Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2010

An XML based scalable implementation of temporal databases using parametric model

Kartic Ramesh
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Ramesh, Kartic, "An XML based scalable implementation of temporal databases using parametric model" (2010). *Graduate Theses and Dissertations*. 11600.
https://lib.dr.iastate.edu/etd/11600

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

An XML based scalable implementation of Temporal Databases using Parametric Model

by

Kartic Ramesh

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Shashi K. Gadia, Major Professor
Ying Cai
Wallapak Tavanapong

Iowa State University

Ames, Iowa

2010

Copyright © Kartic Ramesh, 2010. All rights reserved.

TABLE OF CONTENTS

TABLE OF CONTENTS ii
LIST OF FIGURES iv
ABSTRACT v
CHAPTER 1. Introduction 1
CHAPTER 2. Related Work 5
CHAPTER 3. Parametric model 6
3.1. Temporal elements 6
3.2. Attribute Values 6
3.3. Associative navigation op -comparisons 7
3.4. Tuples and relations 8
3.5. XML-based physical representation. 9
CHAPTER 4. Parametric SQL 10
4.1. Relational Expressions 10
4.2. Domain Expressions. 10
4.3. Boolean Expressions 10
4.4. Weak Equality 13
4.5. Weak Operators 13
CHAPTER 5. Existing database implementation technologies 15
5.1. Storage System 15
5.2. CanStoreX 16
5.3. Cyclone Database Implementation workbench (CyDIW) 18
CHAPTER 6. Implementation of the TDB System 21
6.1. General architecture 21
6.2. Prior Work 22
6.3. The need for a new prototype. 23
6.4. The database catalog 24
6.5. ParaSQL Query Parser 26
6.6. Expression Tree Generator. 27
6.7. Iterator 29
6.8. ParaSQL Query Executor. 31
6.9. Query Result Writer 32
6.10. Commands for the CyDIW GUI 32
6.11. Overview of CyDIW 33

6.12. TDB Command Format 34
6.13. Some Important TDB Commands 35
6.13.1. Command for generation of synthetic temporal data 35
6.13.2. Command to load a dataset 35
6.13.3. Command to open a database 35
6.13.4. Command to display TDB catalog 35
6.13.5. Command to parse query 36
6.13.6. Command to display Parse Tree 36
6.13.7. Command to build expression tree 36
6.13.8. Iterator based command for query execution 36
6.13.9. Execute Query 38
6.13.10. Close Database. 38
6.14. TDB Command Execution 39
CHAPTER 7. Conclusion 40
CHAPTER 8. Future Work 41
8.1. Physical Representation 41
APPENDIX A. 48
ACKNOWLEDGEMENTS 50

LIST OF FIGURES

Figure 1. Personnel database 7
Figure 2. XML representation of Emp relation 8
Figure 3. BNF for expressions in ParaSQL 11
Figure 4. Snapshot of the Dept Relation at NOW 13
Figure 5. Pagination using CanStoreX 17
Figure 6. The Cyclone Database Implementation Workbench. 19
Figure 7. TDB System Architecture 22
Figure 8. TDB System Catalog. 25
Figure 9. Snapshot of the randomly generated data 26
Figure 10. XML Representation of a Parse Tree 28
Figure 11. XML Representation of Expression Tree 29
Figure 12. Query Execution Algorithm 32

ABSTRACT

A parametric model and a query language ParaSQL for temporal databases has been ptoposed in the past. As the attribute values in the model can vary in length, it is difficult to use existing relational storage technology. To address this, CanStoreX, our XML-based storage technology has been deployed in a prior implementation. In parallel, the storage technology as well as our style of implementation for database prototypes have gone through an evolution. This has necessitated the previous implementation to be revisited. In addition, a new parser has been developed using JavaCC. Furthermore, a larger subset of ParaSQL has been imlemented. For testing, a utility to generate synthetic temporal relations has been developed. Conforming to the new style, the present implementation has been encapsulated in terms of high level commands. This allows end-users to system developers on one hand and various database protototypes on the other, to interact with a central storage system from a common GUI that facilitates execution of batches of commands. Our implementation has helped to identify pragmatic issues in temporal database implementation as well as as the storage technology more clearly.

1. Introduction

Conventional databases are only capable of storing and querying the current perception of reality and relationship among objects, such database systems exclude old data values when the data is updated. Unlike conventional databases, a temporal database is capable of storing evolution of data, thereby allowing users to examine complete object histories.

Temporal databases are one of the active research areas in the database community and tremendous research work has been done by many researchers. Applications of temporal databases include organizational record-keeping, economics, and scientific applications.

In the temporal database literature, we can find three types of timestamps – instants, intervals, and temporal elements. A temporal element is a finite union of intervals [20, 22]. Based on timestamps, temporal data models are termed point-based models, interval-based models, and temporal-element based models, respectively. An interval is obviously a temporal element. An instant can be thought of as a closed interval with the same end-points. Therefore, an instant of time can also be considered a temporal element. Thus a temporal element generalizes all the three data types for time.

In our implementation, we are concerned with developing a temporal database system based on the Parametric model which assumes the presence of a hypothetical underlying space called the Parametric Space. The parametric data model defines attribute values as functions over the parametric space. This leads to a one-to-one correspondence between an object in the real world and a tuple in the database. Such one-to-one correspondence between an object and a tuple can avoid self-joins which are inevitable in temporal data models where temporal elements are not used as exclusive use of intervals and instants fragment an object into multiple tuples. Although at the model level the use of temporal elements gives rise to a natural framework for users, implementation becomes more challenging because of variable size of attribute values. Whereas with intervals and instant time stamps several tuples would reside on a page on the disk, when temporal elements are used tuples would vary in length from several tuples fitting in a page to a tuple spanning several pages. In particular, it is difficult to adapt the parametric temporal data model within relational database storage technology that is deployed commonly in interval and point based models.

In order to meet the implementation challenge, we use XML and our own XML-based storage technology called CanStoreX (Canonical Storage for XML). CanStoreX is capable of paginating large XML data in order to store them in pages on the disk. Once the data is stored in a binary paginated form as tree, it is processed using CsxDOM, which is our version of the classical DOM API. In addition for storing data that is possibly very large in size, we make other interesting uses of XML Many artifacts, such as parse trees (also called abstract syntax tree) and expression trees are also represented in XML, which are human readable and more reliable by removing the reliance on traditional linked list-based binary structures that are accessible only when some module implemented in some specific programming language is being executed. These XML documents are small and we use the usual text-based representation as ordinary files in the operating system and use a conventional DOM API. In addition, we use XML for representing many configurations, catalogs, and complex parameter passing.

Several query languages exist that can be used to query temporal databases. One such query language, ParaSQL which is based on the parametric model is used in our approach. The way in which a natural language query is expressed varies significantly from one query language to another. ParaSQL has three types of expressions: relational expressions, domain expressions, and boolean expressions that, when evaluated, result into relations, temporal elements, and boolean values True and False. There three types of expressions are mutually recursive and natural to use. For example, the help realize what appear to be selection in a natural language as a selection in a ParaSQL rather than requiring joins as in other language – thereby leading to convoluted expressions.

It is interesting to note that in the parametric model the boundary between those queries that can and those that cannot be expressed in classical framework is very clear. The two kinds of are called *weak* and *strong* (meaning non weak) queries, respectively. It turns out that the classical boolean expressions map directly to domain expressions in ParaSQL. The classical queries are absorbed by relational and domain expressions in our model. Clearly, this implies that the relational and domain expressions in ParaSQL are weak. The boolean expression in ParaSQL, however, have no counterpart in the classical framework. Therefore, the queries involving boolean expressions in ParaSQL are strong. We describe these concepts in detail later on. In the rest of this section we present an outline of the work undertaken, putting it in the context of existing work.

A prior implementation of a temporal database system on the Parametric model using CanStoreX as the storage was undertaken by Seo-Young Noh. CanStoreX pagination requires every page to be a self-contained XML document on its own right. In an early version of CanStoreX although the interpage navigation was in terms of binary pointers, the pages themselves were stored in text format and traditional DOM API was used for internal navigation with in pages. This is the version used by Noh in his implementation. Because pages were binary objects their instreaming beyond one GByte caused serous problems. Later on the pages were also implemented as trees in a binary format. Also, the prior implementation was a stand-alone system, having its own storage and buffer managers as well as a customized graphical interface. The GUI consisted of consisted of text boxes and buttons with very specific functionality for stepping through the life-cycle of a single query, from creation of expression tree to its execution. In the recent past, our style of implementing database prototypes has also gone through some evolution leading to CyDIW, the Cyclone Database Implementation Workbench. CyDIW enables unified handling of multiple database subsystems. Subsystems can be existing database systems such as various SQL-platforms such as Oracle as well database prototypes under development. In order to facilitate database development a storage consisting of a heap of pages, the storage manager, and buffer manager are available. Rudimentary file system and file services are available to client modules. In addition CanStoreX technology that includes pagination utility to store XML documents in binary format and consume them vis csxDOM are available. A simple, yet powerful integrated GUI has also been developed that is used for interacting will all database subsystems. The integrated GUI is command based and executes batches of commands for every registered database subsystem.

To conform to the above mentioned changes in CyDIW the previous implementation has been revised in several respects. The temporal database system now uses the binary version of CanStoreX. Also, the common storage and buffer managers are used. Following our new style of implementation, the system is encapsulated in terms of commands. In order to port our new implementation to use the binary version of CanStoreX, we have re-written the execution engine to interact directly with the new CanStoreX engine. Also, to integrate our system with the CyDIW, we have developed a suite of commands and a command parser specifically designed for our system.

Noh implemented a parser manually using SJM. In our group we use the highly popular JavaCC for our parsing needs. Therefore, we have implemented a new parser in using JavaCC. In addition, we have implemented some additional language constructs of ParaSQL.

For testing generation of synthetic datasets was undertaken. The previous version available to us generated relations with fixed amount of variability where all tuples would have the same variability and length. The new implementation is more versatile. The desired details for generation of data can be included in the XML-based database catalog.

The organization of the rest of this thesis is as follows. Chapter 2 briefly describes some of the related work in temporal databases. Chapter 3 and 4 describe the Parametric Model and Parametric SQL respectively. Chapter 5 briefly discusses the XML based storage system and the pagination technique. Chapter 6 provides details of our implementation spanning from generation of test data to revamping the ParaSQL parser to re-writing the query execution engine. Integration of our temporal database system with CyDIW in terms of a suite of commands, specifically designed for our system, is described. Chapter 7 summarizes our findings and Chapter 8 describes further work that can be done to extend our system.

2. Related Work

Several approaches have been developed to represent time in databases. These include point based [2] and the interval based [2] approached on one hand and the parametric approach on the other, which represents domain of an object as a temporal element consisting of a finite union of time intervals.

TSQL2 [18] and IXQL[38] are examples of works that use interval timestamps. Temporal element based representation has been used in ParaSQL [20]. Tinsel's model also uses temporal elements, but the attribute values are not explicitly formalized as functions NTC [39]. They use nesting within attribute values to describe variation in values over time. Comparison of the different approaches toward utilizing XML in parametric model for temporal data have been done by Noh et. al. in [8] and [24].

