

Schemes to make ARIES and XML work in harmony

A Writing Project

Presented to

The Faculty Of Department Of Computer Science

San Jose State University

In Partial Fulfillment of the Requirement for the Degree

Master Of Science

By

Thien An Nguyen

February 2005

© February 2005

Thien An Nguyen

Thien_an9@yahoo.com

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Christopher Pollett

Dr. Melody Moh

Dr. Tsau Young Lin

APPROVED FOR THE UNIVERSITY

Abstract

One of the most important issues for a database product is its availability. When disaster happens, the Recovery Component of a database product relies on its log records to recover from the crash. During the time of recovery, the database will be unavailable to user applications as it has to process all of the log records between the last check point and the point where system crash. Thus, the more log records it has to process, the longer the unavailability to user application.

A simple and efficient recovery method for native XML databases has recently been derived from ARIES - Algorithms for Recovery and Isolation Exploiting Semantics. Natix uses techniques called Subsidiary Logging and Annihilator Undo for writing log records. These techniques reduce a number of log records needed for system recovery based on the tree structure of XML document. Thus, it gives better performance and availability for an XML database.

In this project, we built a toy XML database from scratch to study the differences in recovery algorithm. We defined the syntax for our DDL/DML and built a parser for the syntax. This DDL/DML is the communication bridge between user application and our database. We put queries and commands in an input file. We used NetBeans and JUnit to run test cases (scenarios) for our database. We put control and environment variables in a properties file so users can experiment with our database. For example, user can specify which recovery method to be used (ARIES or NATIX) and which test scenario, the size of a page, or the number of pages in buffer pool.

Table of Contents

1	INTRODUCTION.....	8
2	OVERVIEW OF ARIES AND NATIX.....	13
	2.1 Overview of ARIES.....	13
	2.2 Overview of Natix.....	14
3	DESIGN ARCHITECTURE.....	16
	3.1 System Architecture.....	16
	3.2 Query Engine.....	17
	3.3 Execution Engine.....	17
	3.4 Buffer Manager.....	17
	3.5 Tree Storage Manager.....	19
	3.5.1 Logical Model.....	20
	3.5.2 Physical Model.....	20
	3.5.3. Page Split.....	21
	3.6 Storage Manager.....	23
	3.7 Transaction Manager.....	23
	3.8 Log Manager.....	24
	3.9 Recovery Manager.....	24
4	IMPLEMENTATION.....	25
	4.1 The role of dbsystem.properties File.....	25
	4.2 Query Engine.....	28
	4.2.1 Query Language.....	28
	4.2.2 Commands.....	29
	4.2.3 Parser.....	29
	4.3 Execution Engine.....	31
	4.4 Buffer Manager.....	31
	4.4.1 Implementation for a Data Page with ARIES logging – ThPage.....	34
	4.4.2 Implementation for a Data Page with NATIX logging – ThPageInterpreter..	36
	4.5 Tree Storage Manager.....	37
	4.6 Storage Manager.....	38
	4.7 Transaction Manager.....	41
	4.8 Log Manager.....	41
	4.7 Recovery Manager.....	42
5	EXPERIMENTS.....	45
	5.1 Overall Test System.....	45

5.2 Tests.....	47
5.3 The differences between Natix and ARIES log records.....	53
5.4 Performance Experiments.....	55
5.4.1 Case Study with Large Structure, Small Data.....	55
5.4.2 Case Study with Small Structure, Large Data.....	57
6 CONCLUSION.....	61
7 BIBLIOGRAPHY.....	63
8 APPENDIX A.....	64
9 APPENDIX B – Query and Commands Syntax.....	68
10 APPENDIX C – JUnit Testcase.....	70
11 APPENDIX D – Scenarios/Input Files, Data Files.....	72
12 APPENDIX E – Output Results.....	74
13 APPENDIX F – Source Code.....	80

List of Figures

Figure 1 - Phases during Restart Recovery.....	8
Figure 2 - Compensation Log Record.....	8
Figure 3 - Subsidiary Logging.....	9
Figure 4 - Annihilator Undo.....	10
Figure 5 – Overall Architecture.....	14
Figure 6 – Buffer Manager.....	15
Figure 7 – Relationship between ThPage and ThPageinterpreter.....	16
Figure 8 – Logical Tree Associate with Its XML Document.....	17
Figure 9 – XML tree structured in memory.....	19
Figure 10a – A Page Header.....	19
Figure 10b – Mapping in a ThStructMapPage.....	20
Figure 11a – A ThStructMapPage before Split.....	21
Figure 11b – A ThStructMapPage after Split.....	21
Figure 12 – Physical Version of a Data Page.....	22
Figure 13 – UML Diagrams for Overall Architecture.....	24
Figure 14 – Different Version of the same Data Page.....	36
Figure 15 – A Data Page on disk.....	36

1. Introduction

As eXtensible Markup Language (XML) is gaining in popularity as a way of sharing information over the Internet, the need for a database that stores XML document natively is increasing.

Relational and hierarchal databases converted to store XML, often use a separate layer that is built on top of their existing storage structure. For native XML databases, most research papers study the area of storage management, specifically logical and physical storage but rely on existing relational database recovery algorithms, for example, Algorithms for Recovery and Isolation Exploiting Semantics (ARIES). Based on a recent proposal paper for native XML database from NATIX, in this project, we built a toy XML database with concentration on recovery management sub-system to experiment with ARIES and NATIX's. In this section, we introduce the ideas behind ARIES and NATIX's recovery management.

XML documents are semi-structured. They are stored as a tree with labeled edges and leaves. With such structures, mapping it to a relational database might results in either a large number of columns with null values (which wastes space) or a large number of tables (which is inefficient for searching and retrieving data). Storing XML natively offers better performance in terms of retrieving and accessing XML document, because native XML database understands the structure of XML documents and so preserves their data hierarchy and meaning.

Consider the XML document as in Example 1 below. In a relational database, the document might be stored in three tables: Billing, Customers, and Parts. Retrieving the document would require joins across these tables. In a native XML database, the entire document might be stored in a single place on the disk. Thus it requires a single lookup and a read to retrieve the data.

Example 1:

```
<Billing BNumber="12345">
  <Customer Id="0021">
    <Name>Surgical Optics ltd</Name>
    <Street>947 Mission St.</Street>
    <City>San Jose</City>
    <State>CA</State>
    <Code>95054</Code>
  </Customer>

  <Part PNumber="000-675-2311">
    <Description>Objective Lens System </Description>
    <Price>119.95</Price>
    <Quantity>5</Quantity>
  </Part>

  <Part PNumber="000-232-1000">
    <Description>Ocular Lens</Description>
    <Price>111.00</Price>
    <Quantity>9</Quantity>
  </Part>
</Billing>
```

In general, there are two broad categories for recoveries: recover to previous state (known as rollback operations) and recover from crash (restart recovery operation). In either case, the recovery manager goes through three phases: read and analyze the log records, redo the changes for the committed transactions, and undo the changes for the uncommitted transactions. Figure 1 was modified based on the original graphic from [Mohan92] to show the three phases performs by ARIES and Natix. According to ARIES, all updates of transactions are logged, including those performed during rollback. Through appropriate chaining of the log records written during a rollback to those written during forward progress, a bounded amount of logging is ensured during the rollback even in the case of repeated failures during restart or of nested rollbacks [Mohan92].

