An XML-based implementation of the parametric model for ad-hoc query of temporal and spatiotemporal data

Seo-Young Noh
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
An XML-based implementation of the parametric model for ad-hoc query of temporal and spatiotemporal data

by

Seo-Young Noh

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Shashi K. Gadia, Major Professor
William J. Gutowski
Suraj C. Kothari
Glenn R. Luecke
Wallapak Tavanapong

Iowa State University
Ames, Iowa
2006
Copyright © Seo-Young Noh, 2006. All rights reserved.
Graduate College
Iowa State University

This is to certify that the doctoral dissertation of

Seo-Young Noh

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Committee Member
Signature was redacted for privacy.
Committee Member
Signature was redacted for privacy.
Committee Member
Signature was redacted for privacy.
Committee Member
Signature was redacted for privacy.
Major Professor
Signature was redacted for privacy.
For the Major Program
DEDICATION

To Jungsoo Noh and Inyeop Lee,

their endless sacrifice for family will live in my heart

forever ...
TABLE OF CONTENTS

LIST OF TABLES ................................................................. ix
LIST OF FIGURES ............................................................... x
ACKNOWLEDGEMENTS ........................................................... xiii
ABSTRACT ........................................................................... xv
CHAPTER 1 INTRODUCTION ..................................................... 1
  1.1 Overview ........................................................................ 1
    1.1.1 Dimensional Databases ............................................. 1
    1.1.2 Parametric Data Model .............................................. 2
  1.2 Objectives ....................................................................... 2
  1.3 Approach ......................................................................... 3
    1.3.1 Justification of XML-based Implementation ................. 3
    1.3.2 Validation of Parametric Data Model for Temporal Data .. 4
    1.3.3 Validation of Parametric Data Model for Spatiotemporal Data ........................................................................ 4
    1.3.4 Implementation of NC-94 Spatiotemporal Database ........ 5
  1.4 Contributions ................................................................... 6
  1.5 Related Work .................................................................. 7
    1.5.1 Temporal Databases ................................................... 7
    1.5.2 Spatial Databases ....................................................... 8
    1.5.3 Spatiotemporal Databases .......................................... 8
  1.6 Organization .................................................................... 9
CHAPTER 2 OVERVIEW OF PARAMETRIC DATA MODEL

2.1 Introduction ........................................... 11
2.2 Parametric Elements .................................... 12
   2.2.1 Temporal Elements ............................... 13
   2.2.2 Spatial Elements ................................. 13
   2.2.3 Spatiotemporal Elements ......................... 13
2.3 Parametric Relations ................................... 14
   2.3.1 Temporal Relations ............................... 14
   2.3.2 Spatial Relations ................................. 15
   2.3.3 Spatiotemporal Relations ......................... 16
   2.3.4 Dimension Alignment ............................. 17
2.4 Parametric Structured Query Language ................. 17
   2.4.1 Relational Expressions ............................ 17
   2.4.2 Domain Expressions ............................... 18
   2.4.3 Boolean Expressions .............................. 19
   2.4.4 User-Friendliness ............................... 20
   2.4.5 ParaSQL Examples ................................. 22
2.5 Summary ................................................ 23

CHAPTER 3 STORAGE EFFICIENCY OF XML FOR PARAMETRIC DATA MODEL

3.1 Introduction ........................................... 25
3.2 Implementation Platforms ............................... 26
   3.2.1 Conventional Databases ......................... 26
   3.2.2 Object-Oriented Storages ......................... 27
   3.2.3 XML-based Storages .............................. 28
3.3 Data Representations .................................. 28
   3.3.1 Relational Storage ............................... 28
   3.3.2 Object-Oriented Storage ......................... 29
3.3.3 XML-based Storage ........................................... 31
3.4 Storage Costs ................................................. 32
  3.4.1 Notations .................................................... 32
  3.4.2 Cost of Relational Storage ............................... 32
  3.4.3 Cost of Object-Oriented Storage ......................... 37
  3.4.4 Cost of XML-based Storage .............................. 38
3.5 Storage Cost Comparisons ................................... 38
3.6 Summary and Discussion ..................................... 44

CHAPTER 4 XML-BASED IMPLEMENTATION OF PARAMETRIC TEM-
PORAL DATABASE .................................................... 45
  4.1 Introduction .................................................. 45
  4.2 Formalization of the Parametric Data Model Schema .......... 46
    4.2.1 Definitions ................................................. 47
    4.2.2 Transformation ........................................... 48
  4.3 Dynamic Pagination Algorithm ................................ 49
  4.4 ParaSQL Examples ............................................. 53
  4.5 System Architecture .......................................... 57
    4.5.1 Query Processing Layer .................................... 57
    4.5.2 Query Execution Layer ..................................... 61
    4.5.3 Storage Management Layer ................................. 62
  4.6 Performance Evaluation ....................................... 63
    4.6.1 Test Data Configuration .................................... 63
    4.6.2 Effectiveness of the Model Schema ......................... 64
    4.6.3 Experimental Results ..................................... 65
  4.7 Summary and Discussion ...................................... 66

CHAPTER 5 VALIDATION OF PARAMETRIC DATA MODEL FOR TEM-
PORAL DATA ........................................................... 68
  5.1 Introduction .................................................. 68
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1</td>
<td>Notations</td>
<td>33</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>The test data for the model schema effectiveness</td>
<td>63</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Data setup information</td>
<td>80</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Experimental results</td>
<td>84</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Properties of the NC-94 dataset</td>
<td>108</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 Emp temporal relation ........................................ 14
Figure 2.2 County spatial relation [15] .............................. 15
Figure 2.3 Well spatiotemporal relation [18] .......................... 16
Figure 2.4 BNF of relational expression ............................... 18
Figure 2.5 BNF of domain expression ................................. 19
Figure 2.6 BNF of Boolean expression ............................... 20

Figure 3.1 Emp relation in the parametric temporal data model ...... 29
Figure 3.2 Emp relation .................................................... 30
Figure 3.3 Class templates for Emp relation in the attribute versioning approach .... 31
Figure 3.4 XML representation of Emp relation ........................ 32
Figure 3.5 Time intervals for attributes a and b ..................... 34
Figure 3.6 Combined intervals for attribute a and b .................. 35
Figure 3.7 Storage efficiency for default values ....................... 40
Figure 3.8 Storage efficiency for p variations ......................... 40
Figure 3.9 Storage efficiency for U variations ......................... 41
Figure 3.10 Storage efficiency for average tag size variations .......... 41
Figure 3.11 Storage efficiency for object overhead variations ........ 43
Figure 3.12 Storage efficiency for average attribute sized variation .......... 43

Figure 4.1 Parametric model schema for parametric relation ........ 48
Figure 4.2 size(current) ≥ minimum threshold [35] ............... 51
Figure 4.3 size(current) + size(sibling) ≥ minimum threshold [35] .... 51
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>size(current) + size(sibling) ≥ maximum threshold</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>size(current) + size(sibling) ≥ minimum threshold</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>Final paginated XML document</td>
<td>53</td>
</tr>
<tr>
<td>4.7</td>
<td>System architecture</td>
<td>58</td>
</tr>
<tr>
<td>4.8</td>
<td>A parse tree for the ParaSQL query</td>
<td>59</td>
</tr>
<tr>
<td>4.9</td>
<td>Expression tree and its XML representation</td>
<td>60</td>
</tr>
<tr>
<td>4.10</td>
<td>Schema effectiveness (logarithmic scale)</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>Interval-based Emp relation</td>
<td>70</td>
</tr>
<tr>
<td>5.2</td>
<td>Encapsulated tuples</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>Simplified BNF of ISQL</td>
<td>71</td>
</tr>
<tr>
<td>5.4</td>
<td>System architecture</td>
<td>79</td>
</tr>
<tr>
<td>5.5</td>
<td>An example of special treatment for 3-way self-join</td>
<td>83</td>
</tr>
<tr>
<td>6.1</td>
<td>BNF of SQL\textsuperscript{ST}</td>
<td>88</td>
</tr>
<tr>
<td>6.2</td>
<td>BNF of STSQL</td>
<td>89</td>
</tr>
<tr>
<td>6.3</td>
<td>Forest Lake fire development</td>
<td>91</td>
</tr>
<tr>
<td>6.4</td>
<td>ForestFire spatiotemporal relation</td>
<td>93</td>
</tr>
<tr>
<td>6.5</td>
<td>Function $f(x)$</td>
<td>96</td>
</tr>
<tr>
<td>6.6</td>
<td>Triangulation of spatial data in the ForestFire relation of SQL\textsuperscript{ST}</td>
<td>98</td>
</tr>
<tr>
<td>7.1</td>
<td>Field information of Climate relation</td>
<td>106</td>
</tr>
<tr>
<td>7.2</td>
<td>Climate relation</td>
<td>107</td>
</tr>
<tr>
<td>7.3</td>
<td>GML representation of Story county</td>
<td>110</td>
</tr>
<tr>
<td>7.4</td>
<td>Four examples of multi-polygons</td>
<td>111</td>
</tr>
<tr>
<td>7.5</td>
<td>GML geometry representation of Figure 7.4-(d)</td>
<td>111</td>
</tr>
<tr>
<td>7.6</td>
<td>The nine intersection model</td>
<td>113</td>
</tr>
<tr>
<td>7.7</td>
<td>Logical view of HCube</td>
<td>114</td>
</tr>
<tr>
<td>7.8</td>
<td>Page formats in HCube</td>
<td>115</td>
</tr>
<tr>
<td>7.9</td>
<td>Logical view of a hybrid relation</td>
<td>116</td>
</tr>
</tbody>
</table>
Figure 7.10  HCube storage manager ........................................ 119
Figure 7.11  Expression tree of Query 1 ........................................ 121
Figure 7.12  Changed expression tree for the Boolean expression .......... 122
Figure 7.13  Evaluation node ..................................................... 123
Figure 7.14  Binary evaluation tree of Query 3 when only constants are involved ... 124
Figure A.1  Schemas of Crop and Soil relations in the NC-94 dataset ............ 134
Figure B.1  Geometry class hierarchy [47] ....................................... 135
ACKNOWLEDGEMENTS

I want to sincerely express my thanks to those who contribute to this work with various aspects, and to my reaching this milestone. Shashi K. Gadia, my major advisor and mentor, guided my graduate study at Iowa State University. He stimulated my research interest in multi-dimensional data models and implementation of database systems. He spent numerous hours with me discussing many ideas and technical details that eventually led to this dissertation. His scholarly attitude inspired me to rethink about what a computer scientist should be and how to approach real world problems more elegantly as a computer scientist. I also thank my other committee members, William J. Gutowski, Suraj C. Kothari, Glenn R. Luecke, and Wallapak Tavanapong for their efforts and contributions to this work. They gave me many valuable comments and insights on this work, which helped me to improve the quality of this work.

I thank Daryl Herzmann, a program assistant of Iowa Environmental Mesonet in Agronomy Department, for providing the NC-94 dataset and GML for counties of the north central regions in the United States. His preparation for the dataset contributed to this dissertation.

I thank the present and past members of the Laboratory of Database Models and Access. Thanks to Shihe Ma and Daniel Patanroi for their implementation of the dynamic XML pagination algorithm and improvement. Thanks to Srikanth Krithivasan, Jia Tao, and Qian Zhang for their comments for my presentation material. Sharing the same interests with my colleagues made for a pleasant experience at Iowa State. Although we have different regional, cultural, and ethical backgrounds, we share good memories which will enrich my life.

I thank the Korean students in the Computer Science Department. Thanks to Tae-hyung Kim and Kihwan Kim for their sincere hearts, sharing my joyfulness and sorrow while I was at Iowa State. I was fortunate to have such good friends and to have good memories with them.
I thank Hector Leiva for many discussions to go through the notorious headache courses—Algorithms and Theory of Computation—encountered when I just launched my scholarly adventure in the graduate school at Iowa State. Discussions with him played a key role to successfully pass them. They were one of the most dangerous obstacles in my journey toward Ph.D.

Finally, I thank my family; in particular, my parents for their endless love for family and support; Youngcho, for always being there for me with love and patience.

This work has been sponsored in part by Information Infrastructure Institute, a grant from Baker Foundation at Iowa State University, and a studying abroad fellowship from Chungbuk National University.
ABSTRACT

Data arising in many real world applications have space and time dimensions that require database support. The parametric model is one of the data models for dimensional data. The data model assumes that there is an underlying space, called parametric space, consisting of points in a multidimensional space. This makes it suitable for uniform treatment of dimensional data. A value in the parametric model is modeled as a function over the parametric space. Viewing a value as a function helps one achieve a one-to-one correspondence between objects in the real world and records in a database. One of the important requirements in the parametric model is that domains of values should be closed under the set theoretic operations such as union, intersection, and complementation. Due to this modeling concept, ParaSQL—the query language of the parametric model—is able to mimic natural languages more closely. In this dissertation we validate and implement the parametric model for temporal and spatiotemporal data. We also develop a preliminary prototype for the users of NC-94, an interesting dataset in agriculture.

A value in the parametric model is viewed as a function over the parametric space; therefore it does not have a fixed size. Potentially, such values vary in size ranging from a few bytes to gigabytes and beyond. This makes implementation of the parametric model a challenging problem. Whereas other models where records are small and have fixed length tend to use storage technologies offered by conventional database systems, for the parametric model this option is impractical. In order to meet the implementation challenge, we develop our own XML-based storage technology and deploy it in our implementation. Incidentally, XML is also used for interfacing various modules and artifacts like parse tree, expression tree, and iterators to fetch data from a disk.
The NC-94 dataset, mentioned above, contains the most complete record of spatiotemporal variables that characterize the dynamics of agriculture covering the north central region in the United States. However, in this dataset, inclusion of geography is marginal at best. Counties are only known through their symbolic names and do not include geographical maps. In order to support ad-hoc query of data in its geospatial context, a novel hybrid structure is designed and implemented in our XML-based storage. We use XML-based Geography Markup Language (GML) to describe geospatial information. GML supports OpenGIS standard of the Open Geospatial Consortium. Use of GML is a good match, because it is XML-based. More importantly, the OpenGIS standard meets the set theoretic closure requirements proposed by the parametric model.

It is expected that the validation and the implementation methodologies introduced in this dissertation will contribute to database and GIS (Geographical Information Systems) communities. The validation demonstrates the ease of use and efficiency of the parametric model in dealing with temporal and spatiotemporal data. This should help settle a debate in temporal database community which has continued since the mid 1980s. The findings also extend to spatial and spatiotemporal data. It is an important baby-step toward full-fledged implementation of the parametric model. We hope that this work will also help bring database and GIS communities together.
CHAPTER 1 INTRODUCTION

Various real world applications need to handle and process information with space and time dimensions. The parametric data model is one of the data models for dimensional data, which is capable of handling heterogeneous dimensions in a uniform way. The objective of this dissertation is to validate the parametric data model for temporal and spatiotemporal data and provide an implementation methodology. In this chapter, we present the main goals of this dissertation and provide overall introduction to the approach used to achieve the goals.

1.1 Overview

1.1.1 Dimensional Databases

Conventional relational databases play an important role in processing business data, which capture the current perception of reality. Although they work efficiently with ordinary data, they are not suitable for dimensional data. Databases capable of handling dimensional data have many advantages over conventional databases when real world problems are associated with time and space. Databases that deal with time, space, and spatiotemporal dimensions are termed temporal databases, spatial databases, and spatiotemporal databases, respectively.

Many researchers have focused on temporal databases and spatial databases, separately. Temporal databases manage past, present, and future data over time. In contrast, spatial databases handle data over space. Space and time dimensions, however, tend to be intertwined so that separating space and time dimensions is contrary to their characteristics. Because of this, there is growing attention to the area of spatiotemporal databases. Spatiotemporal databases handle and manage spatially and temporally referenced data.
1.1.2 Parametric Data Model

The parametric data model is one of the data models for dimensional data. It has been actively studied since the mid 1980s for its usability in dimensional data. The parametric approach is simple, yet elegant. This data model assumes that there is an underlying hypothetical space, called parametric space. The parametric space is simply viewed as a set of points. Due to this, the parametric data model can handle heterogeneous dimensions like time and space uniformly at the abstract level and allow parametric data to extend to any assortment of dimensions.

The parametric data model defines attribute values as functions over the parametric space, allowing the data model to capture spatial and temporal variabilities. The domain of an attribute is represented as a parametric element which is a subset of the parametric space. The parametric data model proposes no particular point of view about the nature and representation of parametric elements. However, an important requirement is that the set of all parametric elements should be closed under set theoretic operations such as union, intersection, and complementation. Satisfying the closure property of the set operations reduces the complexity of ParaSQL—a query language of the data model—by mimicking natural languages. The set operations—union, intersection, and complementation—can be directly mapped to or, and, and not in natural languages, respectively. In addition to this, the parametric data model captures an object in a single tuple, leading to one-to-one correspondence between objects in the real world and tuples in a database. This also helps reduce the complexity of ParaSQL, avoiding multi-way self-joins which are frequently invoked by other dimensional data models.

1.2 Objectives

Modeling an object in the real world in a single tuple articulates the concept of user friendliness in ParaSQL. However, such a modeling feature makes it difficult to implement the parametric model on top of conventional databases, which is the most popular approach for implementation of other dimensional models.

In order to meet the implementation challenge of the parametric data model, we need a
flexible data description mechanism. Since XML does not have any boundary limitations for objects, XML is an elegant option for the parametric data model. Such flexibility of XML helps encapsulate an object in a single tuple and simplifies the implementation. Furthermore, human-readable XML can lead to more reliable and less bug-affected codes. Therefore, XML is qualitatively an excellent implementation platform. Based on the XML technology, we can build our own storage without relying on existing storage structures.¹

Before we undertake implementation of the parametric data model, it is advisable to validate the data model and query language for usability. Therefore, we addresses the four main goals in this dissertation as follows:

1. Justification of the XML-based implementation.
2. Validation of the parametric data model for temporal data.
3. Validation of the parametric data model for spatiotemporal data.
4. Implementation of a spatiotemporal database within the parametric framework.

1.3 Approach

1.3.1 Justification of XML-based Implementation

XML is a promising option for the parametric data model to encapsulate a real world object into a single tuple because of its flexibility. However, it seems that there exists a myth about XML—XML is not suitable for a storage structure because XML is verbose. Since we consider XML as an implementation platform, it is required to justify the XML-based implementation. For the justification, we will compare and estimate storage costs for relational, object-oriented, and XML-based storages for the parametric data model. Such comparison will quantify the storage requirements and provide sound evidence for our XML-based approach.

¹Note that there was an attempt to build the parametric temporal model on top of ERAM storage [14]. However, the system lacked a buffer manager and a tuple should be always less than the size of a page.
1.3.2 Validation of Parametric Data Model for Temporal Data

In order to validate the parametric data model for temporal data, we will consider an interval-based data model which is one of popular data models in the temporal database community. The interval-based data model uses relational databases by simply adding special attributes to relations. We evaluate the ease of use of two data models with a query suite which is independent of query languages and data models. The evaluation will show us why the parametric data model is effective as a temporal data model.

In addition to this, we will implement two temporal database systems. For the parametric data model, we develop an XML-based storage. Our storage is capable of storing variable-length tuples in a relation, allowing a tuple to reside in multiple pages. Such circumstance is frequently occurring in dimensional data because an object in the real world can be in size ranging from a few bytes to gigabytes and beyond. For the interval-based data model, we follow the industry standard binary page format for a storage structure.

We will also compare the system performance of the two systems by measuring disk block accesses. The performance comparison will provide a reasonable clue to determine whether the parametric data model sustains its modeling advantages in the implementation as well.

1.3.3 Validation of Parametric Data Model for Spatiotemporal Data

In order to validate the parametric data model for spatiotemporal data, we will introduce two other data models. They are point-based and interval-based spatiotemporal data models. The former updates object information every time instant, creating tuples in a spatiotemporal relation. The latter, however, creates a maximum time interval when object status has been changed. Each update of an object is associated with a maximum time interval. It should be noted that such spatiotemporal models fragment objects into multiple tuples in order to utilize conventional database systems. In addition to this, we will compare the user-friendliness of spatiotemporal query languages by using a use case. The comparison will help us to determine which data model is more appropriate for spatiotemporal data.
1.3.4 Implementation of NC-94 Spatiotemporal Database

In addition to the validation of the parametric data model in terms of usability and efficiency, we also validate it for an interesting use case. In this dissertation, we provide an implementation methodology used to build a spatiotemporal database system for the NC-94 dataset.

The NC-94 dataset contains the most complete records of temporal and spatial variables for climate, crop, and soil in the north central region in the United States. The fundamental inputs in the dataset are used to run a variety of agricultural simulations for various applications.

The NC-94 database system allows users to pose ad-hoc queries. Note that supporting ad-hoc queries can significantly enhance the usefulness of the NC-94 dataset in the public domain because such data, in general, is stored in scientific data formats which do not support ad-hoc queries. The database system can be utilized by even researchers to extract data without using third party software packages.

The NC-94 database system maintains geographical information in GML (Geography Markup Language) which is an XML-based encoding standard for geographic information. Storing, accessing, and processing GML-based geospatial data are seamlessly integrated in our XML-based implementation for the parametric data model.

The design and implementation of the database system raises many interesting issues including a hybrid storage which maximizes storage efficiency regardless of the characteristics of dimensional data like homogeneous or heterogeneous data. In the implementation, we take the following subtopics into account.

- **Hybrid storage:**
  
  We design and develop a hybrid storage called *HCube*. The HCube is capable of storing and accessing homogeneous, heterogeneous, and hybrid data.

- **Buffer manager:**
  
  We develop a buffer manager working in the HCube. In our implementation, LRU (Least Recently Used) buffer replacement is considered. However, the buffering scheme is designed to work under different strategies.
• **Iterators:**
  We implement iterators which are used to evaluate ParaSQL queries. In the development, a primitive iterator is used to design more complex iterators.\(^2\)

• **Parse and expression trees:**
  The implementation is XML-oriented. XML is utilized to design various module interfaces, represent parse and expression trees.

### 1.4 Contributions

The outcomes and significance of this research can be summarized as follows:

• **A new spatiotemporal database system:**
  Since it is the first implementation of the parametric data model for spatiotemporal data, this research breaks new ground in spatiotemporal databases.

• **Validation of the parametric data model's advantages:**
  The validation demonstrates the ease of use and efficiency of the parametric model in dealing with temporal and spatiotemporal data. This should help settle a debate in temporal database community which has continued since the mid 1980s. Furthermore, our findings can also extend to spatial and spatiotemporal data.

• **The cornerstone of extensive research for successors:**
  The fruitful outcomes will lead to extensive research, for example, optimization and strategy issues under different buffering and caching schemes. Since many new implementation methodologies are introduced in this dissertation, they will attract researchers to adapt the technology in their implementation.

• **Useful outcomes to database and GIS communities:**
  Since few systems for spatiotemporal databases exist, the successful outcomes will contribute to the database community. In addition to this, our XML-based implementation

\(^2\)In this dissertation, joins for spatiotemporal relations are not considered, but joins for temporal relations are implemented. Since spatiotemporal joins are full fledged topics, they will be considered in the future work.
methodology seamlessly integrates GML into the parametric data model. Although the NC-94 database system is a baby-step toward meeting the parametric data model, it is an important leap for database and GIS communities. Therefore, our work will help bring database and GIS communities together.

1.5 Related Work

1.5.1 Temporal Databases

A temporal database is capable of storing evolution of data, thereby allowing users to examine complete object histories [7]. Applications of temporal databases include financial, record-keeping, and scientific applications [29]. Temporal databases are one of active research areas in the database community and tremendous research work has been done by many researchers. According to Jensen and Snodgrass's survey [29], more than 2,000 research papers on temporal databases had been published. There are many different types of temporal data models and they have their own merits in their specific applications.

In temporal database literature, we can find three types of timestamps—instants, intervals, and temporal elements. Based on timestamps, temporal data models are termed point-based models, interval-based models, and temporal element-based models.

It is important to emphasize that only temporal elements are legitimate domains of objects and events in the real world. However, a temporal element cannot be represented using a fixed length because it is defined as a finite union of intervals. Therefore, intervals or instants are used to timestamp fragments of objects that are stored in multiple fixed-length tuples. Typically such timestamps are attached at tuple level rather than value level.

In general, temporal data models are introduced with their query languages. Among many temporal query languages, SQL/TP [60] and SQL$^T$ [8] use point-based data models; TSQL2 [55] and IXSQL [34] use interval-based data models; and ParaSQL [18] and NRTC [59] use temporal element-based data models.
1.5.2 Spatial Databases

One of the most prominent examples of spatial data is geospatial objects located in a spatial frame such as the Earth’s surface. Many researchers have been motivated to store and query spatial data, resulting in spatial databases.

There are two communities which extensively research on spatial data—the spatial database community and the GIS community. The former community has actively researched on spatial data types, data models, and query languages which support ad-hoc queries over the data models with the data types. On the other hand, the GIS community more focuses on developing practical methodologies to effectively represent and process geospatial data.

A key issue in object-based models of spatial information is the choice of a basic set of spatial data types required to model common shapes on maps. Although many proposals have been made, a consensus is slowly emerging in terms of the OGC’s standard (Open Geospatial Consortium).3 The OGC has standardized spatial feature geometry and spatial operations. The spatial data is based on the OGC Geometry Object Model in the simple feature specification for SQL [47]. In the model, class Geometry serves as the base class and many subclasses such as Point, Curve(Line) and Surface(Polygon) are extended from the base class [51].