There has been a significant growth in the usage of XML to model databases. Several XML databases systems exist today which include Xindice[40] and eXist[41]. In these works XML is used to model the temporal data at logical level itself as well as for storing the temporal data at the physical level. In other words users as well as the system think in terms of XML. In the parametric approach XML is used only at physical level to store and access data [7]. In other words XML plays no role in the queries expressed by end users who are oblivious to the role of XML at the physical level. These issues are clarified in [25], where it is argued that use of XML in modeling the variability arising in time dimension offers little advantage to users.

More recently, [37] discusses the general algorithmic approaches to answering similarity queries on spatial, spatiotemporal and temporal databases. [42] discusses the differences between abstract (theoretical languages such as relational algebra) and concrete temporal query languages (for example that are SQL-based). [43] proposes an event based approach to modeling spatiotemporal data. [44] propose a unique theoretical annotated temporal algebra which is convenient for specifying how algebraic operations should behave.

3. Parametric model

Parametric data model uses temporal element based, explicit time domain, and attribute level time-stamping. The main objective of the parametric data model is to obtain a query language that is most natural in favor of users.

To further delve into the details of the parametric model, let us consider a Personnel database is shown in Figure 1 as a running example. The Emp relation lists the employees in an organization. Name serves as the key and the relation lists the Salary and DName (department name) attributes. Dept relation consists of DName and MName attributes that are names of the department and the manager, respectively with DName as its key. The Personnel database consists of the Emp and Dept relation, we note that alternatively MName can be chosen as the key of the information in the Dept relation. Thus an alternative to Dept is Manager relation, also shown in Figure 1. Whereas Dept relation represents the history of departments, the Manager relation represents the history of Managers. The information in thethe relation is intertwined; the snapshots of Dept and Manager at some given time t is the same.

3.1. Temporal elements

The parametric model [29] assumes that there is a universe of time that consists of an interval [0,NOW] and for the sake of simplicity we assume that to be equal to the set {0,1,…,NOW}. A temporal element, introduced in [20,22], is defined to be a finite union of intervals. As is clear from the definition, a single interval is also a temporal element and so is an instant t as this instant can be identified as the interval [t, t].

The set of all temporal elements is closed under union, intersection, and complementation ($\cup$, $\cap$ and $-$, respectively). The complementation is computed with respect to [0,NOW]. The set of temporal elements along with $\cup$, $\cap$, $-$, $\varnothing$ and [0,NOW] is obviously a Boolean algebra.

3.2. Attribute Values

To capture changing value of an attribute we define a temporal value of an attribute A to be a function from a temporal element into the domain of A [22]. For example, the salary might change from 50,000 in [0,10] to 65,000 in [11,25].

Name	Salary	DName
[11,60] John	[11,49] 50K [50,54] 55K [55,60] 60K	[11,44]∪[55,60] R&D [45,54] Test
[0,20]∪ [41,51] Tom	[0,20] 45K [41,51] 50K	[0,20] Test [41,51] Sales

Emp Relation

DName	MName
[0,50]∪[71,NOW] R&D	[0,50] Kim [71,NOW] Lee
[11,NOW] Test	[11,44] Leu [45,NOW] Inga
[45,NOW] Sales	[45,NOW] Leu

Dept Relation

DName	MName
[0,50] R&D	[0,50] Kim
[71,NOW] R&D	[71,NOW] Lee
[11,44] Test [45,NOW] Sales	[11,NOW] Leu
[45,NOW] Test	[45,NOW] Inga

Manager Relation as an alternative to Dept relation

Figure 1. Personnel database

3.3. Associative navigation op –comparisons

Our counterpart to the construct A op B of the relational model is [[A op B]], which captures the time when A is in op-relationship with B [30].

[[A op B]] = {t : A(t) op B(t) is True}.

We also allow the construct [[A op b]] where b is a constant, which is evaluated by identifying the constant b with the constant function [0,NOW] b.

For example the temporal element during which Salary was equal to 50,000 is represented as [[Salary=50000]], which in this case evaluates to [0,10]. Another example is [[([25,32] Toys, [33,NOW] Shoes) = ([0,NOW] Shoes)]] = [33,NOW].

3.4. Tuples and relations

Informally a tuple or a record is said to be a concatenation of attribute values. The first row in the Emp relation shown in Figure 1 is an example of a tuple. It consists of the salary history and the department history group together by the key which in this case happens to be the name of the employee. If we assume that all the temporal values in a tuple have the same domain, then it is called a homogeneous tuple [22].

Relations are defined as a set of temporal tuples [28]. It has to be noted that not every set of tuples is a relation. This is because, a relation is always defined together with a key.

```
<Relation name="Emp">
 …
   <tuple>
      <attribute name="Name">
         <value> John
            <pdom><dunit start="11" end="60"/></pdom>
         </value>
      </attribute>
      <attribute name="Salary">
         <value> 50K
            <pdom><dunit start="11" end="49"/></pdom>
          </value>
          <value> 55K
            <pdom><dunit start="50" end="54"/></pdom>
         </value>
         <value> 60K
            <pdom><dunit start="55" end="60"/></pdom>
         </value>
      </attribute>
      <attribute name="DName">
         <value> R&D
            <pdom>  <dunit start="11" end="44"/>
                    <dunit start="55" end="60"/>
            </pdom>
         </value>
         <value> Test
            <pdom><dunit start="45" end="54"/></pdom>
         </value>
      </attribute>
      <pdom><dunit start="11" end="60"/></pdom>
   </tuple>
…
</Relation>
```

Figure 2. XML representation of Emp relation

3.5. XML-based physical representation

Following [7], the XML representation of the Emp relation is shown in Figure 2. Only a snapshot of the xml file is shown in the figure. This file actually contains all the tuples from the relation. The <tuple> tag encodes a tuple in the relation. The <attribute> encodes an attribute value that is broken into atomic values included in the <value> element together with its temporal domain <pdom> consists is a temporal element. The temporal element consists of several intervals with their start and end points described in the <dunit> elements.

4. Parametric SQL

Parametric SQL consists of relational, domain, and boolean expressions [19]. These expressions evaluate to relations, temporal elements, and boolean values (True and False), respectively. We explain each of these in detail in this section.The simplified BNF of Relational, Domain and Boolean expressions in the parametric model is shown in Figure 1.

4.1. Relational Expressions

Relational expressions are the syntactic counterpart of temporal relations. A relational expression returns a relation that is a set of temporal tuples. They include union, difference, restructuring, projection, selection and natural Join. These are defined in detail in [27]. In this thesis we do not consider union and difference operators. We only consider the SQL-style select operator which absorbs the selection, projection, and join (SPJ) operators as well as the restructuring operator. The restructuring operator, discussed later on, allows one to apply a different key to a relation.

4.2. Domain Expressions

Domain expressions are the syntactic counterpart of temporal elements [27]. They are formed using temporal elements, $[\![e]\!]$, $[\![A]\!]$, $[\![A \text{ op } B]\!]$, $[\![A \text{ op } b]\!]$, $\cup$, $\cap$, and –.Here e is a relational expression. It has to be noted here that a domain expression could contain a relational expression as described in its BNF. This gives us the ability to nest one query inside another. Detailed explanation of relational expression and domain expressions can be found in [23].

4.3. Boolean Expressions

Boolean expressions are formed by $\mu \subseteq \nu$ where μ and ν are domain expressions. More complicated Boolean expressions can be formed using “and, “or”, and “not”. Any Boolean expression evaluates to True or False. This is the condition that is used to qualify tuples before they are restricted and projected.

As shown in the BNF of the select statement, we see that the where clause takes a Boolean expression as input, evaluates the result and hence either qualifies or disqualifies a tuple based on the result.

```
<expression>::=
        <relational expression> |
        <domain expression> |
        <boolean expression>
<relational expression>:== stored_relation> |
         <relational expression> union <relational expression> |
         <relational expression> difference <relational expression> |
        select <attribute list>
        [restricted To <domain expression>]
        from <relation list>
        [where <boolean expression>]
<domain expression> ::=
        <temporal element>
        [[<attribute>]] |
        [[<attribute> op <attribute>]] |
        [[<attribute> op <value>]] |
        [[<value> op <value>]] |
        [[<relational expression>]] |
        (<domain expression>) union <domain expression |
        (<domain expression>) intersection <domain expression |
        (<domain expression>) minus <domain expression |
<boolean expression> ::=
        True |
        False |
        <atomic boolean expression> |
        (<boolean expression>) and <boolean expression |
        <boolean expression> or <boolean expression> |
        not <boolean expression>
<atomic boolean expression> ::=
        <domain expression><set op><domain expression> |
        <attribute> <op> <attribute> |
        <attribute> <op> <value> |
        <function> <op> <value> |
        <function_identifier> ( <domain expression> ) <op> <value>
<set op> ::=
        subset |
        superset |
        equal to |
        not equal to
```

Figure 3. BNF for expressions in ParaSQL

Our implementation includes a rule that helps apply a function to a domain expression and comparing the resulting value with another value.

Example 1. To illustrate the select statement, let us consider a simple query shown below:

```
select e.Name, e.Salary
restricted to [[e.DName='Toys']]
from Emp e
where NOW ⊆ [[e]];
```

The query aims to find the name and salary of those employees who currently work in the Toys department. The where clause here contains the condition NOW ⊆ [[e]] that verifies that the employee is currently working. The restricted to clause limits the information to be retrieved for such employees to the time when they worked in Toys.

Example 2. Now, consider the two queries written in ParaSQL

```
select e.Salary
from Emp e
where e.Dept='Sales';                              //--Q1

select e.Salary
restricted to [[e.Dept='Sales']]
from Emp e;                                        //--Q2
```

Let's consider Q1. As shown here, it is not required to have a restriction clause in all queries in ParaSQL. This query aims to retrieve all the salary details of the employees who have worked sometime in the Sales department. Note that e.Dept='Sales' is simply a short hand for "⟦e.Dept='Sales'⟧ != emptySet" Therefore all the tuples will be scanned and from those that have worked in the Sales department, the Salary details will be retrieved. Tuples that have never worked in Sales will be deleted.

Now consider the query Q2. Here the where condition is absent. This query aims to find the salary details only during the time the employee worked in the Sales department. Therefore all the tuples will be scanned the salary details will be restricted to the time when an employee worked in the Sales department. Obviously, the salary details of an employee who has never worked in the Sales department will not be fetched. The difference between the two queries is that the first one will retrieve the entire salary history whereas the second will retrieve salary history only during the employee worked in Sales.

DName	MName
NOW R&D	NOW Lee
NOW Test	NOW Inga
NOW Sales	NOW Leu

Figure 4. Snapshot of the Dept Relation at NOW

4.4. Weak Equality

To show the significance of where clause, we have to discuss the concept of weak equality. [22] defines two relations r and s to be weakly equal if at every instant t, the snapshots of r and s at t are equal. The parametric approach injects the classical concepts into a temporal database through snapshots of that database. Such concepts, in the context of temporal databases are termed *weak*. The weak equality between two relations means that the two relations have the same snapshots. An example of a weak identity is $r \cup s = s \cup r$. It turns out that all notions in temporal databases are not weak. For the sake of emphasis, the notions that are not weak are termed strong [36].