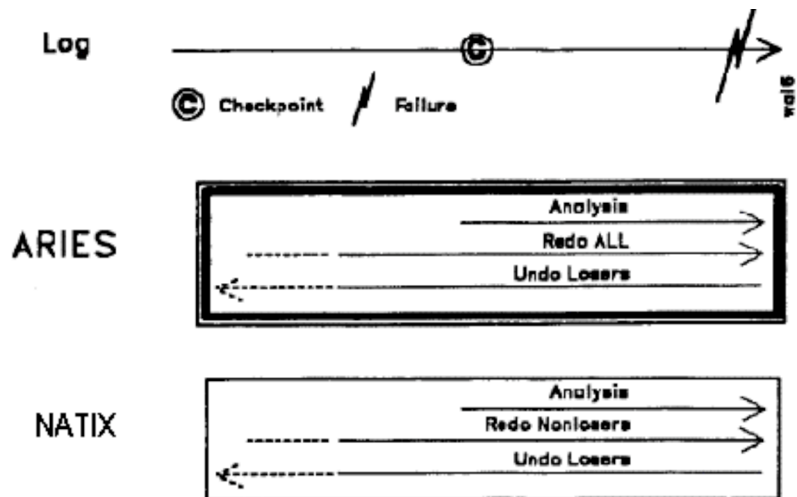
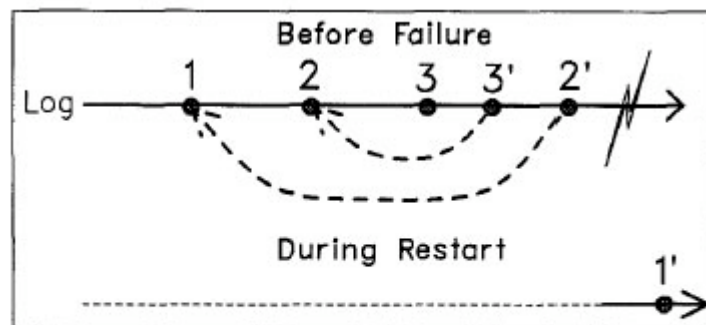


Figure 1 - Phases during Restart Recovery

The log records that are written during rollback operations during forward phase are called compensation log records (CLR). Figure 2 is an example of how CLR are written. During the forward phase, Operation 2 and 3 are undone by a rollback operation lead to CLR 3' and 2' be written. During the restart recovery phase, when CLR 2' is analyzed, the process will skip Log Record 2 and 3 to perform operation in Log Record 1.



1' is the Compensation Log Record for 1
1' points to the predecessor, if any, of 1

Figure 2 – Compensation Log Record [Mohan92]

Log records are logged using write-ahead logging (WAL). WAL protocol says that an update operation cannot be completed unless all log records for the operation has been written to disk. With this feature, the buffer manager is allowed to flush dirty page(s) by a transaction onto disk before the transaction commit (steal). Also, page(s) modified by a transaction does not have to be flushed onto disk when the transaction commits (no-force).

NATIX is a native XML database management system has been recently introduces the idea behind its recovery algorithm is to use subsidiary logging and a so-called annihilator undo. Suppose a record is updated multiple times by the same transaction, causing a sub-tree to be added. The log size would be reduced if we log only the composite update operation. That is, instead of writing a log record for each update by the same transaction into the same sub-tree, we can just accumulate the log information into one big record. Thus, we can save some header storage and limit the amount of log records the recovery has to process. This is called subsidiary logging. The following examples and figures from NATIX's paper are presented to verify this idea. Consider operations performed by one transaction on the same sub-tree in a record as in Figure 3 below. Instead of writing five log records for Operations 1-5, NATIX writes one log record that describes the changes for the sub-tree R1 after the 5th step.

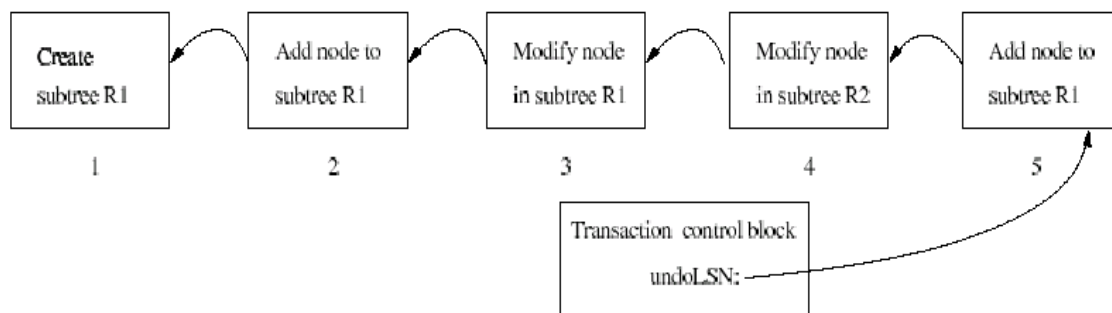


Figure 3 - Subsidiary Logging [Kan 02]

Undo operations that imply the undo of other operations following them in the log are called annihilators. For example, the update operation to a record that has been created by the same transaction does not need to be undone when the transaction is aborted. A delete to the record would be faster and simpler than an undo of updates. Consider the update operations in Figure 4. During a recovery, we can perform undo of 5, 4, 3, 2, and 1 as the reverse of the forward phase updates. On the other hand, if we undo modification 4 and deletion 1, we would receive the same result as performed in the previous case, but only with fewer operations which is an improvement in performance in terms of few operations to performs and increase concurrency.

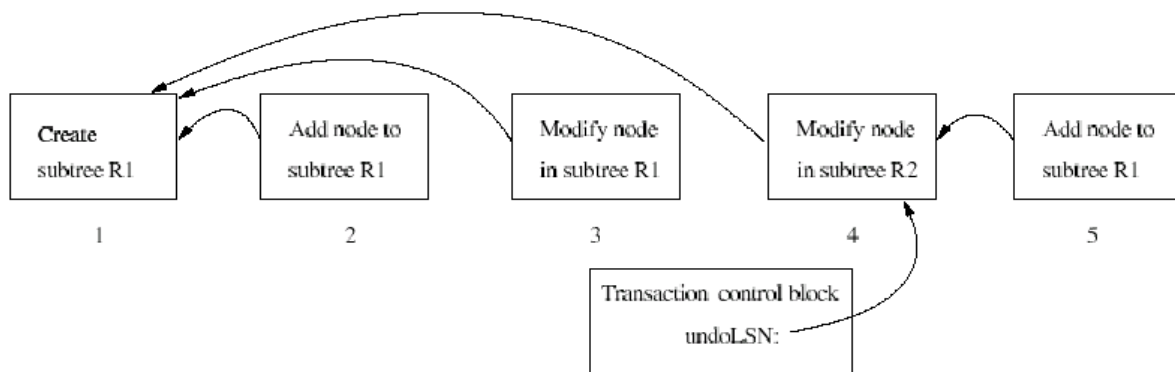


Figure 4 – Annihilator Undo [Kan02]

We now discuss the organization of this report: Section 2 gives an overview of ARIES and Natix’s recovery algorithms. Section 3 describes in detail our design architecture. Section 4 gives details of our implementations. Section 5 gives detail of our deployment and test scenarios. Section 6 is our conclusion and references.