The OGC specification defines a standard for SQL which supports the storage and query of spatial data. Most spatial databases do not stand on their own, but instead are just an extension to relational databases. They use a dialect of SQL called Spatial Feature Structured Query Language which simply adds spatial functions to SQL such as distance, touches, centroid, inside, area, and extent [48].

1.5.3 Spatiotemporal Databases

Many real world objects are associated with time and space. GPS (Global Positioning Systems), mobile phone users within mobile networks, wireless communication network, and environmental monitoring systems are some examples of applications which handle and manage

---

3 The OGC’s proposal for spatial data types is shown in Figure B.1 of Appendix.
spatially and temporally referenced data. Because many applications require an ability of managing spatiotemporal data, there is growing attention to spatiotemporal databases [1].

Spatiotemporal databases have gained considerable attention and been researched actively over a significant period. However, there still exist very few prototypes of complete systems, and far less products that provide effective support for applications tracking changes to spatial and aspatial (or ordinary data) over time because the design and implementation of a complete spatiotemporal database is a challenging undertaking, involving extensions to all aspects of a non-spatiotemporal architecture such as data model, query language, query optimizer, query evaluator, programming environment, storage manager, and indexes [24].

In the past, research in spatial and temporal data models and database systems has mostly been done independently. Spatial database research has focused on modeling and querying geometries associated with objects while temporal databases have focused on modeling and querying temporally evolving data. Nevertheless, many researchers have tried to combine the two areas because they are all dealing with dimensions and closely related [13].

Since temporal and spatial data models have been intensively researched in the temporal and the spatial database communities, one may consider approaches to combine temporal and spatial data models to build spatiotemporal data ones. There are two directions to accommodate temporal and spatial data models—1) the embedding of a temporal awareness in spatial data models and 2) the accommodation of space into temporal data models. According to the survey of Roddick and Spiliopoulou [50], the former approach is more popular because of the relative maturity of geographic information systems.

1.6 Organization

The rest of this dissertation is organized as follows.

In Chapter 2, we introduce the parametric data model, focusing on the advantages of the data model over other dimensional data models.

In Chapter 3, we discuss the usability of XML. We quantify the storage requirements for relational, object-oriented, and XML-based storages and provide sound evidence which supports
our XML-based approach.

In Chapter 4, we discuss an XML-based implementation methodology, explaining how to overcome the implementation challenges of the parametric data model.

In Chapter 5, we validate the parametric data model for temporal data and compare the system performance of two temporal database systems.

In Chapter 6, we validate the parametric data model for spatiotemporal data by evaluating three spatiotemporal data models.

In Chapter 7, we present a spatiotemporal database system for the NC-94 dataset, which demonstrates the usefulness of the parametric data model for the real world application which can be fertilized by the data model.

In Chapter 8, we conclude this dissertation with a summary of findings, followed by an outline of future research directions.
CHAPTER 2  OVERVIEW OF PARAMETRIC DATA MODEL

The parametric data model has been studied since the mid 1980s by Gadia and his students. It handles multi-dimensional data in a uniform way and reflects the properties of natural languages, which reduces the query complexity at the user level. In this chapter, we introduce the general concept of the parametric data model for dimensional data. We also introduce ParaSQL—the query language of the parametric data model, focusing on its user-friendliness.

2.1 Introduction

The parametric data model is one of data models for dimensional data which is capable of handling heterogeneous dimensions. It has been introduced in [18] to model ordinary, temporal, and spatiotemporal data in a uniform way. It has been also studied to apply the data model to multi-level security databases [5]. One of features of the parametric data model is its extendibility to another dimension while the most of other data models focus on one particular form of dimensional data.

In the parametric data model, there is an underlying parametric space that is simply viewed as a set of points. The data model defines an attribute as a function over the parametric space. Such a modeling feature makes it possible to capture an object in the real world in a single tuple in a database.\(^1\) Therefore, it can sustain one-to-one correspondence between tuples and objects in the real world.

Domains of values in the parametric data model are represented by parametric elements which are subsets of the parametric space. It is important to note that the concrete represen-

\(^1\)The term, object, is loosely used in this dissertation. It is not the same as the concept of objects in the object-oriented database paradigm.
tation of parametric elements is open to implementation. However, the set of all parametric elements should be closed under set theoretic operations such as union, intersection, and complementation. Satisfying the closure property for the set operations articulates ParaSQL in more precise term. It naturally minimizes the complexity of ParaSQL by mapping union, intersection, and complementation to or, and, and not in natural languages. In addition, the parametric data model allows one to mix values into different dimensions by dimension alignment. Dimension alignment is automatically taken care of by system.

We must note that the parametric data model is orthogonal to database paradigms such as relational, object-oriented, and XML. However, in this dissertation, we consider the parametric relational data model. Based on the parametric data model, we can implement databases called parametric databases. Parametric databases can be seen as temporal databases, spatial databases, or spatiotemporal databases depending on dimensions handled by the databases.

The rest of this chapter is organized as follows. Section 2.2 discusses the concept of parametric elements. Section 2.3 discusses parametric relations including temporal, spatial, and spatiotemporal relations. Section 2.4 discusses the ParaSQL syntax and query examples. Section 2.5 summarizes this chapter.

2.2 Parametric Elements

In order to model an object in the real world in a single tuple as naturally as possible, parametric elements are introduced in the parametric data model. Parametric elements are subset of the parametric space. Representing parametric elements is left open, but they should satisfy the closure property of union (U), intersection (\(\cap\)), and complementation (\(\neg\)). We may call parametric elements temporal elements in the temporal context, spatial elements in the spatial context, and spatiotemporal elements in the spatiotemporal context. It is worth noting that space and time dimensions are intertwined in the spatiotemporal context so that spatiotemporal elements in the parametric data model can be constructed by combining temporal elements and spatial elements.
2.2.1 Temporal Elements

Time intervals are inadequate to model the history of an object in a single tuple, and they lead to query languages that are difficult to express natural language queries [18]. To obtain timestamps that are closed under the set theoretic operations of union, intersection and complementation, the concept of *temporal elements* is introduced in the parametric data model [20, 18, 58]. The parametric data model assumes that there is a universe of time that consists of an interval \([0, \text{NOW}]\) of instants with a linear order \(<\) on it. Here \text{NOW} denotes the current instant of time. For our purposes, a temporal element is defined as a finite union of time intervals. An interval is obviously a temporal element. An instant \(t\) can be identified with the interval \([t, t]\); thus it can be regarded as a temporal element. Examples of temporal elements are \([11, 60]\) and \([0, 20] \cup [41, 51]\). The set of all temporal elements is closed under union, intersection, and complementation.

2.2.2 Spatial Elements

The parametric data model assumes an underlying universal region \(\mathcal{R}\) in spatial database context. The user views it as a set of points. Let define \(\text{REG}\) be a set of subsets of \(\mathcal{R}\) which is of interest to users, and that \(\text{REG}\) is closed under union, intersection, and complementation. A *spatial element* is an element of \(\text{REG}\). There are no specific assumptions about the constitution of \(\mathcal{R}\). \(\mathcal{R}\) can be an \(n\)-dimensional Euclidean space, surface of a sphere, portion of a plane, a curve and so on. Main hypothesis is that the regions in \(\text{REG}\) should have some reasonable description [15].

2.2.3 Spatiotemporal Elements

We assume that we are given the universal region \(\mathcal{R}\). To this, we add the universe of time and obtain the spatiotemporal universe \(\mathcal{R} \times \mathcal{T} = \mathcal{R} \times [0, \text{NOW}]\).

In case of space, we are interested in spatial elements which are closed under union, intersection, and complementation. We have seen that the temporal elements are also closed under these set operations. These closure properties are essential for seamless querying. In order to
maintain this seamlessness, we define a spatiotemporal element to be of the form

\[ \text{reg}_1 \times \mu_1 \cup \text{reg}_2 \times \mu_2 \cup \cdots \cup \text{reg}_n \times \mu_n, \]

where for each \( i, 1 \leq i \leq n, \text{reg}_i \) is a spatial element and \( \mu_i \) is a temporal element. Clearly, spatiotemporal elements are closed under the three set operations [16].

2.3 Parametric Relations

Just as in a classical database, a parametric tuple is a concatenation of values. The main difference here is that values are parametric values, and they can be very large. Informally, a parametric relation can be defined as a set of parametric tuples. In this section, we will show examples of temporal, spatial and spatiotemporal relations.

2.3.1 Temporal Relations

Figure 2.1 shows an example of a parametric temporal relation whose attributes are Name, Salary, and DName (department name). This Emp relation maintains the history of employees.

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>DName</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[50,54] 55K</td>
<td>[45,60] Test</td>
</tr>
<tr>
<td></td>
<td>[55,60] 60K</td>
<td></td>
</tr>
<tr>
<td>[0,20] Tom</td>
<td>[0,20] 45K</td>
<td>[0,20] Sales</td>
</tr>
<tr>
<td>U [41,51]</td>
<td>[41, 51] 50K</td>
<td>U [41,51]</td>
</tr>
</tbody>
</table>

Figure 2.1 Emp temporal relation

To capture the changing value of an attribute in the temporal context, a temporal value of attribute \( A \) is defined as a function from a temporal element into the domain of \( A \). An example of a temporal value of attribute DName is \(([11,44] \text{ R&D}, [45,60] \text{ Test})\). The semantic of the attribute value is that an employee worked in R&D department from 11 to 44 and Test
department from 45 to 60. If $\xi$ is a temporal value, $[\xi]$ denotes its domain. Thus $[(11,44]$ R&D, $[45,60]$ Test$] = [11,60]$. $\xi \downarrow \mu$ denotes the restriction of $\xi$ to the temporal element $\mu$. For example, if $\mu = [28,55]$, $\xi \downarrow \mu = ([28,44] \text{ R&D}, [45,55] \text{ Test})$.

A key has to be designated for a relation. The key identifies an object uniquely by values that remain invariant in the parametric domain of the object. Formally, a relation $r$ over a scheme $R$, with $K \subseteq R$ as the key of $r$, is a finite set of tuples such that no key attribute value in a tuple changes from one point in its domain to another, and no two tuples assume the same key value. Such keys are termed \textit{parametrically invariant}. There is one object per key and the parametric model preserves the one-to-one correspondence between the tuples in the database and objects in the real world.

\subsection*{2.3.2 Spatial Relations}

Figure 2.2 shows an example of a parametric spatial relation whose attributes are CName (county name) and Crop. This County relation manages spatial and crop information. It shows which crops are being cultivated in a county. Note that $creg_1, \cdots, creg_6$ can be quite complex on their own right, each consisting of multiple disjoint regions having complex shapes.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
CName & Crop \\
\hline
$creg_1 \cup creg_2$ & $creg_1$ wheat  \\
& $creg_2$ corn \\
\hline
$creg_3 \cup creg_4$ & $creg_3$ wheat  \\
& $creg_4$ barley \\
& $creg_5$ rice \\
\hline
$creg_6$ & $creg_6$ wheat \\
\hline
\end{tabular}
\caption{County spatial relation [15]}
\end{table}

A \textit{spatial tuple} is a concatenation of spatial values whose spatial domains are the same. A \textit{spatial value} is a function from a spatial element into a domain of an attribute. An example of a spatial value of attribute Crop is $(creg_1 \text{ wheat}, creg_2 \text{ corn})$. This spatial value means that
wheat is being cultivated in reg₁ while corn is being cultivated in reg₂. The domain of this spatial value is expressed as \( [(\text{reg₁ wheat, reg₂ corn})] = \text{reg₁} \cup \text{reg₂} \).

### 2.3.3 Spatiotemporal Relations

Figure 2.3 shows a parametric spatiotemporal relation which arises in agriculture. This well relation contains information about the concentration of chemicals in the wells taken at different times. The attribute UGConc and DGConc are up-gradient and down-gradient wells, depending on the direction of ground water flow [18].

<table>
<thead>
<tr>
<th>ChemName</th>
<th>UGConc</th>
<th>DGConc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atrazine</td>
<td>( p₁ \times [0, \text{NOW}] ) 1.0</td>
<td>( p₁ \times [0, \text{NOW}] ) 0.9</td>
</tr>
<tr>
<td></td>
<td>( p₂ \times [0, 5] ) 1.5</td>
<td>( p₂ \times [0, 10] ) 1.4</td>
</tr>
<tr>
<td></td>
<td>( p₂ \times [6, \text{NOW}] ) 3.5</td>
<td>( p₂ \times [11, \text{NOW}] ) 2.9</td>
</tr>
<tr>
<td>Simazine</td>
<td>( p₁ \times [0, 9] ) 10.0</td>
<td>( p₁ \times [0, \text{NOW}] ) 9.2</td>
</tr>
<tr>
<td></td>
<td>( p₁ \times [10, \text{NOW}] ) 12.2</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.3 Well spatiotemporal relation [18]

A spatiotemporal value is a spatiotemporal assignment to an attribute to capture the space and time varying value of an attribute. A spatiotemporal value of attribute \( A \) is a function from a spatiotemporal element into the domain of \( A \). In the well relation, the value of attribute UGConc of the first tuple is \( (p₁ \times [0, \text{NOW}] \times 1.0, p₂ \times [0, 5] \times 1.5, p₂ \times [6, \text{NOW}] \times 3.5) \). The semantics of this spatiotemporal value is that from time 0 to now the concentration of Atrazine in well \( p₁ \) is 1.0; the concentration of Atrazine in well \( p₂ \) was 1.5 from time 0 to 5, but it has been changed to 3.5 since time instant 6. The domain of the spatiotemporal value is expressed as follows:

\[
[(p₁ \times [0, \text{NOW}] \times 1.0, p₂ \times [0, 5] \times 1.5, p₂ \times [6, \text{NOW}] \times 3.5)] = p₁ \times [0, \text{NOW}] \cup p₂ \times [0, \text{NOW}]
\]
2.3.4 Dimension Alignment

The parametric data model allows one to mix values into different dimensions. The values can be space independent, time independent, or space and time independent, which means that sometimes it is necessary to align dimensions when a user query is evaluated. The parametric data model does not require a user to align dimensions in a query. It makes the system align different dimensions automatically as needed, by plugging the whole space of missing dimensions into dimensional data. For example, a user can write \( reg \cap \mu \), where \( reg \) and \( \mu \) are a spatial element and a temporal element, respectively. This domain expression can be evaluated as \( (reg \times T) \cap (R \times \mu) \), where \( T \) is the universe of time, and \( R \) is the universe of space. Such alignment applies to all data types, e.g. attribute values, relations. Another interesting corollary of this phenomenon is that the domain operator can be applied to ordinary data. Thus, in the context of spatial alignment \([a]\) evaluates to \( R \), and in spatiotemporal context it evaluates to \( R \times T \) \([10, 16]\).

2.4 Parametric Structured Query Language

ParaSQL (Parametric Structured Query Language) is a query language of the parametric data model. It consists of three expressions—relational expression, domain expression, and Boolean expression. They evaluate to relations, parametric elements, and Boolean values, respectively. These three expressions are mutually recursive. In this section, we will discuss the simplified BNF (Backus Naur Form) of the three expressions, ParaSQL’s user-friendliness, and some ParaSQL examples. An algebra leading to semantics of ParaSQL in terms of the relational operations is given in \([19]\).

2.4.1 Relational Expressions

A relational expression returns a relation that is a set of tuples. It can be expressed by union, intersection, difference, and select statement. Figure 2.4 shows the simplified BNF of the relational expression of ParaSQL.
\[
<\text{relational expression}> ::= <\text{select statement}> | \\
<\text{relational expression}> \{\text{UNION} <\text{relational expression}>\}+ \\
<\text{relational expression}> \{\text{DIFFERENCE} <\text{relational expression}>\}+ \\
<\text{relational expression}> \{\text{INTERSECTION} <\text{relational expression}>\}+
\]

\[
<\text{select statement}> ::= \text{SELECT} <\text{attribute list}> \\
[\text{RESTRICTED TO} <\text{domain expression}>] \\
\text{FROM} <\text{relation list}> \\
[\text{WHERE} <\text{boolean expression}>]
\]

Figure 2.4  BNF of relational expression

In our discussion, we only focus on the select statement of ParaSQL because it is the most interesting relational expression. A select statement in ParaSQL is an SQL-style select statement. Unlike classical SQL, it has \text{RESTRICTED TO} clause that restricts the domain of tuples qualified by \text{WHERE} clause. \text{WHERE} clause in ParaSQL has the same functionality of SQL, that is, it returns a tuple if a Boolean expression is satisfied.

In the relational expression, we must note that \text{RESTRICTED TO} and \text{WHERE} clauses can be omitted. However, they are recovered by a ParaSQL parser as follows:

\[
\begin{align*}
\text{SELECT} & \quad \rightarrow \quad \text{SELECT} \\
\text{FROM} & \quad \rightarrow \quad \text{RESTRICTED TO} \, \text{R} \times \text{T} \\
& \quad \rightarrow \quad \text{FROM} \, <\text{relation list}> \\
& \quad \rightarrow \quad \text{WHERE} \, \text{TRUE}
\end{align*}
\]

2.4.2 Domain Expressions

A domain expression returns the domain of a tuple, an attribute or a relation. Figure 2.5 shows the simplified BNF of domain expression.

The atomic domain expression \([<\text{attribute}>]\) is a counterpart of \([A]\) in the algebra of the parametric data model, where \(A\) is an attribute. It collects temporal domain of a specified attribute. The domain expression, \([<\text{attribute}><\text{op}><\text{attribute}>]\), returns domains such that two attributes have \(\text{op}\) relationship. For example, domains that one employee's salary is greater than another employee's salary can be expressed by \([E1.\text{Salary} > E2.\text{Salary}]\). The
domain expression, \([\langle\text{attribute}\rangle\langle\text{op}\rangle\langle\text{value}\rangle]\), returns domains such that \(<\text{op}\rangle\) relationship is satisfied by an attribute and a constant. For example, domains that an employee's salary is greater than $60,000 can be expressed by \([E.Salary > 60,000]\). The domain expression \([\langle\text{relational expression}\rangle]\) collects all domains of tuples returned by a relational expression. The last type of domain expression is a parametric element. Because a domain expression returns a parametric element, a parametric element itself is a domain expression. The domain expressions in ParaSQL can be connected by or ~ which correspond to set operators union (\(U\)), intersection (\(\cap\)), difference (\(\setminus\)), and complementation (\(\neg\)), respectively.

### 2.4.3 Boolean Expressions

A Boolean expression evaluates if a given Boolean condition is true or false. Figure 2.6 shows the simplified BNF of Boolean expressions.

A Boolean expression has the same functionality as classical SQL in that it either qualifies or disqualifies a tuple. But it differs from classical SQL because it can be constructed by domain expressions with set operators. For example, suppose a user wants to retrieve the information
of Software department. In WHERE clause, it can be expressed like DName = "Software". However, this expression is abbreviation of [DName=“Software"] ≠ ∅, meaning that sometime the domain of an event such that the department name is Software is not empty. Boolean expressions are connected by Boolean operators such as AND, OR, and NOT.

2.4.4 User-Friendliness

The parametric data model describes a real world entity in a single tuple rather than multiple tuples. This property reduces query complexity at the user level without invoking unnecessary self-joins to combine scattered object fragments residing in multiple tuples. Gadia [18, 15] showed that ParaSQL represents English queries seamlessly. For example, a user can change or condition to and condition in an English query. In ParaSQL, it can be expressed in the same way as the English query, that is, it simply changes or condition to and condition in WHERE clause. For example, consider following two queries:

- Q1: Retrieve complete information about employees who worked in R&D or Software department.
- Q2: Retrieve complete information about employees who worked in R&D and Software
department.

The only difference between the two English queries is the Boolean condition. However, expressing Q2 is difficult for a query language whose base model describes an object in multiple tuples. In such a data model, it is unavoidable to circumvent the use of self-joins [45]. In ParaSQL, the Boolean expressions of the queries can be expressed as follows:

\[ [\text{DName} = "R\&D"] \neq \emptyset \text{ OR } [\text{DName} = "Software"] \neq \emptyset \]

\[ [\text{DName} = "R\&D"] \neq \emptyset \text{ AND } [\text{DName} = "Software"] \neq \emptyset \]

The domain expression \([\text{DName} = "R\&D"] \neq \emptyset\) means that sometime the name of a department was R\&D. For ease of use, these two expressions can be abbreviated in WHERE clause as follows:

\text{DName} = "R\&D" \text{ OR } \text{DName} = "Software"

\text{DName} = "R\&D" \text{ AND } \text{DName} = "Software"

In addition to this, a user may pose a query to find information about employees while an event did not happen. For example, consider the following query:

- Q3: Retrieve complete information about employees while they did not work in R\&D or Software department.

In the parametric data model, the restriction is expressed by domain expression as follows:

\((\sim [\text{DName} = "R\&D"]) \cup (\sim [\text{DName} = "Software"])\)

Note that the complementation as well as unions of parametric elements is a parametric element. If the data model uses intervals in the temporal context, the complementation or the union of intervals may not be an interval. Furthermore, in spatial databases, even union of connected regions may not be connected; thus, the corresponding query of such data model for
Q3 would be complicated. Therefore, we can emphasize that satisfying the closure property of union, intersection, and complementation reduces the query complexity of ParaSQL.

2.4.5 ParaSQL Examples

For an illustration of ParaSQL, consider the following three English queries which assume underlying relations are temporal, spatial, and spatiotemporal relations, respectively.

1. Give details about employees while they worked in either R&D or Software and did not earn more than $60,000.

   ```sql
   SELECT * 
   RESTRICTED TO ( [[E.DName = "R&D"]]
   + [[E.DName="Software"]]
   ) * (~ [[E.Salary > 60000]])
   FROM Emp E
   ```

   This ParaSQL query retrieves tuples from `Emp` relation which is a parametric temporal relation. For every tuple, it restricts the tuple to a temporal domain such that an employee worked in R&D or Software department as well as the employee earned less than $60,000 as a salary.

2. Retrieve complete information about counties whose land acres are greater than 500,000 and which grow wheat and corn.

   ```sql
   SELECT * 
   FROM County C 
   WHERE Area([[C.CName]]) > 500000 
   AND C.Crop = "wheat" 
   AND C.Crop = "corn"
   ```

   In this ParaSQL query, the relation `County` is a parametric spatial relation which contains crop information with spatial regions such that crops are being cultivated. In `WHERE` clause, the ParaSQL checks if the area of a county is greater than 500,000 acres. The spatial function `Area([[C.CName]])` returns the area of the domain of a county because
\text{C.CName} \) returns a spatial element. It also checks if the county grows wheat and corn. As we discussed in Section 2.4.3, the Boolean expression is the abbreviation of \([\text{C.Crop}=\text{"wheat"}] \neq \emptyset \ \text{AND} \ [\text{C.Crop}=\text{"corn"}] \neq \emptyset\).

3. Give current information about up-gradient wells which contain the concentration of Atrazine less than 1.5 and corn is being cultivated in a region to which the wells belong.

\[
\begin{align*}
\text{SELECT} & \ast \\
\text{RESTRICTED TO} & \left\{ \right. \\
& \left. \text{SELECT} \ast \\
& \text{FROM County C} \\
& \text{WHERE C.Crop} = \text{"wheat"} \\
& \right\} \ast \left[ \left[ \text{W.UGCons} < 1.5 \right] \right] \ast \text{NOW} \\
\text{FROM Well W} \\
\text{WHERE W.ChemName} = \text{"Atrazine" AND W.UGCons} < 1.5
\end{align*}
\]

This ParaSQL query retrieves tuples from spatiotemporal relation \text{Well} such that the chemical name is Atrazine and up-gradient wells contain the chemical less than 1.5. Tuples satisfying the Boolean condition is restricted to the spatiotemporal element returned by the domain expression in the \text{RESTRICTED TO} clause. The domain expression intersects three parametric elements. The first parametric element is a spatial element which is the domain of counties that grow wheat. The second parametric element is a spatiotemporal element which is the domain of the up-gradient well which contains less than 1.5 Atrazine. The last parametric element is the temporal element NOW. By the dimension alignment discussed in Section 2.3.4, the missing universe of time and space dimension \text{T} and \text{R} are padded into the first and the last parametric elements, respectively. The qualified tuples by the Boolean expression are restricted to the spatiotemporal element.

\subsection{2.5 Summary}

We have introduced the parametric relational data model for dimensional data that handles assortments of dimensions uniformly. This conceptual approach makes it possible to view
different forms of dimensional data as data which has considerable similarities.

We have discussed the concept of parametric elements which are subsets of the parametric space. The parametric data model leaves parametric elements open to implementation. However, the set of parametric elements should be closed under set theoretic operations such as union, intersection, and complementation. Satisfying such closure property reduces the complexity of ParaSQL by mapping union, intersection, and complementation to or, and, and not in natural languages. It helps ParaSQL to mimic the grammar of natural languages, resulting in a user-friendly query language.

In addition, the parametric data model defines an attribute value as a function over the parametric space. Such modeling concept enables to capture an object in the real world in a single tuple, sustaining one-to-one correspondence between an object and a tuple. This prevents the parametric data model from invoking self-joins which are inevitable in other dimensional data models which fragment a real world object into multiple tuples.

We have also discussed the concept of dimension alignment which plugs the universes of time and space into dimensional data which misses such dimensions. The dimension alignment is automatically processed by the system. This mechanism also helps the parametric data model to reduce the complexity of ParaSQL.

