_m is close to n and n is close to b_n, there is no significant cost saving. Therefore we can conclude that for OODBs that support for time-varying data, when the number of epochs is big enough, i.e. $n \gg b_n$, there is certainly a need of provision of facilities for temporal query processing and optimization.

6. Conclusions

In this article, an extensible approach was explored to processing and optimizing temporal object queries within the object-oriented query processing framework. A temporal query that involves associations of both aggregation hierarchy and time-reference is processed by employing a decomposition strategy. The algorithms for processing the decomposed sub-components of the query have been implemented with stream processing techniques and presented with cost analysis. Simulation results are also provided. Through the description of our query processing algorithms with cost analysis and the provision of simulation results, we have demonstrated that the decomposition strategy provides a convenient means to analyse and evaluate the performance of execution algorithms that take account of the time dimension and provides an opportunity for optimization that makes use of the order of information. Temporal and non-temporal queries can be handled in a uniform way. The temporal optimizer plays a role only when time-related operations or temporal predicates exist.

Both cost analysis and simulations show that the join time cost is linearly increased with the expansion in the number of the time-epochs (or the time dimension, in the case of a regular TS). Utilising heuristics could result in a significant cost saving. We would also conclude that solely treating a temporal object as a 'blob' object that is managed by the system, but interpreted by the user is not a strategy for temporal support in OODBs. OODBs should provide query facilities to support the query processing and optimization on time-varying data, especially when the number of epochs is big enough.

Future work will include a detailed study of temporal predicate optimization and global optimization.

Acknowledgements

The authors would like to thank the anonymous referees, whose comments and suggestions helped to improve the content and the presentation of the article.

References

[1] E. Bertino, L. Martino, Object-Oriented database systems: Concepts and Architectures, Addison-Wesley, Reading, MA, USA, 1993.
[2] J.A. Blakeley et al., Experiences building the open OODB query optimizer, Proc. of ACM SIGMOD Conf., 1993 pp. 287–296.
[3] F.M. Carrano, Data abstraction and problem solving with C++, Benjamin, Reading, MA, USA, 1995.
[4] T.Y. Cliff Leung, R.R. Muntz, Stream processing: Temporal query processing and optimization, in: A.U. Tansel (Ed.), Temporal Databases: Theory, Design, and Implementation, Benjamin, Reading, MA, USA, 1993, pp. 329.
[5] A. D'Andrea, P. Janus, UNISQL's next-generation object-relational database management system, SIGMOD RECORD, 25 (3) (1996) 70–76.
[6] U. Dayal, G.T.J. Wuu, A uniform approach to processing temporal queries, Proc. of the 18th Int. Conf. on VLDB, Canada, 1992, pp. 407–417.
[7] S. Ginsburg, A temporal data model based on time sequences, in: A. U. Tansel (Ed.), Temporal databases: Theory, Design, and Implementation, Benjamin, Reading, MA, USA, 1993, pp. 248.
[8] G. Gardarin et al., Cost-based selection of path expression processing algorithms in object-oriented databases, Proc. of the 22nd Int. Conf. on VLDB, Mumbai, India, 1996, pp. 390–401.
[9] W. Kim, Next-generation database systems: objects and beyond, Proc. of IISF/ACM Japan Int. Symposium, Computers as our Better Partners, World Scientific Publishing, Singapore, Tokyo, Japan, March 1994, pp. 188–196.
[10] W. Kim (Ed.), Modern database systems: the Object Model, Interoperability, and Beyond ACM Press, New York, 1995.
[11] M.T. Ozsu, J.A. Blakeley, Query processing in object-oriented database system, in: W. Kim (Ed.), Modern Database Systems: the Object

Model, Interoperability, and Beyond, ACM Press, New York, 1995, pp. 146.
[12] N. Pissinou, et al., Towards an infrastructure for temporal databases: report of an invitational ARPA/NSF workshop, SIGMOD RECORD, 23 (1) (1994) 35–51.
[13] A. Segev, Join processing and optimization in temporal relational databases, in: A.U. Tansel (Ed.), Temporal Databases: Theory, Design, and Implementation, Benjamin, Reading, MA, USA, 1993, pp. 356.
[14] A. Segev, et al., Report on the 1995 international workshop on temporal databases, SIGMOD RECORD, 24 (4) (1995) 46–52.
[15] P. Seshadri et al., The design and implementation of a sequence database system, Proc. of the 22nd Int. Conf. on VLDB, India, 1996, pp. 99–110.
[16] E.J. Shekita, M.J. Carey, A performance evaluation of pointer-based joins, Proc. of ACM SIGMOD Conf., Atlantic, NJ, May 1990, pp. 300–311.
[17] R. Snodgrass, Temporal object-oriented databases: a critical comparison, in: W. Kim (Ed.), Modern Database Systems: the Object Model, Interoperability, and Beyond, ACM Press, New York, 1995, pp. 386.
[18] D.D. Straube, M.T. Ozsu, Queries and query processing in object-oriented database systems, ACM Trans. on Information Systems, 8 (4) (1990) 387–430.
[19] A.U. Tansel (Ed.), Temporal databases: Theory, Design, and Implementation Benjamin, Reading, MA, USA, 1993.
[20] L. Wang, M. Wing, C. Davis, N. Revell, An algebra for a temporal object data model, LNCS 1134, Database and Expert Systems Applications, (DEXA '96), Proceedings, Springer-Verlag, Zurich, Switzerland, September 9–13, 1996, pp. 667–677.
[21] L. Wang, M. Wing, C. Davis, N. Revell, Query processing in object-oriented databases, Proc. of 13th European Meeting on Cybernetics and Systems Research, 9–12 April 1996, Vienna, Austria, pp. 803–808.
[22] L. Wang, M. Wing, C. Davis, N. Revell, A uniform framework for processing temporal object-oriented queries, Technology of Object-Oriented Languages and Systems TOOLS 24, Proceedings, IEEE Press, Beijing, China, 1997, pp. 33–42.
[23] L. Wang, M. Wing, C. Davis, N. Revell, Temporal query processing and optimization in object-oriented databases, Database and Expert Systems Applications (DEXA 97), Proceedings, IEEE Press, France, August, 1997, pp. 474–481.