2 Overview of ARIES and Natix's Recovery Management

In the section above, we provided the ideas behind ARIES and Natix. In this section, we introduce more fundamental concepts about ARIES, and then we give an overview of Natix.

2.1. ARIES – A Recovery Management Algorithm

ARIES has the following properties:

- Flexible buffer management using steal/no-force policy.
- All update operations must be logged using log records.
- Use write-ahead logging protocol (an update operation cannot be completed unless all log records for the operation has been written to disk).
- Undo operations are recorded using compensation log records.

During a forward phase of the system, ARIES writes log records for all updates made by transactions. As discussed in Example 3, ARIES writes log records for Operations 1 – 5.

During a restart of the system, if ARIES detects that there was a crash or an un-expected system shut down, it performs the following actions:

- **Analysis Phase:**
 - In this phase, the system blocks all access into objects in the system (the databases, the documents, etc...).
 - The recovery manager finds information about the last system's checkpoint¹. It uses this information to determine the set of dirty pages and transactions at the checkpoint to determine the starting point for the system.

¹ If the last image copy is not available, it will try to get try to get the one before that.

- Next, the recovery manager scans through log records from the point of system crash back to the last system checkpoint.
- It builds a list of actions to be done for the dirty pages made by all transactions for the redo phase.
- It uses information about the transactions to determine which transaction (the un-committed transactions) will be rolls back during undo phase.
- **Redo Phase:** Before we can continue describing the redo phase, we want to give reader definition of a consistency point. A consistency point is a point when there is no update to the database. Therefore, the system is not available for access. The recovery manager gets the last image copy (taken by the last system checkpoint) of the database and use redo log records to bring the dirty pages to the consistency point. This phase also known as repeat history.
- **Undo Phase:** During this phase, the recovery manager uses the undo log records (analyzed during the analysis phase) to undo the changes made by un-committed transactions.

2.2. Natix Recovery Management Overview

The Natix recovery management has similar properties to ARIES properties:

- Use steal/no-force policy to manage page caching (same as ARIES).
- Use log records to describe changes for update operations.
- Instead of writing log records for every update operation, it writes a log record describe the composite changes for a sub-tree. Please refer to Figure 3 for more details.
- Undo operations also be recorded using compensation log records.

The Natix, however, has different techniques for logging during the forward phase and different approach for processing the log records during the redo phase:

During a forward phase of the system, Natix also writes log records for all updates made by the same transaction, and to the same page but it writes them locally in its log buffer. Consider the Example 3 above, Natix writes log records for Operations 1 – 5 in its log buffer. If the page is full or if the transaction is ready to commit, the local log buffer is resized to reflex only the composite change of Operations 1 – 5. That means it writes a log record to say that sub-tree R1 has been added into the document.

During restart recovery of the system, it performs the following actions:

- **Analysis Phase:** (Same as ARIES)
- **Redo Phase:** The recovery manager uses redo log records to bring the dirty pages of *only* the committed transactions to a consistency point in time.
- **Undo Phase:** (Same as ARIES)

3 Design Architecture

This section describes the design of our toy XML database management system. We carefully go through each component and its role in our system.

3.1 System Architecture.

Figure 5 is our general database management (DBMS) system. It includes Query Engine (responsible for submitting queries from user application and interpreting the result return from

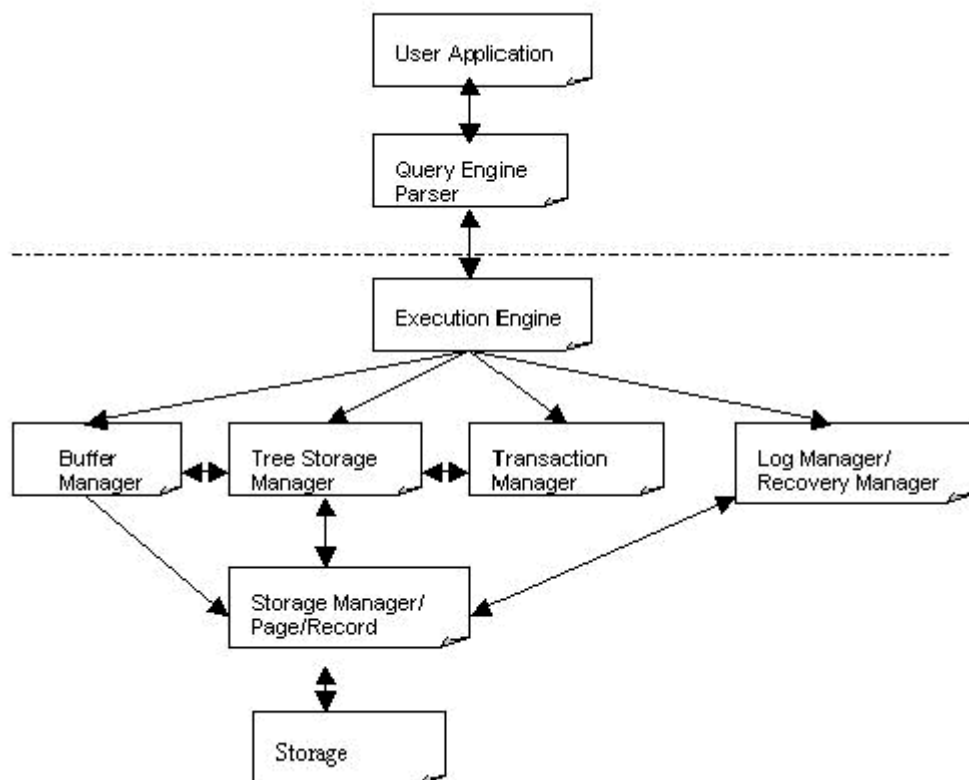


Figure 5 – Overall Architecture

the database), Parser (checking query for grammar and syntax errors), Execution Engine (direction execution of queries), Buffer Manager (caching/swapping pages in/out of memory),

Tree Storage Manager (for mapping tree structured into records), Transaction Manager (handle transactions), Storage Manager (storing and retrieving data pages to and from physical storage), Log Manager (reading/storing log records) and Recovery Manager (handle transaction rollbacks and crash recovery).

3.2 Query Engine

For an enterprise DBMS, Query Engine includes query compiler, query parser, query preprocessor, and query optimizer. Since this is not the scope of this project, we only handle query parser. It parses an input file for queries and commands into an internal form called a plan. A plan is a structure that contains statements before and after parsing. It also keeps the highest return code from the parsing.

3.3 Execution Engine

This component directs the execution for each query/command and return the result back to user application.

3.4 Buffer Manager

The buffer manager controls the number of pages in the buffer pool. It allocates and de-allocates pages in the buffer pool. As mentioned in the introduction section, the WAL algorithm gives buffer manager more flexibility toward page caching (the Steal/No-force policy). For page replacement, we implement the simplified version of Least Recently Use (LRU) algorithm.

Therefore, a page will not be written onto disk unless it is full and a page won't be de-allocated if it is recently updated. Figure 6 bellow is our design for buffer management.