Despite the advantages of the parametric data model, it is worth talking the challenge of implementation. As we discussed in this chapter, the parametric data model describes a real world entity in a single tuple whose attribute sizes are unknown at the definition time. This property makes implementation of the parametric data model a challenging problem because it is impractical to directly use conventional databases which is the most popular approach in other dimensional data models. In the following chapters of this dissertation, we will show how to overcome this implementation challenge by utilizing XML technology.
CHAPTER 3 STORAGE EFFICIENCY OF XML FOR PARAMETRIC DATA MODEL

It seems that there exists a myth about XML—XML is not suitable for a storage structure because of its verbalism. In this chapter, we will discuss the XML's storage efficiency for the parametric data model. We only focus on temporal data in this chapter, but storage costs of the other dimensional data can be studied in similar manner. By comparing storage efficiency for various storage platforms, we show that the myth about XML is not true for the parametric data model. Furthermore, we hope that our discussion in this chapter provides valuable insights to resolve the question about the XML's usability in implementation of the parametric data model.

3.1 Introduction

In 1995, Jensen, Snodgrass, and Soo [30] pointed out some difficulties of temporal data models capturing an object in a single tuple such that the models “may not be capable of directly using existing relational storage structures or query evaluation techniques that depend on atomic attribute values”.

The difficulties result from two properties of the parametric data model. First, it captures an object in the real world in a single tuple, resulting in variable-length tuples in a relation. Second, it uses temporal elements to timestamp objects, which cannot be represented in a fixed length. These two properties require a flexible data description mechanism to implement the parametric data model.

1They distinguished temporal data models into 1NF and non-1NF data models. An interval-based data model is 1NF while the parametric temporal data model is non-1NF.
XML is a promising option for the parametric data model to encapsulate a real world object into a single tuple because XML does not have any boundary limitations for objects. However, one may raise a question about storage efficiency because XML is verbose. In order to justify the XML-based implementation of the parametric data model, we evaluate and estimate storage costs for relational, object-oriented, and XML-based storages.

In our discussion, we mainly concentrate upon the amount of storage needed to store parametric data. Therefore, we do not count additional page information such as previous page, next page, and index of tuples. We only consider pure data used to represent relations. We also limit ourselves to temporal data to simplify our discussion.

In terms of ease of use, XML is the best as storage primitives which are easily encapsulated by using tags. Another side benefit is use of XML to modulate various layers. Even parse and expression trees can be represented in XML. We can create, analyze, and process parse trees and expression trees without relying on linked-list data structures, leading to more human readable and reliable code.

The rest of this chapter is organized as follows. Section 3.2 introduces various implementation platforms. Section 3.3 discusses the three storages. Section 3.4 estimates the storage costs of the three platforms. Section 3.5 evaluates storage efficiency with various inputs. Section 3.6 summarizes and discusses our findings.

3.2 Implementation Platforms

3.2.1 Conventional Databases

Conventional databases are popular implementation platforms for temporal database systems because of its fast implementation by utilizing existing technologies. Relational databases, one of the most representative conventional databases, can be used by temporal data models which use intervals as timestamps by adding Start and End attributes to a relation. However, such approach is not appropriate for the parametric data model because of variable-length tuples in a relation.

Object-oriented databases (OODBMS) are another popular implementation platforms of
temporal data models. In an object-oriented design, the world to be modeled is thought of as composed of objects [22]. One may regard an OODBMS as an effective candidate implementation platform because the parametric data model captures an object into a single tuple. However, there is a main disadvantage in implementation perspectives such that a resulting system can be too restrictive when incorporating particular temporal models into a given OODBMS because of complex tasks for extending an object-oriented language supported by the OODBMS to time supporting object-oriented language [57].

3.2.2 Object-Oriented Storages

In addition to extending existing conventional databases, we can also consider object-oriented storages because the parametric data model loosely considers a tuple as an object. We can find many object-oriented storages such as $O_2$ [3], ODE [2], Shore [52], and GemStone Facets [23].

Despite the advantage of object-oriented storages in handling objects, there are three major problems to utilize them for the parametric data model. First, it creates too many objects for a single tuple. To address this problem, consider an employee object which has salary history. Since the parametric data model uses value-level timestamping, each value of salary should be created as a single object with a timestamp which is a union of intervals. Moreover, the timestamps (temporal elements) are not primitive data types, but objects. Therefore, creating a tuple generates too many objects. Second, accessing values may cause additional accesses required. Since each value is a persistent object in object-oriented storages, searching for an attribute value for an object may result in several disk accesses because the objects and their timestamps may reside in different locations [6, 46]. Third, accessibility of object-oriented storages is very limited. The first and second problems can be resolved by modifying internal feature of object-oriented data storages. But, the most of object-oriented storages are transparent externally and only accessed through their object-oriented languages. Even if there

---

2We do not consider relational-object databases because of impedance mismatch which occurs when objects are mapped to tables stored in a relational database. Since the underlying database is a relational database, this approach is sufficiently covered by the relational storage platform.
are some open source object-oriented storages, it would not be a trivial task to restructure the storages for the parametric data model or support the problems addressed above.

### 3.2.3 XML-based Storages

XML is an attractive implementation platform for the parametric data model because of its flexibility. By using XML, we can describe a real world entity in a single tuple and construct relations as a set of tuples. However, there are two major obstacles to overcome when XML is used in implementation of the parametric data model—storage efficiency and pagination. Since XML is text-based and tag-based, it raises questions about storage efficiency. In addition to this, an XML storage must have an efficient way of handling and paginating larger XML data into small pages.

Therefore, it is worth discussing the storage efficiency of XML for the parametric data model. The rest of the following sections devotes the discussion about the storage costs of the three storages. In order to resolve the second problem, a dynamic XML pagination algorithm will be introduced in Chapter 4.

### 3.3 Data Representations

In this section, we will discuss how a parametric temporal relation can be represented in three different storages. Figure 3.1 shows Emp relation which maintains the history of employees with name, salary, and department information. We transform the Emp relation into relational, object-oriented, and XML-based storages, respectively.

#### 3.3.1 Relational Storage

A relational storage uses intervals and tuple-level timestamping, where all attributes in a tuple share a same domain. In addition to usual attributes, Start and End attributes are used to represent intervals.

Figure 3.2 shows Emp relation in the relational storage. Every tuple has Start and End attributes representing a valid period of a tuple. In Figure 3.2, an object consists of multiple
tuples, where each tuple represents an event. For example, two tuples, \((\text{Tom}, 50000, \text{R&D}, [0,10])\) and \((\text{Tom}, 50000, \text{Software}, [11,21])\), are distinct events and the latter tuple was created when Tom moved to Software department from R&D department. Therefore, no time intervals are overlapped.

### 3.3.2 Object-Oriented Storage

An object-oriented storage can also be used as a base storage for a temporal database. There are two representations in the object-oriented storage—*object versioning* and *attribute versioning* [54]. The former associates time to the whole object while the latter associates time to object attributes.

These two representations can be divided into four finer representations based on the relationship between objects—(a) object versioning with relationship object, (b) object versioning without relationship object, (c) attribute versioning with relationship object, and (d) attribute versioning without relationship object. Among the four approaches, approach (a) and (d) are known as the promising approaches to object-oriented temporal representations [12]. Therefore, we will discuss the two approaches—(a) and (d). We call them simply the object versioning and the attribute versioning, respectively.

However, the object versioning approach is not suitable for the parametric temporal data model because of two reasons. First, the object versioning approach creates multiple objects
<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>DName</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td>50000</td>
<td>R&amp;D</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Tom</td>
<td>50000</td>
<td>Software</td>
<td>11</td>
<td>21</td>
</tr>
<tr>
<td>Tom</td>
<td>60000</td>
<td>Software</td>
<td>22</td>
<td>40</td>
</tr>
<tr>
<td>Tom</td>
<td>65000</td>
<td>Software</td>
<td>41</td>
<td>50</td>
</tr>
<tr>
<td>Tom</td>
<td>65000</td>
<td>Hardware</td>
<td>51</td>
<td>70</td>
</tr>
<tr>
<td>Tom</td>
<td>70000</td>
<td>Hardware</td>
<td>71</td>
<td>73</td>
</tr>
<tr>
<td>Tom</td>
<td>70000</td>
<td>R&amp;D</td>
<td>74</td>
<td>90</td>
</tr>
<tr>
<td>Jane</td>
<td>45000</td>
<td>Software</td>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>Jane</td>
<td>45000</td>
<td>R&amp;D</td>
<td>33</td>
<td>50</td>
</tr>
<tr>
<td>Jane</td>
<td>50000</td>
<td>R&amp;D</td>
<td>51</td>
<td>70</td>
</tr>
<tr>
<td>Jane</td>
<td>55000</td>
<td>R&amp;D</td>
<td>71</td>
<td>90</td>
</tr>
</tbody>
</table>

Figure 3.2 Emp relation

for representing a real world entity. But, the parametric temporal data model treats a real world entity as a single tuple, which means history information should be embedded in a tuple. Second, this approach creates objects whenever there are updates in attributes, leading to multiple identical information in previous objects only except the updating attributes. But, in the parametric data model, updating an attribute does not affect the other attributes. Because of these two reasons, the object versioning approach cannot be used for the parametric temporal data model.

The attribute versioning approach is considered as the pure temporal object-oriented representation because each object captures the full history of a real world object [12]. Figure 3.3 shows the class representation of the attribute versioning for Emp relation.³

³In order to simplify our discussion, we denote time instants as a sequence of positive integers instead of using Date type.
tuples, where each tuple consists of the pair of a salary and a temporal element [12]. It is worth noting that, in the parametric data model, a temporal element is a finite union of intervals, which means that the size of a temporal element is not fixed. Therefore, a temporal element should be a class. Like \(t_{\text{salary}}\) and \(t_{\text{dept}}\) in Figure 3.3, a temporal element is a list of tuples, where each tuple contains starting and ending time instants.

### 3.3.3 XML-based Storage

Unlike relational storages, XML does not have any data boundary restrictions. Such flexibility makes it possible to encapsulate an object in the real world in a single tuple. The concept of relations, tuples, attributes, and temporal elements can be mapped to XML elements so that the features of the data model can be simply transformed into XML without convoluting any properties. XML can even add additional information derived from the data model to achieve fast information retrieval of a query execution engine [44, 41].

Figure 3.4 shows the XML representation of Emp relation. Element \(<e>\) represents a temporal element. It may contain several time intervals. Each value element, \(<v>\), has its own domain expressed by element \(<e>\). The union of all domains of element \(<v>\) is the domain of attribute element \(<a>\). For example, Salary attribute has four temporal elements and the union of the temporal elements is the same as the domain of Salary attribute. Each tuple
element, `<t>`, encapsulates its attribute elements. Each relation element, `<r>`, encapsulates its tuple elements. Note that there are many alternative XML representations for the parametric temporal data model and Figure 3.4 is one of the possible representations. For example, the attribute element `<a n="Name">` can be expressed as `<a>` without using XML attributes because the attribute information can be stored in a system catalog.

```xml
<r n="Emp">
  <t>
    <a n="Name">
      <v>Tom</v>[0,90]
    </a>
    <a n="Salary">
      <v>50000</v>[0,21]
      <v>60000</v>[22,40]
      <v>65000</v>[41,70]
      <v>70000</v>[71,90]
    </a>
    <a n="Dept">
      <v>R&D</v>[0,20]
      <v>Software</v>[11,50]
      <v>Hardware</v>[51,73]
    </a>
  </t>
</r>
```

Legends:
- r: relation
- t: tuple
- n: name
- a: attribute
- v: value
- e: temporal element

Figure 3.4 XML representation of Emp relation

### 3.4 Storage Costs

#### 3.4.1 Notations

In order to estimate storage costs, we will establish mathematical formulas for the three storage platforms. Table 3.1 shows the notations used in the formalization.

#### 3.4.2 Cost of Relational Storage

In order to find the storage cost of the relational storage, we first need to consider the size of a single tuple. The size of a single tuple can be calculated by summing the sizes of attributes
Table 3.1 Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>an attribute</td>
</tr>
<tr>
<td>u</td>
<td>update frequency of an attribute</td>
</tr>
<tr>
<td>t</td>
<td>a tuple</td>
</tr>
<tr>
<td>o</td>
<td>an object</td>
</tr>
<tr>
<td>p</td>
<td>% of intervals sharing ending time instants with another</td>
</tr>
<tr>
<td>A</td>
<td>average attribute size</td>
</tr>
<tr>
<td>T</td>
<td>average size of temporal elements</td>
</tr>
<tr>
<td>I</td>
<td>the size of an interval</td>
</tr>
<tr>
<td>C</td>
<td>average object overhead size</td>
</tr>
<tr>
<td>Tag</td>
<td>average tag size</td>
</tr>
<tr>
<td>U</td>
<td>average update frequency</td>
</tr>
<tr>
<td>N_a</td>
<td>the number of attributes</td>
</tr>
<tr>
<td>N_t</td>
<td>the number of tuples for an object</td>
</tr>
<tr>
<td>N_o</td>
<td>the number of objects</td>
</tr>
<tr>
<td>S_R</td>
<td>the storage cost of the relational storage</td>
</tr>
<tr>
<td>S_O</td>
<td>the storage cost of the object-oriented storage</td>
</tr>
<tr>
<td>S_X</td>
<td>the storage cost of the XML-based storage</td>
</tr>
</tbody>
</table>

in the tuple. Since the relational storage needs an interval for every tuple, we can define the size of a tuple by Eq. 3.1:

\[
Size(t) = \sum_{k=1}^{N_a} (Size(a_k)) + I \tag{3.1}
\]

We have seen that an object in the relational storage is fragmented into multiple tuples. The size of an object can be measured by summing all tuples for the object. Therefore, the object size in the relational storage can be defined by Eq. 3.2.

\[
Size(o) = \sum_{i=1}^{N_t} Size(t_i) \tag{3.2}
\]

Since there are total \(N_o\) objects in a single relation, we can estimate the cost of the relational
storage as shown in Eq. 3.3

\[ S_R = \sum_{j=1}^{N_o} \text{Size}(a_j) \]
\[ = \sum_{j=1}^{N_o} \sum_{i=1}^{N_t} \left( \sum_{k=1}^{N_a} (\text{Size}(a_k)) + 1 \right) \]
\[ = \sum_{j=1}^{N_o} \sum_{i=1}^{N_t} (N_a \cdot A + I) \cdot \sum_{k=1}^{N_a} \text{Size}(a_k) = N_a \cdot A \]
\[ = \sum_{j=1}^{N_o} N_t \cdot (N_a \cdot A + I) \]
\[ = N_o \cdot N_t \cdot (N_a \cdot A + I) \quad (3.3) \]

The term \( N_t \) in Eq. 3.3 is the number of tuples for an object. However, the term is not used in the other storage costs; thus, we need to replace \( N_t \) with attribute-related terms.

In order to understand how many tuples should be created for an object, let consider an object which has two attributes \( a \) and \( b \). Let \( \Gamma_{<a>} \) and \( \Gamma_{<b>} \) be time lines for attribute \( a \) and \( b \), respectively. Suppose the attribute \( a \) and \( b \) are updated every \( x \) and \( y \) times.\(^4\) Figure 3.5 illustrates the intervals for the two attributes. The bars on the time lines represent ending time instants.

![Figure 3.5 Time intervals for attributes a and b](image)

The number of tuples for an object, \( N_t \), is the exactly same as the distinct number of intervals, which can be derived by overlapping time lines for all attributes in the object. By

---

\(^4\)Although we assume that \( a \) and \( b \) are regularly updated in our example, it is not necessarily to be true in general.
overlapping time line $\Gamma_{<a>}$ and $\Gamma_{<b>}$ shown in Figure 3.5, we can find the number of intervals for the object whose attributes are $a$ and $b$ as shown in Figure 3.6, where $\Gamma_{<a,b>}$ is the combined time line for attribute $a$ and $b$.

![Combined intervals for attribute a and b](image)

Figure 3.6 Combined intervals for attribute $a$ and $b$

As shown in Figure 3.6, the number of intervals is the number of bars on the time line $\Gamma_{<a,b>}$. Therefore, finding $N_t$ is the same as counting bars on the time line. In our example, the two attribute $a$ and $b$ are updated every $x$ and $y$ times with respect to $n$ time instants. The total number of bars on $\Gamma_{<a,b>}$ can be counted as follows:

$$N_t = \left\lceil \frac{n}{x} \right\rceil + \left\lceil \frac{n}{y} \right\rceil - \left\lceil \frac{n}{x \cdot y} \right\rceil$$

For example, suppose that the two attributes are Salary and DName attributes and $x$, $y$, and $n$ are 2, 3, and 10, respectively. The total number of tuples for this object should be:

$$N_t = \left\lceil \frac{10}{2} \right\rceil + \left\lceil \frac{10}{3} \right\rceil - \left\lceil \frac{10}{6} \right\rceil = 7$$

Therefore, there are 7 tuples for the object. We can generalize this for $q$ attributes by using the principle of inclusion-exclusion as shown in Eq. 3.4, where $B_i$ is the set of ending time instants of the $i$-th attribute and $|B_i|$ an update frequency of the attribute.
In order to simply Eq. 3.4, let us define a relationship between $\sum_{i=1}^{q} |B_i|$ and the rest of the terms in Eq. 3.4 as follows:

$$p \cdot \left( \sum_{1 \leq i \leq q} |B_i| \right) = \sum_{1 \leq i < j \leq q} |B_i \cap B_j| - \sum_{1 \leq i < j < k \leq q} |B_i \cap B_j \cap B_k| + \cdots + (-1)^{q+1} |B_1 \cap B_2 \cap \cdots \cap B_q|$$  \hspace{1cm} (3.5)

Now, we can derive Eq. 3.6 from Eq. 3.4 and Eq. 3.5 as follows:

$$|B_1 \cup B_2 \cup \cdots \cup B_q| = \sum_{1 \leq i \leq q} |B_i| - p \cdot \left( \sum_{1 \leq i \leq q} |B_i| \right)$$

$$= \left( 1 - p \right) \sum_{1 \leq i \leq q} |B_i|$$

$$1 - p = \frac{|B_1 \cup B_2 \cup \cdots \cup B_q|}{\sum_{1 \leq i \leq q} |B_i|}$$  \hspace{1cm} (3.6)

From Eq. 3.6, we can inference the meaning of $p$. Suppose $p = 0.3$ so that Eq. 3.6 is 0.7. It means that 30% of intervals share same ending time instants with some intervals. Note that the overlapping ending time instants are not necessarily in all sets, which means $x \in \bigcup_{1 \leq i < j \leq q} B_i \cap B_j$, where $x$ is an ending time instant.

We can rewrite Eq. 3.6 in terms of $N_a$, $U$, and $p$ because $q$ is the number of sets which is the same as the number of attributes and $|B_i|$ is the update frequency of the $i$-th attribute in the relational storage. Eq. 3.7 shows the replacement.
\[ |B_1 \cup B_2 \cup \cdots \cup B_q| = (1 - p) \sum_{1 \leq i \leq q} |B_i| \]
\[ N_{i.} = (1 - p) \cdot N_a \cdot U \quad (3.7) \]

Therefore, \(S_R\) can be rewritten as shown in Eq. 3.8:

\[ S_R = N_0 \cdot (1 - p) \cdot N_a \cdot U \cdot (N_a \cdot A + I) \quad (3.8) \]

### 3.4.3 Cost of Object-Oriented Storage

In the object-oriented storage, the size of an object can be estimated by summing the sizes of all attributes, the sizes of all temporal elements, and the sizes of all object overheads for the attributes. Eq. 3.9 shows how to estimate the size of an object in the object-oriented storage.

\[ \text{Size}(a) = \sum_{i=1}^{N_a} ((\text{Size}(a_i) + T + 2 \cdot C)) \cdot u_i + C \quad (3.9) \]

Note that each attribute is an object, but \(\text{Size}(a_i)\) denotes the size of the \(i\)-th attribute field. It does not include class information or an object overhead. To compensate this, we add an object overhead \(C\) for every attribute. In Eq. 3.9, the term \(2 \cdot C\) is used for the object overheads for an attribute and a temporal element. Since each object has also an object overhead, we add \(C\) for every object at the end of Eq. 3.9.

Now, we can define the storage cost of the object-oriented storage as shown in Eq. 3.10.

\[ S_O = \sum_{j=1}^{N_o} \left( \sum_{i=1}^{N_a} ((\text{Size}(a_i) + T + 2 \cdot C)) \cdot u_i + C \right) \]
\[ = \sum_{j=1}^{N_o} (N_a \cdot (A + T + 2 \cdot C) \cdot U + C) \]
\[ = N_o \cdot (N_a \cdot (A + T + 2 \cdot C) \cdot U + C) \quad (3.10) \]
3.4.4 Cost of XML-based Storage

In the XML-based storage, a single tuple encapsulates a real world object. The size of a single tuple in the XML-based storage can be expressed as follows:

\[
\text{Size}(o) = \sum_{i=1}^{N_a} \left( (\text{Size}(a_i) + T + 4 \cdot \text{Tag}) \cdot u_i + 2 \cdot \text{Tag} \right) + 2 \cdot \text{Tag} \quad (3.11)
\]

In Eq. 3.11, the term, \(\text{Size}(a_i) + T + 4 \cdot \text{Tag}\), measures the size of an attribute value and its domain. The domain is a temporal element and is included in the value. To represent a value and a temporal element, the XML-based needs 4 tags. The XML-based storage needs an opening and a closing tags for every attribute, which requires \(2 \cdot \text{Tag}\) in the summation. Finally, a tuple also needs its opening and closing tags so that \(2 \cdot \text{Tag}\) is added at the end of Eq. 3.11.

Using Eq. 3.11, the cost of the XML-based storage can be defined as follows:

\[
S_X = \sum_{j=1}^{N_a} \text{Size}(o_j) = \sum_{j=1}^{N_a} \left( \sum_{i=1}^{N_a} \left( (\text{Size}(a_i) + T + 4 \cdot \text{Tag}) \cdot u_i + 2 \cdot \text{Tag} \right) + 2 \cdot \text{Tag} \right)
\]

\[
= \sum_{j=1}^{N_a} \left( N_a \cdot ((A + T + 4 \cdot \text{Tag}) \cdot U + 2 \cdot \text{Tag}) + 2 \cdot \text{Tag} \right)
\]

\[
= N_o \cdot \left( N_a \cdot ((A + T + 4 \cdot \text{Tag}) \cdot U + 2 \cdot \text{Tag}) + 2 \cdot \text{Tag} \right) \quad (3.12)
\]

3.5 Storage Cost Comparisons

In the previous section, we have estimated the storage costs for the three platforms. The following shows the equations for storage costs:
In this section, we will simulate the storage efficiency based on various settings. We default the variables in the formulas as follows:

\[ S_R = N_0 \cdot (1 - p) \cdot N_a \cdot U \cdot (N_a \cdot A + I) \]
\[ S_O = N_0 \cdot (N_a \cdot (A + T + 2 \cdot C) \cdot U + C) \]
\[ S_X = N_a \cdot \left( N_a \cdot ((A + T + 4 \cdot Tag) \cdot U + 2 \cdot Tag) + 2 \cdot Tag \right) \]

Figure 3.7 shows the storage efficiency for the default values. We increase the number of objects, \( N_0 \), up to 3,000. For the default values, the XML-based storage shows better storage efficiency than the object-oriented storage, but it is worse than the relational storage. However, the graph shows a positive indication that XML can be used as an implementation platform for the parametric data model because the relational storage is marginally better than the XML-based storage for the default values.

Figure 3.8 shows the storage efficiency changes for \( p \) value variations. The graph shows that the relational storage achieves better storage efficiency than the XML-based storage when \( p \geq 4.0 \). It means that more than 40% of intervals should share same ending time instants with another interval in order to achieve better storage efficiency than the XML-based storage.

Figure 3.9 shows the storage efficiency for \( U \) value variations. We increase \( U \) value from 0 to 200. As we can see, the relational storage shows the best storage efficiency than the others. However, the graph shows that the relational storage is marginally better than the XML-based storage.

The XML's myth is mainly caused by its verbose tags. Therefore, it is important to investigate the relationship between tag sizes and storage efficiency. Figure 3.10 shows the storage efficiency for \( Tag \) value variations. It shows that the XML-based storage can expect better storage efficiency than the relational storage if \( Tag < 3 \) bytes while it achieves better stor-
Figure 3.7  Storage efficiency for default values

Figure 3.8  Storage efficiency for $p$ variations
Figure 3.9 Storage efficiency for $U$ variations

Figure 3.10 Storage efficiency for average tag size variations
age efficiency than the object-oriented storage if $Tag < 10$ bytes. The parametric data model uses less than 20 predefined terms when representing a relation such as tuple, attribute, and parametric element, which means that we can represent the terms in 1-byte characters. Therefore, we can sufficiently define XML elements for the terms with 4 bytes including opening and closing brackets.

The XML-based storage can save XML's tag size further without using opening and closing tags. It can use a single tag approach [49]. Figure 3.10 also shows the variations of storage efficiency for the single tag approach. If we use the single tag approach, the XML-based storage achieves better storage efficiency than the relational storage if $Tag < 6$ bytes while it achieves better storage efficiency than the object-oriented storage if $Tag < 20$ bytes.

When an object is stored in a disk, the object-oriented storage requires that an object be stored with additional object information such as a serial number, type information, and size information for the object. Therefore, it is worth discussing how much such object overhead affects storage efficiency. Figure 3.11 shows the storage efficiency of the object-oriented storage for $C$ value variations. The graph shows that the object-oriented storage is better than the XML-based storage if the additional information is less than 8 bytes while it is better than the relational storage if the information is less than 5 bytes. From the graph, we can expect that the XML-based storage can achieve generally better storage efficiency than the object-oriented storage because an 8-byte object overhead can be considered as very small amount of information.