The snapshot of relation r at time instant t is the relation obtained by restricting each tuple of the relation r to t. For example, consider our Dept relation shown earlier. If we restrict each of the tuples in that relation to NOW, then we get a snapshot of the Dept relation at time instant t = NOW. This relation is shown in Figure 4

To illustrate the concept of weak equality, we restructure the Dept relation by changing the key of the relation from Dept to MName. This relation is called Manager which is also shown in Figure 1.

4.5. Weak Operators

A unary relational operator O is said to be weak if $O(r_1)$ is weakly equal to $O(r_2)$ whenever r_1 is weakly equal to r_2. Similarly, a binary relational operator O is said to be weak if $O(r_1, s_1)$ is weakly equal to $O(r_2, s_2)$ whenever r_1 is weakly equal to r_2 and s_1 is weakly equal to s_2 [5,27].

Example 3. Consider the two queries written in ParaSQL.

```
select D.MName
from Dept D
where [11,60] ⊆ [[D.MName]];                                   //--Q3

select M.MName
from Manager M
where [11,60] ⊆ [[M.MName]];                                   //--Q4
```

The grammatical structures of the two queries are identical. The only difference is that the former selects the names of the manager from the Dept relation such that the interval [11,60] is contained in the domain of MName and the latter does the same for Manager relation. This translates to list all the managers that managed between [11,60] in plain English. For the state of the database as in Figure 1, the query Q3 will return an empty relation while the query Q4 will return one tuple matching the name of 'Leu'. We know that the two relations Dept and Manager are weakly equal to each other but the result that we get from running the queries Q3 and Q4 are not equal. This means that the where clause is strong. The implication of this is the queries Q3 and Q4 cannot be expressed in classical SQL. The where clause separates weak and strong queries. In general, queries that do not involve the where clause are weak and those that do include a where clause are strong. The separation between weak and strong concepts is clear in the parametric model.

5. Existing database implementation technologies

In this section we give a brief introduction to CanStoreX (Canonical Storage for XML) and the pagination algorithm used to store our relations. In the temporal database prototype temporal relations are stored as XML documents using CanStoreX pagination technology. The query language is implemented by processing the paginated document using CsxDOM, the DOM API for the paginated XML document in CanStoreX.

XML [46] is a simple, natural but powerful language to describe data and metadata. An XML document can be viewed as a rooted tree with nodes having a varying number of ordered child nodes. Recursively, child nodes can be trees of varying size and complexity as XML elements on their own right. XML is capable of representing relational and objected-oriented data, scientific data, documents such as this thesis, web content, and metadata.

Traditionally, there have been two important APIs for XML. These are SAX and DOM API. SAX is a parser that scans a documents as a text stream from beginning to end linearly and as events are detected, actions pre-specified by the user are taken. DOM API on the other hand is quite powerful and recognizes the hierarchical nature of XML. It allows traversal from a node to its parent, siblings and children and is a versatile tool for processing XML documents.

5.1. Storage System

A general purpose computer system consists of a CPU (Central Processing Unit), main memory and a disk. When information residing on the disk is to be processed by the CPU, it has to be first brought into the main memory in order for CPU to access it. A page size is chosen and used as a unit of transfer between main memory and disk. A page once read in main memory, should be large enough to keep CPU busy until it is ready to handle the data on the next page. Page sizes in a given environment are typically fixed.

In many applications, a page may be large enough to hold several logical units of information called records. This is a standard situation in relational databases where logical records are typically much smaller than a page. In other applications, logical units of information such

as records could be larger than pages. For example a tuple in temporal database may not only not fit in a page, it may span several pages.

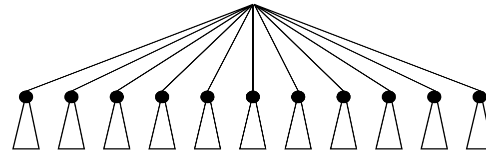
The place where a page is stored in main memory for processing is called a buffer. Thus the size of a buffer is same as that of the page. A buffer is a window in main memory to the disk. Typically, a system sets a certain amount of memory aside for buffers called buffer space, buffer pool or cache.

Since main memory is small and more expensive relative to the disk, the buffer space is a premium resource. Buffers have to be managed wisely for obtaining a good performance and the objective is to reduce the number of disk accesses that is often the bottleneck when processing large amounts of information. This gives rise to the need for a buffer manager whose job is to manage buffer space wisely.

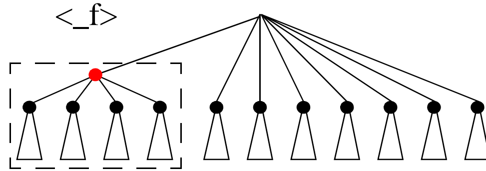
5.2. CanStoreX

CanStoreX (a Canonical Storage for XML) paginates an XML document into pages [45]. An interesting characteristic of CanStoreX is that each page itself is organized as an XML document on its own right. The page-based organization of the document directly mimics the hierarchical structure of the original document. Figure 5 shows the basic idea behind pagination. Figure 5(a) shows an XML document where the root has a variable number of children. The child trees are XML elements on their own right and vary in complexity and physical size. Recursively the base case is when the whole document fits on a page. If this is not true than there can be one of two reasons for it. First, considered in Figure 5(b), is where the number of children is so large that even pointer to all of them will not fit in a page. In this case some children can be grouped together and a dummy parent can be created to represent all of them via a single pointer. Second, shown in Figure 5(c), is when a child is so large that it would not fit in a page. In this case the child can be differed to a page of its own and be represented by a pointer. This argument is the basis from where CanStoreX starts. Here, the two types of nodes have been used to facilitate pagination.

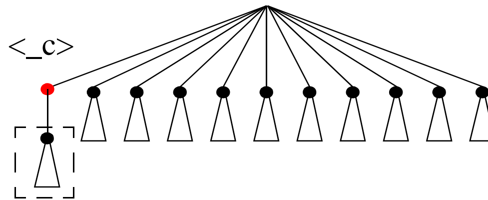
CanStoreX does not place any limits or expectations on what kinds of nodes will be needed or used to accomplish pagination. However, CanStoreX requires an XML document to be bro-



(a) The source XML document



(b) Case A. The fanout is too large



(c) Case B. A child is very large

Figure 5. Pagination using CanStoreX

ken into pages where the pages themselves are XML documents on their own right. (Except when information has no structure and requires multiple pages).

Natix is another technology for storing XML documents. Using what are called helper and proxy nodes, Natix breaks up a documents into small chunks. Several chunks, that are possibly unrelated, may share a page for their storage. A page is not required to be a self-contained XML document. Natix uses helper and proxy nodes in specific way, where as CanStoreX considers the types and use of storage facilitating nodes to be open to future evolution.

Since XML is a technology that enables other technologies, the CanStoreX system can be used to implement a general-purpose seamless storage for relational and object-oriented databases, XML, and metadata.

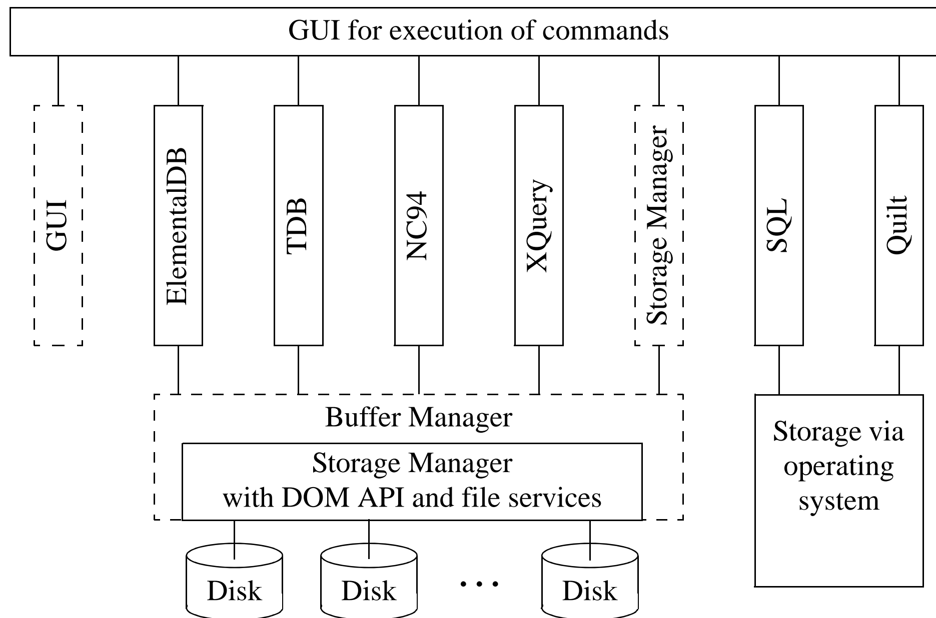
Recall that a page in CanStoreX is required to be a self-contained XML document in its own right. In the earliest version of CanStoreX, implemented by Shihe Ma [12], the pages

were stored as plain text. This pagination algorithm took an input XML file and converted it to a pff (page formatted file) file. The inter-page navigation was supported via binary pointers. But for navigation within pages the traditional DOM API was used. Due to use of Java, the pages were treated as objects and due to heap based storage environment in Java the pagination did not scale to large documents in several gigabyte range.