Lecture Notes in Computer Science 1134

Roland R. Wagner, Helmut Thoma (Eds.)

Database and Expert Systems Applications

7th International Conference, DEXA '96
Zurich, Switzerland, September 1996
Proceedings

DEXA 96

Springer

An Algebra for a Temporal Object Data Model

L. Wang, M. Wing, C. Davis, N. Revell

School of Computing Science, Middlesex University
Bound Green Road, London N11 2NQ, United Kingdom
email: {lichun1, michael47, colin11, norman}@mdx.ac.uk

Abstract: In this paper, we present a temporal object data model, which has been adapted from the unified model of OODB and RDB in UniSQL/X so that a time dimension can be easily added to form temporal relational-like cubes but with aggregation and inheritance hierarchies. A query algebra, that accesses objects through the associations of aggregation, inheritance and time-reference, is thereby defined. Due to the adaptation of the unified model of RDB and OODB, the temporal object data model supports both homogeneity and heterogeneity in the time dimension, and the algebra reflects the spirit of both temporal relational algebra and object algebra. Data query examples through "The Wood Panel Deformation Measurement Database" illustrate algebraic operations and a brief evaluation of algebra has been given.

Keywords: Object-oriented databases, temporal object, query algebra.

1 Introduction

Temporal properties play an essential role in many real world applications and therefore a recent trend in data modelling is the representation of temporal aspects in database schema and the support of the corresponding data manipulation facilities directly by a database management system [21,18,19]. The vast majority of research on temporal database systems is the incorporation of time elements into relational and pseudo-relational database models [13]. However, the widely recognised semantic limitations of relational databases (RDBs) suggest that they are not suitable for advanced database applications [2,10,11,7,22,23]. Object-oriented technology forms a good basis for a rich data model for the advanced database applications such as computer-aided design, engineering systems, artificial intelligence, multimedia, etc., but little work has been reported on time in object-oriented databases (OODBs), compared to temporal relational models [13]. Management of temporal data is one of the key challenges that today's OODBs need to address [10,11,22]. To meet this challenge, the project on query processing in temporal object-oriented databases has been carried out at Middlesex University.

An OODB is a database system based on object-oriented data model concepts. One approach in introducing time into an object data model is to extend the semantics of a pre-existing snapshot model to incorporate time directly [14]. However, there is currently no commonly accepted object data model, nor is there a commonly accepted object algebra [12]. We adopt the unified model of RDB and OODB from UniSQL/X [10,11] as a snapshot object data model, and then incorporate within it with a time dimension. A temporal object query algebra is thereby defined, making use of the research results of temporal extensions to RDBs for OODBs.

The remainder of this paper is organised as follows. Section 2 describes the adaptation of the unified data model of RDB and OODB with the inclusion of a time dimension.

Section 3 defines the algebra. Section 4 gives query examples and a brief evaluation. Section 5 discusses related work and Section 6 includes concluding remarks.

2 The Model

2.1 The Unified Model of OODB and RDB

The unified data model of RDB and OODB from UniSQL/X [10,11] extends the relational data model in three important ways, each reflecting a key object-oriented concept: (1) nested predicates; (2) inheritance; (3) methods. In this model, "relation" equates to "class", "tuple of relation" equates to "instance of a class", "column" equates to "attribute", "procedure" equates to "method", "relation hierarchy" to "class hierarchy", "child relation" to "subclass", "parent class" to "superclass", and "nested relation" to "aggregation hierarchy". Allowing a column of a relation to hold a tuple of another relation directly leads to a nested relation. Allowing the users to attach procedures to a relation and to have the procedures operate on the column values in each tuple achieves the combination of data with program. Allowing the users to organise all relations in the database into hierarchy, such that between a pair of relations P and C, P is made the parent of C, if C takes (inherits) all columns and procedures defined in P (besides those defined in C), the relational model integrates the object-oriented concept of inheritance. This model is an object-oriented model and it is adopted as a basis for extending temporal extensions of RDBs to OODBs.

Besides, we preserve the basic object concepts such as "any real-world entity is uniformly modelled as an object", "each object is associated with a unique identifier", etc., so that *heterogeneity* in the time dimension and the *grouped completeness* of algebra can be maintained.

2.2 A Temporal Object

In order to address the temporal issues, we adopt the ideas of using temporal sets (temporal elements) as timestamps [8,21] and associating the lifespan at both attribute and tuple level [5]. Let $T = \{..., t_0, t_1, ...\}$ be a set of times, at most countably infinite, over which is defined the linear (total) order $<^T$, where $t_i <^T t_j$ means t_i occurs before (is earlier than) t_j. For the sake of simplicity, we can assume that T is isomorphic to the set of natural numbers. Any subset of T is called a temporal set. A temporal set can be represented as a union of disjoint time intervals. The most basic property of temporal sets is that they are closed under finite unions, intersections, and complementation. That is, if T_1 and T_2 are temporal sets, then so are $T_1 \cup T_2$, $T_1 \cap T_2$, $T_1 - T_2$, *and* $\neg T_1$.