In the relational storage, an object is represented in multiple tuples so that the size of attributes may affect storage efficiency of the relational storage. Therefore, it is worth discussing the storage efficiency for the average attribute changes. Figure 3.12 shows the storage efficiency for the attribute size changes. The graph indicates that the XML-based storage is better than the relational storage if $A > 13$ bytes. The relational storage is even worse than the object-oriented storage if $A > 28$ bytes.
Figure 3.11  Storage efficiency for object overhead variations

Figure 3.12  Storage efficiency for average attribute sized variation
3.6 Summary and Discussion

The parametric data model captures a real world object in a single tuple. This feature reduces query language complexity by avoiding self-joins. Despite its modeling advantages, such a feature makes it difficult to implement the data model on top of an existing relational database, which is the most popular approach for other dimensional data models. The implementation challenge is mainly because of unfixed attribute sizes. In order to implement the parametric data model, we need more flexible storage structures than existing relational databases.

XML is an emerging technology and many researchers have utilized it in various research areas. Because of XML's flexibility, we can use XML as an implementation platform for the parametric data model. However, one may raise a question about XML's storage efficiency because of its verbalism. In order to investigate whether XML is suitable for the parametric data model, we have evaluated the three storages—relational, object-oriented, and XML-based storages.

Our simulation results have revealed that the XML-based storage is reasonable for the parametric data model. For default values, the XML-based storage has shown comparable storage efficiency to the relational storage. However, we have noted that the XML-based storage is better than the relational storage if less than 40% of intervals share their ending time instants with another; in other words, more than 40% of updates should occur at least two attributes at the same time in order for the relational storage to be better. We have also noted that the average attribute size significantly affects the storage efficiency of the relational storage, which means that the XML-based storage can achieve the best storage efficiency than the others if the average attribute size is greater than 13 bytes. In general, the XML-based storage has shown better storage efficiency than the object-oriented storage.

Our simulations of storage costs quantify the usability of XML for the parametric data model. Moreover, the results reveal a strong evidence such that the myth about XML is not true for the parametric data model. Therefore, our XML-based approach is a promising solution to meet the implementation challenge of the parametric data model.
CHAPTER 4 XML-BASED IMPLEMENTATION OF PARAMETRIC TEMPORAL DATABASE

One-to-one correspondence between an object in the real world and a tuple in the parametric data model provides various advantages in modeling dimensional data. Despite its modeling advantages, it is challenging to implement the parametric data model because it is impractical to build the data model on top of conventional databases. However, such implementation challenge can be resolved by utilizing XML technology. In this chapter, we will discuss the methodology used to overcome the implementation challenges of the parametric data model for temporal data.

4.1 Introduction

Many dimensional data models tend to fragment a real world object into multiple tuples to utilize conventional databases. As pointed by Nørvåg [46], however, such implementation approach cannot be a long term solution for dimensional databases because conventional databases are designed to support ordinary data, not dimensional data, leading to complex query languages as well as expensive query execution. Steiner and Norrie [57] also noted that implementing a database system without relying on existing database systems (or storages) can lead to various advantages from storage efficiency to optimization.

In the previous chapter, we have discussed the storage efficiency of XML for the parametric data model. The storage efficiency of XML provides the practical soundness of our XML-based approach to implementing the parametric data model. In this chapter, we focus on the practical implementation of the data model by using XML technology without using any conventional databases and existing storage technologies. We develop our own XML-based storage to manage
XML documents for the parametric data model.

In general, a storage manager divides a large data into smaller ones called pages which can be retrieved into a main memory. We introduce a dynamic pagination algorithm which can paginate large XML documents into small self-contained XML pages. The dynamic pagination algorithm plays an important role in building an XML-based storage called CanStoreX (Canonical Storage for XML) [35]. The XML-based temporal database system to be discussed in this chapter is implemented on top of CanStoreX. Therefore, our implementation is different from other approaches in that it is built on its own storage, not using conventional databases including relational, object-oriented, and native XML databases.

In addition to this, many artifacts, such as parse tree and expression tree are also represented in XML. We will also show how to utilize XML in developing a query execution engine. The primitives for the query language are implemented using the DOM parser. We validate our system with nine queries characterized into five categories.

The rest of this chapter is organized as follows. Section 4.2 formalizes a schema for the parametric temporal data model. Section 4.3 describes the dynamic XML pagination algorithm. Section 4.4 introduces some examples of ParaSQL queries. Section 4.5 provides the methodology to build the XML-based parametric temporal database system. Section 4.6 discusses the system performance for ParaSQL queries. Section 4.7 summarizes and discusses this chapter and our findings.

4.2 Formalization of the Parametric Data Model Schema

In this section, an XML-based schema for the parametric data model will be introduced, which is called the model schema. To enhance readability, the schema is represented as a tree. Since the schema is an XML document, it consists of nodes and edges like an ordinary XML document. The schema, however, has a few definitions which provide a transformation from the parametric relational model to an XML document without losing any properties of the parametric data model.
4.2.1 Definitions

- **Definition 1: The model schema for the parametric data model**
  Let $S$ be a schema for the parametric temporal data model. $S$ is defined as $S = (N, E)$, where $N$ is the set of nodes and $E$ is the set of edges.

- **Definition 2: Node**
  Let $N$ be the set of nodes in $S$. There are two types of sets—element sets and value set, denoted as $N^e$ and $N^v$, respectively. An element node is a non-terminal node, which has at least one child node. In contrast, a value node is a terminal node that has no child. Two sets are disjoint, that is, $N^e \cap N^v = \emptyset$.

- **Definition 3: Element**
  The set of elements, $N^e$, consists of five types of elements—$Dunit$ (Domain Unit), $Pdom$ (Parametric Domain), $Tunit$ (Terminal Unit), $Xunit$ (Attribute Unit), and $Iunit$ (Internal Unit) elements.
  
  - **Dunit element**: It is a parent element of a value node that represents a maximal continuous time domain of a value (time interval). The set of Dunit elements is denoted as $N^d$.
  
  - **Pdom element**: It is a parent element of Dunit elements and represents a temporal element of an object. The set of Pdom elements is denoted as $N^p$.
  
  - **Tunit element**: It is a parent element of a Pdom element and a value node. It represents a data value and its domain, that is, a temporal value. The set of Tunit elements is denoted as $N^t$.
  
  - **Xunit element**: It is a parent element of a Pdom element and a set of Tunit elements. It is used to represent an attribute. The set of Xunit elements is denoted as $N^x$.
  
  - **Iunit element**: It is a parent element of a Pdom element and either Iunit or Xunit elements. It is used to internally connect Iunit elements or Tunit elements. It
represents either a tuple or a relation. The set of Iunit elements is denoted as \( N^i \).

- **Definition 4: Edge**

  Let \( E \) be the set of edges in \( S \). An edge \((p, c) \in E\) is one of following, where \( p \) is a parent node and \( c \) is a child node: 
  \((p \in N^d, c \in N^v)\); 
  \((p \in N^p, c \in N^d)\); 
  \((p \in N^i, c \in N^p)\); 
  \((p \in N^x, c \in N^i)\); 
  \((p \in N^i, c \in N^i)\); 
  \((p \in N^i, c \in N^p)\); 
  \((p \in N^i, c \in N^v)\).

### 4.2.2 Transformation

Figure 4.1 shows the model schema for parametric temporal relations. A relation is mapped to the root node in the tree. Each tuple is mapped to a Tuple node that is an Iunit element. Each Tuple node has attribute nodes represented by Xunit elements. Each attribute node has a parametric value and its domain represented by a Tunit element and a Pdom element, respectively.

![Figure 4.1 Parametric model schema for parametric relation](image)

We must note that the schema represents the parametric relational data model. However, the schema is designed to reduce search time for domain-related queries, providing domain information at attribute-level, tuple-level, and relation-level. Because domain-related queries are frequently posed by users, fast responses to such queries are critical in temporal database systems. The effectiveness of the model schema will be discussed in Section 4.6.2.
The effective transformation from one data model to another model can be determined by the investigation on whether the target modeling scheme satisfies the base model's properties. The formal proof can be found in a technical report [40].

4.3 Dynamic Pagination Algorithm

In this section, we introduce the dynamic XML pagination algorithm which can paginate a large XML document into multiple self-contained XML pages. The algorithm has been used in building XML-based storage called CanStoreX [35]. Its variation version, CanStoreX's variable-size node strategy, manages XML documents in binary formats and inflates only 28% for an 1GB XML document when paginating it into 4KB multiple pages with 70% page utilization [49].

Algorithm 1 shows the dynamic XML pagination algorithm. This algorithm partitions an XML document into small pages by depth first traversing. In this algorithm, $\text{MIN}$ and $\text{MAX}$ stand for the minimum and maximum size thresholds of a small self-contained XML document. Once the threshold is met, storage facilitating nodes, $f$-node and $c$-node, are added to a paginated XML document and a pruned subtree. A $c$-node is added to combine subtrees and points to a page containing its subtree(s). An $f$-node is added to the pruned subtrees as a root node. It is an artificial parent node of the subtree. An $f$-node has a unique page identification so that the buffer manager can identify which page should be uploaded into a buffer pool.

In this algorithm, the function $\text{TAG}$ tags a node with a number to indicate how many times the node has been visited (at most 2). Since this algorithm uses the depth first search, a node tagged with 2 means that all of the node's subtrees including itself have already been traversed. Once a node has been visited twice, the algorithm evaluates the size of tree to check if it meets the threshold of pagination. If this is the case, the subtree will be pruned and stored in a page, which is a self-contained XML document. The page is identified by an $f$-node. The pruned node in the main XML document is replaced with a $c$-node pointing to the $f$-node.

Figure 4.2 through Figure 4.6 illustrate how the dynamic XML pagination algorithm paginates an XML document into small pages. There are four cases that the function $\text{PAGINATION}$ is invoked as follows:
Algorithm 1 Dynamic XML Pagination Algorithm

1: procedure DYNAMIC_PAGINATION(node)
2:     if node is null then
3:         return
4:     else if node has been visited twice then
5:         TAG(node, 2) ▷ tag the node with '2'
6:         if node is the root then
7:             PAGINATION(node)
8:             exit
9:     end if
10:    curSize ← SUBTREE_SIZE(node) ▷ get size of node's subtree
11:    sibSize ← SiblingTREE_SIZE(node) ▷ get size of sibling's subtree
12:    if MIN ≤ curSize + sibSize ≤ MAX then ▷ check the threshold
13:        PAGINATION(node)
14:     end if
15:    else if node has been visited first then
16:        TAG(node, 1) ▷ tag the node with '1'
17:        node ← node.getFirstChild() ▷ get the first child
18:        DYNAMIC_PAGINATION(node) ▷ recursive call
19:        node ← node.getSibling() ▷ get the sibling of the node
20:     end if
21: end procedure

1. when the size of a subtree is greater than the minimum threshold, but less than the maximum threshold.
2. when the size of a subtree and its siblings is greater than the minimum threshold, but less than the maximum threshold.
3. when the size of a subtree and its siblings is greater than the maximum threshold.
4. when the size of a subtree and its siblings is less minimum threshold and the current node is a root node.

Figure 4.2 illustrates case 1. Figure 4.2-(a) shows an original XML document, which is not paginated. Since our pagination algorithm uses the depth first search, shaded nodes indicate that the nodes have been visited once while black nodes indicate that they have been visited
twice.

Figure 4.2 \( \text{size(current)} \geq \text{minimum threshold} \) [35]

Since the size of current subtree exceeds a threshold, the subtree in Figure 4.2-(a) is pruned.

The root of the subtree is replaced with a \textit{c-node} in the original tree. The pruned subtree is attached as a child node of an \textit{f-node} which indicates a page identification.

Figure 4.3 illustrates case 2. The subtree and its sibling(s) are replaced with a \textit{c-node}. In our example, it says that the pruned subtree and its siblings are stored in page 2. We must note that if the subtree's sibling has been pruned in a previous stage, the sibling node is a \textit{c-node} indicating a page that the actual subtree is stored.

Figure 4.3 \( \text{size(current)} + \text{size(sibling)} \geq \text{minimum threshold} \) [35]

Figure 4.4 shows case 3, but the size of current subtree and its siblings exceeds the maximum threshold. In this case, only the current subtree is pruned and paginated. In our example, the current subtree is stored in page 3.

Figure 4.5 shows case 2. In this example, only the root node and its rightmost child are
not completely visited yet. The subtrees visited twice from Figure 4.4 are pruned and they are replaced with a c-node pointing to page 4. The pruned subtrees are stored at page 4 with an f-node. As shown, two c-nodes are pointing to page 2 and page 3, respectively.

Figure 4.5 size(current) + size(sibling) ≥ minimum threshold [35]

Figure 4.6 shows a final paginated XML document and illustrates case 4. Once the root node is visited twice, the root node and its children are stored in a reserved page called root page.

Kanne and Moerkotte [32] introduced the Natix storage that paginates a large XML document into a set of pages. Natix is a storage technology for XML documents. In order to facilitate pagination of an XML document, Natix as well as CanStoreX adds some auxiliary
nodes to the document. However, there is a significant difference between the two approaches. In Natix, a page consists of several (small) XML elements while a page generated by the dynamic pagination algorithm is a self-contained XML document on its own right. To a client of our DOM API, auxiliary nodes and page boundaries are transparent. The fact that each page is a self-contained XML document has some interesting advantages for system development.\footnote{Comparing Natix with CanStoreX is beyond the scope of this dissertation because our main aim of this chapter is to illustrate our XML-based implementation of the parametric data model for temporal data.}

### 4.4 ParaSQL Examples

In this section, we introduce nine ParaSQL queries such as relation scan, snapshot, interval, temporal element, NOW query for all information, NOW query for current information, history, nested query, and temporal join. We also group this nine queries into five types based on query properties. ParaSQL queries are based on $\textit{Emp}$ and $\textit{Dept}$ relations, where Name is the key of
Emp relation and DName is the key of Dept. relation, respectively.

\[
\text{Emp(}\text{Name, Salary, DName)} \quad \text{Dept(DName, MName)}
\]

- **Type 1: Relation Scan**

Query 1-(a) Retrieve entire employee information. (Simple scan)

\[
\text{SELECT *}
\quad \text{FROM Emp E}
\]

Query 1-(b): Retrieve employees who worked at time instant 10. (Snapshot)

\[
\text{SELECT *}
\quad \text{RESTRICTED TO } [10, 10]
\quad \text{FROM Emp E}
\]

Query 1-(c): Find all employees who worked throughout 10 to 200. (Interval)

\[
\text{SELECT *}
\quad \text{RESTRICTED TO } [10, 200]
\quad \text{FROM Emp E}
\]

Query 1-(d): Find all employees who worked when Software department ([20,200]) or Hardware department ([301,400]) existed. (Temporal element)

\[
\text{SELECT *}
\quad \text{RESTRICTED TO } [20, 200] + [301, 400]
\quad \text{FROM Emp E}
\]

In this example, we assume that Software department existed from 20 to 200 and Hardware department existed from 301 to 400. In the RESTRICTED TO clause, the notation \( + \) represents a union. Therefore, \([20, 200] + [301, 400]\) means \([20, 200] \cup [301, 400]\).

Query 1-(a) through Query 1-(d) are grouped into a same query because they are all simple relation scans. As discussed in Section 2.4.1 in Chapter 2, all omitted clauses such
as RESTRICTED TO and WHERE clauses are recovered. Therefore, Query 1-(a) through Query 1-(d) share the same property.

• Type 2: NOW Query

Query 2-(a): Give name and salary (history) of all employees working currently. (Now query for all information)

SELECT E.Name, E.Salary
FROM Emp E
WHERE NOW SUBSET \[[E]\]

In WHERE clause, this ParaSQL query uses a Boolean condition expressed by domain expressions with a set operation. If an employee is working currently, all information about the employee will be retrieved. However, if we want to retrieve current employee information, then the ParaSQL should be rewritten as Query 2-(b):

Query 2-(b): Give current name and salary information of all employees. (Now query for current information)

SELECT E.Name, E.Salary
RESTRICTED TO NOW
FROM Emp E

Query 2-(b) restricts the domain of employees to a current time. Therefore, only employees who are currently working will be retrieved with their current information.

• Type 3: History Query

Query 3: Retrieve the salary history of employee Bob.
SELECT E.Salary
FROM Emp E
WHERE E.Name = "Bob"

• Type 4: Nested Query

Query 4: Give employee’s name and salary history during John was a manager of Software department.

SELECT E.Name, E.Salary
RESTRICTED TO [[
  SELECT *
  RESTRICTED TO [[[D.DName="Software"]]]
  FROM Dept D
  WHERE D.MName = "John"]]
FROM Emp E

This ParaSQL query retrieves employee tuples from Emp relation and restricts the domain of each tuple a temporal element constructed by RESTRICTED TO clause. The RESTRICTED TO clause has a domain expression, the type of [[<relational expression>]]. The relational expression is a select statement which returns a set of tuples or a relation whose MName is John. Such tuples are restricted to the domain expression [D.DName="Software"]. The union of tuple’s domains is a temporal element used to restrict domains of employee objects.

It is worth noting that Emp relation is not used by the inner query. Since the inner query is independent from the outer query, they can be computed separately. Therefore, relation scans for inner and outer relations are enough to execute this ParaSQL query. This observation explains the conceptual difference between the temporal element-based data model and interval-based data models because in the interval-based data models, this query requires a join.
• Type 5: Temporal Join

Query 5: List all pairs of employees and departments such that the salary of an employee is greater than $60,000 and the employee worked or are working in the department.

```sql
SELECT E.Name, D.DName
RESTRICTED TO [[E.Dept = D.DName]]
FROM Emp E, Dept D
WHERE E.Salary > 60000
```

This query needs to join Emp and Dept relations. The join condition is expressed by a domain expression in RESTRICTED TO clause because a joined tuple should be restricted to the domain such that the employee's DName is equal to the department's DName.

4.5 System Architecture

In this section, we will discuss the system architecture of an XML-based temporal database system. The database storage and all internal exchange of data are XML-based. Figure 4.7 shows the three-layer system architecture of the framework [45, 42]. Layer-1 focuses on query processing such as parsing ParaSQL queries, generating expression trees and query plans. Layer-2 executes ParaSQL queries using DOM API. Parse trees, expression trees, and query plans are all XML documents. Layer-3 is a paginated XML storage. This storage provides DOM API to access and retrieve temporal tuples represented by XML.

4.5.1 Query Processing Layer

The purpose of Query Processing Layer is to generate a physical query plan from ParaSQL queries.\(^2\) The ParaSQL Parser parses ParaSQL queries and generates tree structures of the queries. Suppose a user poses Query 5 discussed in Section 4.4. The ParaSQL query is parsed as shown in Figure 4.8. The parse tree is internally represented in XML.

\(^2\)Since our proposed system is early stage of new implementation, Optimizer and Physical Query Planner are not discussed in this dissertation. They will be covered in our future work. Therefore, an expression tree is directly passed to Query Execution Layer.
The XML-based parse tree is passed to the Logical Query Planner that generates an expression tree for the parse tree. The expression tree is also internally represented in XML. Figure 4.9-(a) shows the expression tree for Query 5. The expression tree is an abstract-level description for the query. Figure 4.9-(b) shows a simplified XML representation for the expression tree. In the XML-based expression tree, the nodes for projection, restriction, and where are at the same level. They do not have parent-child relationship, but they are siblings, which is different from the abstract expression tree. However, they have the same functionality because the ParaSQL query execution engine traverses the expression tree in depth first order.³

³It depends on implementation choices.
Figure 4.8  A parse tree for the ParaSQL query
The XML representation has annotation nodes to provide information on a query execution. For example, Iterator element has Annotation element that has information about which iterator should be selected from an iterator pool.

Representing parse trees and expression trees in XML has an implementation benefit. It can help reduce our reliance on linked-list-based implementation for parse trees and expression trees because XML provides an effective navigation mechanism with DOM API.

![Expression tree for Query 5](image)

(b) A simplified XML-based expression tree

(a) Expression tree for Query 5

Figure 4.9 Expression tree and its XML representation

We must note that the expression tree shown in Figure 4.9-(a) is a standard form, not an optimized form. The standard form is to illustrate how our proposed approach handles expression trees. It is up to an optimizer to eliminate or combine nodes from the expression tree. Therefore, $\Pi_x$, $\mu_{universal}$, and $\sigma_{true}$ in the expression tree can be eliminated by the optimizer. It is also worth noting that $n$-way join can be represented in a standard form of expression trees.
4.5.2 Query Execution Layer

The second layer is Query Execution Layer. This layer consists of five subsystems as shown in Figure 4.7. The Query Executor manages the other subsystems. It calls either Data Manipulator or Data Definer. Data Manipulator calls the iterators implemented by using DOM API. Data Definer creates databases and tables. The Query Executor processes a query plan based on an expression tree.

Algorithm 2 shows the query execution procedure. It determines which iterator should be used to process a given expression tree. The expression tree has information on iterators and the Query Executor extracts the information. Once an iterator is determined, the iterator retrieves tuples from temporal relation(s) and qualifies them with EVALUATION function. A qualified tuple is passed to RESTRICTION function as an argument. The RESTRICTION function restricts the domains of tuples to a temporal element constructed from the expression tree.

```
Algorithm 2 Query Execution Algorithm

1: procedure QUERYEXECUTION(e) ▷ e: expression tree
2:   if e has join condition then
3:     it ← JOIN(e) ▷ it: iterator
4:   else if e has a relation scan then
5:     it ← RELATIONSCAN(e)
6:   end if
7:   while it.hasNext() = true do
8:     tuple ← it.getNext() ▷ retrieve a tuple
9:     tuple ← RESTRICTION(e, tuple, null)
10:    if tuple ≠ null then
11:       output(tuple) ▷ write a tuple
12:   end if
13: end while
14: end procedure
```

The most important functions in executing queries are evaluation functions for RESTRICTED TO and WHERE clauses. Even though they are different functionalities, they share the base function because they are mutually recursive. Algorithm 3 explains how to process WHERE and RESTRICTED TO clauses. The functions, EVALUATION and RESTRICTION, call NEXTEVAL
Algorithm 3 Query Evaluation Algorithm

1: procedure EVALUATION(e, tup1, tup2)  \(\triangleright e: \text{exp. tree, } \text{tup}: \text{tuple}\)
2: \[\text{result} \leftarrow \text{NEXTEVAL(e.firstChild(), tup1, tup2)}\]
3: \[\text{return result} \quad \triangleright \text{Boolean value}\]
4: end procedure

5: procedure RESTRICTION(e, tup1, tup2)
6: \[\text{domain} \leftarrow \text{RESTRICTIONDOMAIN(e, tup1, tup2)}\]
7: \[\text{tup} \leftarrow \text{NEXTEVAL(e.firstChild(), tup1, tup2)}\]
8: \[\text{result} \leftarrow \text{RESTRICTTUPLE(tup, domain)}\]
9: \[\text{return result} \quad \triangleright \text{time domain}\]
10: end procedure

11: procedure NEXTEVAL(c, tup1, tup2) \(\triangleright c: \text{condition node}\)
12: \[\text{if } c \text{ is a type of Boolean expression then}\]
13: \[\text{result} \leftarrow \text{BOOLEANEXPRESSION(c, tup1, tup2)}\]
14: \[\text{else if } c \text{ is a type of domain expression then}\]
15: \[\text{result} \leftarrow \text{DOMAINEXPRESSION(c, tup1, tup2)}\]
16: \[\text{else if } c \text{ is a type of relational expression then}\]
17: \[\text{result} \leftarrow \text{RELATIONALEXPRESSION(c, tup1, tup2)}\]
18: \[\text{end if}\]
19: \[\text{return result} \quad \triangleright \text{result is an abstract type}\]
20: end procedure

function. But they are different in that EVALUATION function returns either \texttt{true} or \texttt{false} while RESTRICTION function returns a \texttt{domain} (or temporal element). It is worth noting that the NEXTEVAL function reflects the three expressions of ParaSQL such as relational expression, domain expression, and Boolean expression.

4.5.3 Storage Management Layer

The Storage Management Layer handles page requests from its upper layer. It provides a requested page from a disk. This layer manages paginated XML data. Whenever it receives a request, it retrieves one page at a time from the disk. It provides DOM API so that iterators can retrieve nodes from loaded pages. Storage Management Layer has a buffer manager to reduce the number of disk accesses for repeatedly used pages.
4.6 Performance Evaluation

4.6.1 Test Data Configuration

There are many XML data synthesizers such as ToXgene [4] and IBM XML Generator [27]. However, we found that it is difficult to synthesize XML data that has time features. Therefore, we created the test data by our own definitions as follows:

1. We only increase employee information when increasing the size of databases.

2. Each employee tuple has salary and department history information for more than 30 years (day is granule).

3. In employee tuples, salary increases $100 every year and department information is updated every five years.

Based on our definitions, we have created six data for measuring the effectiveness of the model schema and system performance for the nine ParaSQL queries. Table 4.1 shows data setup information about 6 different test cases.