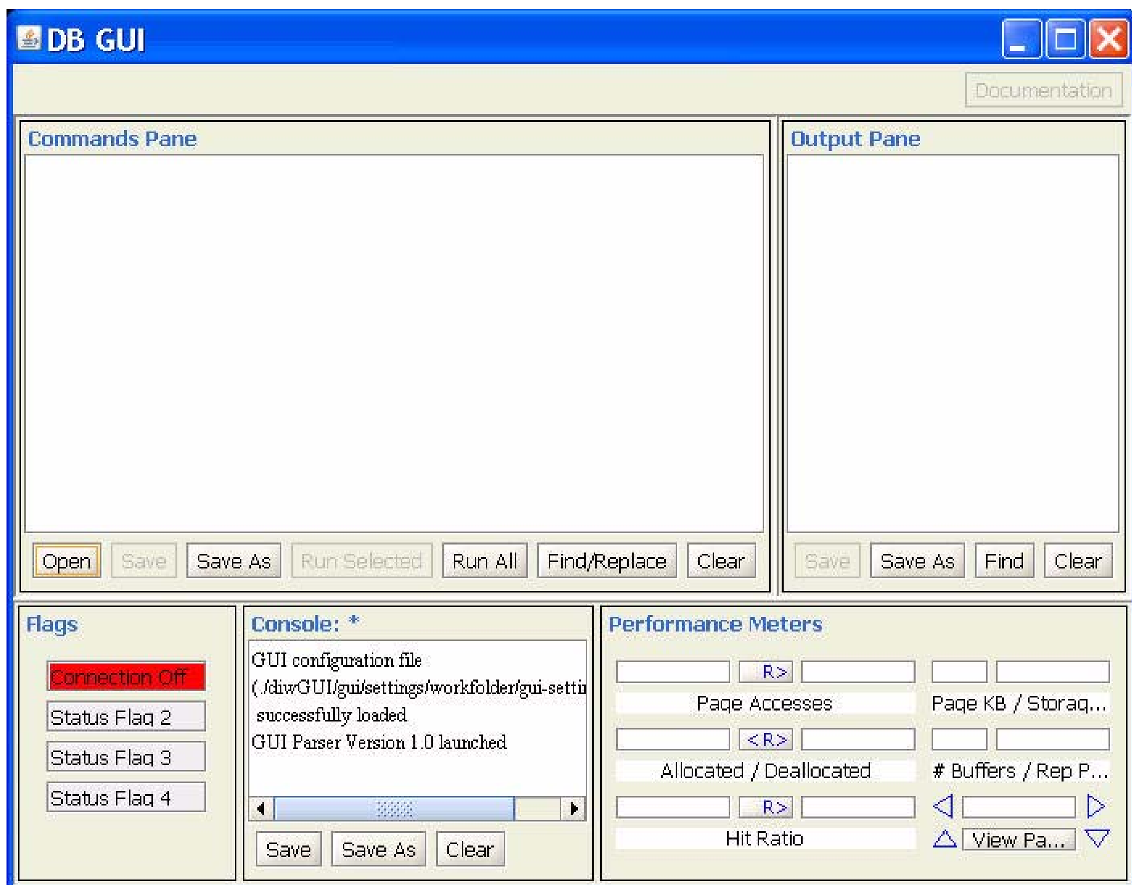
Using the CanStoreX developed by Ma and making some minor improvements, a prototype for NC94, a spatiotemporal dataset for agriculture, was implemented by Seo-Young Noh. Noh also implemented a prototype for temporal databases. The work under this thesis builds with the prototype developed by Noh. Patanroi[14] implemented pages in a binary format. The binary page implementation removed major memory issues and allowed CanStoreX to parse paginate and process terabyte range XML documents. Further development was continued by Bob Stark, Srikanth Krithivasan, and Matt Swanson. Krithivasan made significant addition making traversal of XML-axes iterator based. He also implemented an XQuery engine on the top of CsxDOM using a parser for XQuery, developed earlier by Satyadev Nandakumar. [16]. Niranjan Kumar ported NC94 prototype to binary version of CanStoreX.

5.3. Cyclone Database Implementation workbench (CyDIW)

We are developing several database prototypes. In past each prototype has been developed separately, having its own storage and buffer management and a dedicated GUI. The GUIs were button-based where buttons were customized to render some specific functionalities. In a paradigm shift CyDIW, the Cyclone Database Implementation Workbench has been developed. The storage and buffer management for all database prototypes have been centralized. CanStoreX pagination and CsxDOM have been turned into services available to all prototypes that may need to use large XML documents to store data. Artifacts consisting of binary pages are stored as files. Artifacts include traditional relations that are stored in industry standard binary pages but viewed as XML documents, indexes, and spatiotemporal datasets. The GUI provides a framework from where a batch of commands can be executed. Every prototype and subsystem is assigned a prefix. Prefixed commands belonging to different prototypes can be interleaved in the same batch. The workbench provides services to facilitate command execu-



(a) Organization of the workbench



(b) The centralized GUI

Figure 6. The Cyclone Database Implementation Workbench

tion. Commands can be stored in variables and commands on such commands can be executed. As some results of some commands can be very large, they can be redirected to operating system as well files internal to CyDIW. Customized buttons are not needed anymore and the same time the GUI is far more powerful and natural with hardly any learning curve. Storage of command containing variables that store commands, together with ability to highlight and selectively execute it, is like having custom designed recursive buttons within a prototype. The GUI provides ability to display XML and text documents. As many of our artifacts, such as syntax trees, expression trees, catalogs, metadata, global variables, and system configurations are stored as XML documents this makes the development as well as end user environments highly visible, readily accessible, and self-contained. The GUI also facilitates performance benchmarking. XML-based logs can be created, performance commands can be logged, and generation of performance reports can be automated.

The entire behavior of a system is encapsulated in terms of commands. A batch of commands can be used to encapsulate an experiment. An experiment may consist of creation of storage spanning multiple disks consisting of specified page size, start storage and buffer managers with a desired number of buffers, create synthetic datasets or load datasets, start a database prototype, execute commands, do benchmarking, prepare reports, and close a database prototype. The entire experiment can be repeated with the click of a button. This would make project management a very high level activity, make prototypes self documenting at very high level, reduce learning curves – all via a very simple GUI that is used by students, instructors, developers, and researchers.

All our prototypes are being ported to CyDIW. Narayanan [17] ported the NC94 prototype to CyDIW. As mentioned before, CanStoreX style pagination and CsxDOM have become services. XQuery engine has been ported and being further developed by Xinyuan Zhao. Porting of the TDB, originally developed by Noh, and its further development are being accomplished under this thesis and discussed in the next section.

6. Implementation of the TDB System

The project under this thesis covers several aspects of the TDB (temporal database system) prototype. These are listed below and described later in greater detail in the remainder of this section. It helps to keep in mind that the temporal relations are stored as XML documents that reside in the central storage of CyDIW in paginated format. For processing CsxDOM API is used.

• As mentioned in previous section, the existing prototype developed by Seo-Young Noh was a stand alone system with its own storage manager, buffer manager, and the GUI. This prototype has been ported to our Cyclone Database Implementation Workbench (CyDIW).

• The parser has now been implemented in JavaCC.

• The query language ParaSQL has been extended to include nesting of the domain of a relational subexpression in the restricted to clause.

• Noh's implementation generated tuples of fixed size. The data generation now is more flexible. The configuration information for the synthetic data to be generated is added to thee relation catalog. For preparing datasets, first ordinary .xml files are generated and then they are copied into .bxml file using copyfile command in CyDIW. The command internally invokes pagination.

• The entire implementation has been made command based.

6.1. General architecture

The general architecture of a Temporal Database system is shown in Figure 7. Most implementations of a database system along with a query language follow a similar procedure. The user query is first parsed using a query language parser. The parser checks the query string for valid syntax. If the syntax is correct a parse tree is built, otherwise an error is returned. The parser creates a parse tree using only fixed rules about the syntactic structure of the language. It does not consult the system catalogs, so there is no opportunity to understand the detailed semantics of the requested operations. The parse tree is XML-based.

After the parser completes, an expression tree generator takes the parse tree handed back by the parser as input and does the semantic interpretation needed to understand which relations, functions, and operators are referenced by the query by looking up the system catalog. The data structure that is built to represent this information is called the expression tree. The expression tree is also XML-based. Implemented by Noh, the conversion of a parse tree into an expression tree is done by using prune and graft operations in DOM API.

The generated expression tree is then passed on to the query executor. The query executor evaluates tuples returned by the iterator which acts a demand-pull pipeline. Each time the iterator is called, it has to deliver one or more tuples or report that it is done delivering tuples. The tuples are evaluated to determine if they qualify the conditions specified in the user query and are returned as a result.

6.2. Prior Work

In order to implement TDB, a prototype for temporal databases, Noh [7] used XML to model the temporal relations on which the temporal queries would be executed. We are using the same XML-based representation. Whereas Noh [13] used CanStoreX as a self contained

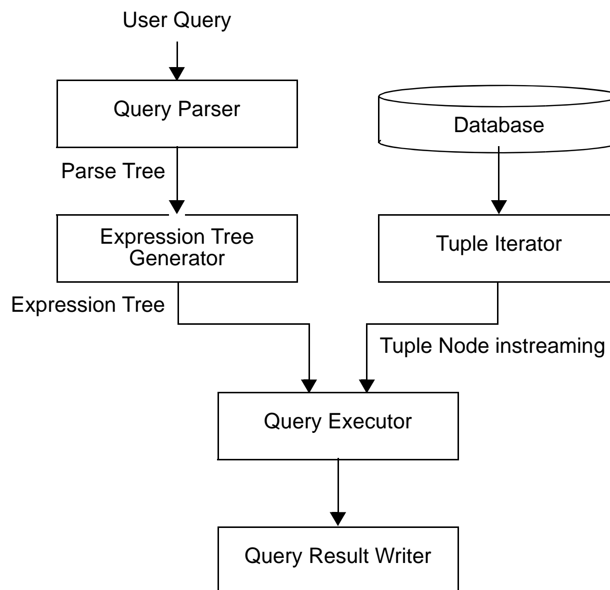


Figure 7. TDB System Architecture

storage technology we are using the centralized storage in CyDIW and CanStoreX as a utility to paginate the XML documents.

The query processing layer consisted of a ParaSQL query parser written using SJM parser API which produced an abstract syntax tree. This syntax tree is then converted to an expression tree. Both the syntax tree and the expression trees are internally represented using XML.

The storage system used in this implementation was CanStoreX, which at that time used a text based pagination scheme. DiskNode iterators developed by Ma [12] were used to in-stream the tuple nodes to the execution engine. The query execution engine then proceeded to evaluate the tuples in-streamed by the iterator and write the result.

The user interface for this system was a command button based graphical input method. The end user was provided with a set of buttons to perform various activities from selecting the database to executing the query. The system itself was a self contained stand alone module which took a page file formatted file (pff) consisting of the relations as input. The features of this system included either a graphical or XML representation of the parse and expression trees.

6.3. The need for a new prototype

During the time of this prior implementation, the storage system CanStoreX used a textual pagination scheme. In this scheme, the paginated XML file was stored as characters. In other words, the page file formatted file (pff) would be human readable if opened using a text editor. From then onwards, CanStoreX has gone through a series of changes geared toward providing a more powerful storage system. The evolution of CanStoreX has been discussed in detail in the previous chapter.

Also during this time, there was a strong need to integrate different database prototypes to work on a common platform. The Cyclone Database Implementation Workbench has been in development by our research group for the past few years. This platform provides us with a centralized storage and buffer manager, a customized implementation of DOM API called the CxDOM API to traverse our custom files in the storage and a simple but elegant command based user interface which can execute batches of commands.

In keeping with our current implementation style of developing database prototypes on top of the Cyclone Database Implementation Workbench it was decided to port the existing temporal database system to the workbench and to use the current binary version of CanStoreX.

In the next few sections, we describe some of the important components of a temporal database system along with the changes that have been made to the existing system to completely port it to work on top of the Cyclone Database Implementation Workbench.

6.4. The database catalog

Following [7], the XML-based database system catalog is the place where schema metadata, such as information about the different relations, their keys, attributes, and the physical location of the relation on the disk are stored. The catalog (Personnel.xml) for the Personnel temporal database is shown in Figure 8. The catalog file is of immense importance for a database system as it contains valuable information. The expression tree generator needs the catalog to perform semantic interpretation needed to understand which relations, functions, and operators are referenced by the query. The execution layer needs the physical location of the relation on the disk to instream the tuples to the query executor.