We incorporate the temporal dimension at the object level. If an object o exists in a certain period of time, which is a subset of T (i.e., the temporal set), this period is called the object's lifespan, denoted as $L(o)$ for the object o. In order to support for derived lifespans, we allow the usual set-theoretic operations over lifespans. That is, if L_1 and L_2 are lifespans, then so are $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, and $\neg L_1$.

A temporal object is defined as an ordered pair $<L(o), o>$, where $L(o) \in T$ and o is any real word entity, which asserts that the object o is valid for its lifespan $L(o)$. For a constant object, it may be represented with no timestamp where its time reference is implied as

$L(o)$. It can also be represented with explicit time references as a temporal object $<L(o), o>$. Further, in the real world, even though the properties of an object change with time, we think of it as the same object [16]. We treat an object such as an attribute value as a function of time. This helps us prevent fragmentation of an object description. Even if the time domain is physically fragmented, as in *[11,27]* and *[30,40]*, the attribute value can stay in the database as a single logical entity. We express this as $<L(o), o(t)>$ (or simply $<L(o), o>$ without causing a confusion). In general, the word *object* will refer to *a temporal object* within this paper.

In OODBs, every real world entity is uniformly modelled as an object that is grouped into a class (relation) and interrelated to other objects through association. Now we isolate a class (relation) C from its association relationship, as shown in **Fig. 1**. The $value_{m,n}$ is an object with lifespan $l_{m,n}$. The $tuple_m$ is also an object, its lifespan is denoted as $L(t_m)$. We have

$$L(t_m)=l_{m,1}\cup l_{m,2}\cup\ldots\cup l_{m,n}$$

The lifespan of attribute A_n is

$$L(A_n)=l_{1,n}\cup l_{2,n}\cup\ldots\cup l_{m,n}$$

The lifespan of relation C is

$$L(C)=L(A_1)\cup L(A_2)\cup\ldots\cup L(A_n)=L(t_1)\cup L(t_2)\cup\ldots\cup L(t_m)$$

Relation	A_1	A_2	...	A_n
$tuple_1$				
$tuple_2$				
...	...	...	...	
$tuple_m$				$value_{m,n}$

Fig. 1. Interaction of Tuple Lifespan and Attribute Lifespan

It is obvious that

$$l_{i,j}=L(t_i)\cap L(A_j)$$

This implies that there is no value for an attribute in a tuple for any moment in time outside the intersection of the life spans of the tuple and the attribute. Obviously our temporal object model can support a completely heterogeneous temporal dimension, but at the cost of maintaining a distinct lifespan for each value. This is important because homogeneity is sometimes difficult to maintain, although homogeneity is necessary as no timeslices of a homogenous relation produce null values [14].

It is possible to refer to the components of a temporal object. For a temporal *object* $o=<T,o>$, $o.\upsilon$ and $o.T$ refer to its value and temporal set components, respectively. Sometimes we omit υ, i.e., $o.\upsilon=o$, (or $o.\upsilon(t)=o(t)$) to refer to the value of the object o without causing a confusion. Let A be the name of an attribute that can take a temporal object for its values, then $A.\upsilon$ and $A.T$ represent names for the value and temporal set components of the attribute A. Further, the same notation may be applied to class (relation) C. If C is a temporal relation, then $C.\upsilon$ and $C.T$ represent names for the value and temporal set components of the class C. Thus a 2-dimensional relation (class) "table" becomes a 3-dimensional "cube", as shown in **Fig. 2**, which is also a set.

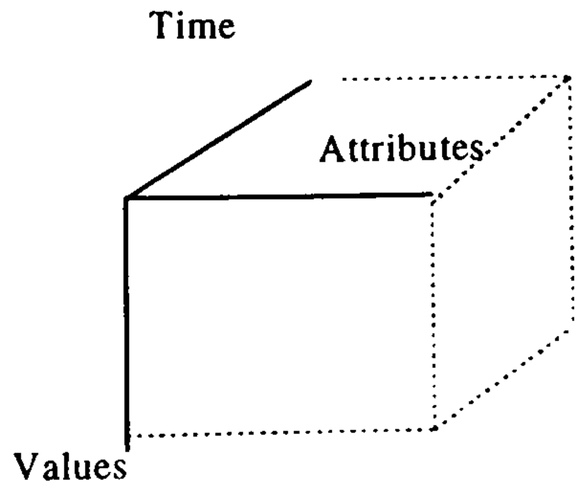


Fig. 2. A 3-Dimensional Class

If the domain of attribute A_i of class C is another class C', then implicitly, $L(A_i)=L(C')$. If

class C is a subclass of class C', then $L(C)=L(C')$. Moreover, if a database consists of n classes (relations) C_1, C_2,...C_n, the lifespan of the database schema is $L= L(C_1)\cup L(C_2)\cup...\cup L(C_n)$.