Table 4.1 The test data for the model schema effectiveness

<table>
<thead>
<tr>
<th>Notation</th>
<th>Data Size</th>
<th>DB Size</th>
<th># of pages</th>
<th>Capacity</th>
<th># of Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>10KB</td>
<td>12</td>
<td>20</td>
<td>5</td>
<td>0.59</td>
<td>3</td>
</tr>
<tr>
<td>100KB</td>
<td>105</td>
<td>152</td>
<td>38</td>
<td>0.69</td>
<td>36</td>
</tr>
<tr>
<td>1MB</td>
<td>1,027</td>
<td>1,472</td>
<td>368</td>
<td>0.70</td>
<td>363</td>
</tr>
<tr>
<td>10MB</td>
<td>10,242</td>
<td>14,680</td>
<td>3,670</td>
<td>0.70</td>
<td>3,633</td>
</tr>
<tr>
<td>100MB</td>
<td>102,420</td>
<td>146,784</td>
<td>36,696</td>
<td>0.70</td>
<td>36,330</td>
</tr>
<tr>
<td>1GB</td>
<td>882,489</td>
<td>1,495,012</td>
<td>373,753</td>
<td>0.70</td>
<td>372,385</td>
</tr>
</tbody>
</table>

* The unit of data and database sizes is kilo bytes (KB).
* Page size = 4KB.

The capacity column in Table 4.1 shows the average occupation of pages in each database file. The column of the number of employees shows how many employee tuples reside in each paginated database.
4.6.2 Effectiveness of the Model Schema

In Section 4.2, we have discussed the mapping of the parametric data model into the model schema. The proposed schema differs from the parametric relational data model because it has domain information at attributes, tuples, and even relations. Since domain expressions are pivotal in temporal databases, it was advisable to design the model schema to reduce response times to domain queries.

With the model schema, we can expect that domain related query can be processed without traversing entire data. For example, if we want to restrict the domain of a department, we must traverse the department’s child nodes to get its domain. However, if we use the model schema, retrieving a domain node is enough to check if there exists a common domain to restrict the department.

Figure 4.10 shows the experimental result for two different approaches such that one has the domain information at the tuple-level and the other one does not. For each database, our proposed schema saved more than 60% of disk requests for Query 1 and Query 2 except Query 1-(a).

\[ \text{Effectiveness of the XML Schema} \]

![Figure 4.10 Schema effectiveness (logarithmic scale)](image-url)
4.6.3 Experimental Results

We evaluated the system for the nine ParaSQL queries discussed in Section 4.4. For those queries in Type 1, they are all relation scans. They request 747,505 pages. Among those requests, total 375,120 pages are actually accessed. Despite the fact that Query 1-(a) through Query 1-(d) are different types of queries in the user's viewpoint (relation scan, snapshot, interval, and temporal element), they are the same query types in the ParaSQL's viewpoint. As discussed in Section 4.4, ParaSQL recovers the omitted clauses such as `RESTRICTED TO` and `WHERE` clauses. If there is no `WHERE` clause, the Boolean condition is set to true. If there is no `RESTRICTED TO` clause, the domain expression is set to the universal time. In ParaSQL, the universal time, interval, and time instant are all temporal elements. Therefore they have similar expression trees, hence they have the similar performance results.

For queries in Type 2, these queries can explained in the same way of Type 1 because they are also relation scan queries. However, we separate them as a different group because of its temporal property, e.g., NOW query.

Type 3 is a history query. Without an index, this query shows the same performance as Type 1 and Type 2. However, with an index, we can significantly reduce the number of disk accesses. To achieve this, a B+-tree can be implemented to support faster retrievals of tuples by their Name-values. Assuming 30 bytes to store a name value, 4 bytes to store a pointer, page size of 4 KB, and a 75% space utilization, it can be shown that the height of the B+-tree will be 4. That means to look for a tuple of an employee whose Name is known, 4 block accesses will suffice.

Query 4 is a nested query. This query requests total 747,505 pages, but 375,122 pages are actually accessed. In this query, we must note that, for every tuple from `Emp` relation, its domain is restricted to a temporal element constructed by a nested query. This approach is similar to a join operation. However, if the inner query is independent from the outer query, we can construct a domain of the inner query at the first time, not for every tuple of the outer

\footnote{The test platform is a Pentium 4 2GHz PC with 512MB memory and 80GB IDE hard drive under Windows XP operating system.}
query. Therefore, the performance for Query 4 is similar to those of Type 1. If the outer and the inner queries are connected by any variables, relation is used in the inner relational expression, the performance will the same as a temporal join.

Query 5 is a temporal join query. This query requests 1,745,658 pages for the execution and total 876,758 pages are actually accessed.

4.7 Summary and Discussion

In this chapter, we have introduced an implementation methodology for the parametric temporal data model. Our approach is different from other approaches because we have developed our own storage technology and query language, rather than using existing storage technologies including relational, object-oriented, and native XML database systems.

In addition to XML's usability in modeling dimensional data, we have noted that parse and expression trees can be represented in XML. Since XML models hierarchical structures naturally, use of XML to represent parse and expression trees reduces our reliance on linked-list data structures, leading to significant savings in design and implementation effort, at the same time, making code more human-readable, hence more reliable.

In Section 4.2.2, it has been shown that the parametric relational data model can be transformed to the model schema. The schema has domain information at the level of attributes, tuples, and even relations. This additional information reduces response times to domain-related queries. Because domain expression is frequently posed by users, reducing disk requests to domain expressions is important in temporal databases. Our experiment shows that the proposed schema saved more than 60% of disk requests.

For the system validation, we have measured the page requests and accesses for the nine ParaSQL queries. We have noted that although some of these queries seem different, they are the same types from the ParaSQL's viewpoint. For example, universal time, time interval, and time instant are all temporal elements in the parametric data model.

Despite extensive research work in temporal databases, it is difficult to find native implementation of temporal database systems rather than extending conventional database systems
or existing storage technologies. As we discussed in this chapter, XML can be used for the native implementation of a database system which is complex to be implemented on top of conventional database systems. Therefore, we hope that our implementation paradigm will provide an elegant solution to meet implementation challenges of other complex dimensional data models.
CHAPTER 5 VALIDATION OF PARAMETRIC DATA MODEL FOR TEMPORAL DATA

Starting from the mid 1980s, there has been a debate about what data model is most appropriate for temporal databases. A fundamental choice one has to make is whether to use intervals of time or temporal elements to timestamp objects and events with the periods of validity. The advantage of using interval timestamps is that Start and End columns can be added to relations for treating them within the framework of classical databases, leading to quick implementation. Temporal elements are finite unions of intervals. The advantage of temporal elements is that timestamps become implicitly associated with values, tuples, and relations. Furthermore, since temporal elements, by design, are closed under union, intersection and complementation, they lead to query languages that are natural. Here, we investigate the ease of use as well as system performance for the two approaches to help settle the debate.

5.1 Introduction

Despite extensive research on temporal data models, it is hard to find model comparisons in ease of use and system performance. In this chapter, we will compare interval-based and temporal element-based data models, evaluate their usability, and measure system performance in terms of disk block accesses. We consider an interval-based data model that uses time-interval and tuple-level timestamping which is a popular data model in the temporal database community.

In this chapter, we introduce ISQL, a hypothetical query language for interval-based data models, to compare with ParaSQL. For carrying out the comparisons, we have developed a query suite which extends the primary version of queries introduced in [31]. The query suite is
stated in plain English; thus, it is independent of data models and query languages.

In addition to this, we implement the two temporal data models. In our implementation, XML is the main implementation platform for the systems. For the storage of the interval-based data model, we follow the industry standard binary structure. However, tuples are encapsulated into XML elements by a primitive iterator. Because of this, a query execution engine of the data model considers tuples as XML elements so that it can seamlessly utilize XML for query evaluations. For the parametric temporal data model, we use the system introduced in Chapter 4.

Our benchmarks essentially compare the complexity of queries of ISQL and ParaSQL and system performance of the two data models. Our benchmarks reveals that ParaSQL is more user-friendly for the query suite than ISQL. The ISQL queries frequently invoke self-joins to combine scattered tuples for an object, resulting in complex queries. One interesting outcome is that although self-joins increase query complexity of ISQL, they do not significantly degrade system performance if a special treatment for self-joins is used. Such special treatment is one-pass algorithm and gives ISQL maximum benefit of the doubt. However, it must be emphasized that ISQL queries perform marginally better than those in ParaSQL where only literal execution is considered.

In addition, we remark that the interval-based data model is dependent upon order properties of instants of time, whereas the parametric data model is based upon set operations on domains. For this reason, the parametric data model extends to spatial and spatiotemporal data whereas interval-based model does not.

The rest of this chapter is organized as follows. Section 5.2 briefly introduces an interval-based temporal data model. Section 5.3 introduces query suite and compares the easiness of query languages. Section 5.4 discusses the implementation of the interval-based temporal database. Section 5.5 compares the system performance of the two systems. Section 5.6 summarizes and discusses our findings.
5.2 Interval-based Model

5.2.1 Overview

We consider an interval-based data model that uses time-intervals and tuple-level timestamping, which is the most popular approach for interval-based data models. In the interval-based approach, in addition to usual attributes, Start and End attributes are used to specify the period of validity for the information in the tuple [53, 37].

Figure 5.1 shows an interval-based temporal relation. The Emp relation maintains the history of employees with name, salary, and department information. As shown in Figure 5.1, every tuple has Start and End attributes indicating the valid time of a tuple.

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>DName</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td>45000</td>
<td>Hardware</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Tom</td>
<td>50000</td>
<td>Sales</td>
<td>41</td>
<td>51</td>
</tr>
<tr>
<td>John</td>
<td>50000</td>
<td>Software</td>
<td>11</td>
<td>44</td>
</tr>
<tr>
<td>John</td>
<td>50000</td>
<td>R&amp;D</td>
<td>45</td>
<td>49</td>
</tr>
<tr>
<td>John</td>
<td>55000</td>
<td>R&amp;D</td>
<td>50</td>
<td>54</td>
</tr>
<tr>
<td>John</td>
<td>60000</td>
<td>R&amp;D</td>
<td>55</td>
<td>60</td>
</tr>
</tbody>
</table>

Figure 5.1 Interval-based Emp relation

XML is the base platform for the interval-based data model. Although the storage handles binary pages, iterators retrieve tuples and encapsulate them into elements because its upper layers handle elements for query executions. The storage layer is transparent to the upper layers, assuming it as an XML storage.

Figure 5.2 shows how tuples in Emp relation of the interval-based data model can be encapsulated in XML. Note that its physical view is binary pages containing tuples like relational databases. As shown in Figure 5.2, Relation element consists of multiple tuple elements. Every tuple element has five attributes. XML-tagged tuples are generated by the storage
manager and passed to the query execution engine which processes XML elements.

5.2.2 Interval-based Structured Query Language

For comparison purposes, ISQL (Interval-based Structured Query Language) is defined. ISQL is a hypothetical query language and a place holder for many interval-based data models. Because it is impractical to implement and compare all interval-based query languages, a hypothetical query language is desired for our purposes. Figure 5.3 shows the skeleton of ISQL select statement.

```
ISQL := SELECT <attribute list>
       [RESTRICTED TO <interval> | <instant>]
FROM <relation list>
       [WHERE <boolean expression>]
```

Figure 5.3 Simplified BNF of ISQL

All clauses in ISQL have the same functionality of classical SQL except RESTRICTED TO
clause. RESTRICTED TO clause is to capture specific time domain information. We must note that the clause can only have either a single time interval or a single time instant. It does not allow a nested form in the clause because not all domains of nested query can be an interval. For example, some tuples in an object can be screened by a Boolean expression so that the domain of a resulting relation should be the union of intervals which is not an interval. In ISQL, nested queries are allowed in WHERE clause like classical SQL. A Boolean expression determines if a given tuple satisfies the Boolean condition. It has the same functionality of classical SQL in that it either qualifies or disqualifies a tuple.

5.3 Query Suite and Query Comparisons

5.3.1 Query Suite Development

Developing a query suite is important in model comparisons and system evaluations. In the literature of temporal databases, we can find many different types of query suites. Jensen et al. [31] have provided a benchmark query suite for temporal databases. The query suite is dependent on TSQL. Wang and Zaniolo [62] have introduced a query suite which is categorized into five types such as relation scan, history, snapshot, interval, and temporal join.

We have developed our own query suite which is extended from the preliminary version of queries introduced in the appendix of [31]. It includes categories introduced by Wang and Zaniolo. However, our query suite has some additional types. For example, some query characteristics like an instant and an interval are known before executing a query (parsing time or explicit). However, sometimes, the characteristics may be implicit before executing a query.

5.3.2 Query Comparisons

Our proposed query suite consists of 10 queries and the schemas used in our benchmarks are as follows:

\[
\text{ISQL:} \\
\text{Emp (Name, Salary, DName, Start, End)} \\
\text{Dept (DName, MName, Start, End)}
\]
ParaSQL:
Emp(Name, Salary, DName)
Dept(DName, MName)

Query 1: Give name and salary (history) of all employees.

ISQL:
SELECT E.Name, E.Salary
FROM Emp E

ParaSQL:
SELECT E.Name, E.Salary
FROM Emp E

ISQL and ParaSQL are the same for the query. However, their internal procedural are different because ISQL retrieves partial information of an object at a time while ParaSQL does whole information for the object.

Query 2: Retrieve the salary history of employee Bob.

ISQL:
SELECT E.Salary
FROM Emp E
WHERE E.Name="Bob"

ParaSQL:
SELECT E.Salary
FROM Emp E
WHERE E.Name="Bob"

ISQL and ParaSQL queries are identical. However, we must note that ParaSQL's WHERE clause and ISQL's WHERE clause are different in that ParaSQL's WHERE clause potentially considers the whole information about an object while ISQL's WHERE clause considers a tuple at a time.

It is worth noting why ISQL and ParaSQL have the same structure for Query 2. It is because the key of Emp relation is used in a Boolean condition. If the Boolean condition compares a non-key attribute, then we can find the difference between ISQL and ParaSQL as shown in Query 2*.

Query 2*: Retrieve the history of employees whose salary was (is) greater than $50,000.
Since an object in the interval-based data model is scattered in multiple tuples, the ISQL needs a self-join to combine tuples for non-key attribute condition. However, the ParaSQL is the same as Query 2 except the Boolean condition.

**Query 3:** Give name and salary (history) of all employees working currently.

**ISQL:**

```
SELECT E1.Name, E1.Salary
FROM Emp E1, Emp E2
WHERE E1.Name = E2.Name
    AND E2.End = NOW
```

**ParaSQL:**

```
SELECT E.Name, E.Salary
FROM Emp E
WHERE NOW subset [[E.Name]]
```

Since it is nonpointic query\(^1\), we must provide a Boolean condition in \textit{WHERE} clause. It is worth noting that ISQL introduces a self-join operation while ParaSQL is a relation scan.

In ParaSQL, the Boolean expression, \(\text{NOW} \subset \text{[[E.Name]]}\), means that it is true if the current time is subset of the tuple's domain because Name is the key of Emp relation.

The Boolean expression can be rewritten as \(\text{NOW} \subset \text{[[E]]}\).

**Query 4:** Give name and salary at instant 60 of all employees.

**ISQL:**

```
SELECT E.Name, E.Salary
RESTRICTED TO [60,60]
FROM Emp E
```

**ParaSQL:**

```
SELECT E.Name, E.Salary
RESTRICTED TO [60,60]
FROM Emp E
```

The ISQL and ParaSQL are identical because Query 4 requests information for a simple time instant which is atomic granule. But, we can note the fundamental difference between

\(^1\) A nonpointic query extracts all information about an object while a pointic query extracts a part of information about an object.
two data models if Query 4 is changed to Query 4*, by requesting information for two disjoint time instants.

**Query 4**: Give name and salary at instant 60 or 70 of all employees.

**ISQL**:

```sql
SELECT E.Name, E.Salary
FROM Emp E
RESTRICTED TO [60,60]
UNION
SELECT E.Name, E.Salary
FROM Emp E
RESTRICTED TO [70,70]
```

**ParaSQL**:

```sql
SELECT E.Name, E.Salary
FROM Emp E
RESTRICTED TO [60,60] + [70,70]
```

In ISQL, `RESTRICTED TO` clause only allows a single time instant or a single interval so that the union of two time instants cannot be expressed in the clause. Note that ISQL's base model is the interval-based temporal data model. Therefore, it is required to union two select statements for two time instants. In general, ISQL requires `k` select statements for `k` disjoint intervals, making ISQL more complex.

In contrast to ISQL, ParaSQL expresses Query 4* in the same way of Query 4, which is more naturally reflecting natural language. The domain expression, `[60,60] + [70,70]`, means the union of two time instants, i.e., `[60,60] \cup [70,70]`.

**Query 5**: Find all employees throughout interval [60, 80].

**ISQL**:

```sql
SELECT E.Name, E.Salary
FROM Emp E
RESTRICTED TO [60,80]
```

**ParaSQL**:

```sql
SELECT E.Name, E.Salary
FROM Emp E
RESTRICTED TO [60,80]
```

The query structures of the ISQL and ParaSQL are the same, but the internal procedures are different as discussed in Query 4. If the English query needs multiple disjoint intervals,
the ISQL query should be changed to multiple unions of select statements while the ParaSQL simply adds multiple unions of intervals in RESTRICTED TO clause.

**Query 6:** Retrieve names of employees and their managers.

**ISQL:**

```sql
SELECT E.Name, D.MName
RESTRICTED TO [[E]]*[[D]]
FROM Emp E, Dept D
WHERE E.DName = D.DName
```

**ParaSQL:**

```sql
SELECT E.Name, D.MName
RESTRICTED TO [[E.DName=D.DName]]
FROM Emp E, Dept D
```

It must be noted that ISQL restricts joined tuples with the intersection of domains of tuples from Emp and Dept relations. The expressions \([E]\) and \([D]\) can be replaced with \([E.Name]\) and \([D.DName]\) because all attributes in a same tuple share a same interval. Note that \([E]\)*\([D]\) is legal because the intersection of two intervals are interval. In ParaSQL, RESTRICTED TO clause is used to join two relations.

**Query 7:** Give name and salary history during the time John was a manager of Software department.

**ISQL:**

```sql
SELECT E.Name, E.Salary
RESTRICTED TO [[E]]*[[D]]
FROM Emp E, Dept D
WHERE E.DName = D.DName
  AND D.MName = "John"
  AND D.DName = "Software"
```

**ParaSQL:**

```sql
SELECT E.Name, E.Salary
RESTRICTED TO [[
    SELECT *
    RESTRICTED TO [[D.DName="Software"]]
FROM Dept D
WHERE D.MName = "John"]]
FROM Emp E
```
ISQL needs to join two relations while ParaSQL scans a relation with the nested select statement. ParaSQL is closer to natural languages in that a user does not think about the query in terms of tuples, but in terms of objects.

In the ParaSQL query, it is worth noting that the nested query is separated from the outer query, that is, there are no relationship between Dept and Emp relation. Therefore, a smart query executor can process them separately to reduce disk accesses.

**Query 8:** Give name and salary history during the time John was not a manager of Software department.

**ISQL:**

```sql
SELECT E.Name, E.Salary
FROM Emp E
DIFFERENCE
SELECT E.Name, E.Salary
RESTRICTED TO [[E]]*[[D]]
FROM Emp E, Dept D
WHERE E.DName = D.DName
    AND D.MName = "John"
    AND D.DName = "Software"
```

**ParaSQL:**

```sql
SELECT E.Name, E.Salary
RESTRICTED TO ~[
    SELECT *
    RESTRICTED TO [[D.DName="Software"]]
FROM Dept D
WHERE D.MName = "John"
]
FROM Emp E
```

The notation, '~, in RESTRICTED TO clause, represents the complementation of a domain expression. This ParaSQL is exactly same as Query 7 except it has the complementation of a nested select statement whose output is a temporal element. It was made possible because the set of temporal elements are closed under complementation. Therefore, ParaSQL naturally expresses Query 8. In contrast, the ISQL query needs two steps.
It first retrieves all name and salary information of employees while John was a manager of Software department. Second, it subtracts the tuples of the first step from Emp relation. It does not follow the features of natural languages.

**Query 9:** Give current departments of employees who have worked in Hardware or Software department.

**ISQL:**

```sql
SELECT E1.DName
RESTRICTED TO NOW
FROM Emp E1, Emp E2
WHERE E1.Name=E2.Name
AND (E1.DName="Hardware"
OR E1.DName="Software")
AND E2.End=NOW
```

**ParaSQL:**

```sql
SELECT E.DName
RESTRICTED TO NOW
FROM Emp E
WHERE E.DName="Hardware"
OR E.DName="Software"
```

In ParaSQL, domains with which a condition is satisfied and the domain to be retrieved can be independent of each other because both domains are in the same tuple. However, in ISQL the domains reside in different tuples and self-joins are inevitable.

**Query 10:** Give current departments of employees who have worked in Hardware and Software departments.

**ISQL:**

```sql
SELECT E1.DName
RESTRICTED TO NOW
FROM Emp E1, Emp E2, Emp E3
WHERE E1.Name = E2.Name
AND E2.Name=E3.Name
AND E1.DName="Hardware"
AND E2.DName="Software"
AND E3.End=NOW
```

**ParaSQL:**

```sql
SELECT E.DName
RESTRICTED TO NOW
FROM Emp E
WHERE E.DName="Hardware"
AND E.DName="Software"
```

This natural language query only changes "or" condition of Query 9 to "and" condition.

ParaSQL mimics the natural language's behavior. Thus the complexity of the query dose

---

2In this query, we assume that \([a,b] - [c,d]\) is equal to \([a,b] \cap [c,d]\).
not change. In ISQL, it needs 3-way self-join for 2 conjunctive conditions. In general, the interval-based data model needs \((n + 1)\)-way self-joins for conjunction of \(n\) conditions [16].

### 5.4 System Implementation

We have developed two temporal database systems named I-based (interval-based) system and the XML-based parametric temporal database system which was introduced in Chapter 4, respectively. Since we discussed the XML-based parametric temporal database system in detail, in this section we more focus on the system implementation of the I-based system. The two systems have similar high level components except storages which are binary relation storage and XML storage, respectively. Figure 5.4 shows the abstract level system architecture which combines the two temporal database systems.

![System architecture](Image)

Figure 5.4 System architecture

In order to execute an ISQL query, the I-based system transforms the ISQL query into a parse tree represented in XML by the ISQL parse tree generator. By the ISQL expression
tree generator, the parse tree is again transformed to an expression tree which is also an XML document. The expression tree is executed by the ISQL query execution engine. As shown in the figure, when the system generates an XML document and transforms it to another form, it uses DOM API.

The I-based system uses binary relation storage while the XML-based parametric temporal database system uses paginated XML storage. Although the two storages are internally different, they follow the general functions of database storages at high level. They communicate with their space managers with pages. The space managers communicate with the buffer managers. The two storages have their own iterators which differently behave. The iterator of the I-based system retrieves a tuple at a time in binary format, makes the tuple as an XML element by adding necessary XML tags, and returns the encapsulated tuple to the ISQL query execution engine.

5.5 Performance Evaluation

5.5.1 Test Data Configuration

We follow the same data generation scheme introduced in Section 4.6.1 of Chapter 4. Table 5.1 shows the information of data used in our benchmark.3

<table>
<thead>
<tr>
<th>Table 5.1 Data setup information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Employees</td>
</tr>
<tr>
<td>Tuples</td>
</tr>
<tr>
<td>Pages</td>
</tr>
<tr>
<td>DB Size</td>
</tr>
</tbody>
</table>

* DB size includes dept relation and catalog.
* Page size: 4KB. Header size: 20bytes. Page utilization: 70%.

3Note that two databases have the same page utilization. Various page utilizations are not considered in this dissertation and they will be considered in our future work.
5.5.2 Special Treatment for Self-Join

In conventional databases, joins are important, but one of the most expensive operations. Joins are more serious in temporal databases because temporally-varying data dramatically increases the size of a database depending on time duration as well as time granularity. Some methodologies for temporal joins can be found in literature. Gao [21] et al. introduced and summarized join operations in temporal databases including Nested-loop join, Sort-merge join, and Partition-based join. These algorithms are all based on relation scans.

Most research work has focused on join operations for two heterogeneous relations so that it is hard to find methodologies for self-joins. A self-join is a join between a relation and itself. Soo et al. [56] introduced a linear ordered valid-time natural join algorithm called Partition-based algorithm. This algorithm partitions relation $r$ and $s$ into $n$ partitions. The join $r \bowtie s$ is computed by unioning the joins $r_i \bowtie s_i$, where $1 \leq i \leq n$. However, this algorithm cannot guarantee the linear ordered time complexity if $r$ contains many tuples with long intervals. For example, if $|\text{buffer}| - 2 < |r_i|$, the entire partition $r_i$ cannot be loaded in the buffer. Even though $r_i$ fits in the buffer, the partition-based algorithm needs $n$ time scans for $r_i$ for $n$-way self-joins. If the size of partitions are small, scanning multiple times may not be problematic. However, temporal data is so accumulative that the partition size can easily exceed the buffer size.

In this dissertation, we will introduce a self-join algorithm which is not affected by partition sizes as well as implemented by a single scan rather than multiple scans. This algorithm gives ISQL maximum benefit of the doubt. As we discussed in Section 5.3.2, the interval-based data model frequently introduces self-joins, increasing query complexity. Since joins are expensive, introducing multiple joins in a query is a critical issue for system performance. It seems obvious that the system performance of the interval-based data model may be affected by self-joins. However, self-joins can be implemented as a relation scan by a special treatment. The special treatment for self-joins is based on the following observations [43]:

1. We can construct a relation by clustering tuples that represent an object.
2. In self-joins, there is an one-to-one correspondence between an object and a group of tuples for the object in a database.

3. Objects are not mixed in a relation. Therefore, an object \( o_i \) is not related to an object \( o_j \) if \( i \neq j \).