An important change in the catalog from the prior implementation is the inclusion of <DataSet> element to describe the nature of the dataset in the database. The Type attribute describes if the dataset consists of live operational data, a real dataset, or synthetic one. It is intended that the catalog will evolve over time to include more useful information. As our current preoccupation is with generation of synthetic data, we have considered further expansion of annotations for configuration of synthetic data. An algorithm for generation of synthetic temporal data has been implemented. The algorithm takes many characteristics of temporal data into consideration in order to generate a random data. The need for such a procedure was felt when it was noted that the prior implementation of the system did not generate a truly random data set.

It is possible to generate a relation with either desired number of tuples or a desired size of the XML data file depending on the user requirements. The data generation algorithm produces the same data set using the fixed seed value to provide a consistent data set to reproduce our experiment. The tupleOverride attribute helps us choose between the two options stated

```
<Database name="Personnel">
  <RelationList number="2">
    <Relation name="Emp">
      <DimensionInfo/>
      <AttributeInfo NumberofAttributes="3">
        <Attribute name="Name" pos="0" length="32" type="string"/>
        <Attribute name="Salary" pos="1" length="4" type="float"/>
        <Attribute name="DName" pos="2" length="32" type="string"/>
      </AttributeInfo>
      <Key><Attribute>Name</Attribute></Key>
      <StorageInfo FileName="Emp.bxml"/>
    </Relation>
    <Relation name="Dept">
      <DimensionInfo/>
      <AttributeInfo NumberofAttributes="2">
      <Attribute name="DName" pos="0" length="32" type="string"/>
      <Attribute name="Manager" pos="1" length="32" type="string"/>
      </AttributeInfo>
      <Key><Attribute>DName</Attribute></Key>
      <StorageInfo FileName="Dept.bxml"/>
    </Relation>
  </RelationList>
  <DataSet Type = “Synthetic” Method=”RandomA” Now=”16000”>
    <Relation name = “Emp”>
      <Parameters FileSize=”100” Use=”Yes”/>
      <Parameters NumOfTuples = “1000” Use="No"/ >
      <Variations>
        <Attribute name="Salary" MinVal="40000" MaxVal="140000"
                    Step = "1000" MaxVars="100"/>
        <Attribute name="DName" NumOfValues="5" MaxVars=”10”/>
      </Variations>
    </Relation>
    <Relation name = “Dept”>
      <Parameters NumOfTuples = “5” Use="Yes"/ >
      <Variations>
        <AttributeInfo name="MName" MaxVars="10"/>
      </Variations>
    </Relation>
  </DataSet>
</Database>
```

Figure 8. TDB System Catalog

above. If tupleOverride is set to true, then a fixed file size document is generated and vice versa. Also, it possible to specify individual variations in the attributes may be specified as required. In the example shown, the start value specifies the lowest value that salary can take, the end value specifies its maximum value and maxvar specifies the maximum number of variations that can occur in a single tuple for that particular attribute.

```
<Database>
  <Relation name="Emp">
    <t>
    <a name="Name">
      <v>Emp_0<p><d>[0,16000]</d></p></v>
    </a>
    <a name="Salary">
      <v>3000<p><d>[0, 200]</d></p></v>
      <v>3300<p><d>[201,400]</d></p></v>
      <v>5000<p><d>[401,600]</d></p></v>
    …
    < /a>
    <a name="DName">
      <v>Dept_0<p><d>[0, 500]</d><d>[2001, 2500]</d> …</v>
    …
    </a>
    <p><d>[0,16000]</d></p>
    </t>
    …
  </Relation>
</Database>
```

Legend:
t-> tuple
a->attribute
v->value
p->pdom
d->dunit

Figure 9. Snapshot of the randomly generated data

A snapshot of the randomly generated Emp relation is shown in Figure 9. The legend in the figure explains the meaning of the tags used in the file.

6.5. ParaSQL Query Parser

As mentioned earlier, the parser module checks the syntax of the query and generates an abstract syntax tree if the parsing is successful or returns an error if the parsing is a failure. The existing version of the parser was implemented manually using SLJM Because of its popularity and ease of use to implement parsers, JavaCC compiler-compiler tool has been adopted in our database implementation workbench. Therefore a parser for a subset of ParaSQL has now also been implemented in JavaCC. Moreover, the subset of ParaSQL that has been implemented has been expanded over the previous version. The importance of Boolean expression was discussed earlier and we felt that in addition to the already existing Boolean expression rules, it would be very useful to answer more complicated queries if we add more functionalities to the same. To elucidate this point, consider the following English language queries.

Query 1: List the names of the current employees

Query 2: List the names of the employees that have a combined working experience of greater than 10 years in ‘Toys’ or ‘Shoes’ departments.

In order to support above queries we have added support for the ‘subset’ set operation of ParaSQL and a Length function respectively. With the subset operation, it will be possible for us to find out if NOW is contained in the domain of a particular employee. If this evaluates to true, then it means that the employee is currently working in the firm. The Length function can be used to measure the time during which the employee worked in a particular department. Using these two new additions, one can easily express the queries in ParaSQL as shown below.

Query 1:

```
Select e.Name
From Emp e
Where NOW subset [[e.Name]];
```

Query 2:

```
Select e.Name
From Emp e
Where Length ( [[e.Dept=’Toys’]] union [[e.Dept=’Shoes’]]) > 10;
```

The ParaSQL parser checks the syntax and generates a parse tree. This parse tree is internally represented using XML. Representing parse trees using XML has several benefits. An XML-based parse tree is a self-contained entity that exists on its own. It is independent of the language and the environment that is used to create it at runtime environment and accessible outside of such environment. It is easy to read and can be visualized through tools readily available for XML documents. Lastly, it is easy to process it using code in DOM API that provides a much higher level interface. Such code is also easy to read and maintain. Because of the use of XML navigation is very logical. It can help reduce our reliance on linked-list based structures. An XML representation of a parse tree is shown in Figure 10.

6.6. Expression Tree Generator

The expression tree generator takes the abstract syntax tree as its input and does the semantic interpretation needed to understand which relations, functions, and operators are referenced by the query by looking up the system catalog. As in [7], in XML based expression tree,

```
<ParseTree>
  <QueryExpression>
    <SelectStatement>
      <SelectClause>
        <AttributeList>
          <AttributeTerm>
            <ObjectName>Emp</ObjectName>
            <AttributeName>Name</AttributeName>
          </AttributeTerm>
        </AttributeList>
      </SelectClause>
      <RestrictClause/>
      <FromClause>
        <ObjectList>
          <Object>
            <ObjectName>Emp</ObjectName>
            <ObjectNickName>e</ObjectNickName>
          </Object>
        </ObjectList>
      </FromClause>
      <WhereClause>
        <AtomicBooleanExpression>
        <FuncOpConstBooleanExpression>
          <Function>
            <Name>Length</Name>
            <Arg>
            <AtomicDomainExpression>
              <AttrOpValAtmDomainExpression>
              <AttributeTerm>
              <ObjectName>Emp</ObjectName>
              <AttributeName>Dept</AttributeName>
              </AttributeTerm>
                <Operation>=</Operation>
                <String>Dept_0</String>
              </AttrOpValAtmDomainExpression>
            </AtomicDomainExpression>
            </Arg>
          </Function>
          <Operation>greaterThan</Operation>
          <Number>10</Number>
        </FuncOpConstBooleanExpression>
        </AtomicBooleanExpression>
      </WhereClause>
    </SelectStatement>
  </QueryExpression>
</ParseTree>
```

Figure 10. XML Representation of a Parse Tree

the nodes representing projection, restriction and where clauses are at the same level. They do not have a parent-child relationship, but instead are siblings of each other as shown in Figure 11.

6.7. Iterator

Iterators offer a mechanism to access one object at a time, on demand, from a collection of objects. Iterators go hand in hand with streaming (instreaming and outstreaming) that is a primary paradigm for query processing in databases. In our case, they're used to iterate over tuples in relations. Iterators help hide tedious details implicit in dealing with page boundaries. This makes the implementation of consumer modules simpler that would otherwise turn into spaghetti-like code. Base iterators instream and outstream tuples from and to the disk, respectively. Relational operators, including the select statement is a complex composition of iterators. A tuple outstreamed by a query is assembled based upon instreaming from base iterators and other operators.

It has to be noted here that we're concerned with instances of an iterator that instreams data

```
<expression>
  <Iterator type="scan" relname="Emp">
    <projection isAll="false">
      <AttributeList num="1">
        <attribute relation="Emp" name="Name" attrPos="0" type="string"/>
      </AttributeList>
    </projection>
    <restrict isUniversal="true"/>
    <relationlist>
      <relation name="Emp" nickname="e"/>
    </relationlist>
    <condexp isTrue="False">
      <Function name="Length">
        <DomainExp opType="unary">
        <BinaryOp type="=">
        <attribute relation="Emp" name="Dept" attrPos="2" type="string"/>
        <const value="Dept_0" type="string"/>
        </BinaryOp>
        </DomainExp>
        <op>></op>
        <number>100</number>
      </Function>
    </condexp>
  </Iterator>
</expression>
```

Figure 11. XML Representation of Expression Tree

directly from the disk and an iterator that outstreams tuples assembled by the select statement. As our relations are stored using XML, the base iterators require XML processing. In the current version of our TDB prototype, we use the iterators for the CanStoreX system developed by Krithivasan and others. These base iterator were implementations of an interface called DOMNodeIterator. It has to be noted here that this DOMNodeIterator returns a DOMNode which is a custom implementation of DOM API called CxDOM API. The custom implementation is designed specifically with CanStoreX in mind. These base iterators traverse through the bxml (binary XML) files that reside in paginated format of our stored relations. The base iterator that we have used extensively is the ChildNodeIterator. This iterator is used to return all the children of the current node. Internally, the iterator would return the first child of the node initially followed by the right siblings of the first child until it either reaches a node with no right sibling or the user explicitly invokes the close function. This iterator helps us iterate through all the tuples in a relation. It has to be noted here that all our tuple nodes are siblings of each other and are children of <Relation> node. Thus running the ChildNodeIterator by passing the relation node to it, will instream all the tuples in our relation. Note that the ChildNodeIterator that accesses data from the disk is very high level compared to the usual iterators in classical relational databases. Whereas classical relations are stored as sequences of tuples that are themselves flat byte sequences, in our case the tuples have a complex tree-based structure. Even so our temporal database system is a client for which such details are low level.

We have implemented iterators to instream tuples from our temporal relations. These iterators are used in the query execution algorithm. One such iterator is called RelationScan. Following the methodology of iterator implementation, the basic functionalities that we have implemented are: open, close, hasNext, getNext and getRemaining. The open() method of an iterator is used to instantiate and start a new iterator. The close() method destroys the iterator and makes it void. The hasNext() returns a boolean value of either true if there are more tuples available to be fetched or false if there are no more tuples. The getNext() method increments the iterator to point to the next available tuple and fetches that tuple and finally, the getRemaining() method fetches all the available remaining tuples starting from the current location. The use of getRemaining is mainly when instreaming is done by executing commands via the GUI.