2.3 A Case Study

Using this model, a database is designed for "The Wood Panel Deformation Measurement System"[3,4,15,23], which brings together the results of recent research in art observation, electro-optical image processing and advanced database management in order to increase knowledge and understanding of deformation and cracking of wood panel paintings (which lead to paint loss) caused by changes in ambient conditions. A deformation analysis of movement occurring in wood panels was required by the Hamilton Kerr Institute of the Fitzwilliam Museum, University of Cambridge, where 74 wood panels used for supporting fine art painting were tested. An automated 3-D measuring system using photogrammetric and machine vision techniques has been developed at City University. The panels to be measured were divided according to wood type: linden; oak; poplar; and Scots pine. Each type was supported by a number of different reinforcement type to give 74 panel reinforcement combinations. An array of retro-reflective targets were placed on each test panel. The number and disposition of the targets on each test panel varied from 175 to 464 according to the pattern of auxiliary supports. The total number of epochs (the number of sequential images) at which the initial set of images of the 74 test panels would be acquired was 25 (i.e. 25 humidity levels at different time). For each epoch, there were about 400 images in total to be grabbed by 5 cameras at different positions, which occupied about 170M storage. Therefore over 10,000 images were grabbed and processed. The average number of targets on each test panel was 250, resulting in a total of 2,500,000 targets to be processed.

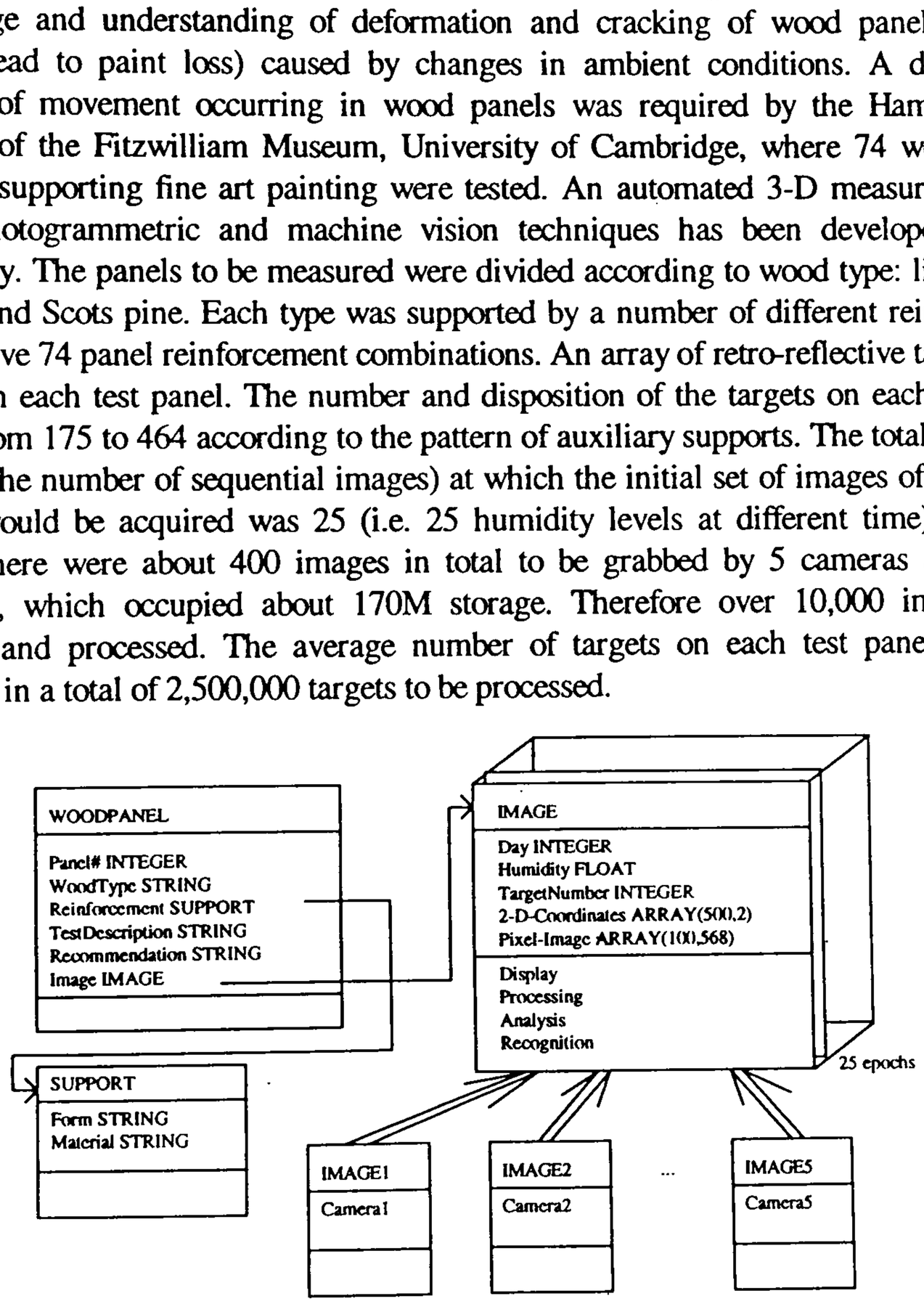


Fig. 3. Database Schema of "Wood Panel Deformation Measurement System"

The database schema is shown in a simplified form in **Fig. 3**, which is a collection of classes. Each node is a class (synonymously a relation). A node is subdivided into three levels, the first of which contains the name of the class, the second the attributes and the third the methods or procedures attached. Two nodes C and C' can be connected by two types of arcs: (1) a thin arc, indicating that C' is the domain of an attribute of A of C, or

that C' is the class of the result of a method of C (resulting in the aggregation hierarchy); (2) a thick arc, indicating that C is the superclass of C' (resulting in the inheritance hierarchy). Arrows indicate the directions of connection. We use the epoch to represent the time dimension t, where $t \in L$ and $L=\{1,2,...,25\} \in T$ is a lifespan. t can be viewed as a normalised time and may (or may not) be related to the attribute *Day*. *IMAGE1* to *IMAGE5* are subsets of *IMAGE*, so they inherit all three dimensional attributes from *IMAGE*. <*L, IMAGE1*> represents the *25* sequential images grabbed by *Camera 1*. *IMAGE1(3)* represents the 3rd image grabbed by *Camera 1*.