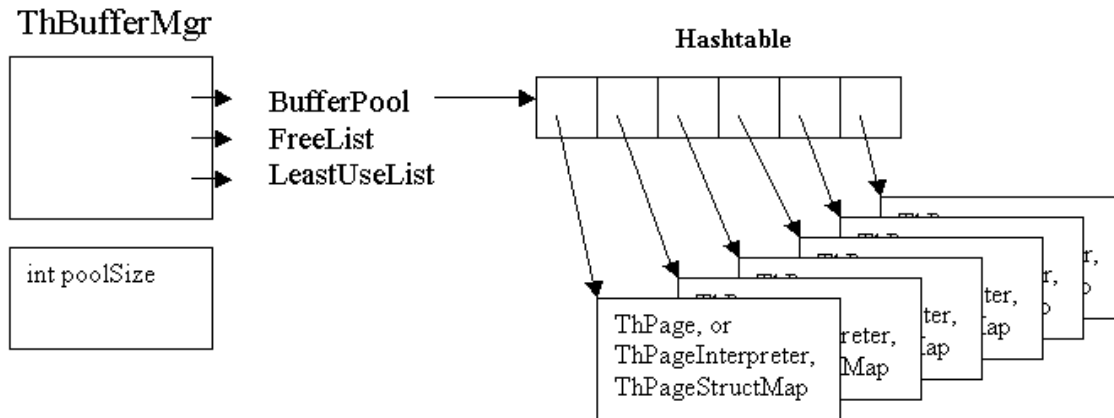


Figure 6 – Buffer Manager

There are three types of pages that are managed by buffer manager: ThPage, ThPageInterpreter, and ThStructMapPage. ThPage and ThPageInterpreter are data pages. Their relationships are shown in Figure 7. ThStructMapPage contains a tree like document structure. More details about pages are in tree storage manager and storage manager section below. Buffer manager responsibilities are page allocation, page deallocation, and page swapping (in and out of the buffer pool). It maintains a list of pointers to free pages in buffer pool and keep track of least recently used page numbers in buffer pool. Buffer pool is a pool of logical pages. The number of data pages and struct-map pages in buffer pool are pre-defined in the properties file. Buffer manager will maintains buffer pool size through out the program's life.

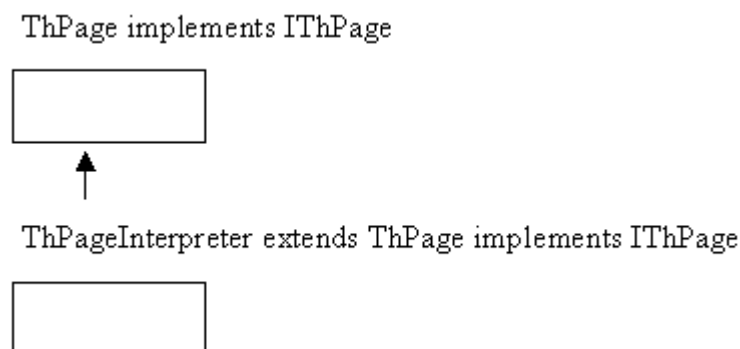


Figure 7 – The Relationship between ThPage and ThPageInterpreter.

3.5 Tree Storage Manager

Tree Storage Manager controls the mapping between an XML document and tree structured in memory. For simplicity, our XML document contains only 3 types of tags: `<G></G>`, `<L></L>`, and `<XML></XML>`. Please see the following example as our possible XML document.

Example 2:

```
<XML ID="00000001">  
  <G name="CS297">  
    <G id="123-45-6789">  
      <L>Jane Eyre</L>  
    </G>  
    <G id="234-56-7890">  
      <L>Irene Hugh</L>  
      <L>3.0</L>  
    </G>  
  </G>  
  ....  
</XML>
```

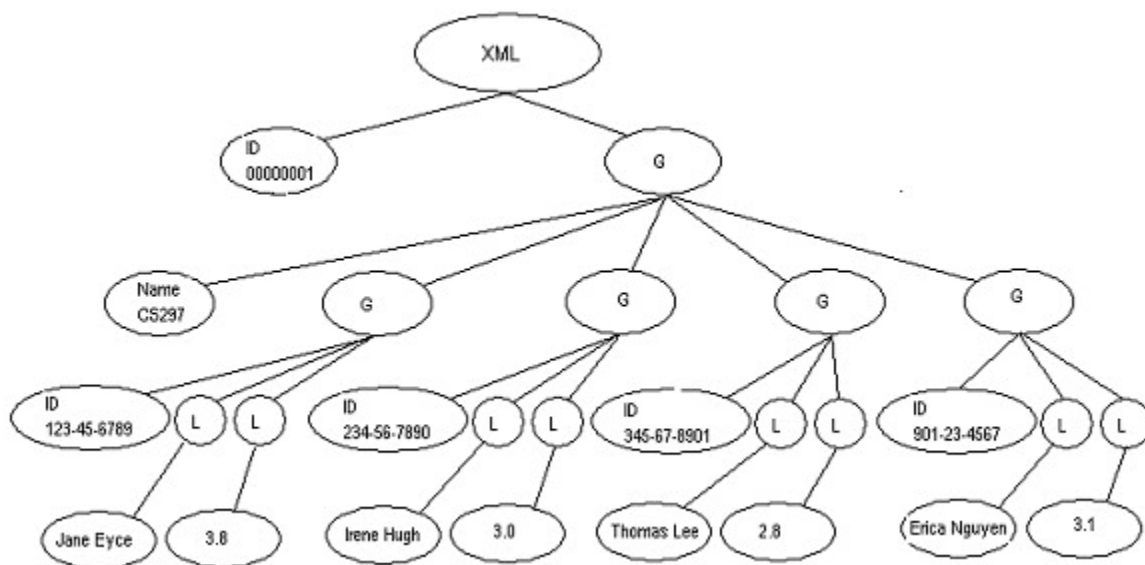


Figure 8 – Logical Tree Associate with Its XML Document

3.5.1 Logical Model

A logical XML document can be interpreted as a tree. An example of such tree is shown in Figure 8. Internally, we built the tree of node Ids (NIDs). Each node contains a node id, the page number where the data node stored (data page number), a pointer to the child node and a pointer to the next node. We map XML element tree nodes one-to-one onto a logical data model. Attribute and leave nodes are mapped to child nodes. Figure 9 below shows a NID content.

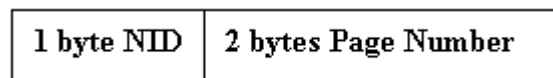


Figure 9 – a Node ID

3.5.2 Physical Model

Figure 10b shows how a sub-tree can be stored on a page. The ThStructMapPage objects are used to store mapping of the document tree structures. Like the ThPage and ThPageInterpreter, when this page is written onto the disk, it will be written as the contiguous bytes of data. The toHex() method of this object will perform this task. To create an object of the ThStructMapPage from a byte array of data, one of the constructors of this class will perform this task. Figure 10a shows the data in a page header.