These observations provide clues to implement a self-join operation without multiple relation scans. We create \( n \) pointers for an \( n \)-way self-join. These pointers move forward, pointing to tuples in a buffer. This treatment constructs a condition table to check Boolean conditions. For example, consider ISQL’s Query 10 discussed in Section 5.3.2. In this ISQL query, the Boolean expression in \texttt{WHERE} clause consists of five Boolean conditions as follows:

\[
\text{E1.Name} = \text{E2.Name} \land \\
\text{E2.Name} = \text{E3.Name} \land \\
\text{E1.DName} = \text{"Hardware"} \land \\
\text{E2.DName} = \text{"Software"} \land \\
\text{E3.End} = \text{NOW}
\]

Figure 5.5 illustrates how the special treatment avoids multiple scans for the 3-way self-join operation.

Each page in \texttt{Emp} relation is allocated to a frame in a buffer pool when it is requested. Since it is 3-way join, there are 3 pointers that point to tuples in the buffer. Each pointer moves forward while checking Boolean conditions in the condition table which are related to the pointers. It is worth noting that each pointer stops if its corresponding Boolean conditions are qualified. Whenever Boolean values of predicates in the condition table is changed, the predicate \( P \) for \texttt{WHERE} clause is evaluated. In addition to the pointers, the stream-based self-join algorithm uses a special pointer which points to the first page of an object. Once \( P \) is set to true, the algorithm uses the special pointer to retrieve tuples for the object in streaming fashion. By using this special treatment for self-joins, we can process self-joins as at most two relation scans, reducing tremendous number of disk block access.
5.5.3 Experimental Results

Table 5.2 shows the disk block requests and actual block accesses measured from the two temporal database systems.

From Table 5.2, we observe that the interval-based data model achieved 8% lower disk requests and actual disk accesses than the parametric data model, respectively. However, we must note that the performance of the interval-based data model is marginally better than the parametric data model; thus, the performance of two systems is comparable.

If a query contains self-join operations, the benchmarks show that the interval-based model needs 935% more block requests and 809% more actual accesses than the parametric data model for Q3 and Q9. The interval-based data model needed much higher disk requests and accesses than the parametric data model for Q10. However, if the special algorithm for self-joins is used, the block requests and accesses of the self-joins are significantly dropped to those of a relation scan, showing similar performance to the temporal element-based data model.
<table>
<thead>
<tr>
<th>Query</th>
<th>Interval-based data model</th>
<th>Parametric data model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Request</td>
<td>Access</td>
</tr>
<tr>
<td>Q1</td>
<td>519,608</td>
<td>261,143</td>
</tr>
<tr>
<td>Q2</td>
<td>519,608</td>
<td>261,143</td>
</tr>
<tr>
<td>Q3</td>
<td>527,402,492</td>
<td>229,311,284</td>
</tr>
<tr>
<td>Q3*</td>
<td>1,072,147</td>
<td>261,143</td>
</tr>
<tr>
<td>Q4</td>
<td>519,608</td>
<td>261,143</td>
</tr>
<tr>
<td>Q5</td>
<td>519,608</td>
<td>261,143</td>
</tr>
<tr>
<td>Q6</td>
<td>850,526</td>
<td>427,485</td>
</tr>
<tr>
<td>Q7</td>
<td>850,526</td>
<td>427,485</td>
</tr>
<tr>
<td>Q8</td>
<td>1,370,134</td>
<td>602,329</td>
</tr>
<tr>
<td>Q9</td>
<td>527,402,492</td>
<td>229,311,284</td>
</tr>
<tr>
<td>Q9*</td>
<td>1,072,147</td>
<td>261,143</td>
</tr>
<tr>
<td>Q10</td>
<td>31,617,544,810</td>
<td>15,808,772,405</td>
</tr>
<tr>
<td>Q10*</td>
<td>1,072,147</td>
<td>261,143</td>
</tr>
</tbody>
</table>

* The special treatment for self-join is applied.

### 5.6 Summary and Discussion

Satisfying the closure property makes the parametric data model reduce query complexities at the user level because it avoids invoking self-joins to gather information about an object. The parametric data model also allows more versatile expressions in the `RESTRICTED TO` clause.

As we discussed, for some queries, the ISQL required self-joins while the ParaSQL needed simple relation scans. We have shown that self-joins are inevitable in ISQL if queries have Boolean expressions containing multiple conjunctions. Such conjunctions frequently appear in temporal queries. ParaSQL for Query 8 can be expressed without significant changes from Query 7 (adding a negation was enough) while in ISQL it requires the union of two select statements.

In the disk block request and access tests, two data models showed similar performance even though the interval-based data model showed slightly better performance than the parametric data model for relation scans. To achieve the similar performance, the special treatment for self-joins was required in the interval-based data model. Without the special treatment, the
data model was unable to compensate its disadvantages due to multi-way joins.

Our query suite is conservative in that it only hints at the difficulties to be faced by users of interval-based approach. However, in practice the situation is expected to be far more serious. For example, let us reconsider Query 10 where we were seeking employees who had worked in Software as well as Hardware departments. What if we wanted to ensure that such employees have had at least 10 years of cumulative experience in respective departments? In ISQL, it becomes even more difficult to express this query because domains to be considered collectively are scattered in multiple tuples. ISQL may require complex use of SQL-style aggregate operations. Aggregate operations are also available in ParaSQL, but this query can be expressed without invoking aggregates as follows:

\[
[[E.DName = "Software"] > 10 \text{ AND } [[E.DName = "Hardware"] > 10
\]

It is should be emphasized that the parametric data model extends seamlessly to spatiotemporal databases. In spatial databases, the luxury of using intervals is not even available; spatial domains are far more complex. Our query suite can be readily extended to spatiotemporal data with the advantage that the user complexity of queries will remain similar. For example, the crop production depends upon extended periods of hot days. One may like to query for such periods in different counties in a state. So a county may be required to satisfy a where clause such as \([\text{Temp} \geq 70 \text{ degrees}] > 20 \text{ days}\). This is a simplified version of queries that in practice can be quite a bit more difficult.

One may raise a question about the usability of native XML database systems for temporal data because they also use XML-based storages. In order to answer the question, we conducted the comparisons between the XML-based parametric temporal database system and native XML database systems. Our experiments [41] showed that the XML-based parametric temporal database system is more efficient and easy for processing and expressing temporal queries.

In this chapter, we have investigated the ease of use as well as system performance for the two temporal data models. We hope that our findings help settle a debate which has continued since the mid 1980s for determining which data model is most appropriate for temporal databases.
CHAPTER 6 VALIDATION OF PARAMETRIC DATA MODEL FOR SPATIOTEMPORAL DATA

There is growing attention to spatiotemporal databases because many real world problems are associated with time and space dimensions. Space and time dimensions tend to be intertwined in spatiotemporal data so that separating them is contrary to their characteristics. Spatiotemporal elements in the parametric data model are constructed by combining temporal and spatial elements. Therefore, the parametric data model can naturally express domains of spatiotemporal values. In this chapter, we will introduce two other spatiotemporal data models. They use instants and intervals for temporal domains. In addition to this, we will also compare their spatiotemporal query languages with ParaSQL for user-friendliness.

6.1 Introduction

Research on spatiotemporal databases has been done independently in the temporal database community and the spatial database community [13]. Thus, there are two directions to realize spatiotemporal databases—extension from a temporal data model and extension from a spatial data model. In our discussion, we consider the former approach. Furthermore, we consider relational spatiotemporal data models because of their popularity in the database community.

In the temporal database literature, there are three types of timestamps for temporal domains such as instants, intervals, and temporal elements. Based on timestamps, temporal data models can be classified as point-based, interval-based, and temporal element-based data models. By the same token, we distinguish spatiotemporal data models based on timestamps—a point-based spatiotemporal data model and an interval-based spatiotemporal data model. Note that temporal elements and spatiotemporal elements are all parametric elements in the
parametric data model.

The validation of the parametric data model for spatiotemporal data can be reduced to the validation of ParaSQL for spatiotemporal data because the complexity of ParaSQL is directly influenced by the parametric data model; in other words, the way of modeling spatiotemporal data affects the complexity of query languages. We consider two other spatiotemporal query languages—SQL$^{ST}$ and STSQL. SQL$^{ST}$ uses a point-based spatiotemporal data model while STSQL uses an interval-based one. We will compare the three spatiotemporal query languages including ParaSQL for user-friendliness. In our comparisons, we use a spatiotemporal use case which is independent of data models and query languages.

The rest of this chapter is organized as follows. Section 6.2 provides the idea behind our approach. Section 6.3 discusses the syntaxes of the three spatiotemporal query languages. Section 6.4 compares and discusses the complexity of the query languages for the use case. Section 6.5 summarizes and discusses our findings.

6.2 Background

Güting et al. [25] proposed spatiotemporal data types which represent time-dependent geometries as attribute data types. They showed how the data types can be embedded in an SQL style query language for spatiotemporal data (STSQL). They also introduced an interesting application. They provided English queries for the application and expressed them into STSQL by using their spatiotemporal data types.

Chen and Zaniolo [9] introduced a spatiotemporal data model and query language called SQL$^{ST}$. They showed how to express Güting's use case in SQL$^{ST}$. The data model of SQL$^{ST}$ uses time instants for temporal domains and triangles for spatial domains. Therefore, it requires seven attributes for spatiotemporal domains; one for a time instant and six for a triangle.

The data model of STSQL implicitly uses intervals for temporal domains and links for spatial domains. Therefore, it requires three attributes for spatiotemporal domains; two for a

\[1\] In [25], they did not give a name to the SQL style query language. We name it STSQL for our comparison.
time interval and one for a symbolic spatial link. Note that the two data models fragment a spatiotemporal object in the real world into multiple tuples; thus, they can be implemented on top of conventional database systems.

Within this context, we express Güting’s use case into ParaSQL which uses spatiotemporal elements for spatiotemporal domains. Therefore, we can establish an evaluation framework for the query languages for user-friendliness in order to validate the parametric data model for spatiotemporal data.

6.3 Query Language Syntaxes

In this section, we introduce the syntaxes of SQL^{ST} and STSQL. For the syntax of ParaSQL, refer to Section 2.4 of Chapter 2.

6.3.1 SQL^{ST}

SQL^{ST} consists of two components, SQL^{T} and SQL^{S}, which evaluate temporal data and spatial data, respectively. SQL^{T} supports valid-time and uses a polygon-oriented representation. SQL^{ST} views reality as a sequence of snapshots of objects [9]. It minimizes the extensions of classical SQL for the point-based spatiotemporal data model. Figure 6.1 shows the simplified BNF of SQL^{ST}.

```
<select statement> ::= SELECT <attribute list>
    FROM <relation list>
    [WHERE <boolean expression>]
    [GROUP BY <attribute list>]
    [HAVING <boolean expression>]]
```

Figure 6.1 BNF of SQL^{ST}

It should be noted that their spatiotemporal types are orthogonal to underlying data models and STSQL is one of the possible query languages in which the spatiotemporal data types can be embedded.
6.3.2 STSQL

The attribute types introduced by Gütting et al. [25] are time-dependent geometries and orthogonal to spatiotemporal data models. STSQL is one of the possible query languages in which the attribute types can be embedded. Figure 6.2 shows the simplified BNF of STSQL.

\[
\text{<assignments>} ::= \text{LET } \text{<name> } = \text{<query>} \mid \\
\text{LET } \text{<name> } = \text{<function expression>} \mid \\
\text{LET } \text{<name> } = \text{<conversion>}
\]

\[
\text{<function expression>} ::= \text{FUN}(<\text{parameter list}>) \text{<expression>}
\]

\[
\text{<conversion>} ::= \text{ELEMENT } (<\text{query}>) \mid \\
\text{SET } (<\text{attribute name}>, \text{<value>} )
\]

\[
\text{<query>} ::= \text{SELECT } <\text{attribute list}> \mid <\text{derived attribute}> \\
\text{FROM } <\text{relation list}> \mid \\
[\text{WHERE } <\text{boolean expression}>]
\]

\[
\text{<derived attribute>} ::= <\text{new attribute name}> \text{AS } <\text{expression}>
\]

\[
\text{<multiple query>} ::= \{<\text{assignments}>;\} + [<\text{query}>]
\]

Figure 6.2 BNF of STSQL

<assignments> can be a query assignment, a function assignment, or a conversion assignment. A query assignment assigns the results of <query> to a new object called <name> which can be used in further steps of a query. A function assignment defines a new operator derived from existing ones. A conversion assignment makes it possible that a relation—consisting of a single tuple with a single attribute—can be converted into a typical atomic value [25]. <query> is closer to the classical SQL.
6.4 Query Language Comparisons

In this section, we compare the three spatiotemporal query languages for user-friendliness by using Güting’s use case—Forest Fire Control Management.

6.4.1 Scenario: Forest Fire Control Management

- Background

Güting et al. [25] justified the necessity of a spatiotemporal database which can be used in the forest fire control management as follows:

In a number of countries like the U.S.A., Canada, and others, fire is one of the main agents of forest damage. Forest fires are often caused by the carelessness of people abandoning campfires in and around forests. Another essential reason is self-ignition through lightening strikes, long drought, or underground fire sources like coal seams. Forest fire control management mainly pursues the two goals of learning from past fires and their evolution and of preventing fires in the future by studying weather and other factors like cover type, elevation, slope, distance to roads, and distance to human settlements. Specialized geographical information systems enriched by a temporal component and by corresponding analysis tools could be appropriate systems to support these tasks.

- An Example

The following illustrates an example of the forest fire control management. The Forest Lake fire occurred in Wyoming in 1981.

The Forest Lake fire was reported to the park dispatcher by the Mt. Sheridan lookout at 1850 on August 29, 1981. South District Ranger Mernin located the fire on the ground and confirmed that it was lightning-caused. The air patrol reported the development of the fire as follows: from 0 to 3, 20 acres; from 10 to 50, 312 acres; from 13 to 15, 54 acres; and from 16 to 25, 331 acres burned [26].
Figure 6.3 shows the Forest Lake fire development. The fire's development is represented as fire extents with a time period and an area burned in acre, where fire extent is defined as the area burned per a time period or an event [36].

6.4.2 Schemas and Representations

In the use case, the forest fire management system maintains three relations such as Forest, ForestFire, and FireFighter. The relation Forest records the location and the development of different forests growing and shrinking over time through clearing, cultivation, and destruction processes. The relation ForestFire contains information about the evolution of different fires from their ignition up to their extinction. The relation FireFighter describes the motion of fire fighters being on duty from their start at the first station up to their return [25]. The schemas of SQL$^ST$, STSQL, and ParaSQL for the use case are as follows:
**SQL**

Forest (ForestName CHAR(30), Territory REGION, VTime DAY)
ForestFire (FireName CHAR(30), Extent REGION, VTime DAY)
FireFighter (FighterName CHAR(30), Location POINT, VTime DAY)

**STSQL**

Forest (ForestName: STRING, Territory: MREGION)
ForestFire (FireName: STRING, Extent: MREGION)
FireFighter (FighterName: STRING, Location: MPOINT)

**ParaSQL**

FOREST (ForestName)
ForestFire (FireName)
FireFighter (FighterName)

The schemas for SQL$^{ST}$ use spatial data types REGION and POINT to represent a region and a location, respectively. The schemas for STSQL use spatiotemporal data types MREGION and MPOINT to represent a moving region and a moving location, respectively. In order to use conventional database systems, SQL$^{ST}$ needs seven attributes for spatiotemporal domains; six for spatial domains and one for temporal domains while STSQL needs three attributes for one for spatial domains and two for temporal domains. In contrast, the schemas for ParaSQL are all single-attribute relations for the use case. Figure 6.4 shows the conceptual representations of the relation ForestFire of SQL$^{ST}$, STSQL, and ParaSQL.

### 6.4.3 Queries

**Query 1** When and where did Forest Lake Fire have its largest extent?

---

3. The schemas for SQL$^{ST}$ and STSQL simplify spatial domains as REGION. However, in order to implement the data models on top of conventional database systems, they need to separate spatial and temporal dimensions. Therefore, they must have multiple attributes.
Figure 6.4 ForestFire spatiotemporal relation

$\text{SQL}^{ST}$:

```sql
SELECT F1.VTime, F2.Extent, AREA(F1.Extent) 
FROM ForestFire AS F1 F2 
WHERE F1.FireName = "Forest Lake Fire" 
AND F2.FireName = "Forest Lake Fire" 
AND F1.VTime = F2.VTime 
GROUP BY F1.VTime 
HAVING AREA(F1.Extent) = 
(SELECT MAX(AREA(Extent)) 
FROM ForestFire 
WHERE FireName = "Forest Lake Fire")
```
STSQL:

\[
\text{LET ForestLakeFire = ELEMENT (}
\text{SELECT Extent}
\text{FROM ForestFire}
\text{WHERE FireName = "Forest Lake Fire";}
\text{LET max_area = initial((atmax(area(ForestLakeFire))));}
\text{atinstant(ForestLakeFire, inst(max_area));}
\text{val(max_area)}
\]

ParaSQL:

\[
[[\text{SELECT *}]
\text{RESTRICTED TO [[Max(Area(DomPrj(S,F.FireName)))]}}]
\text{FROM ForestFire F}
\text{WHERE F.FireName = "Forest Lake Fire"}]
\]

The SQL$^S_T$ query uses a user-defined spatial aggregate function named AREA and a built-in aggregate function MAX to express the query. Since the temporal data model of SQL$^S_T$ is point-based, to retrieve all information about Forest Lake Fire, it must perform 2-way self-join to group the tuples recorded in every time instant.

The STSQL query consists of four parts. First, it extracts all extents for Forest Lake Fire and assigns the elements to the variable ForestLakeFire. Second, it finds the maximum area among the extents and assigns the result to the variable max_area. Note that the variable max_area contains value and spatiotemporal domain. Third, it finds time instants when the extents of Forest Lake Fire were maximum. Last, it extracts the value of max_area to return the largest extent. Note that the values pointed by the variables—ForestLakeFire and max_area—should be materialized in order to be used by another step of the query.

The ParaSQL query uses the function DomPrj to project a specific domain from a parametric element. The prototype of DomPrj is as follows:

\[
\text{DomPrj(target dimension, target parametric element)}
\]
**Target dimension** indicates which dimension should be projected from **Target parametric element**. In the spatiotemporal context, it can be either space dimension or time dimension.

Let \( P \) and \( X \) be a parametric element and a target dimension which should be extracted from \( P \), respectively. A parametric element can be viewed as \( P = (d_1, d_2, \cdots, d_n) \), where \( d_i \) is a domain for dimension type \( i \). For example, in the spatiotemporal context, \( d_1 \) and \( d_2 \) can be time and space dimensions, respectively. Let \( \Psi \) be the domain projector. The function \( \Psi \) is defined as follows:

\[
\Psi_X(P) = \{ x \mid (d_1, d_2, \cdots, x, \cdots, d_n) \in P \text{ and } x \text{ is a domain for dimension } X \}
\]

In ParaSQL, \( \Psi_S \) projects a spatial domain from a spatiotemporal element. For example, \( \Psi_S([\text{FireName}]) \) projects the spatiotemporal element \([\text{FireName}]\) as follows, where \( \text{FireName} = \text{Forest Lake Fire} \):

\[
\Psi_S([\text{FireName}]) = \begin{cases} 
[0,3] \times e_1 & \rightarrow e_1 \\
[4,9] \times e_2 & \rightarrow e_2 \\
[10,12] \times e_3 & \rightarrow e_3 \\
[13,15] \times e_4 & \rightarrow e_4 \\
[16,25] \times e_5 & \rightarrow e_5 
\end{cases}
\]

It is worth discussing the return values of a function whose input is another function. Such circumstance frequently occur when data should be processed in advance by another function. Furthermore, such compose functions can lead to less complex query languages and express queries more naturally. For our discussion, suppose that there is a function \( f(x) \) shown in Figure 6.5.
Let \( g \) be a function. A compose function \( g \circ f \) can be designed into three types—1) it returns values for given domains, 2) it returns domains such that \( g \circ f \) is true, and 3) it returns values as well as domains. Suppose \( g \) is \( \text{Max} \) function which finds the maximum values of \( f(x) \). We can express \( \text{Max}(f(x)) \) as follows:

\[
\begin{align*}
\text{Max}_1(f(x)) &= y_1 & : \text{Type 1} \\
\text{Max}_2(f(x)) &= \{x_1, x_2\} & : \text{Type 2} \\
\text{Max}_3(f(x)) &= \{(x_1, y_1), (x_2, y_1)\} & : \text{Type 3}
\end{align*}
\]

In the ParaSQL query, we assume that the function \( \text{Max} \) is \( \text{Max}_3 \). It should be clear that the function \( \text{Area} \) is also the type 3. The type-3 functions can reduce some additional notations in ParaSQL by removing \([\cdot]\), articulating ParaSQL in more precise terms.

The procedural step of the ParaSQL query is straightforward. It uses the function \( \text{Area} \) which returns the area of a symbolic spatial point as well as a spatiotemporal domain. By feeding the pairs of an area and a domain to the function \( \text{Max} \), the ParaSQL can find the maximum areas. The function \( \text{Max} \) is the type-3 function, it contains the domain of the maximum areas. Therefore, the domain expression in \text{RESTRICTED TO} clause is able to restrict qualified tuples to the spatiotemporal domain such that an extent is maximum.
Since the result of the relational expression is a tuple, the ParaSQL takes $\llbracket \cdot \rrbracket$ to extract the spatiotemporal domain for the tuple. The time complexity of this query is a relation scan.

**Query 2** When and where was the spread of fires larger than 500 $km^2$?

**SQL$^{ST}$:**

```sql
SELECT F1.VTime, F2.Extent
FROM ForestFire AS F1 F2
WHERE F1.VTime = F2.VTime
    AND F1.FireName = F2.FireName
GROUP BY F1.VTime, F2.Extent, F1.FireName
HAVING AREA(F1.Extent) > 500
```

**STSQL:**

```sql
LET big_part = SELECT big_area AS extent
    WHEN [FUN (r: region) area(r) > 500]
FROM ForestFire;
SELECT
    FROM big_part
WHERE not(isempty(deftime(big_area)))
```

**ParaSQL:**

```sql
[[SELECT *
    RESTRICTED TO [[Area(DomPrj(S,[[F.FireName]])) > 500]]
FROM ForestFire F]]
```

The `ForestFire` relation of SQL$^{ST}$ shown in Figure 6.4-(a) is a conceptual relation. When the relation is stored in a relational database, the model of SQL$^{ST}$ triangulates spatial objects. Therefore, the actual `ForestFire` relation can be seen as shown in Figure 6.6.

In Figure 6.6, $(x_{ij}, y_{ij})$ represents a point of a triangle of extent $e_i$, where $j$ indexes one of three points in a triangle and $1 \leq j \leq 3$. Due to the triangulation, SQL$^{ST}$ needs self-joins.
The STSQL query consists of two parts. The first part extracts extents such that the area of an extent is greater than 500 km² and assigns the set (or a relation) to a variable big_part. The relation big_part has a single attribute whose name is big_area. The second part checks if the temporal domain of big_area is empty because some fires may not have such extents.

The ParaSQL query simply expresses the English query using RESTRICTED TO clause. Similarly to Query 1, the domain expression, \[\text{Area(DomPrj(S, F.FireName))} > 500\], restricts the domain of a tuple to a spatiotemporal element such that the size of the extent is greater than 500 km². If there are no such extents in a tuple (note that it is a single tuple), the domain will be empty and the tuple will be excluded. If there are such extents, the relational expression retrieves the extents. Since we need the spatiotemporal domain of the extents, the ParaSQL query takes \[\cdot\] to extract the domain of the extents.

**Query 3** How long did fire fighter Miller work to extinguish Forest Lake Fire, and which distance did he cover there? The STSQL query may avoid the self-join for this query if it stores un-partitioned spatial data in the relation instead of triangulating the data.

The original English query is "How long was fire fighter Miller enclosed by the fire called Forest Lake Fire,"
\textbf{SQL}^{ST}:

\begin{verbatim}
SELECT DURATION(FireFighter.VTime),
    MOVING_DISTANCE(FireFighter.Location, FireFighter.VTIME)
FROM ForestFire, FireFighter
WHERE ForestFire.VTime = FireFighter.VTime
    AND FireName = "Forest Lake Fire"
    AND Fightername = "Miller"
GROUP BY ForestFire.VTime
HAVING INSIDE(Location, extent)
\end{verbatim}

\textbf{STSQL}:

\begin{verbatim}
LET ForestLakeFire = ELEMENT {
    SELECT Extent
    FROM ForestFire
    WHERE FireName = "Forest Lake Fire"};
SELECT time As duration(
    deftime(intersection(location, ForestLakeFire))),
    distance As length(trajectory(
        intersection(location, ForestLakeFire)))
FROM FireFighter
WHERE Fightername = "Miller"
\end{verbatim}

\textbf{ParaSQL}:

\begin{verbatim}
SELECT Duration([Fightername]), Distance([Fightername])
RESTRICTED TO [[
    SELECT FireName
    FROM ForestFire
    WHERE FireName = "Forest Lake Fire"
]]
FROM FireFighter
WHERE Fightername = "Miller"
\end{verbatim}

and which distance did he cover there?" We clarified the original English query to Query 3 because the original English query is unclear. However, the clarified English query does not required any structural changes of SQL^{ST} and STSQL queries.
The SQL$^S_T$ query joins the two relations, ForestFire and FireFighter, based on valid times. It groups tuples by valid time instants and checks if Miller's location is inside extents. For only those qualified tuples, it calculates the moving distance and the time length by using MOVING\_DISTANCE and DURATION, respectively.

The STSQL query consists of two parts. The first part extracts all extents for Forest Lake Fire and assigns the elements to the variable ForestLakeFire. The second part retrieves a fire fighter tuple whose name is Miller and calculates the length of distance and the duration such that he covered and he worked to extinguish the fire, respectively.