3 Query Algebra

From the algebra point of view, a temporal OODB can be viewed as a collection of temporal objects, grouped together in classes (relations) and interrelated through associations: aggregation, generalisation and time-reference. Each temporal relation can be viewed as a 3-dimensional cube. If the existing hierarchical structure of "inheritance" hierarchy and "aggregation" hierarchy between classes is not considered, the structure of queries is essentially the same in both the RDB and OODB paradigm. Temporal aspects only transform some tables or even only some attributes to cubes (many relational databases already support timestamps). We therefore have a common base to expand (temporal) relational algebra to temporal object algebra.

Basically, the standard relational algebra provides a unary operator for each of its two dimensions: *select* for the value dimension and *project* for the attribute dimension. An object operator allows the predicate of the *select* operation on a contiguous sequence of attributes along a branch of class-aggregation hierarchy. We will add more operators for the added time dimension. The algebra is defined against set objects, and "a class" = "set objects" = "relation" concept is preserved so that it can readily take advantage of inheritance and enable application to automatically reach any existing objects of interest, without acquiring explicit references to those objects [24].

3.1 Predicate

There are three types of predicate: a simple predicate, a nested predicate and a temporal predicate.

A simple predicate is of the form <*attribute-name operator value*>. The value may be an instance of a primitive class (type) (e.g., string, integer, etc.) or an object identifier (OID) of the instance of some class. The latter is important because it may be used for testing the object equality, that is, equality of referenced objects. The operator is a scalar comparison operator (=, <, >, etc.) or a set comparison ($\in$, $\subset$, $\subseteq$, set-equality, etc.).

A nested predicate is a predicate on a contiguous sequence of attributes along a branch of the class-aggregation hierarchy of a class. Path-expression [2] is used to express the nested predicate. A path is defined as

$$P=C_1.A_1.A_2....A_n \ (n \geq 1)$$

where C_1 is the class in the database schema, A_1 is an attribute of class C_1, A_i is an attribute of a class C_i such that C_i is the domain of the attribute A_{i-1} of class C_{i-1}, $1<i \leq n$. For example, *WOODPANEL.Reinforcement.Form* is a path, and *WOODPANEL*.

Reinforcement.Form = "lattice" is a predicate.

A temporal predicate is a predicate referred to temporal set along the time dimension. There are two types of temporal predicates: a simple temporal predicate and a nested temporal predicate. A simple temporal predicate can be expressed as <*temporal-set operator value*>. The operator could be <, ≤ , =, >, ≥ , and the combination, representing time *before, while/when, after, during*, etc. A nested temporal predicate can be expressed by integrating the path-expression into a simple temporal predicate. If *o* is a attribute name or a path-expression or a predicate, we use function *When* $\varpi(o)$ to express the temporal domain of *o* (we will give a formal definition later). For example, we use the following expression to refer to epoch 4 at which the image's humidity level is, say, *30 % rh:* $\varpi(WOODPANEL.Image.Humidity = 30\%\ rh) = 4$.

We also extend the path-expression [2] to express the nested predicate with a time dimension, using the enhanced path-like expression to refer to the value components of a temporal object. For example, a path-like expression which refers to an image's humidity level=30 % rh is *WOODPANEL. Image.Humidity(t)=30 % rh*, say, *t=4*. More generally, we use $o|_{T1}$ to denote the restriction of *o* to the temporal set T_1. For example, $WOODPANEL.Image.Humidity|_{t=4} = 30\ \%\ rh$, and $WOODPANEL.Image.Humidity\ |=_{4,5,6} = \{30\ \%\ rh,\ 40\%\ rh,\ 50\%\ rh\}$.

A method may be used for any part of a predicate, that is, as the attribute-name, the operator, or the value. We could think *of* $\varpi(O)$ and $o|_{T1}$ as methods as well.

If P_1 and P_2 are predicates, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $\neg P_1$. These constitute complex predicates.

3.2 Identity and Equality

Identity is a property of an object that distinguishes the object from all others. It is important to distinguish between the following different types of equality.

1) Identity equality of objects: two objects *o* and *o'* are identity equal if they are the same object (i.e., they have the same OID), denoted as "==". That is, *o==o'* if *OID(o)=OID(o')*.

2) Value equality of objects: two temporal objects are equal if the values and the temporal sets of all their attributes are recursively equal, denoted by "=". That is, two temporal objects *o* and *o'* are equal if $o.T(A_i)=o'.T(A_i)$ and $o.\upsilon(A_i)=o'.\upsilon(A_i)$ (or $o.\upsilon(A_i)(t) = o'.\upsilon(A_i)(t)$ at every *t*). The term value equality is analogous to the snapshot equivalent/weekly equivalent in temporal RDBs that states that two tuples are snapshot equivalent or weekly equivalent if the snapshots of the tuples at all times are identical. Two identical objects are also equal whereas the reverse is not true.

3) Shallow-equality: two objects are shallow-equal if their attributes share the same value and the same references, and their corresponding temporal sets are equal although they are not identical, denoted as $=_I$.

Duplication in set membership is based on object identity, i.e., a set will not contain two objects with the same identifier. There are many cases in algebra that implicit

comparisons are made using identity equality. There are also some cases that the comparison is made by value equality. Shallow equality is required for join operation.