6.8. ParaSQL Query Executor

As mentioned earlier, Noh in his prior work developed a query execution engine for a temporal database system which directly interfaced with the textual implementation of CanStoreX to execute the query. Our implementation uses the binary implementation of CanStoreX. Therefore, the entire query execution engine has been revised to interface with the binary implementation of CanStoreX.

The query execution algorithm is shown in Figure 12. This algorithm takes the expression tree generated by the expression tree generator as its input. First it checks if the expression tree if there is another query in its restricted to clause. If this is the case, then the inner query is first executed and the domain of that result is stored in a temporal element. It has to be noted here that the inner query has already been parsed by our query parser and the expression tree already contains all the information needed to execute the inner query. Moreover the strategy of executing the inner query first works as currently we only allow such queries to be self contained. This means the variables used in the where and select clauses are those that have been declared in the from clause of the subquery. In other words the where and select clauses do not used variables declared in from clause of the outer query. Under such conditions subquery returns a fixed result that is independent of the variables in the outer query and it is necessary to compute it only once. After this computation is completed, when we check if there is a single relation or two relations in the from clause of the outer query. In case of a single relation a simple relation scan is required. A relation scan is an operation that sequentially traverses a single relation from beginning to end. Presence of two relations invokes a JoinIt iterator.

An example of query that involves a join is given below.

```
Select e.Name, D.DName Restricted To [[e.Dept=D.DName]]
From Emp e, Dept D
Where length([[e.Salary>60000]]) > 10;
```

It should be noted that many queries that require complex 2-way to multi-way joins in other temporal query languages become simple selections in ParaSQL that in turn lead to simple relation scans. For example, the above query in interval-based approach would involve a join between Emp and Dept relations.

Depending on the type of the query we create either a RelationScan iterator or a Join iterator. These iterators themselves use some of the base iterators to perform a simple relation scan

```
Algorithm 1 Query Execution
1: procedure Execution e                                     e:= expression tree
2:  if e has a nested query in the restricion clause then
3:   te=ProcessInnerQuery(e)                                 te:= temporal element
4:  if e has join condition then
5:   it ← Join(e)                                            it:= iterator
6:  else if e has relation scan condition then
7:   it ← RelationScan(e)
8:  end if
9:  while it.hasNext() =  True do
10   tuple=it.getNext()                                      retrieve a tuple
11:  tuple=Evaluation(e, tuple, NULL)                        evaluate the tuple
12:  tuple=Restriction(e, tuple, te)                         restrict the tuple
13:  if tuple ≠ NULL then
14:   Projection(tuple)                                      write the tuple
15:  end if
16: end while
17:end procedure
```

Figure 12. Query Execution Algorithm

or a join operation. Once an iterator has been created, we can start retrieving tuples. The relation scan iterator and the join iterator both return tuples in a stream. We evaluate these tuples and further restrict them based on the restriction clause before projecting (writing) the tuple as an output.

6.9. Query Result Writer

The query result writer module simply takes the generated tuples from the query executor and writes these tuples into an output XML file. The output of the query is in xml format, which we can easily paginate to recursively execute more queries. This pagination has currently not been undertaken. One way to do this is to implement it by using copyfile command to transform .xml output file to a .bxml file that will internally invoke pagination. A better and far more efficient way of implementing would be to outstream the output directly to the pagination algorithm. Once such an implementation has been accomplished, nested queries in ParaSQL can be implemented.

6.10. Commands for the CyDIW GUI

The Cyclone Database Implementation Workbench requires a command-based interface to all database subsystems. Next we give an overview of existing commands and then go on to list commands that we have developed for the TDB prototype.

6.11. Overview of CyDIW

We are developing several database prototypes that share common needs for storage, storage management, buffer management and GUI-based interface. The Database Integrated Workbench (CyDIW) that meets these needs has evolved over several years encapsulating considerable efforts of many individuals. This integration also minimizes configuration and organizational overlaps among multiple systems that become as independent of each other as possible. The Integrated workbench architecture is shown in Figure 6. The command based architecture follows the convention of using prefixes to differentiate between commands targeted toward different layers of the system. As evident from the figure, the integrated GUI provides us with not only a general-purpose page-based storage and buffer management services but also included are pagination and DOM API services for large XML documents.

The lowest layer in the architecture is the storage system. The storage manager component in the integrated system is responsible for two important functionalities, creation and management of storage. All storage manager commands have the prefix CyDB:>. The command *CyDB:>CreateRawStorage storageConfig.xml* is used to create a storage based on the parameters set in the XML-based storageConfig file. The storage config file specifies the physical location where the storage has to be created, the size of the storage, the size of pages, the number of buffers and the buffer replacement policy to be used. (Currently only policy that is available is LRU (least recently used.)

The top layer of the storage engine is the buffer manager layer. It is responsible for bringing pages from the storage to main memory as required. The size of each buffer is equal to the size of the pages. Buffers are used both for reading pages from the disk and for writing pages to the disk. The collection of buffers is called the buffer pool. The buffer manager manages the buffer pool. When all the buffers in buffer pool are full, the buffer manager uses buffer replacement policy to decide the victim buffer. The buffer manager also keeps track of counters for the page accesses and number of pages allocated and deallocated.

The common GUI component is a command based system. Every sub-system is assigned a prefix and used with the command to indicate the targeted subsystem for the command. The prefix for our temporal database system is *tdb:>*. Every sub-system should have a dedicated

command parser which will be called by the GUI parser when a command for that sub-system is encountered. In our case, this parser is called the TDB Command Parser.

When the GUI parser encounters a command starting with the prefix *tdb:>*, then it immediately delegates the command to the target sub-system, which in this case is TDB Command Parser, for execution.

The GUI itself has some native commands. These commands have the prefix *CyDB:>*. Some of the GUI commands that we're interested in are:

CyDB:>declare string[10] a;

This command is used to declare an array of variables which can hold our temporal data queries. Storing our query in a variable makes it easy to follow its life-cycle from parsing to execution.

CyDB:>set a[0] = <temporal database query written in ParaSQL>;

This command is used to assign a query to a variable. The design of the GUI is such it consists of two panes, one for the commands and one for output. There is a third pane in the lower end which is used to display console information. The command buttons that are available are Open, Save, Save As, Run Selected, Run All, Find/Replace and Clear on the commands pane and Save, Save As, Find and Clear on the output pane.

The GUI also displays some performance parameters like the number of page accesses, the number of pages allocated/deallocated and the hit ratio. We can use these values to compare the execution of different queries.

6.12. TDB Command Format

A suite of commands to execute the core temporal database functionalities has been developed. As mentioned earlier, our temporal database system commands have *tdb:>* as the prefix. The general command format for our temporal database commands is shown below.

tdb:> <command name> <list of arguments>;

The command is loaded into the commands pane of the GUI, selected and executed. The GUI parser delegates this command to the TDB command parser once it encounters the prefix. Once the TDB command parser gets the command, it matches the command name with a list of predefined commands and calls the associated routines.

6.13. Some Important TDB Commands

As mentioned earlier, the TDB command parser is used to parse the TDB commands and execute the appropriate routines associated with them. Here we discuss some of the important TDB commands by giving a brief description of what they do along with their command format.

6.13.1. Command for generation of synthetic temporal data

tdb:> GenerateTDBData Catalog.xml;

With configuration information for the desired data contained in XML-based database catalog, this command creates randomly generated data The data consists of relations represented in XML format.

6.13.2. Command to load a dataset

tdb:> LoadTDBData <relation_name>;

This command loads the specified relation from an xml file onto our general purpose paginated file storage system. The above command takes temporal relation name as input parameter and loads the specific relation into the common storage. *<relation_name>* parameter takes ‘Emp’ or ‘Dept’ as values. On execution of this command, the specific temporal relation data are paginated and loaded into the common storage. This command creates the bxml files in our storage and names them as specified in the catalog file.

Currently, the data is first created in standard XML format and then paginated. In future the generator can directly outstream data to the CanStoreX pagination algorithm.

6.13.3. Command to open a database

tdb:> OpenTDBDatabase <database_name>;

The OpenTDBDatabase command takes the catalog file name as its argument and caches the root pages of all the relations in that catalog. Note that the relations are files in our storage, e.g. Emp is stored as Emp.bxml. The address of the root page is obtained by internally consulting the directory for our storage. For processing the catalog of the system, DOM API is used.

6.13.4. Command to display TDB catalog

tdb:> DisplayTDBCatalog;

The DisplayTDBCatalog command opens the TDB system catalog in the system default web browser.

6.13.5. Command to parse query

tdb:> ParseQuery <variable_name>;

This command calls the ParaSQL parser routine to parse the query associated with a variable. This assumes that the database has already been opened. Otherwise it throws a parser failed exception. The command parses the given query and creates an XML-based parse tree (also called abstract syntax tree) with the name <variable_name>.parsetree.xml.

6.13.6. Command to display Parse Tree

tdb:> DisplayParseTree <variable_name> <view_type>;

DisplayParseTree command is used by the client to display the parse tree that has been created for the query in the previous step. The above commands takes variable name which holds the target query and type of view as parameters. It provides two types of tree views, XML view and graphical view. This command can be executed only after parsequery command has been executed for the target query. The *view_type* parameter can take 'xmlview' or 'graphicalview' as values. Using 'xmlview' opens the xml representation of the parsed tree in a web browser and 'graphicalview' converts the xml representation of parsed tree on the fly to the graphical view and displays in the new pop up window.

6.13.7. Command to build expression tree

tdb:>BuildExpressionTree <variable_name>;

This command calls the expression tree generator routine which builds the expression tree based on the parse tree generated in the previous command. If the parse tree has not been created already then this command will throw an exception. The above command takes variable name which holds the target query as an input parameter. The expression tree is constructed based on the parse tree and is also represented as an XML document with the name. The name <variable_name>.exptree.xml.

6.13.8. Iterator based command for query execution

Query execution in databases is stream based. Tuples from operand relations are instreamed and the tuples resulting from the query are outstreamed. Whereas instreaming is internal to the execution environment and not directly visible to a user, outstreaming is. Typically, and iterator consists of the OpenIterator, GetNextTuple (or PutNextTuple), CloseIterator functions. In addition there is a boolean HasNextTuple that remains True while there are more tuples and becomes

False when no tuples are left in the stream. Consequently we have implemented the following commands.

tdb:>OpenIterator <variable_name>;

tdb:>HasNextTuple <variable_name>;

tdb:>GetNextTuple <variable_name>;

tdb:>CloseIterator <variable_name>;

These commands take the variable name which holds the target query as input parameter. Before iterating through the result of the query, the expression tree for the query should have already been created. If an associated expression tree does not exist, then commands will report an error. The commands work as follows.