The ParaSQL query retrieves a fire fighter tuple whose name is Miller and restricts. It restricts Miller's information to a domain such that Forest Lake Fire occurred by intersecting two spatiotemporal domains. The ParaSQL query uses the function Duration and Distance to find the duration and the distance, respectively.

The procedural steps of STSQL and ParaSQL for this query are similar because two query languages iterate the two different relations independently, avoiding joins. However, if we closely investigate the underlying steps of STSQL, we can find that a join is involved. The variable ForestLakeFire is a relation name and it is scanned when intersecting with Location attribute of FireFighter relation. Since STSQL is interval-based, for every location, all tuples of ForestLakeFire should be intersected. It is the same as a cartesian product. In the STSQL query, we can see there are two cartesian products to find the duration and the distance. In contrast, ParaSQL needs only 2 relation scans between the inner and the outer queries which are independent each other. Formally speaking, STSQL requires $O(n + 2(n \times m)) = O(n \times m)$ time complexity while ParaSQL requires $O(n + m)$, where $n$ and $m$ are the numbers of tuples for Forest Lake Fire and Miller.$^6$

**Query 4** Determine the time and location when Forest Lake Fire started.

---

$^6$In ParaSQL, there are two tuples for the fire and the fire fighter. For brevity of our discussion, we assume that $n$ and $m$ represent the size of the two tuples.
**SQL\textsuperscript{ST}:**

\[
\begin{align*}
&\text{SELECT VTime, Extent} \\
&\text{FROM ForestFire} \\
&\text{WHERE FireName = "Forest Lake Fire"} \\
&\quad \text{AND VTime = (SELECT MIN(VTime) FROM ForestFire} \\
&\quad \text{WHERE FireName = "Forest Lake Fire"})
\end{align*}
\]

**STSQL:**

\[
\begin{align*}
\text{LET ForestLakeFire = ELEMENT (} \\
&\quad \text{SELECT Extent FROM ForestFire} \\
&\quad \text{WHERE FireName = "Forest Lake Fire"}); \\
\text{LET start_extent = initial(min(deftime(ForestLakeFire)))}; \\
\text{val(start_extent)}
\end{align*}
\]

**ParaSQL:**

\[
\begin{align*}
\text{[[SELECT FireName} \\
&\quad \text{RESTRICTED TO [[StartTime(FireName)]]} \\
&\quad \text{FROM ForestFire} \\
&\quad \text{WHERE FireName = "Forest Lake Fire"}]}
\end{align*}
\]

The SQL\textsuperscript{ST} query finds the minimum valid time instant of the fire which is the time instant that the fire occurred. Finding the minimum valid time instant can be expressed as a nested query in the \textit{WHERE} clause.\footnote{The original query in [25] assumes that a fire can start at different times with different initial regions. However, SQL\textsuperscript{ST} query introduced in [9] implicitly assumes that a fire started at a single time instant. For the sake of our discussion, we modified the STSQL query to make it follow the same assumption of SQL\textsuperscript{ST}.}

The STSQL query consists of three parts. First, it extracts all extents of the fire. Second, it finds the first extent of the fire. Last, it returns the time and the location of the extent.

ParaSQL simply expresses the query by restricting the tuple of the fire to the starting time instant using the function \textit{StartTime}. The domain of the fire is restricted to a
spatiotemporal domain such that the time domain is the starting time instant and the spatial domain is the location at the time instant.

6.5 Summary and Discussion

We have introduced SQL$^ST$ and STSQL which use instants and intervals for temporal domains. These data models easily utilize conventional database systems by simply adding some attributes for spatiotemporal domains. But, such modeling approach fragments a spatiotemporal object into multiple tuples. In addition to this, it requires to separate space and time dimensions. However, in spatiotemporal data, space and time dimensions tend to be intertwined so that separating the dimensions is contrary to their characteristics.

In contrast, the parametric approach sustains one-to-one correspondence between a spatiotemporal object in the real world and a tuple in a database. Furthermore, the parametric data model uses spatiotemporal elements which can naturally express domains of spatiotemporal values by combining temporal elements and spatial elements. Such approach maintains the characteristics of spatiotemporal data which intertwines space and time dimensions.

For all queries of Güting's use case, we have seen that ParaSQL simply expressed them and only required relation scans while the others required joins and materializations. In spatiotemporal databases, joins and materializations are expensive operations because spatiotemporal data is relatively huge and far more complex than ordinary data.

Throughout our comparisons for the three spatiotemporal query languages, we have noted that ParaSQL is less complex than the other spatiotemporal query languages; thus, we can conclude that the parametric data model is more appropriate for spatiotemporal data because the complexity of query languages is directly influenced by their data models.
CHAPTER 7 APPLICATION — NC-94 SPATIOTEMPORAL DATABASE

The NC-94 dataset contains the most complete records of temporal and spatial variables for climate, crop, and soil in the north central region in the United States. Scientists store and process the dataset within scientific data formats which are efficient for scientific simulations. However, it is difficult for the public to access the dataset by using ad-hoc queries because the scientific data formats are not database management systems. In this chapter, we present the NC-94 spatiotemporal database system built within the parametric framework and supports ad-hoc queries. In order to store the NC-94 dataset and geospatial information, we introduce a hybrid storage which can efficiently handle uniform data and geospatial data.

7.1 Introduction

The NCRA (North Central Regional Association of Agricultural Experiment Station) in the United States has a strong and extensive history of developing, verifying and validating agricultural databases for last 50 years. After decades of work, this committee has assembled an important, internally consistent dataset called the NC-94. The NC-94 dataset is expected to be used extensively to facilitate crop and risk analysis, pest management and forecasting. In addition, it will be publicly accessed through the Internet, allowing the public to ask ad-hoc queries [17].

In order to use the NC-94 dataset for scientific purpose, many scientific data formats are used with software packages to store and process it. The data formats help scientists to invent new models and validate their methodologies to understand environmental development. However, such data formats are not database management systems which support ad-hoc queries; thus,
it is difficult for the public to directly access the rich dataset.

We present the NC-94 database system which is implemented within the parametric framework for spatiotemporal data instead of using conventional database systems. In order to store the spatiotemporal dataset, we design a hybrid storage called HCube (Hybrid Storage for Homogeneous and Heterogeneous Data). The HCube is capable of storing and retrieving homogenous, heterogeneous, and hybrid data. Homogeneous data has a property such that all records in the data are stored in an identical format. For example, the climate data included in the NC-94 dataset has climate information for every day in a fixed format. To store homogeneous data, the HCube follows the industry standard binary structure (N-ary storage model) used in relational databases. In contrast, heterogeneous data cannot be stored in a fixed format. For example, regions expressed by GML are arbitrarily shapes. To store heterogenous data, the HCube uses the dynamic pagination algorithm discussed in Section 4.3 of Chapter 3. The HCube creates a hybrid data format by combining the N-ary model and XML to store hybrid data. In the hybrid format, XML is used as a directory for homogeneous data; in other words, homogeneous data is accessed through XML.

Many rich datasets used in agriculture and meteorology science are preprocessed and reported on the Internet rather than allowing users to ask ad-hoc queries. For example, Iowa Environmental Mesonet (IEM) [28] provides the public with preprocessed reports for most common climatological questions. Therefore, the implementation of the NC-94 database system will significantly enhance the usefulness of the NC-94 dataset for the public. Note that the database system, however, is a general-purpose database system for spatiotemporal data.

The rest of this chapter is organized as follows. Section 7.2 briefly reviews scientific data formats. Section 7.3 introduces the NC-94 dataset. Section 7.4 discusses how to represent and process spatial data. Section 7.5 presents the hybrid storage design and the methodology of building the NC-94 database. Section 7.6 discusses the query execution engine in the database system. Section 7.7 summarizes and discusses our findings.
7.2 Overview of Scientific Data Formats

There are many scientific data formats such as CDF (Common Data Format), NetCDF (Network Common Data Format), and HDF5 (Hierarchical Data Format). CDF is a file format that facilitates the storage and retrieval of multi-dimensional scientific data. NetCDF is an interface for array-oriented data access and a library that provides the implementation of the interface [61]. The NCSA (The National Center for Supercomputing Applications) has developed HDF5 which is a general purpose library and file format for storing scientific data. HDF5 stores datasets and groups which are the primary objects. A dataset is essentially a multi-dimensional array of data elements, and a group is a structure for organizing objects in an HDF5 file. Using these two basic objects, one can create and store scientific data structures [39].

The scientific data formats are well designed and used for analyzing and storing scientific data. In general, domain specific software is used over the data formats to process scientific data. Despite the advantage of the data formats, they are not database management systems so that they do not support ad-hoc queries to retrieve information from scientific datasets.

Such data formats may be used for a storage of the NC-94 database, but they require complex mechanism to store and handle heterogeneous data because in general they are designed for homogeneous data. For example, spatial information in the NC-94 dataset can be expressed in GML which is an XML based encoding standard for geographic information. It is not obvious how one would store such data in homogeneous data formats.

7.3 NC-94 Dataset

In this section, we discuss the relations of the NC-94 dataset. We also discuss storage efficiency for the relational storage and the XML-based storage for the NC-94 dataset.

7.3.1 Relations

Over the past few years north central 13 states in the United States has led in the development of a set of spatiotemporal data including regional soils, climate and crop productivity
in the NC-94 (North Central) regional project. The NC-94 dataset provides one of the most complete records of temporal and spatial variables that characterize the dynamics of agriculture covering 1,043 counties for 30 years (1971-2000). The data is primarily based on soils, climate and crop production at the county level [38].

Figure 7.1 shows the schema of Climate relation in the NC-94 dataset. The schemas of Crop and Soil relations are found in Appendix A.

<table>
<thead>
<tr>
<th>Field</th>
<th>Unit</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td>County</td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td>Time</td>
<td>day</td>
<td>Long</td>
</tr>
<tr>
<td>Radiation</td>
<td>MJ/m^2</td>
<td>Float</td>
</tr>
<tr>
<td>Max Temperature</td>
<td>°C</td>
<td>Float</td>
</tr>
<tr>
<td>Min Temperature</td>
<td>°C</td>
<td>Float</td>
</tr>
<tr>
<td>Precipitation</td>
<td>mm</td>
<td>Float</td>
</tr>
</tbody>
</table>

Figure 7.1 Field information of Climate relation

We will show how the parametric data model can be used in modeling the NC-94 dataset. In our example, we only consider climate data. However, the other data can be modeled as same as the climate data. Figure 7.2 illustrates how the climate data is represented in the parametric data model. Spatial information is expressed in a symbolic name in the relation, but its actual data is stored as GML in a spatial relation.

7.3.2 Data Structure Choice for NC-94 Relations

In Section 3.4 of Chapter 3, we compared the storage costs for the three implementation platforms such as relational, object-oriented, and XML-based storages. We showed that if less than 40% of intervals share ending time instants with another interval, the XML-based storage is better than the relational storage. Therefore, it is worth evaluating which approach is more appropriate for the NC-94 dataset. For our discussion, let us consider Climate relation present
in Section 7.3. For every day, a value was recorded in the relation, resulting in total 10,957 tuples. As discussed in Section 3.4 of Chapter 3, we can find the total percentages of intervals which share ending time instants as shown in Eq. 7.1:

\[
1 - p = \frac{|B_1 \cup B_2 \cup \ldots \cup B_5|}{\sum_{1 \leq i \leq 5} |B_i|} \\
p = 1 - \frac{|B_1 \cup B_2 \cup \ldots \cup B_5|}{\sum_{1 \leq i \leq 5} |B_i|} \\
= 1 - \frac{10,957}{5 \times 10,957}, \text{ where } |B_i| = \text{total number of intervals} = 10,975 \\
= 0.8
\]  

(7.1)

Note that \(B_i\) is the set of intervals of the \(i\)-th attribute. Since \(p\) is the variable for the percentage of intervals sharing ending time instants with another interval, Eq. 7.1 shows that
80% of intervals share ending time instants with another. Therefore, we can expect that the relational storage will be better than the XML-based storage.

It is worth discussing how much the relational approach is more efficient than the XML-based approach. Eq. 7.2 and Eq. 7.3 show the storage costs of the two approaches. We denote the storage costs of the relational storage and the XML-based storage as $S_R$ and $S_X$, respectively.

\[
S_R = N_o \cdot (1 - p) \cdot N_a \cdot U \cdot (N_a \cdot A + I)
\]  
\[\text{(7.2)}\]

\[
S_X = N_o \cdot \left( N_a \cdot \left( (A + T + 4 \cdot Tag) \cdot U + 2 \cdot Tag \right) + 2 \cdot Tag \right)
\]  
\[\text{(7.3)}\]

The variables shown in Eq. 7.2 and Eq. 7.3 represent the properties of the NC-94 dataset. Table 7.1 shows the values of the variables when the NC-94 dataset is stored in the relational storage and the XML-based storage.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of intervals sharing ending time instants</td>
<td>$p$</td>
<td>80</td>
</tr>
<tr>
<td>average attribute size</td>
<td>$A$</td>
<td>4</td>
</tr>
<tr>
<td>average size of temporal elements</td>
<td>$T$</td>
<td>8</td>
</tr>
<tr>
<td>size of an interval</td>
<td>$I$</td>
<td>4</td>
</tr>
<tr>
<td>average XML tag size</td>
<td>$Tag$</td>
<td>4</td>
</tr>
<tr>
<td>average update frequency</td>
<td>$U$</td>
<td>10,957</td>
</tr>
<tr>
<td>the number of attributes</td>
<td>$N_a$</td>
<td>5</td>
</tr>
</tbody>
</table>

Therefore, we can find the ratio of $S_R$ to $S_X$ as shown in Eq. 7.4.

\[
\frac{S_R}{S_X} = \frac{N_o \cdot (1 - p) \cdot N_a \cdot U \cdot (N_a \cdot A + I)}{N_o \cdot \left( N_a \cdot \left( (A + T + 4 \cdot Tag) \cdot U + 2 \cdot Tag \right) + 2 \cdot Tag \right)}
\]

\[
= \frac{262,968}{1,534,028}
\]

\[
= 0.171
\]  
\[\text{(7.4)}\]
Eq. 7.4 implies that the cost of the relational storage is only 17% of that of the XML-based storage. Therefore, it is necessary to store the relations of the NC-94 dataset in the homogeneous binary format.

7.4 Spatial Data Representation and Processing

The NC-94 spatiotemporal database system represents spatial information in GML. In this section, we briefly introduce GML as well as the spatial analysis methodology for geometries expressed in GML.

7.4.1 Geography Markup Language

The OGC (Open Geospatial Consortium) has developed an abstract data model for geographic data which is called SFS (Simple Feature Specification). The data model describes the world in terms of geographic features which can be viewed as a list of properties and geometries. Properties have the user name, type, value description while geometries are composed of basic geometry building blocks such as points, lines, curves, surfaces, and polygons [33]. The OGC's object model for geometry is shown in Appendix B.

GML is an encoding standard for geographic information, which follows OGC's OpenGIS specification [33]. GML is an emerging technology of representing geographic data because GML is based on a common model of geography which has been developed and agreed to by the vast majority of all GIS vendors in the world. In addition, GML is an XML-based encoding standard. Since it inherits the features from XML, there are four major advantages of GML. First, it is capable of verifying data integrity achieved by using GML Schema. Second, it is text-based so that it is human readable and easy to modify the content. Third, it is readily integrated with non-spatial data. Binary data structures are typically difficult to integrate with one another. Fourth, it is transformable to another format.

Since GML follows the XML's description methodology, it separates the content from the presentation. Geographic data expressed in GML contains information about the properties and geometry of spatial objects. It is worth noting that geographic data is independent from
any particular visualization of that data.

```xml
<featureMember>
  <fips>19169</fips>
  <name>Story</name>
  <state_name>Iowa</state_name>
  <pop1999>75514</pop1999>
  :
  <geometryProperty>
    <MultiPolygon>
      <PolygonMember>
        <Polygon>
          <outerBoundaryIs>
            <LinearRing>
              <coordinates>
                -93.347933,41.863105 -93.367306,41.863189 -93.375557,41.863154
                -93.406006,41.863244 -93.425348,41.863133 -93.444598,41.863179
                :
                -93.309219,41.862996 -93.328469,41.863087 -93.347933,41.863105
              </coordinates>
            </LinearRing>
          </outerBoundaryIs>
        </Polygon>
      </PolygonMember>
    </MultiPolygon>
  </geometryProperty>
</featureMember>
```

Figure 7.3  GML representation of Story county

Figure 7.3 shows an example of GML which describes the geographical information about Story county of Iowa in the United States, including properties of the county and geometry information. In Figure 7.3, the geometry of Story county is described by `<MultiPolygon>` which can consists of multiple `<Polygon>`. The shell or hole of a polygon is represented by `<LinearRing>` which contains $n+1$ coordinates such that the coordinates of 0-th and $n$-th are met, e.g., `coord[0]=coord[n]`.

Figure 7.4 shows four different types of multi-polygons which consist of 1, 3, 2, and 2 polygons, respectively. In Figure 7.4, the multi-polygon (c) consists of two polygons, where inside polygon is a hole which separates a polygon into two polygons. The multi-polygon (d) has two polygons which have holes inside.
GML can express the multi-polygon (d) of Figure 7.4 by using two <Polygon> elements which have the element <innerBoundaryIs> and <outerBoundaryIs> to represent shells and holes of the polygons. The two polygons are the members of a multi-polygon, which represents the multi-polygon (d). Figure 7.5 shows a skeleton of GML for the multi-polygon (d).

```
<geometryProperty>
<MultiPolygon>
  <polygonMember>
    <!-- the outside polygon -->
    <Polygon>
      <outerBoundaryIs>
        <!-- LinearRing for outer boundary is ommitted -->
      </outerBoundaryIs>
      <innerBoundaryIs>
        <!-- LinearRing for inner boundary is ommitted -->
      </innerBoundaryIs>
    </Polygon>
    <!-- the outside polygon -->
    <Polygon>
      <!-- inner and outer boundary -->
    </Polygon>
  </polygonMember>
</MultiPolygon>
</geometryProperty>
```

Figure 7.5  GML geometry representation of Figure 7.4-(d)

The use of multi-polygons guarantees the closure properties under the set theoretic operations of union, intersection, and complementation, which is originally proposed in the parametric data model [15].
7.4.2 Spatial Operations and Relationships

Multi-polygons in GML are closed under the set theoretic operations. In addition to the spatial set theoretic operations, the NC-94 database system follows the nine intersection model to process binary topological relationships between two objects. Egenhofer [11] introduced the nine intersection model to evaluate topological relationships between two objects $A$ and $B$ based on the intersection of $A$'s interior ($A^o$), boundary ($\partial A$), and exterior ($A^–$) with $B$'s ($B^o$), boundary ($\partial B$), and exterior ($B^–$). The relationship between the six parts can be expressed in the nine intersection matrix (IM) as follows [51]:

$$IM(A, B) = \begin{pmatrix} A^o \cap B^o & A^o \cap \partial B & A^o \cap B^- \\ \partial A \cap B^o & \partial A \cap \partial B & \partial A \cap B^- \\ A^- \cap B^o & A^- \cap \partial B & A^- \cap B^- \end{pmatrix}$$

By using the nine intersection matrix, we can evaluate eight relationships between two objects such as disjoint, meets, overlaps, equal, contains, inside, covers, covered by.

Figure 7.6 shows the eight topological relationships. For example, the disjoint relationship explains that there are no sharing points in interior and boundary of the two objects, but there are in exterior [51].

7.5 Hybrid Storage Design

7.5.1 Overview

One of the fundamental issues faced by the agricultural scientific community is logical and relatively easy access to the NC-94 dataset. We have developed the NC-94 database based on the HCube. HCube has been designed to sustain the advantages of the parametric data model without losing storage efficiency.\(^1\)

Figure 7.7 shows the abstract view of the HCube. For storing homogenous data (uniform data), the HCube follows the N-ary storage model which stores records continguously starting

\(^1\)In this chapter, the terms homogeneous and uniform are interchangeable, and so are heterogeneous and less uniform.
from the beginning of each page. When deleting a tuple in a page, the page is compacted. Therefore, by using offset, every tuple can be assessed. For storing heterogenous data (less uniform data), the HCube uses XML. An XML document containing heterogenous data is paginated into small XML pages.

In order to model the NC-94 dataset within the parametric framework, the relations of the dataset should be stored in a hybrid data format which is built by combining XML and the N-ary storage model. In the hybrid data format, XML is used as a directory; thus, homogeneous data is accessed through XML.
7.5.2 Page Structures

Although there are three different data types in the HCube, there are only two different page structures called binary page and XML page. Therefore, it is worth discussing the two page structures.

Figure 7.8 shows the two page formats. The two pages reserve 16 bytes for header information. In the header information, the fields, Page Type and Page ID, are common in the two page formats. The field Page Type indicates whether this page is a binary or an XML page while the field Page ID stores the current page identification. In the binary page format, the field Next Page ID contains the next page identification while the field Number of Tuplets contains the total number of tuplets stored in the binary page, where a tuplet is a record which consists of attributes. A tuplet is atomic data which is a part of a parametric tuple. In the XML page format, the field Parent Page ID contains the parent page identification while the field Data Size contains the size of XML data stored in the
current page. In addition to the header fields, the two page formats have a reserved field to be used if necessary.

### 7.5.3 Hybrid Data and Iterating Tuples

The HCube uses the combination of XML and binary pages to store and access hybrid data. XML is used to traverse homogenous data which consists of binary pages. By using DOM API, the HCube can retrieve a page identification which points to the starting page of homogeneous data.

Figure 7.9 shows how the HCube stores hybrid data. The HCube accesses the first binary pages through XML nodes which contain the addresses of the pages. Every leaf node in XML contains the identification of a starting page of homogeneous data. It is worth noting that XML for navigation is heterogeneous as well as paginated by the dynamic pagination algorithm and the binary pages in the HCube are not required to be continuous on a disk. For example, the homogeneous data starting with \( \text{PageID} = z \) shown in Figure 7.9 consists of multiple pages and two adjacent pages do not keep the order of page sequences in a disk because every page can be randomly accessible through a pointer which is the value of \text{Next Page ID} field. It should be also noted that we may totally ignore the order of leaf nodes in XML or maintain

<table>
<thead>
<tr>
<th>Page Type</th>
<th>Page ID</th>
<th>Parent Page ID</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of Page Formats](image)

(a) Binary page

(b) XML page

Figure 7.8 Page formats in HCube
Figure 7.9 Logical view of a hybrid relation

the order to achieve fast retrieval.

By using the hybrid data structure, we can store the NC-94 relations in the HCube. Each node in XML represents a parametric tuple whose actual data is stored in binary pages. As discussed, a tuple consists of tuplets. Note that if a relation is the same as a relation of conventional relational databases, we can store the relation in binary pages without XML, considered as simple homogeneous data.

The HCube manages three different data structures. For homogeneous data, the procedure of iterating tuples is similar to relational databases. For heterogeneous data, it iterates tuple nodes which are the top nodes of tuples represented in XML. For hybrid data, it iterates tuples consisting of tuplets. The HCube has a primitive iterator used in other iterators which need to access hybrid relations. It simply returns a tuplet at a time and notifies its client whenever
it reaches the end of a tuple. This primitive iterator consists of three functions such as OPEN, HASMORETUPLET, and GETNEXTTUPLET. By using these functions, iterators used in a query execution evaluate user queries.

Algorithm 4 shows the function OPEN. It prepares to retrieve tuples from a relation. The parameter $R$ is relation information which has attribute information and the page identification of the relation. It assigns the first child of the relation stored in XML to the current tuple node. The current tuple node has the page identification of the first binary page of the relation. The first page is loaded into the buffer in the HCube storage manager. From the first page, the next page identification and the total number of tuplets are assigned to variables.

**Algorithm 4 Hybrid data iterator-Open**

```
1: procedure OPEN(R) ▷ R: Relation Information
2:   root ← getRootNode(R.getPageID()) ▷ get a root node of the relation tree
3:   curTupleNode ← root.getFirstChild() ▷ get the first child
4:   curPage ← getPage(curTupleNode) ▷ get a page indicated by curTupleNode
5:   nextPageID ← curPage.getNextPageID() ▷ get the next page ID
6:   numOfTuplets ← curPage.getNumOfTuples() ▷ total tuplets in this page
7:   tupletIndex ← 0 ▷ tuple index to retrieve a next tuplet in this page
8:   tupletSize ← R.getTupletSize() ▷ tuplet size in this page
9: end procedure
```

**Algorithm 5 Hybrid data iterator-HasMoreTuplet**

```
1: procedure HASMORETUPLET(curPage) ▷ curPage: the current page
2:   if nextPageID = -1 then ▷ current page is the last page
3:     if tupletIndex > numOfTuplets - 1 then ▷ no available tuplet exists
4:       return false
5:   end if
6: else if tupletIndex > numOfTuplets - 1 then
7:   tupletIndex = tupletIndex - numOfTuplets
8:   curPage = getPage(nextPageID)
9:   nextPageID = curPage.getNextPageID()
10:  numOfTuplets = curPage.getNumOfTuples()
11: end if
12: return true
13: end procedure
```
Algorithm 5 shows the function HASMORETUPLET. It evaluates whether there is an available tuplet. It checks two cases—1) if the current page is the last page of a tuple and 2) if the current page is not the last page of a tuple. For the first case, it checks if the tuplet index exceeds the total number of tuplets; if this is the case, there is no more available tuplet. For the second case (line 6–10), it simply checks if the tuplet index exceeds the total number of tuplets; if this is the case, it gets the next binary page and resets the variables of the iterator.