3.3 Set-Theoretical Operations

If O_1, O_2 are temporal object sets, then the sets operators Union, Intersection and Difference are identical to Codd's corresponding relational operators:

Difference $O_3=O_1-O_2=\{o \mid o\in O_1 \wedge \neg o\in O_2\}$ *where* $L(O_3)=L(O_1)-L(O_2)$
Union $O_3=O_1\cup O_2=\{o \mid o\in O_1 \vee o\in O_2\}$ *where* $L(O_3)=L(O_1)\cup L(O_2)$
Intersection $O_3=O_1\cap O_2=\{o \mid o\in O_1 \wedge o\in O_2\}$ *where* $L(O_3)=L(O_1)\cap L(O_2)$

Like relational algebra, the duplication in the resulting set is eliminated.

3.4 Special Operations

Select $\sigma_P O$ selects the elements *"o"* of set O such as the predicate $P(o,t)$ holds.

$$\sigma_P O=\{o \mid o \in O \wedge P(o,t)\}$$

Select is a hybrid operation, reducing a class (relation) in both the value and the temporal dimension. If no predicate is referred to time, it then reduces the class along the value dimension.

Map $g:O_1\rightarrow O_2$: for the type of objects in O_1 (i.e., $o\in O_1$), g returns an object of type of O_2 (i.e., $g(o)\in O_2$).

$$g:O_1\rightarrow O_2=\{g(o) \mid o\in O_1\}$$

Project $\pi_{<A1,...,Ai>}O$ extends *Map* by allowing the application of many functions to an object, thus supporting the creation and maintenance of selected relationships between objects.

$$\pi_{<A1,...,Ai>} O =\{<A_1:g_1(o),..., A_i:g_i(o)> \mid o\in O\}$$

where O is of type *set [T]*, the A_i's are unique attribute names, and each g_i takes a single input of type T and returns an object of type T_i. g_1... g_i are similar to g. If $g_i=1$, it returns OID of the domain object of A_i unless A_i is atomic. We retain $g_i=1$ (unless it is specified not) so that we keep our project operator on a set of objects (relation) like the relational project. Therefore the project operator, when applied to class (relation) O, removes from O all but a specified set of attributes. As such it reduces a relation along the attribute dimension.

Time-slice $\xi_{L1}(O)$ defines the relation (set of objects) containing those objects derived by restricting each objects in the operand relation to those times specified by L_1.

$$\xi_{L1}(O)=\{o \mid \forall t\in(L_1\cap L(o))[o(t)\in O]\}$$

Obviously the lifespan of $\xi_{L1}(O)$ is $L_1\cap L(o)$. So the *time-slice* reduces the relation solely along the temporal dimension. If L_1 equals to a time point t_1, i.e., $T_1=t_1$, then $\xi_{L1}(O)$ represents the event $o(t_1)$ happened at t_1.

Offset $\gamma(O, l)$ "shifts" a snapshot relation at t_1 by the number of positions specified by the offset.

$$\gamma(O(t_1), l)= O(t_1+l)$$

When $\varpi(O)$ defines an operator that maps a relation (set of objects) O to its temporal set:

$\varpi(O)=L(O)$

The result of *when* is a time value; it can serve as a parameter or a predicate to those operators, like time-slice, etc.

Join $O_1 \triangleright\!\triangleleft_{P<Ao1,Ao2>} O_2$ is an explicit join operator used to create relationships between objects from two collections in the databases. Unlike relational joins, in which the domains of the join attributes must be identical, we require the join attribute to only be compatible [9]. Two attributes A_i and A_j are compatible, if the value domain and the temporal domain of A_i is identical to those of A_j (or a superclass or subclass of the domain of A_j). Shallow-equality could express this compatibility. Although join attributes are compatible, they have different OIDs. So the join we defined is essentially a θ-*join* as in relational algebra.

$$O_1 \triangleright\!\triangleleft_{P<Ao1,Ao2>} O_2=\{<A_{o1}:o_1, A_{o2}:o_2> \mid o_1 \in O_1 \wedge o_2 \in O_2 \wedge P(o_1,o_2)\}$$

$Unnest^T(\mu_K O)$ Suppose relation (class) O_1 have the scheme $<A_1,...,A_n>=<o_1(1),...,o_1(n)>$ and the scheme of $o_1(k)$ (i.e., $<m,A_k>$) be $<A_k.1,...,A_k.m>, 1\le k\le n$, then $Unnest^T$ is defined as

$$O_2 = \mu_K O_1$$
$$= \{o_2 \mid o_2(i)=o_1(i) \text{ for } 1\le i\le k-1 \wedge o_2(i)=o_1(i+1) \text{ for } k\le i\le n-1$$
$$\wedge\, o_2(i)=A_k(i-n+1) \text{ for } n\le i\le n+m-1\}$$

$Nest^T(\upsilon_y O)$ Let the relation (class) O_1 have the scheme $<A_1,...,A_n>=<o_1(1),...,o_1(n)>$, $y=\{i_1,i_2,...,i_k\}$ is a subset *of* $\{1,2,...,n\}$, and $x=\{1,2,...,n-y\}$. $Nest^T$ has the scheme of $<B_1,..., B_{n-k+1}>= <o_2(1),...,o_2(n-k+1)>$, where $o_2(j)=o_1(r)$ for $1\le j\le n-k$, $r\in x$, and $o_2(n-k+1)$ has the scheme relation: $<B_{n-k+1}.1,...,B_{n-k+1}.k>$. Similar to the unnest operator, the nested component is placed at the last column of $\upsilon_y O$. So $Nest^T$ is defined as

$$O_2=\upsilon_y O_1=\{o_2 \mid o_2(j)=o_1(r) \text{ for } 1\le j\le n-k,\ r\in x$$
$$\wedge\, o_2(n-k+1)=\{z \mid \exists o\ (o\in O_1 \wedge o(r)=o_1(r) \text{ for } r\in x \wedge z(j)=o(i_j) \text{ for } 1\le j\le k)\}\ \}$$

Besides the above database operators, we can easily define some aggregate operators. Suppose *Agg-func* is one of functions Avg, Min, Max, and Sum, then $Agg\text{-}func_{T1}(O)$ returns the function value over the specified period T_1. Null records in tuples are ignored if there is at least one non-Null records otherwise the output is Null record.