- The OpenIterator command is used to instantiate and open an iterator for the query referenced by the variable name. The OpenIterator command is used to instantiate and open an iterator for the query referenced by the variable name. Without opening the iterator, the client can not execute other functions.
- HasNextTuple command indicates whether any more tuples are available in the result of the query. GetNextTuple command is used to retrieve the next tuple of the query result. Once the tuple is retrieved, the iterator attempts to point to the next tuple in the resultant tuples associated with this query.
- GetNextTuple is intended to be executed only when HasNextTuple is True. On execution, GetNextTuple simply assembles a tuple and deposits it to the output stream.
- CloseIterator command is used to close the iterator for the target query associated with the given variable name. The buffers occupied by the iterator are released to the storage. Once the iterator is closed, it is not possible to iterate through the resultant tuples one or more at a time anymore. An iterator can be closed prematurely through the GUI. Ordinarily the iterator will be executed only when GetNextTuple becomes false. For the sake of convenience the following command is also available through the GUI.

tdb:>GetRemainingTuples <variable_name>;

GetRemainingTuples command is used by the client to retrieve all the remaining tuples starting from the iterator’s current position. After all the remaining tuples are outstreamed the boolean HasNextTuple will become False.

It is important to understand the division of labor between the functionalities of HasNextTuple and GetNextTuple. The HasNextTuple must really makes sure that the query will indeed return a tuple when GetNextTuple is executed. This means that it has to preemptively verify that the next tuple is indeed a qualified one. Internally, this can amount to quite a bit of work depending upon the query and the query language. In case of the select statement the where clause has to be verified for the candidate operand tuples. In TDB it also has to make sure that the restricted to clause does not compute to an empty temporal element. Until both of these are ensured, more tuples from the input relations will be instreamed until the required conditions for existence of the next tuples are met. HasNextTuple can go as far as assembling a tuple but that is not a good idea. In relational databases there is a small gap between where HasNextTupe ends its work and where getNextTuple resumes it. In more complex models and query languages the role of HasNextTuple is only to guarantee that there is a next tuple. The fragments of the next tuples reside in the buffers. These fragments have to be assembles according to the requirements of a model. Unlike a tuple in ordinary classical relational databases where several tuples reside in the same page, in more advanced models a tuple can occupy several buffers. In fact this is the case with TDB as well. Preparing the next tuple will require extra memory. Such memory may not be accounted for if it outside of the buffer pool which is hardly desirable.

6.13.9. Execute Query

tdb:>ExecuteTDBQuery <variable_name> <result_filename>;

This command is used to combine the above iterator steps into one by dumping all the resultant tuples into a file name specified in <result_filename>. It take the variable name which holds the target query as another input parameter. When this command is executed, an iterator for the result is automatically created and the query is executed. This result is stored in the file name specified in the command. Note that in the current implementation the output is a standard XML document and not a paginated one. This has the advantage that the XML document can be examined in a browser. Currently a browser for paginated .bxml documents is not available. On the other hand, a paginated document that is desirable if nested queries of ParaSQL were to be implemented. Paginated documents would also be more helpful if the results are large and need to be processed.

6.13.10. Close Database

tdb:>CloseTDBDatabase;

This command is used to close the TDB database system. It closes the handler to the temporal relations in our storage and releases the currently held resources using Java's native garbage collection method.

6.14. TDB Command Execution

To execute a command, the command has to be selected and the [Run Selected] button in the GUI has to be clicked. This method gives us the ability to run one or more sequential commands at a time. Sometimes it might be more appropriate to run all commands at the same time. To meet this need, we can load a batch of commands and click [Run All] button in the GUI. This would execute all the commands in the batch. Extra flexibility can be availed by suppressing execution of commands by making them remarks by adding "//" at the beginning of the line. An individual command that has been converted to a remark can also be executed by highlighting the command by carefully avoiding "//" characters. An example TDB batch is shown in Appendix A.

7. Conclusion

We have presented a temporal database implementation based on the parametric model using XML. The use of XML in our implementation has been helpful. XML has helped us do away with complicated data structures and eliminated our reliance on linked-list based data structures for parse tree, expression tree, and the database catalog. All settings and configuration become explicit and readily available without having to sift through code. Furthermore, any changes that might be made to these artifacts would not affect the overall working of the system.

The storage is drawn from the central storage facility provided by the Cyclone Database Implementation Workbench (CyDIW). The parsing has now been implemented in JavaCC making future development much easier. We have implemented a function to determines the length of a temporal element. Additional functions can be parsed and implemented in the same style. A query with local variables can be nested inside the restricted to clause. A rudimentary implementation of the subset boolean operation has been started. These additions when fully expanded will help one to harness greater power of ParaSQL for temporal databases.

We have also developed a more random temporal data generation algorithm which can be used to generate a relation with desired number of tuples or a relation size. This algorithm takes input from the XML-based catalog file and generates data based on the configuration information. After the physical representation of the datasets is revisited for efficiency, benchmarking with large scale temporal datasets can take place.

Finally, in order to conform to the requirements of CyDIW and its GUI, we have encapsulated the implementation in terms of commands that can be executed via the GUI without needing any specialized graphical elements such as buttons. As required by CyDIW, an entire experiment starting from creation or using existing storage, a synthetic temporal dataset, execution of queries, and benchmarking can be encapsulated in terms of a beach of commands. This also makes experiments entirely reproducible by anyone anywhere and anytime.

Our work provides a framework that will make the ongoing and future development of our temporal databases prototype easier in order to harness the latent promise of the parametric approach.

8. Future Work

Our implementation of the temporal database system prototype TDB is now in conformance with the requirements of Cyclone Database Implementation Workbench (DWI) and its GUI. This would make future development of the TDB prototype much easier. Here we list several directions.

8.1. Physical Representation

A more efficient support storage and consumption of temporal elements, values, tuples, and relations is needed. A temporal element such as [11,20] ∪ [21,36] ∪ [45,81] ∪ [106,202] can be efficiently represented as a sequence of integers 11, 20, 21, 36, 45, 81, 106, 202 which at 4 bytes per integer would require 32 bytes. The same idea can be applied to a value such as < [11,20] ∪ [21,36] ∪ [45,81] ∪ [106,202] “Toys” >. This type of representation would help efficient computation of expressions such as ⟦DName = “Toys”⟧. Another efficient representation of values can be in sorted order of the instants. A value such as < [21, 40] ∪ [75,86] Toys, [41,50] Shoes> could be stored as a sequence of triplets 21, 40, Toys, 41, 50, Shoes, 75, 86, Toys. The advantage of this is that it keeps the value sorted by time. This is advantageous because it would allow stream-oriented processing that is a core idea in database implementation. One would expect support for such storage strategies to be shared between CanStoreX as a general-purpose storage technology on one hand and temporal database as a specific domain application on the other. This would greatly improve performance as currently the binary pagination inflates the size of XML representation significantly.

This may seem like one is circumventing XML. But one can have virtual XML document with different storage methodologies underneath. A case in point is with middle-ware a client can see a relation, stored internally in industry standard format with pages containing tu[ples as byte sequences, as an XML document. This is on one end of the spectrum with a completely heterogeneous artifact on the other end a great deal of interpolation is need to take full advantage of XML.

In databases pinning is a very important mechanism to implement caching strategy of an algorithm. While a buffer is pinned to a page, it cannot be removed prematurely. Only a client knows when it is done using a page and must bear the responsibility of unpinning the buffer. As far as

CyDIW provides the support for pinning and unpinning as long as it is requested. This needs to be exploited by iterators. The main thing is that an iterator must do its own bookkeeping to keep track of pages that it wishes pinned and unpin them when those pages are no longer needed. After a tuple has been processed the resource needed exclusively for the tuple can be released. But tuples can be large, spanning multiple pages and one should be able to release pages even while a tuple is being processed. This would require us to visit primitive and composite expressions in ParaSQL of temporal databases to be visited carefully. Note that iterators can do pinning directly or have CyDIW do the pinning on its behalf but unpinning has to be requested explicitly by the iterators.

Some gaps between the parser and query processing that was available in [7] has not been incorporated in our implementation. This includes set operations ($\cup$, $\cap$, and $-$) on domain expressions and boolean operators (an, or, and not) on boolean expressions.

The structure of the expression tree needs to be revisited. Following [7], the restricted to, from, and where clauses are treated as siblings in the parse tree and the same relationship is carried on to the expression tree. This needs to be changed to reflect the customary representation of the expression tree to reflect its underlying algebraic structure.

To execute recursive queries, it becomes necessary to paginate the resulting xml file back to bxml format. This bxml file is directly stored in our common storage. Once this has been done, it would be possible to query this result file just as if it were another stored relation. Currently the output of a query is an ordinary XML document. This can be copied into a paginated binary bxml document by using the copyfile command available in CyDIW. This intermediate step is not necessary as the pagination can be invoked directly. Know-how for accomplish this is already available in the createfile command in CanStoreX that creates an XML document of any pre-specified size by using the well known XMark benchmark. Instead of storing the document first in XML and then paginate it, the document is paginated on the fly as fragments of the document become available. As XMark is a C program, for direct pagination JNI (Java Native Interface) is used to pass the character stream created by XMark directly to the Java-based pagination algorithm. An additional thing to keep in mind here is that the XML document being produced should conform to the chosen representation of temporal elements and temporal values discussed above if such representations are deployed. In a nutshell the format of a computed temporal relation should be

no different from the format of the stored temporal relation.

Although we have enlarged the subset of the ParaSQL language that has been implemented, there are still several much needed additions. We have implemented subqueries inside the restricted-to clause in our language, but the current implementation assumes that the inner query variables are independent from the outer query variables. This need not be the fact as the inner query can reference variables from the outer query. Subqueries can appear in many other places. Example include addition of constructs such as in (L), not-in (L), exists(L), all(L), and any(L) where L is a subquery. Our implementation is primarily targeted towards executing queries that retrieve information from the stored relations. Update operations have been implemented in distant past but the nature of our platform has simplified implementation to such an extent that a fresh implementation should be quite easy. For users visualization of temporal relations in tabular form would also be useful.

ParaSQL itself can be extended in significant and useful ways. For example, in addition to classical style inter-tuple aggregates, ParaSQL offers tremendous opportunities to tuple level aggregates. The latter would allow a construct such as e.TAvg(e.Salary) to be used seamlessly at par this attribute values such as e.Salary. (Here, “T:” in “TAvg” indicates tuple level aggregation.) In human experience, periodicity of time plays an important role. For example a year may be seen as a cycle and one may like to compile analyze and compare events year-wise. Linguistics support for such constructs would be a welcome addition.

A much needed addition to our system would be the implementation of an algebraic query optimizer. A theoretical framework for algebraic query optimization is available in [27]. The next steps would be development of indexes and plan generation. This would lead to a full-fledged implementation of a query optimizer. A query statement can be executed in many different ways, such as full table scans, index scans, nested loops, and hash joins. The query optimizer determines the most efficient way to execute a query statement after considering many factors related to the objects referenced and the conditions specified in the query. The output from the optimizer is a plan that describes an optimum method of execution. This determination is an important step in the processing of any query statement and can greatly affect execution time.

Finally, a comparison between our parametric model based ParaSQL and the interval model based ISQL based on usability and efficiency can also be done. This would probably help settle

the debate for determining which data model is most appropriate for temporal databases.