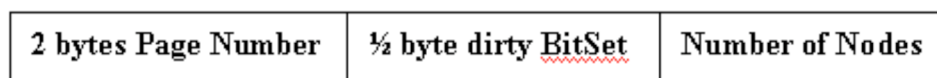


Figure 10a – A Page Header

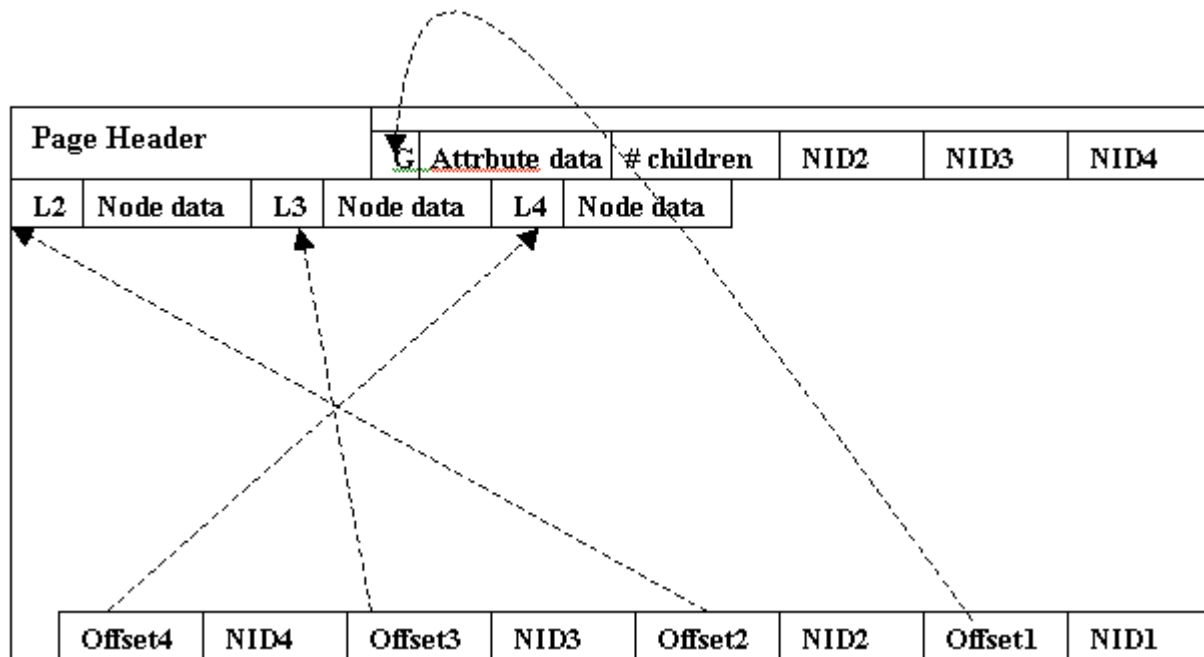


Figure 10b – Mapping in a ThStructMapPage

3.5.3. Page Split

Consider the tree structure as in Figure 11a. Suppose we add a node L13 into the sub-tree G7 causes the ThStructMapPage Page1 overflow. We split the page by performing the following actions:

- Allocate Page2 and add this page number into this XML document's page set ².
- Remove sub-tree G7 from Page1.
- Add sub-tree G7 into Page2.
- Create a proxy node ³ (P1) that points to G7 in Page2.
- Add P1 into Page1 as child of G4.

The result of Page1 splitting is shown in Figure 11b.

² Page Set: the set of page numbers that the XML document tree spans over.

³ Proxy Node: a tree node that contains a pointer to a node on another page.

Page1

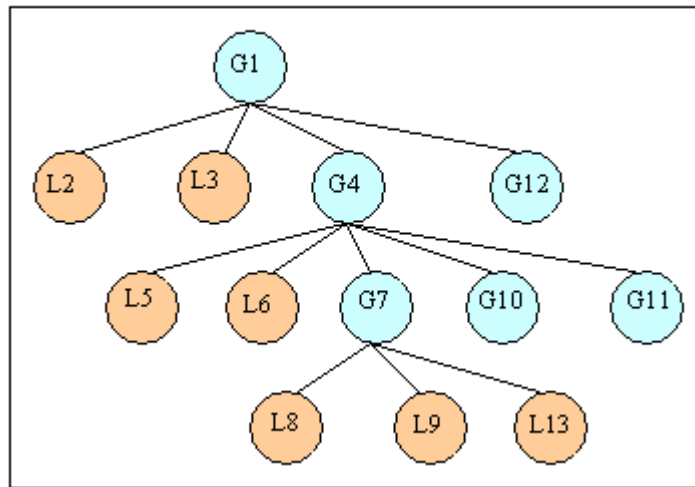


Figure 11a – StructMapPage before Split

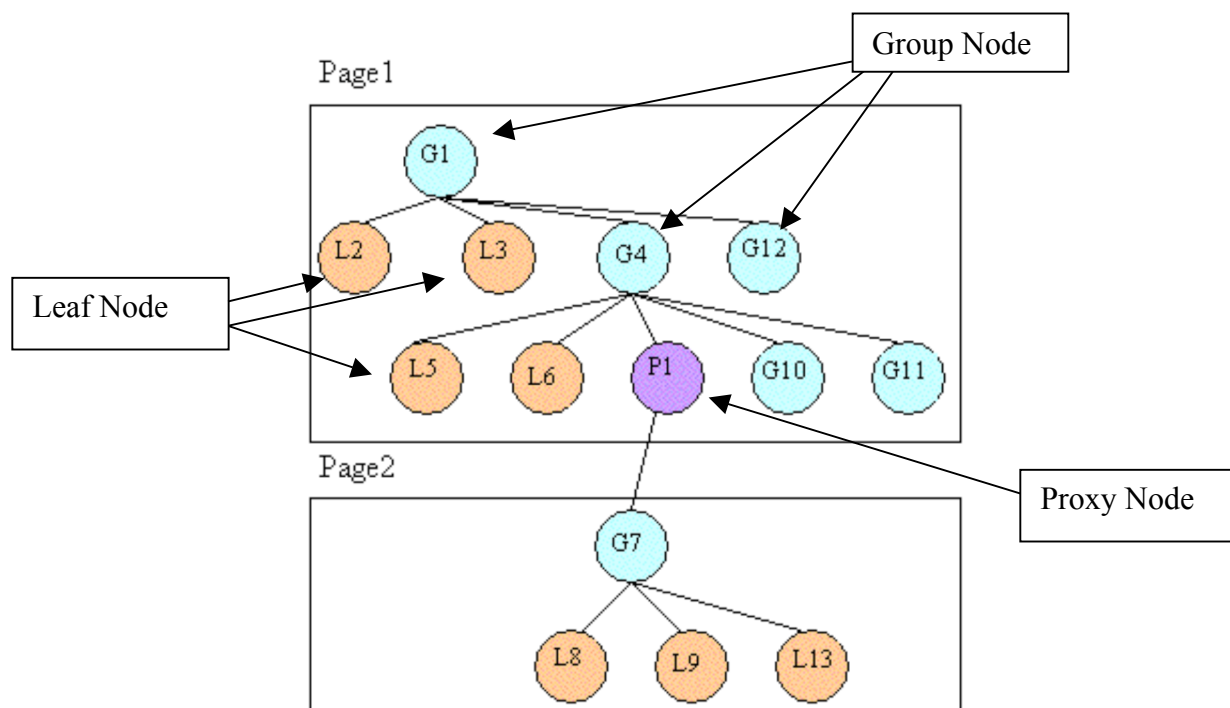


Figure 11b – StructMapPage after Split

3.6 Storage Manager

The Storage Manager manages the way data blocks are read/write from/onto disk efficiently. In this project, we designed two kinds of data pages: the ThPage's and the ThPageInterpreter's. A ThPage contains an ArrayList of data nodes. A ThPageInterpreter inherits all of the capabilities of a ThPage but manages its own log records before public them to log manager as described in Section 1. Figure below shows a physical version of a data page:

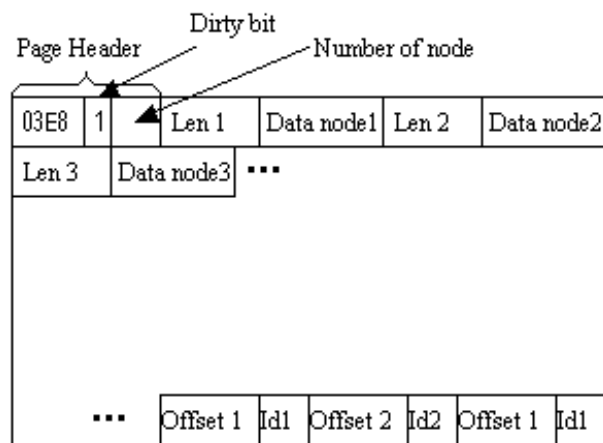


Figure 12 – Physical Version of a Data Page

3.7 Transaction Manager

This component contains a transaction table of transactions and its current state (committed or not committed). Each transaction also has a list of Log Serial Numbers (LSNs) accumulated by each update of the same transaction.

3.8 Log Manager

The log manager reads and writes log records. Both ARIES's and NATIX's log manager implements write-ahead logging, which makes sure that log records are written to physical storage before an update operation can be completed.

3.9 Recovery Manager

Please refer to Section 1 and 2 for the roles of recovery management.

4. Implementation

In this section, we explain in great detail the implementation of each component. The following is UML diagram for our overall system architecture:

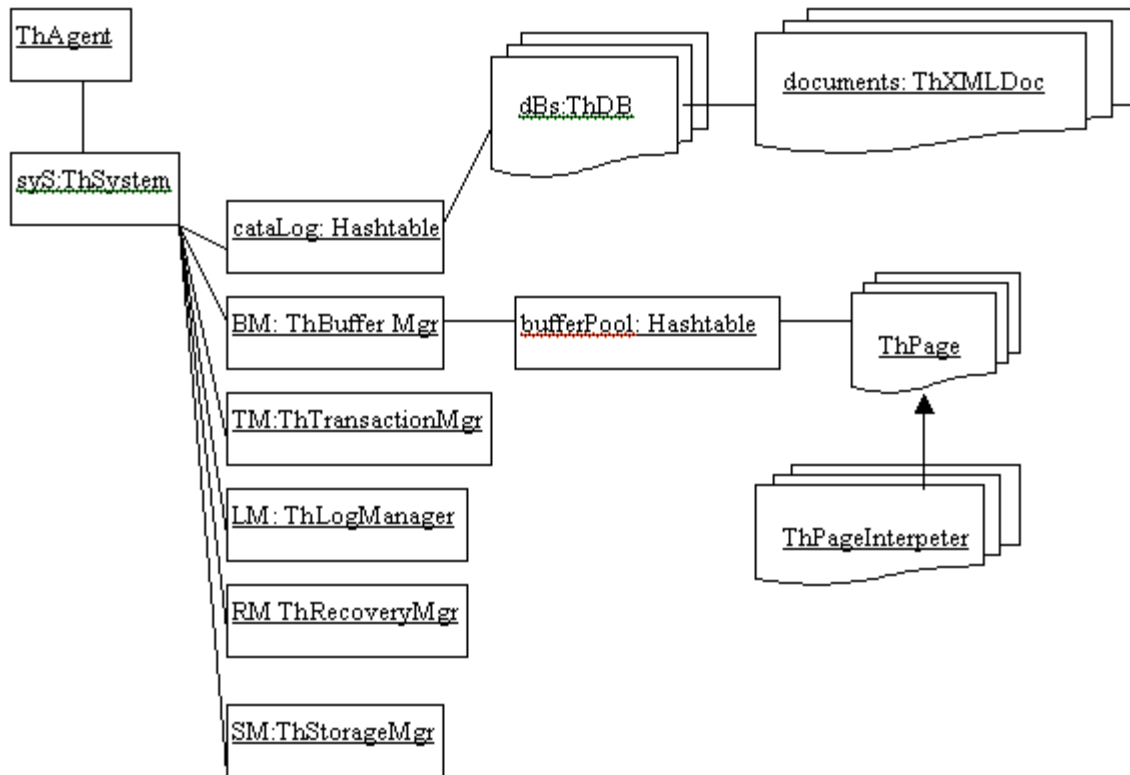


Figure 13 – UML Diagrams for Overall Architecture

4.1 The role of dbsystem.properties File

The dbsystem.properties file contains the environment that initializes our program. We are going to explain the meaning of each entry in the Listing 1 below:

Listing 1: dbsystem.properties

```
1  unittest = c://an//cs297//xmldb//unit
2  storageDirectory = c://an//cs297//xmldb//src//sm//data//
3  systemDirectory = c://an//cs297//xmldb//src//sm//system//
```

```

4  archiveDirectory = c://an//cs297//xmldb//src//sm//archive//
5  logRecords = c://an//cs297//xmldb//src//sm//logs//logrecords.txt

6  testFile = c://an//cs297//xmldb//test11.txt
7  verifyFile = c://an//cs297//xmldb//unit//verify//verify11.txt
8  outputFile = c://an//cs297//xmldb//src//sm//output//output.txt
9  resultFile = c://an//cs297//xmldb//unit//verify//result.txt
10 IO = NO
11 pageSize = 1000
12 poolSize = 4
13 numberOfStrucPage = 2
14 startPageNum = 1000
15 pageLSN = 555
16 recoverMethode = ARIES
17 IsolationLevel = UR
18 SpaceMapPage = OFF

19 SYSTEM_CRASH = -111
20 SYS_DOWN = -100
21 DB_EXISTED = -110
22 DB_NOT_FOUND = -111
23 DOC_EXISTED = -112
24 DOC_NOT_FOUND = -113
25 OPEN_FILE_ERROR = -114
26 UPDATE_LENGTH_MISMATCH = -115

27 UNKNOWN_COMMAND = -200
28 TOKEN_cOUNT_ERROR = -201

29 MISS_MATCH_LEAVE = -300
30 MISS_MATCH_GROUP = -301
31 MISS_MATCH_XML = -302
32 UNKNOWN_TAG = -303
33 UNKNOWN_PATH = -304
34 PATH_NOT_FOUND = -305
35 PARENTID_NOT_FOUND = -306

36 EC0 = 000
37 EC1 = 001
38 EC2 = 002
39 EC3 = 003

```

Lines 1, 2, 3, 4, and 5 are directories and file names for our program.

Line 6 is the test file name. This line can be change before executing the program to pick up the new test scenario. If reader refers back to our Figure 5 (Overall Architecture), we consider user application the test file. Line 8 is the program output file. This line is valid only if IO (line 10) specified NO.