4 Data Query Examples and Evaluation

In this section the applicability of our algebra to data queries is illustrated through following query examples and a brief evaluation is made at the end of the section.

Query 1 "Find all the wood panels whose type is 'pine' and was reinforced in form of 'lattice' by 'oak, glued' ". This query did not involve any temporal aspect. We can treat it like a constant object query while its lifespan implies the same as the lifespan of temporal objects. We express this query in the following algebra:

$$O_{W1}=\sigma_{P1}\, WOODPANEL$$
$$=\{o \mid o\in WOODPANEL \wedge WOODPANEL.WoodType="pine"$$

$\wedge$ *WOODPANEL.Reinforcement.Form="lattice"*
$\wedge$ *WOODPANEL.Reinforcement.Material="oak, glued"}*

For the attributes of Reinforcement and Image, this query only gives the OIDs instead of all the SUPPORT objects and IMAGE objects (sequential images).

Query 2 "Decide when the humidity level of above selected wood panel is 30% rh".

$Ot=\varpi(o \in O_{w1} \wedge o.IMAGE.<Humidity.\upsilon>=30\ \%\ rh)$

Query 3 "Select the wood panel image whose wood type is 'pine' and reinforced in form of 'lattice' by 'oak, glued', while its humidity level is 30 % rh (at the 4th epoch) and grabbed by Camera 1". We have the following algebra:

$O_{W2} = \pi_{<WOODPANEL.Image>} (\sigma_{P1}\ WOODPANEL) = \pi_{<WOODPANEL.Image>}\ O_{W1}$
$O_{W3} = Map: O_{W2} \rightarrow \{o \mid o \in IMAGE1\}$
$O_{W4} = \sigma_{P2}\ (O_{W3}) = \{o \mid o \in IMAGE1 \wedge Ot\}$

Query 4 "Find the humidity level values of above selected IMAGE1 which appeared before time Qt". This query involves the temporal reasoning. We apply the following algebra operators to support this reasoning.

$O_{W5} = \pi_{<IMAGE1.Humidity>}\ (O_{W3})$
$O_{W6} = \sigma_{p3}\ (O_{W5}) = \{o \mid o \in Humidity \wedge t < Ot\}$

Query 5 "Get the average humidity level of above selected panel".

$Agg\text{-}func_{T1}\ O_{W6} = Avg_{T1=[1,\ Qt]}\ Ow6$

The closure property states that *output from one operation can become input to another* [7]. Our algebra imposes operators (except *when* $\varpi(o)$, we already treat it as a method) on relations (sets of objects). So the output is also a relation. In this sense our algebra is closed. Besides, because our algebra supports object identity, it is also polymorphic in the sense that it is defined across all "objects"[24].

These models which employ tuple-time-stamping are termed *temporally ungrouped* whereas those models that employ complex attribute values bearing the temporal dimension are termed *temporally grouped* [14]. While the expressive power of *ungrouped complete* was generally accepted as a desirable property for TSQL, there were considerable concerns on *grouped complete* [14]. The benefit of being *grouped complete* is that it supports a rather strong notion of the "history of an attribute". For example, one can talk about "Panel #1's humidity history" as a single object, and ask to see it, or define constraints over it, etc. In temporal RDBs, as stated in [5,14], there is no algebra that has been shown to be *grouped complete*. In our temporal object data model, every object is associated with a OID. If every OID is maintained in a database (in some data models, primitive entities such as integers, or characters, are represented by values and have no OID associated with), then our algebra will be *grouped complete*.

5 Related Work

Although temporal databases have been an active area of research for over fifteen years, there is no commonly accepted consensus data model, nor is there well-accepted temporal

database algebra [14]. It is advocated [14] that a very simple conceptual data model is adopted that captures the essential semantics of time varying relations, but has no illusions of being suitable for representation, storage, or query evaluation. Our work pursues this.

There are quite a few reports on defining a temporal relational data model and algebra. Clifford [5] associated lifespans at both attribute and tuple level as timestamps to define the historical relational data model and algebra. Tansel [21] adopted temporal sets for timestamps and a temporal atom for a timestamped object. Gadia *et al.* [16] provided a parametric data model, treating the attribute value as a function of time. These features of the above work have been integrated and applied to our data model and algebra. But because our data model is a temporal object data model, the *heterogeneity* in time dimension and *grouped completeness* of algebra can be supported.

Over two dozen proposals have been made for an object algebra [12]. No query algebra thus defined is based on any unified model of RDBs and OODBs, although it has been claimed that an object algebra should extend relational algebra consistently [21,17]. None of these object algebra consider the temporal dimension. Our temporal object algebra reflects the spirit of object algebra [17,20,6,1]. But in addition to support the access through aggregation and inheritance associations, our algebra accesses objects through time dimension. These are embodied in the enhanced nested predicates and (nested) temporal predicates.