In this thesis we have only considered the parametric model for historical data to capture the evolution of objects in the real world. The parametric approach has been extended to other forms of time in databases, spatial databases, spatiotemporal databases, and belief databases of which multilevel security is a special case. It would be fruitful to extend the implement to such cases as well.

References

[1] R.Bourret, XML and Databases, April 2005

[2] Bohlen, M.H., Bussato, R., Jensen, C.S., Point versus Interval based temporal data models. In:Proceedings of the fourteenth international conference on data engineering.

[3] W3C, 2007. Document Object Model Website:http://www.w3.org/DOM

[4] Toman, D., 1997. Pointe based temporal extension of temporal SQL. In: Proceedings of the 5th International Conference on Deductive and Object Oriented Databases.

[5] Tansel, A.U et. al. 1993.A glossary of temporal database concepts. In: Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings.

[6] Snodgrass, R.T., 1987. The temporal query language TQuel. ACM transactions on Database Systems.

[7] Noh, S.-Y., Gadia, S.K., Ma, S.,2007: An XML based methodology for parametric temporal database model implmentation. Journal of System and Software.

[8] Noh, S.-Y., Gadia, S.K., 2006. A comparison of two approaches to utilizing XML in parametric databases for temporal data. Information & Software Technology 48(9): 807-819.

[9] Noh, S.-Y., Gadia, S.K., 2005b. Efficient self-join algorithm in interval based temporal data models. Tech. Rep. 05-22, Department of Computer Science, Iowa State University.

[10] Noh, S.-Y., Gadia, S.K., 2005a: An XML-based framework for temporal database implementation. In: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, Burlington, Vermont, USA.

[11] Noh, S.-Y., Gadia, S.K.,2004. A parametric framework for implementing spatiotemporal databases. In: Proceedings of the 12th International Conference on Computer Science and its Applications, San Diego, California, USA.

[12] Ma, S., 2004: Implementation of a Canonical Native Storage for XML, Master's Thesis, Department of Computer Science, Iowa State University, Ames, Iowa

[13] Noh, S.-Y., 2006. An XML-based implementation of the parametric model for ad-hoc query of temporal and spatiotemporal data, PhD Thesis,Department of Computer Science, Iowa State University, Ames, Iowa .

[14] Patanroi, D., 2005. Binary page implementation of a canonical native storage for XML,Master's Thesis, Department of Computer Science, Iowa State University, Ames, Iowa

[15] Krithivasan, S., 2007. Implementation of a XQuery engine for large documents in CanstoreX, Master's Thesis, Department of Computer Science, Iowa State University, Ames, Iowa.

[16] Kumar, N., 2008. Implementation of the NC-94 hybrid storage prototype on a binary version of CanStoreX, Master's Thesis, Department of Computer Science, Iowa State University, Ames, Iowa.

[17] Narayanan, V., 2009: A workbench for advanced database implementation and benchmarking, Master's Thesis, Department of Computer Science, Iowa State University, Ames, Iowa.

[18] Jensen, C.S., et. al 1993. The TSQL benchmark. In: Proceedings of the International Workshop on an Infrastructure for Temporal Databases, Arlington, Texas, USA.

[19] Gadia, S.K., Chopra, V., Tim, U.S., 1993. An sql-like seamless query language for spatio-temporal data. In: Proceedings of the International Workshop on an Infrastructure for Temporal Databases, Arlington, Texas, USA.

[20] Gadia, S.K.,Vaishnav, J.H., 1985. A query language for a homogeneous temporal database. In: Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, Oregon, USA.

[21] Gadia, S.K., Chopra, V., 1993. A relational model and SQL-like query language for spatial databases. In: Advanced Database Systems. Lecture Notes in Computer Science vol 759, Springer.

[22] Gadia, S.K., 1988. A homogeneous relational model and query languages for temporal databases. ACM transactions on Database Systems 13(4).

[23] Gadia, S.K., Nair, S.S., 1993. Temporal Databases, A prelude to paramteric data, In: Temporal Databases, Theory, Design and Implementation, Benjamin/Cummings.

[24] Noh, S.-Y., Gadia, S.K., 2008. Benchmarking temporal database models with interval-based and temporal element-based timestamping. Journal of Systems and Software 81(11): 1931-1943

[25] Noh, S.-Y., Gadia, S.K., 2007. User-Friendly Extendibility of Two Temporal Data Models to Spatiotemporal Data Models. MUE 2007: 241-246.

[26] Noh, S.-Y., Gadia, S.K., 2005.The Usability of XML in Parametric Databases for Temporal Data. IKE 2005: 113-119

[27] Gadia, S.K., Nair, S.S., 1998. Algebraic Identities and Query Optimization in a Parametric Model for Relational Temporal Databases. IEEE Trans. Knowl. Data Eng. 10(5): 793-807

[28] Gadia, S.K., Nair, S.S., 1992. Incomplete Information in Relational Temporal Databases. VLDB 1992: 395-406.

[29] Gadia, S.K., 1988. The Role of Temporal Elements in Temporal Databases. IEEE Data Eng. Bull. 11(4): 19-25

[30] Gadia, S.K., 1986. Weak Temporal Relations. PODS 1986: 70-77

[31] Jensen, C.S., Snodgrass, R.T., Temporal Database. Encyclopedia of Database Systems 2009: 2957-2960.

[32] Jensen, C.S., Snodgrass, R.T., Temporal Data Models. Encyclopedia of Database Systems 2009: 2952-2957.

[33] Jensen, C.S., Snodgrass, R.T., Temporal Element. Encyclopedia of Database Systems 2009: 2966.

[34] Jensen, C.S., Snodgrass, R.T., Temporal Query Languages. Encyclopedia of Database Systems 2009: 3009-3012.

[35] Jensen, C.S., Snodgrass, R.T, Time Interval. Encyclopedia of Database Systems 2009: 3112-3113.

[36] Jensen, C.S., Snodgrass, R.T, Weak Equivalence. Encyclopedia of Database Systems 2009: 3455-3456.

[37] Chomicki, J., Toman, D., Abstract Versus Concrete Temporal Query Languages. Encyclo-

pedia of Database Systems 2009:1-6.33

[38] Alda Lopes Gançarski, Pedro Rangel Henriques: IXQuery: Interactive and Information Retrieval Xquery. ELPUB 2005

[39] Abdullah Uz Tansel, Erkan Tin: Expressive Power of Temporal Relational Query Languages and Temporal Completeness. Temporal Databases, Dagstuhl 1997: 129-149

[40] Apache Xindice, xml.apache.org/xindice/

[41] eXist-Open Souce Native XML Database, http://exist.sourceforge.net/xquery.html

[42] Mirella M. Moro, Vassilis J. Tsotras: Bi-Temporal Indexing. Encyclopedia of Database Systems 2009:239-243

[43] Christian S. Jensen, Richard T. Snodgrass: Fully Temporal Relation. Encyclopedia of Database Systems 2009:1193

[44] Alex Dekhtyar, Robert B. Ross, V. S. Subrahmanian: Probabilistic temporal databases, I: algebra. ACM Trans. Database Syst. (TODS) 26(1):41-95 (2001)

[45] Gadia, S.K., A canonical native storage architecture for XML.

[46] Worldwide Web Consortium. www.w3.org

APPENDIX A

This section shows a batch file of commands for the Temporal Database system. The batch is completely self-contained from creation of storage, to creation and loading of a synthetic dataset, creation of parse and expression trees for queries, their visualization, step-by-step as well as complete execution of queries, and creation of a log of performance. The batch represents an experiment that can be reproduced by anyone, anywhere, and at anytime. The batch is a high level encapsulation of the system behavior.

```
/* Title : TDB Demo File
   Author : Kartic Ramesh
   Date   : Mar 25, 2010 */

// Config for the storage is in StorageConfig.xml ;
// Create storage from scratch, format with fixed size pages, start buffer manager ;

// CyDB:>createrawstorage StorageConfig.xml;
// CyDB:>formatStorage 16;
// CyDB:>startBufferManager 100;

// Use storage that has alredy been created  ;
CyDB:>usestorage StorageConfig.xml;

// Show contents of directory ;
CyDB:>showdirectory;

// Create synthetic dataset with configuration in TDB_Catalog.xml ;
// tdb:>GenerateTDBData TDB_Catalog.xml;

// Load the database;
// tdb:>LoadTDBData Emp;
// tdb:>LoadTDBData Dept;

//Open the TDB Database;
//The connection is opened and the root of the database is cached ;
tdb:>OpenTDBDatabase;

//Display the catalog information for the database ;
CyDB:>displayXML C:\Users\ramesh\workspace\TDB_Workspace\TDB_Catalog.xml;

//Declare a variable;
 CyDB>declare string[10] a;

// Set a query to the variable declared above ;
CyDB>set a[0] := tdb:>
Select *
From Emp E;

// Parse the query, build expression tree, and display them;
```

```
tdb:>ParseQuery a[0];
tdb:>BuildExpressionTree a[0];
CyDB:>DisplayXML C:\Users\ramesh\workspace\TDB_Workspace\a[0].parsetree.xml;
CyDB:>DisplayXML C:\Users\ramesh\workspace\TDB_Workspace\a[0].exptree.xml;

// Compute tuple at-a-time ;
tdb:>OpenIterator a[0];
tdb:>HasNextTuple a[0];
tdb:>GetNextTuple a[0];
tdb:>GetRemainingTuples a[0];
tdb:>CloseIterator a[0];

// Execute the query using the expression tree';
// tdb:>ExecuteTDBQuery a[0] queryresult.xml log>> <Stat1> MyLogTDB.xml;

// Another query;
CyDB>set a[1] := tdb:>
Select Emp.Name
From Emp E
Where Length([[Emp.Dept="Dept_0"]])>100;

tdb:>ParseQuery a[1];
tdb:>BuildExpressionTree a[1];

tdb:>OpenIterator a[1];
tdb:>HasNextTuple a[1];
tdb:>GetNextTuple a[1];
tdb:>GetRemainingTuples a[1];
tdb:>CloseIterator a[1];

tdb:>ExecuteTDBQuery a[1] queryresult.xml log>> <Stat1> MyLogTDB.xml;

// Close the TDB Database ;
tdb:> CloseTDBDatabase;
```

ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my adviser Dr. Shashi Gadia whose wide knowledge and logical analysis of the problem have been of great value to me. This thesis would not have been possible but for his encouragement and personal guidance.

I would also like to thank Dr. Wallapak Tavanapong and Dr. Ying Cai for agreeing to be my committee members.

During this work I have collaborated with many colleagues in the Department of Computer Science for whom I have great regard. I wish to personally thank my lab mates Xinyuan Zhao and Niranjan Kumar for their guidance in helping me understand the prior work that was done in this field. Also, I would like to thank Valliappan Narayanan for his support.

I would like to express my loving thanks to my wife Sujatha for her constant support and my parents Lakshmi and Ramesh Rajan for their fantastic encouragement during this endeavor.

Finally, I would also like to thank Iowa State University for their partial financial support during this program.