Line 7 is a verify file name. A verify file contains the expected data from the program output. This line is meaningful only if IO is NO. Thus, content of a verify file is used to verify the correctness of the output of the program. User needs to change the verifyFile name based on the testFile name. If not, program might be failed. Line 9 is a file contains the result of verifying output. If verify succeeded, the result.txt will be empty as nothing need to be investigated. If verify failed, result.txt will contain the output file name and the line the failing line from the output file as follows:

*==>Can't find the following line in output file: c://an//cs297/xmlldb/src//sm//output//output.txt
Archive DBIDBI*

Line 10 determines direction of output data: if IO is NO, output data from the program will be appended into an output.txt file. If IO is YES, program will direct it's output to standard I/O (System.out) console window.

Lines 11, 12, and 13 are buffer management related parameters. Line 11 is the size of a page in buffer. Line 12 is the number of all pages in buffer pool. Line 13 is the number of struct map page. Thus, the number of data page in buffer is the subtraction of line 13 from line 12. For debugging purpose, lines 14 and 15 allow user to specify the starting point of page number and LSN number.

Line 16 let user control the recovery method that can be applied to the execution of the program. There are options that user can play with: ARIES, or NATIX. The rest of the lines are error code that used to diagnose user and system errors.

4.2 Query Engine

Before go into the parser implementation, we are going to introduce reader our query language's syntax and grammar. We also briefly describe commands that we use for simulation purpose.

4.2.1 Query Language

We define syntax and grammar for our query language. It composes of two components: Data Definition Language (DDL) and Data Manipulation Language (DML). They enable users to create and manipulate data in a database. For the CREATE XMLDOC statement, at run time, wills triggers an image copy of the database that the document resigns in. Here are examples of our possible DDL and DML statements:

Example 3: an example of DDL statement

```
T1 CREATE DATABASE DB1DB;  
T1 CREATE XMLDOC XML1DOC IN DB1DB;
```

Example 4: an example of DML statement

```
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>Jane Eyre</L>",  
                                                "<L>3.8</L>");  
T1 COMMIT;  
  
T2 UPDATE XML1DOC SET VALUE("<L>392 Lulu Ahh Dr., San Jose CA 95123</L>")  
      WHERE PATH("/G/G[1]/L[3]");
```

The first column is a transaction id. For simulation purpose, each statement, command has to have a transaction id at the beginning of the statement. This way, we can control the environment we want to test with. For Example 3, our transaction id is T1. For further description of our DDL/DML language, please see our Appendix A.

4.2.2 Commands

For simulation purposes, we also put commands in the input file (the file that contains DDL/DML statements). In the Example 5 below, the statement ARCHIVE will create an image copy of database DB1DB. The statement printTree will print XML1DOC document into the output.txt file or on the console window. If the value of IO specified in dbsystem.properties is ON, the result of printTree statement will be flushed into an output file. If IO is OFF, the result of printTree will be seen on the console window of NetBeans. Here is example 5:

Example 5:

```
T1 ARCHIVE DB1DB;  
T2 printTree(XML1DOC);  
T1 CRASH;
```

4.2.3 Parser

We implemented the parser in the class ThParser. There are two static methods in this class: errorCheckDML and scanForStatements. The following is the ThParser prototype:

```
public class ThParser  
{  
    public static ThPlan errorCheckDML(Properties prop, ThSystem sys);  
  
    private static ArrayList scanForStatements(String fileName, ThSystem sys);  
}
```

In the input file that contains DDL/DML statements and commands, the statements and commands are ended by semicolon. A statement can span over several lines. Several statements can be written on one line. The method `scanForStatement()` read the input file line by line. It uses `StringBuffer` to store a statement. The return of this method is an `ArrayList` of statements. Here is the code for `scanForStatement()` method:

```
StringBuffer cmd = new StringBuffer();

while((inLine = in.readLine())!=null)
{
    if (inLine.indexOf("/") >= 0) // this is a comment
        ; // do nothing
    else if (inLine.indexOf(';') < 0)
        cmd.append(inLine);
    else //one or more ';'
    {
        // found on the same line
        int idx = inLine.indexOf(';');
        while (idx > 0)
        {
            //ignore if ';' is at the beginning
            cmd.append(inLine.substring(0,idx)); //of the stmt
            cmdList.add(cmd); //append the first stmt
            cmd = new StringBuffer();

            inLine = new String(inLine.substring(idx+1, inLine.length()));
            idx = inLine.indexOf(';');
        }
    }
} //end while
```

The second method, the `errorCheckDML()` will loop through the `ArrayList` of statements returned by `scanForStatement()`. For each statement, it will check for the correctness of the syntax and grammar. Next, it creates the `ThStatement` object. `ThStatement` object contains a DDL/DML statement or a command, the error message if there is an error, and the highest error

code found during the `errorCheckDML()` process the statements. Please refer to Appendix F for detail of this method as it is fairly robust and easy to understand.

4.3 Execution Engine

This component was implemented in the `ThAgent` and `ThSystem` classes. The `ThAgent` is supposed to represent a unit of execution. It determines the path for each statement to be executed. A `ThAgent` instance contains a reference to an instance of the class `ThSystem`. The `ThSystem` class contains static variables and pointers to other components such as buffer manager, transaction manager, storage manger, log manager, recovery manager, ... and a catalog table. In a relational database, the system catalog is used to store information about the types, lengths of attributes, and access paths available. In this project, the catalog table is implemented as a Hashtable of databases (instances of `ThDB`) as shown in Figure 11.

4.4 Buffer Management

The implementation for the buffer manager is done in the class `ThBufferManager`. Below is our buffer manager's prototype:

```
public class ThBufferMgr
{
    public IThPage allocatePage(ThSystem sys);
    public IThPage allocateStrucPage(ThSystem sys);
    public String flush(IThPage page, ThSystem sys);
    public int flushAllPage(ThSystem sys);
    public int freeAllPages(ThSystem sys);
    public IThPage getPage(String key, ThSystem sys);
    public IThPage getPage(int datlength, ThSystem sys);
    public IThPage loadPage(String pagenum, ThSystem sys);
    public int publicLogs(ThSystem sys);
}
```

For page allocation, the buffer manager first checks to see if there is any free page in its free list (freeList). If there is, it returns that page. If not, it finds the least recently used page (the first element in the leastUse), flushes that page onto disk and returns the new allocated page. The following code segment handles allocation:

```
public IThPage allocatePage(ThSystem sys)
{
    if (freeList.size() < 1)
    {
        String firstKey = (String) leastUse.remove(0);

        //remove the page from buffer
        IThPage page = (IThPage)bufferPool.remove(firstKey);
        flush(page, sys); //will come back. right now null
        page = null;

        if (sys.isNatix())
            page = new ThPageInterpreter(pSize, pSn);
        else
            page = new ThPage(pSize, pSn);

        bufferPool.put(Integer.toString(pSn), page);
        leastUse.add(Integer.toString(pSn));
        pSn++;

        return page;
    }

    //remove the 1st key in freelist
    String key = (String) freeList.remove(0);
    leastUse.add(key);

    return (IThPage)bufferPool.get(key);
}
```