Algorithm 6 Hybrid Data Iterator-GetNextTuplet

1: procedure GETNEXTTUPLET
2:   if HASMORETUPLET(curPage) = true then
3:     tuplet ← curPage.getNextTuplet(tupletIndex, tupletSize)
4:     return tuplet
5:   end if
6:   curTupleNode ← curTupleNode.getNextSibling() > get the next tuple node
7:   if curTupleNode = null then
8:     return "the end of relation" > there is no more tuple
9:   end if
10:  curPage ← getPage(curTupleNode)
11:  nextPageID ← curPage.getNextPageID()
12:  numOfTuplets ← curPage.getNumOfTuplets()
13:  tupletIndex ← 0
14:  return "the end of tuple" > no more tuplets for previous tuple
15: end procedure

Algorithm 6 shows the function GETNEXTTUPLET. This function consists of three cases. First, if there is a next tuplet, it returns the tuplet. Second, if there is no more tuplet, it checks if there is no more tuple (line 6–9); if this is the case, it returns a message to indicate it reaches the end of a relation. Third, if there is an available tuple (line 10–14), it initializes the variables of the iterator and returns a message indicating that it has reached the end of a tuple.

7.5.4 HCube Storage Manager

By combining XML and binary storage technologies, we can build the storage manager for the NC-94 database system. In order to reflect the modeling concept of the parametric data
model, the storage manager uses XML for the system catalog and spatial tables, and the hybrid data format for the relations of the NC-94 dataset.

Figure 7.10 HCube storage manager

Figure 7.10 shows the logical view of the storage manager of the NC-94 database system. The space manager handles pages stored in a disk (or multiple disks or partitions). The buffer manager loads pages into the buffer pool by using the space manager or passes a page to an iterator. The storage manager has an iterator pool which is a collector of iterators for the three types of data. These iterators are transparent to clients and behave differently, but provide the same functionality.
7.6 Query Execution Engine

In this section, we will discuss how the query engine of the NC-94 database executes user queries. The query execution engine evaluates user queries represented in XML. Like the parametric temporal database discussed in Chapter 4, we utilize XML to represent parse and expression trees.

7.6.1 Examples

For our discussion, we present three ParaSQL queries. In the examples, we increase the complexity of queries by adding Boolean conditions which have spatial features.

**Query 1:** Retrieve the climate information of counties while the average of the maximum and minimum temperatures was (or is) less than 0 °C.

```
SELECT *  
RESTRICTED TO [[(C.MaxTemp+C.MinTemp)/2<0]]  
FROM Climate C
```

Query 1 retrieves climate data from the Climate relation and evaluates the parametric domain such that the average of maximum and minimum temperatures are less than 0 °C. For each tuple, the domain of the tuple is restricted to the parametric domain.

Figure 7.11 shows the expression tree of Query 1 which is represented by XML. Since the Boolean condition of Query 1 in WHERE clause is true, the expression tree represents the Boolean condition by using the element `<Condition isTrue="true"/>`.

**Query 2:** Retrieve the climate information of counties such that the population of a county is greater than 20,000 while the average of the maximum and minimum temperatures was (or is) less than 0 °C.

```
SELECT *  
RESTRICTED TO [[(C.MaxTemp+C.MinTemp)/2<0]]  
FROM Climate C  
WHERE Population(C.County) > 20000
```
Query 2 adds a Boolean condition which evaluates the population of each county. The function Population is a spatial function which retrieves population metadata from GML.

Figure 7.11 shows how to express the Boolean condition of Query 2. Since there is a Boolean condition in WHERE clause, the element `<Condition isTrue="false"/>` indicates that there exists a Boolean expression. In ParaSQL, a Boolean expression is expressed by a binary tree. Each Boolean condition is represented by the element `<BoolExp>`. This element has attribute `opType` which indicates if this Boolean condition is terminal or binary. If a Boolean condition is terminal, the attribute value is `unary`; otherwise, it can be `and` or `or`. Since Query 2 uses a spatial function, the expression tree represents the information in the element `<SpatialFunc name="population" args="1">` meaning that it requires the spatial function population and the function needs a single argument.
Figure 7.12 Changed expression tree for the Boolean expression

Query 3: Retrieve the climate information of neighbors of Story county such that its size is greater than 540 \textit{mi}^2 and its population is greater than 20,000 while the average of the maximum and minimum temperatures was (or is) less than 0 °C.

```
SELECT *
RESTRICTED TO [(C.MaxTemp+C.MinTemp)/2<0]
FROM Climate C
WHERE Neighbor("Story", C.County) = TRUE
AND Area(C.County) > 540
AND Population(C.County) > 20000
```

Query 3 is expressed by adding two Boolean conditions to Query 2, which are Neighbor and Area functions. These functions are all spatial functions so that they are expressed by element \(<\text{SpatialFunc}>\) in the expression tree. It is worth discussing the difference between the spatial functions of Query 3 and that of Query 2. Although they are all spatial functions, they have different properties internally. As discussed in Section 7.4.1, GML contains spatial properties and spatial geometries. The population of a county is a spatial property which is directly accessible without processing the data while evaluating neighbors and areas require to process geometry data represented by coordinates. For example, the Neighbor function retrieves coordinates of multi-polygons and evaluates whether two multi-polygons are met by using the nine intersection matrix. The full expression tree of Query 3 is shown in Appendix C.
7.6.2 Boolean Evaluation

In order to evaluate Boolean conditions, the query execution engine creates a Boolean binary tree for a Boolean expression shown in WHERE clause. A Boolean binary tree consists of evaluation nodes. Figure 7.13 shows the structure of evaluation node.

![Evaluation node diagram](image)

The field Parent points to the parent node of the current node. The field Node Type indicates whether this node is a Boolean node or a domain node. In the parametric data model, a Boolean expression is expressed by $\mu \subseteq \nu$, where $\mu$ and $\nu$ are domain expressions. For example, a Boolean expression, $[[\text{C.MaxTemp} > 30]] \subseteq [[\text{C.Precipitation} = 0]]$, evaluates if precipitation is 0 whenever maximum temperature is greater than 30 °C. For some cases, Boolean expressions can be abbreviated. For example, the Boolean expression $A \theta B$ is the abbreviation of $[[A \theta B]] \neq \emptyset$, where $A$ and $B$ are attributes, and $\theta$ is an operation such as $\neq$ and $<$.  

The field Result stores the result of the current node. The result can be either a Boolean value or a parametric element depending on the node type. The field Operation Type stores the type of the current Boolean expression. It can be and, or, subset, or unary, where unary expresses that the current node is a terminal node. If the node type of an evaluation node is not unary. The node is binary and it has two child nodes. The field Left Result stores the result of the left child while the field Right Result stores the result of the right child. The fields, Left Child and Right Child, point to the left and right child evaluation nodes, respectively. The result of the evaluation node is evaluated by the
operator of Operation Type filed with the two values of the child nodes.

In the parametric data model, a Boolean expression is formally expressed by domain expressions with set operations. To find the parametric element of a domain expression, it is required to traverse entire tuple. Note that in the parametric data model, an object is modeled into a tuple which captures changes of attributes. Therefore, evaluating a domain requires the entire information of a tuple. However, if a Boolean expression is abbreviated and contains constants, we can expect a fast evaluation without stepping through an entire tuple. For our discussion, let us revisit Query 3 discussed in Section 7.6.1. Figure 7.14 shows the binary evaluation tree for Query 3.

Figure 7.14  Binary evaluation tree of Query 3 when only constants are involved

Query 3 has three Boolean conditions connected by the two Boolean operator ANDs. Note
that all Boolean conditions are in the abbreviation forms and constants are involved in the
Boolean conditions. In the NC-94 database, a parametric tuple stored in the hybrid format
consists of multiple tuplets. A tuplet has information at a certain parametric point. In Query 3,
the query execution engine can evaluate the three Boolean conditions with only a single tuplet
because a tuplet contains a county name so that the spatial functions can be executed with only
the information. Therefore, if a Boolean expression consists of Boolean conditions which are in
the forms of abbreviations and contain constants, traversing an entire tuple is not necessary to
evaluate the Boolean expression. However, if there is a Boolean condition which is a form of
the domain expressions with set operations, it is required to traverse the entire tuple.

Algorithm 7 shows the procedural steps to evaluate a Boolean expression by using a binary
evaluation tree.

---

**Algorithm 7** Boolean expression evaluator

```plaintext
1: procedure BoolExp(Node expNode, EvalNode evalRootNode, Tuplet aTuplet) ▷
   expNode: expression tree, evalRootNode: binary evaluation tree root node, aTuplet: a
tuplet to be processed
2: if evalRootNode.isBoolEvalNode(expNode) then
3:    return true
4: end if
5: int opType ← getOpType(expNode)
6: if opType = OP_TYPE.UNARY then ▷ terminal node
7:    boolean bResult ← procBoolExp(expNode.getFirstChild(), aTuplet)
8:    if bResult = true then
9:       EvalNode evNode ← evalRootNode.findEvalNode(expNode)
10:      evNode.setResult(true)
11:      evalRootNode.updateParent(evNode)
12:    end if
13: else ▷ binary operation such as and or or
14:    BoolExp(expNode.getFirstChild(), evalRootNode, aTuplet)
15:    BoolExp(expNode.getLastChild(), evalRootNode, aTuplet)
16: end if
17: return evalRootNode.getResult()
18: end procedure
```

---

The function BoolExp is the key function of the query executor when evaluating a Boolean
expression. The function has three arguments—an expression tree, a root node of a binary
evaluation tree, and a tuplet to be processed. The function consists of three parts. First, it checks if the value of the evaluation node, corresponding to the node of the current expression tree, is true (line 2-4). Since a tuplet is the part of a parametric tuple, it is not necessary to evaluate the tuplet if the previous tuplet has been already qualified. If this is not the case, the second step is to check whether the expression node is a unary node (line 5-12). If this is the case, the expression node is a terminal node. The query executor processes the expression node and checks if the result is true or false. If the result is true, then it finds the evaluation node for the current expression node. The evaluation node is set to true and its parent node is updated. When the parent node is updated, updates are propagated to ancestors if the value of a parent node is changed to true. If the node type of the current expression node is not unary, the third step is to call BOOLEXP twice for the first and last children of the expression node (line 13-16). Whenever the root node of the binary evaluation node is set to true, the evaluation stops and returns the parametric tuple.

7.6.3 Domain Evaluation

Domain expressions retrieve the domain of qualified tuples. In the NC-94 database, a tuple consists of tuplets whose time granule is day. Algorithm 8 shows how to evaluate a domain expression in the NC-94 database system.

The function DOMEXP is the key function of the query execution engine when retrieving the domain of qualified tuples. The form of this function is similar to the function BOOLEXP. The evaluation steps of the domain evaluator is as follow. It checks the operation type of the current expression tree node (line 4). Based on the operation type, it calls either procDomExp or DOMEXP functions. If the operation type is unary (line 5-6), the expression node is a terminal node and the function procDomExp is called. If the operation type is either intersection or union (line 7-15), the function DOMEXP is recursively called twice—one for the first child of the current expression node and the other for the last child of the node. Therefore, the domain evaluator builds a parametric element for a given expression tree. The parametric element is returned by the domain evaluator.
Algorithm 8 Boolean expression evaluator

1: procedure DOMExp(Node expNode, Tuplet aTuplet) ▷ expNode: expression tree, 
   aTuplet: a tuplet to be processed
2:     PElement pe1 = null
3:     PElement pe2 = null
4:     int opType ← getOpType(expNode)
5:     if opType = OP.TYPE.UNARY then ▷ terminal node
6:         pe1 = procDomExp(expNode.getFirstChild(), aTuplet);
7:     else if opType = OP.TYPE.INTERSECTION then ▷ intersection
8:         pe1 = DOMExp(expNode.getFirstChild(), aTuplet);
9:         pe2 = DOMExp(expNode.getLastChild(), aTuplet);
10:        pe1 = pe1.intersect(pe2)
11:     else if opType = OP.TYPE.UNION then ▷ union
12:        pe1 = DOMExp(expNode.getFirstChild(), aTuplet);
13:        pe2 = DOMExp(expNode.getLastChild(), aTuplet);
14:        pe1 = pe1.union(pe2)
15:     end if
16:     return pe1
17: end procedure

It is worth noting that domain expressions and Boolean expressions are mutually recursive. The terminal functions, procBoolExp and procDomExp, share core functions which can be used in both functions. Furthermore, the functions may directly call BOOLExp and DOMExp functions. This approach reflects the property of the parametric data model such that the three core expressions are mutually recursive.

7.7 Summary and Discussion

The NC-94 dataset includes the most fundamental inputs needed to run a variety of crop, soil, and climate simulations for various applications. Scientists use scientific file formats to store and process it. However, it is difficult for the public to access data stored in scientific formats because they do not support ad-hoc queries.

In order to support ad-hoc queries, we have implemented the NC-94 spatiotemporal database system. The database system can be used by general users like farmers or even utilized by domain experts like meteorologists. We have found that if we store the NC-94 dataset in pure
XML, it requires 6 times more storage cost than relational storages because too many intervals in the dataset share their ending time instants with another. Therefore, we had to develop a hybrid storage, HCube, which is able to sustain the advantages of the parametric data model without losing storage efficiency.

In evaluating Boolean expressions, we have noted that it is unnecessarily to retrieve entire tuple from a disk if Boolean conditions are in the form of abbreviations with constants. For some cases, only a single tuplet was enough to evaluate a Boolean expression, leading to fast execution. However, if a Boolean condition is expressed in the form of $\mu \subseteq \nu$, it is required to retrieve entire tuple to evaluate the Boolean condition.

In our implementation, we have incorporated GML in the parametric data model. Use of GML is a good mach because it is an XML-based encoding standard for geospatial data. More importantly, GML meets the OGC's OpenGIS standard. It should be noted that the OpenGIS standard satisfies the set theoretic closure requirements for spatial domains proposed by the parametric data model. Therefore, GML was seamlessly integrated to the parametric data model.

In addition, we may embed the paradigm of the HCube into HDF5. HDF5 is well known and designed for scientific data, but it has some limitations on storing heterogeneous data. Although most scientific data are homogeneous in general, their applications tend to cooperate with heterogeneous data. In order to resolve this problem, it is required to design an interface on HDF5 for storing and retrieving less uniform data. By adapting XML methodology in HDF5, we can make HDF5 behave like the HCube, which can handles heterogeneous data effectively. Therefore, we may expect that the approach of the HCube will provide scientists with capability of accessing and handling less uniform data without losing the advantages of HDF5.
CHAPTER 8 CONCLUSIONS AND FUTURE WORK

In this dissertation, we proposed the validation of the parametric data model for temporal and spatiotemporal data, and the implementation of the NC-94 spatiotemporal database system within the parametric framework. In this chapter, we conclude how the proposed goals have been met. In addition to this, we also sketch some directions for future work on issues resulted from the implementation of the parametric data model.

8.1 Conclusions

In this dissertation, we have validated the parametric data model for dimensional data and ParaSQL, its query language for the temporal and spatiotemporal dimensions. In both cases, we have shown ParaSQL to be more natural for users than other existing query languages. There are several reasons for this: in the parametric data model a real world object is not fragmented into multiple tuples; it treats objects with different mixes of dimensions uniformly irrespective of the number and types of dimensions present; parametric domains are closed under set theoretic operations of union, intersection, and complementation; ParaSQL allows relational, domain, and Boolean expressions that are mutually recursive. None of these features are present in any other models for dimensional data.

We hope that the query suite used in this dissertation for benchmarking user friendliness for the temporal dimension will be found useful. More detailed benchmarks can now be developed. For the spatiotemporal dimension, we used Gütting's use case and showed that this too can potentially be expressed more naturally in the parametric data model.

In addition to study of usability, we also implemented the model and ParaSQL for the said dimensions. In order to make a comparison with interval based languages, we implemented
ISQL which is a hypothetical query language for interval-based data models. ISQL queries posed by users routinely require spurious self-joins whose literal execution can be expensive. In order to give ISQL maximum benefit of the doubt, we implemented a stream-based self-join algorithm that requires only one pass and not multiple passes. This makes ISQL queries run much faster. We have shown that even with our single-pass algorithm for self-join ISQL queries perform marginally better than those in ParaSQL where only literal execution is considered. Other single-pass ParaSQL queries require complex joins in other languages; in these cases, ParaSQL outperforms other query languages. In case of time dimension, we happen to have ISQL, a good representative of other query languages, to compare with. In spatiotemporal databases, models and languages make any side by side comparison a very difficult exercise.

We used XML for all our implementation needs. It is easy to define XML models for storage of data. XML provides a very high level interface for implementation of artifacts and linguistics of the model and query languages. The code one obtains is highly human readable and reliable. To store the data, we used CanStoreX, our own storage technology that paginates XML documents leading to excellent control over caching. Even for traditional artifacts like parse tree and expression tree, use of XML is very easy and appropriate compared to traditional implementation of trees in terms of nodes linked through binary pointers. It is also easy to develop and deploy specifications of interfaces among implementation and consumption modules of the software system.

As stated above, XML is qualitatively an excellent implementation platform. In order to quantify the storage requirements, we compared XML with traditional relational and object oriented storage technologies. For heterogeneous data, XML is most appropriate. XML is also best for homogeneous data where properties of objects do not change very frequently. When properties of objects change frequently, the relational storage is more economical than XML. In our implementation, we encountered very homogeneous data such as climatic information and in the implementation of ISQL. Here, binary relational storage was used. But even such data was wrapped in XML through its metadata to provide easy access to its consumer modules. In the temporal database, where data was not very heterogeneous, we successfully deployed XML
directly. Lastly, we deployed geographical data that was available in GML, an XML based format. For this data, use of XML should be considered completely natural.

We developed a prototype for NC-94, an important dataset in agriculture, on top of the parametric data model. As stated above, for implementation of this prototype, we used XML as well. This dataset includes daily observations during 1971-2000 for climatic variables such as precipitation for 1,043 counties in the north central region of the United States. This information is very uniform and changes frequently. Like ISQL data, it was also stored in a binary format and wrapped into XML through its metadata. As stated above, geographical information for counties was directly stored in XML. Thus internally the time and space dimensions are stored very differently. These were brought together by creating a directory that provides direct access to temporal and spatial data for each county. This methodology has been termed HCube which is a hybrid storage for homogeneous and heterogeneous data.

The NC-94 database system maintains GML-based geospatial information. Geometries in GML are represented by multi-polygons. It should be noted that the set of multi-polygons are closed under set theoretic operations like union, intersection, and complementation, which has been proposed by the parametric data model. Integrating GML into the parametric data model was made possible because of closure properties. This creates a symbiotic relationship between GIS and the parametric data model. Therefore, we hope that our work will help bring the GIS and database communities together.

8.2 Future Work

Even though this dissertation breaks new ground in spatiotemporal databases, it is only a baby step. Spatiotemporal data is complex and much research is needed to capture its linguistic features and implementation. Therefore, it is worth discussing them briefly here.

At present ParaSQL treats a single set of dimensions somewhat adequately when all values being queried or being extracted have the same dimensions. A support for varying dimensionality is needed. This requires a clear understanding of how phrases, inner queries, and outer queries interact when different dimensions are interleaved. In dealing with Güting's use case,
we hypothesized a projection of spatiotemporal dimension onto time dimension. A full-fledged framework allowing traversal with in specific sub-dimensions needs further research. We have also differ aggregate for future consideration.

Spatial joins, indices, and optimizers were not considered in this dissertation because they are full-fledged topics required to do extensive research. In order to improve system performance, index structures and optimizers should be integrated into the database system.

A storage manager plays an important role in a database system. The HCube, our hybrid storage, stores geospatial data in the form of GML. Because of this, coordinates are stored as text-based numbers. A binary representation would lead to significant improvement in processing performance when evaluating topological relationships between geometries.

In this dissertation, there was no consideration about multi-granularity in time and space. Supporting multi-granularity will provide great flexibility at the user level. For the sake of our discussion, suppose that the parametric database system supports multi-granularity in time. Now, a user can navigate a single tuple like multiple tuples grouped by a time-granule such as every month or every other year. For example, one may pose the query like “Retrieve daily maximum temperature of counties while monthly rainfall is less than 2.3”.

In order to quantify the system usability and performance, benchmarks should be further developed. One immediate task is to extend the query suite for the temporal dimension to the spatiotemporal dimension. Benchmarks, thus developed, should be publicly available for use of database and GIS communities.

Our spatiotemporal database system presented in this dissertation returns output of a query in a simple format. However, better ways of further analysis, reporting, and visualization should be considered. In addition to only report results of ad-hoc queries in databases, it is common to use general purpose programming language such as Java or C++ for further processing the query results tuple by tuple. A support to facilitate this for the parametric data model should be added. As a first step, this requires output to be streamed as ParaSQL-style tuples. This would ensure that there is no loss of richness in information after a query is executed and users can use linguistic constructs from ParaSQL as well as the general purpose host language.
for further processing of query results. After all processing is completed the result should be presented in a way, most intuitive to a user. As this is typically a final step only meant for human consumption, one has the freedom of reordering, formatting, and visual presentation of the result. With this freedom at hand, it should be easy to integrate many visualization tools available, e.g., those in meteorology and GIS communities.
APPENDIX A  Schemas of Crop and Soil Relations in NC-94 Dataset

<table>
<thead>
<tr>
<th>Field</th>
<th>Unit</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>County</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Corn Planted</td>
<td>ha/county</td>
<td>Float</td>
</tr>
<tr>
<td>Corn Yield</td>
<td>tons/ha</td>
<td>Float</td>
</tr>
<tr>
<td>Soybeans Planted</td>
<td>ha/county</td>
<td>Float</td>
</tr>
<tr>
<td>Soybean Yield</td>
<td>tons/ha</td>
<td>Float</td>
</tr>
<tr>
<td>Wheat Planted</td>
<td>ha/county</td>
<td>Float</td>
</tr>
<tr>
<td>Wheat Yield</td>
<td>tons/ha</td>
<td>Float</td>
</tr>
</tbody>
</table>

(a) Field information of Crop relation

<table>
<thead>
<tr>
<th>Field</th>
<th>Unit</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>County</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Total land acres in county</td>
<td>acre</td>
<td>Float</td>
</tr>
<tr>
<td>Total arable acres in county</td>
<td>acre</td>
<td>Float</td>
</tr>
<tr>
<td>Percent of land area in total county area</td>
<td>%</td>
<td>Float</td>
</tr>
<tr>
<td>Mean slope of arable land in county</td>
<td></td>
<td>Float</td>
</tr>
<tr>
<td>Mean slope of total land in county</td>
<td></td>
<td>Float</td>
</tr>
<tr>
<td>Mean drainage classification of county</td>
<td></td>
<td>Float</td>
</tr>
<tr>
<td>Depth to water table during wettest part of year</td>
<td>cm</td>
<td>Float</td>
</tr>
<tr>
<td>Depth to bed rock</td>
<td>cm</td>
<td>Float</td>
</tr>
<tr>
<td>Maximum root depth</td>
<td>cm</td>
<td>Float</td>
</tr>
</tbody>
</table>

(b) Field information of Soil relation

Figure A.1  Schemas of Crop and Soil relations in the NC-94 dataset
APPENDIX B  Object Model for Geometry

Figure B.1 shows the OGC’s object model for geometry. The base Geometry class has sub­classes for Point, Curve, Surface and Geometry Collection. Each geometric object is associated with a Spatial Reference System, which describes the coordinate space in which the geometric object is defined.

Figure B.1  Geometry class hierarchy [47]
APPENDIX C  Expression Tree

<ExpressionTree>
<Iterator type="relationscan" relname="climate">
  <Projection isAll="true">
    <Annotation/>
  </Projection>
  <Restriction existDomainExp="true">
    <DomainExp opType="unary">
      <ArithOpConst opType="lessThan">
        <Arithmatic opType="div">
          <Arithmatic opType="plus">
            <Attribute attrName="MaxTemp" attrPos="4" type="float"/>
            <Attribute attrName="MinTemp" attrPos="5" type="float"/>
          </Arithmatic>
          <Const value="2" type="float"/>
        </Arithmatic>
        <Const value="0" type="float"/>
      </ArithOpConst>
    </DomainExp>
  </Restriction>
  <WhereCond>
    <Condition isTrue="false">
      <BoolExp opType="and">
        <BoolExp opType="and">
          <BoolExp opType="unary">
            <FuncOpConst opType="equal">
              <Function return="value">
                <SpatialFunc name="neighbor" args="2">
                  <Attribute attrName="County" attrPos="1" type="integer"/>
                  <Const value="Story" type="string"/>
                </SpatialFunc>
              </Function>
              <Const value="1" type="integer"/>
            </FuncOpConst>
          </BoolExp>
          <BoolExp opType="unary">
            <FuncOpConst opType="greaterThan">
              <Function return="value">
                <SpatialFunc name="area" args="1">
                  <Attribute attrName="County" attrPos="1" type="integer"/>
                </SpatialFunc>
              </Function>
              <Const value="540" type="float"/>
            </FuncOpConst>
          </BoolExp>
        </BoolExp>
      </BoolExp>
    </Condition>
  </WhereCond>
</ExpressionTree>
<FuncOpConst opType="">
    <Function return="value">
        <SpatialFunc name="population" args="1">
            <Attribute attrName="County" attrPos="1" type="integer"/>
        </SpatialFunc>
    </Function>
    <Const value="20000" type="float"/>
</FuncOpConst>
BIBLIOGRAPHY