6 Conclusions

In this paper, we have presented a temporal object data model and its algebra. The adaptation of the unified model of RDB and OODB by adding a time dimension to form the relational-like cubes but with aggregation and inheritance associations provides a basis to extends techniques of defining temporal relational algebra for developing the temporal object algebra. The temporal object algebra will become an object algebra when the time dimension is not taken into account, and the object algebra extends the relational algebra consistently. On the other hand, the modelling capability of OODBs that every real world entity can be uniformly modelled as an object, makes it easy to model the temporal aspects. The temporal object we defined can be used to represent any object with a time dimension, such as representing an attribute as a function of time which avoids fragmentation of an object description. In addition, it can support both homogeneity and heterogeneity in the time dimension. This is important because homogeneity sometimes becomes difficult to maintain. Furthermore, the *grouped completeness* of the algebra can be maintained.

Future work will extend the query evaluation and optimization techniques developed for temporal RDBs and OODBs to temporal OODBs.

Acknowledgements

Authors would like to thank J. Chen of City University, UK for his cooperation with the "Wood Panel Deformation Measurement System".

References

1 R. Alhajj and M. E. Arkun. A query model for object-oriented databases. *Proc. of 9th Int. Conf. on Data Engineering*, p163-172, 1993.

2 E. Bertino and L. Martino. *Object-Oriented Database Systems: Concepts and Architectures.* Addison-Wesley Publishers Ltd., 1993.

3 J. Chen, T. A. Clarke and S. Robson. Optimised target matching based on a 3-D space intersection and a constrained search for multiple camera views. *Videometrics III, SPIE Proc.*, Vol 2350, p 324-335, 1994.

4 A. Clarke, *et al.* Automated three dimensional measurement using multiple CCD camera views. *The Photogrammetric Record*, 15(85): 27-42, 1995.

5 J. Clifford, *et al.* The historical relational data model (HRDM) revisited. *Temporal Databases: Theory, Design, and Implementation* (edited by A.U.Tansel, *et al.*), p6-27. Benjamin/ Cummings, 1993.

6 S. Cluet and C Delobel. Towards a unification of rewrite-based optimization techniques for object-oriented queries. *Query Processing for Advanced Database Systems (edited by* J. Freytag, *et al.).* Morgan Kaufmann, 1994.

7 C. J. Date. *An Introduction to Database Systems*, 6th Edition. Addison-Wesley, 1995.

8 S. K. Gadia. A homogeneous relational model and query languages for temporal databases. *ACM Trans. on Database Systems,* 13(4): 418-448, 1988.

9 W. Kim. A model of queries for object-oriented databases. *Proc. of the 5th Int. Conf. on VLDB*, p423-432, Amsterdam, 1989.

10 W. Kim. Object-oriented databases systems: promises, reality, and future. *Proc. of the 19th Int. Conf. on VLDB*, p 676-687, Dublin, Ireland, 1993.

11 W. Kim. Next-generation database systems: objects and beyond. *Proc. of IISF/ACM Japan Int. Symposium, Computers as our Better Partners,* p188-196, Tokyo, Japan, March 1994. World Scientific Publishing.Singapore.

12 M. T. Ozsu and J. A. Blakeley. Query processing in object-oriented database system. *Modern Database Systems: the Object Model, Interoperability, and Beyond,* (edited by W. Kim), p146-174. ACM Press, 1995.

13 N. Pissinou, *et al.* On temporal modelling in the context of object databases. *SIGMOD RECORD,* 22(3): 8-15, Setp. 1993.

14 N. Pissinou, *et al.* Towards an infrastructure for temporal databases: report of an invitational ARPA/NSF workshop. *SIGMOD RECORD,* 23(1): 35-51, March 1994.

15 S. Robson, *et al.* Seeing the wood from the trees-an example of optimised digital photogrammetric deformation detection. *ISPRS Intercommission Workshop: From Pixels to Sequences-Sensors, Algorithms, and Systems*, Vol 30/5W1, p379-384, 1995.

16 S. K. Gadia and S. S. Nair, Temporal databases: a preclude to parametric data, *Temporal Databases: Theory, Design, and Implementation* (edited by A. U. Tansel, *et al.*), p28-66. Benjamin/ Cummings Publishing, 1993.

17 G. M. Shaw and S. B. Zdonik. A query algebra for object-oriented databases. *Proc. of 6th Int. Conf. on Data Engineering*, p 154-162, 1990. IEEE.

18 R. Snodgrass. Temporal object-oriented databases: a critical comparison. *Modern Database Systems: the Object Model, Interoperability, and Beyond,* (edited by W. Kim), p 386-408. ACM Press, 1995

19 M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering*, 2(1): 125-142, 1990.

20 D. D. Straube and M. T. Ozsu. Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems*, 8(4): 387-430, 1990.

21 A. U. Tansel. A generalized relational framework for modelling temporal data. *Temporal Databases: Theory, Design, and Implementation* (edited by A. U. Tansel, *et al.*), p183-201. Benjamin/ Cummings Publishing, 1993.

22 L. Wang, M. Wing, C. Davis and N. Revell. Query processing in object-oriented databases. *Proc. of 13th European Meeting on Cybernetics and Systems Research, p803-808,* April 9-12, 1996, Vienna, Austria.

23 L. Wang, M. Wing, C. Davis, N. Revell and J. Chen. Data modeling and management in sequential image databases: a temporal object-oriented approach. *IEE Colloquium Digest on Intelligent Image Databases*, p1/1-6, 96.

24 L. Yu and S. L. Osborn. An evaluation framework for algebraic object-oriented query models. *Proc. of 7th Int. Conf. on Data Engineering,* p 670-677, 1991. IEEE.