To fetch a page with a given page number, the buffer manager first checks to see if the page is still in the buffer pool. If it is, the page is returned to the caller. If not, buffer manager calls the storage manager to load the page from disk. Here is the code that handles getPage():

```
public IThPage getPage(String key, ThSystem sys)
{
    if (bufferPool.containsKey(key))
        return (IThPage)bufferPool.get(key);

    if (freeList.size() < 1)
    { //write the least use page (block) to disk.
        String firstKey = (String) leastUse.remove(0);

        //remove the page from buffer
        IThPage page = (IThPage)bufferPool.remove(firstKey);
        flush(page, sys);
        page = null;

        //load page from disk.
        page = loadPage(key, sys);
        if (page != null)
        {
            bufferPool.put(key, page);
            leastUse.add(key);
            return page;
        }
    }
    else
    {
        freeList.remove(0);
        IThPage page = loadPage(key, sys);
        bufferPool.put(key, page);
        leastUse.add(key);
        return page;
    }

    return null;
}
```

4.4.1 Implementation for a Data Page with ARIES logging – ThPage

The ThPage class is used to instantiate a data page if we are testing for ARIES. Before a data node is added into a page, the page will be checked to see if there is enough free space to store the new node. If there is enough space, an insert log record will be created and flushed on stable storage (WAL protocol) before the node is inserted into the page. The following code fragment is how the page's insert function is called to insert data into a page:

```
//1. compute the length needed for this data to be on data page
int datalength = ThPage.computeGRecLen(attr1, attr2);
int id = ThSystem.getId(); //unique logical id

//2. allocate page to store data
IThPage page = sys.getBM().getPage(datalength, sys);

//3. store data in the page
String msg = "UINSERTGROUP";
ArrayList data = new ArrayList();
data.add(attr1);
data.add(attr2);
page.insert(msg, txid, id, path, data, xmlname,
            sys.getDB(xmlname), true, sys, natixlog, islog);
page.updateFreeSpace(datalength);
```

Below is the implementation for inserting a data node into a data page:

```
public int insert(String msg, String txid, int id, String path, ArrayList value, String xmlname,
                 String dbname, boolean isgroup, ThSystem sys, boolean natixlog, boolean islog)
{
...
    lsn = createInsertLogRecord(msg, txid, id, path,
                               value, xmlname, dbname, sys);
...

    ThNode node = new ThNode(id, isgroup, value);
    nodeS.add(node); ...
}
```

The following is the implementation of how a page is materialized into a block of data in hexadecimal (HEX):

```
public char[] toHex()
{
...
    //1. page header
    //store 2 bytes page number, pos = 0
    pos += ThUtilities.numberToHex(arr, pNum, pos);

    //store 1 byte dirty bit, pos = 4
    pos += ThUtilities.BitsetToChar(arr, inFo, pos);

    //store 2 bytes number of nodes
    int nnodes = nodeS.size();
    pos += ThUtilities.numberToHex(arr, nnodes, pos);

    //2. id blocks
...
    //write all the node headers
    for (int i=0;i<nnodes;i++)
    {
        node = (ThNode) nodeS.get(i);
        //store node id
        id = node.getId();
        attr = node.getData();
        info = node.getInfoByte();

        backpos -= 4;
        ThUtilities.numberToHex(arr, id, backpos); // 2 bytes for id
        backpos -= 1;
        ThUtilities.BitsetToChar(arr, info, backpos);
        backpos -= 4;

        //store offset to data 2 bytes
        ThUtilities.numberToHex(arr, pos, backpos);

        //store node data
        data = (String) attr.get(0);
        pos += ThUtilities.numberToHex(arr, data.length()*2, pos);
        pos += ThUtilities.dataToHex(arr, data, pos);
        if (info.get(7))
        {
            data = (String) attr.get(1);
```

```

        pos += ThUtilities.numberToHex(arr, data.length()*2, pos);
        pos += ThUtilities.dataToHex(arr, data, pos);
    }
}

return arr;
}

```

4.4.2 Implementation for a Data Page with NATIX logging – ThPageInterpreter

The way data stored on a ThPageInterpreter page is exactly the same as the way data stored on the ThPage. The only difference between the ThPageInterpreter and ThPage is the way log records are written. The following is pseudo-code for the implementation of creating a log record on the ThInterpreter class:

```

public String createInsertLogRecord(String rectype, String txid, int id, String path,
    ArrayList values, String xmlname, String dbname, ThSystem sys, boolean natixlog)
{
    if (this is a leave node &&
        inserting into the same subtree)
    {
        //get the last lsn of the previous transaction
        lsn = getLSN(txid);
        if (lsn != null)
        {
            //get the previous log record out and add this leave into it
            ThLogRecord log = getLog(lsn);
            if (log != null)
            {
                log.addLeave((String) values.get(0));
                return lsn;
            }
        }
    }

    //If all of the above condition failed
    public LogsToLM(sys);

    //Create new log record
    lsn = sys.getLSN();
}

```

```

        ThLogRecord log = new ThLogRecord(rectype, lsn,
            Integer.toString(getPageNum()),
            txid, Integer.toString(id), path, values,
            xmlname, dbname);

        localLogs.add(log);
        addUndoLSN(txid, lsn);
        previousTX = txid;

    return lsn;
}

```

4.5 Tree Storage Manager

As described in Section 3.5, the Tree Storage Manager maps the XML document structure onto a tree structure in memory. This tree structure is stored in ThStructMapPage. Since the tree structure can be spanned over the pages, we keep track of the page numbers (ThStructMapPage) that used to store the tree. These page numbers are stored on a Stack called pageSet in the ThXMLDoc. When an insert into a page that cause the page overflow, the page is split as below:

```

public int splitPage(IThPage page, ThGroup subtree, int id, ThSystem sys)
{
    IThPage rpage = (sys.getBM()).allocateStrucPage(sys);
    pageSet.push(Integer.toString(rpage.getPageNum()));

    page.removeSubTree(subtree);
    rpage.addSubTree(subtree);

    IThNode rprox = new ThProxyNode(subtree.getId(), rpage.getPageNum());
    page.add(rprox);
    page.updateFreeSpace(ThStructMapPage.computePRecLen());

    return rpage.getPageNum();
}

```

4.6 Storage Manager

The implementation for storage manager is handled in class ThStorageMgr. It reads/writes data blocks from/onto disk. Figure 15 is an example of how the content of a page stored on disk.

Below is the code for writing a page onto disk:

```
public String flushPage(IThPage page)
{
    int size = page.getSize();
    String pagenum = Integer.toString(page.getPageNum());
    String physicalID = storageDir+pagenum+".page";

    try
    {
        File file = new File(physicalID);

        if (file.exists())
            file.delete();

        DataOutputStream outStream = new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream(physicalID)));
        char[] hex = page.toHex();
        String str = new String(hex);
        outStream.writeBytes(str);
        outStream.close();

        //also give a viewable verstion of this page
        printPage(size, physicalID, hex);
    }
    catch(Exception e)
    {
        e.toString();
        return null;
    }

    return physicalID;
}
```

We use the file name that contains the page as the page's physical identification (PID). Figure 14 is a snapshot of how page's physical ids are used to represent pages on disk. There are two versions of the page 1002: the 1002.page and the 1002.pageformatted. For debugging purposes, the content in both of the versions are the same but the presentations are different. The file 1002.pageformatted is viewable by human eyes as HEX characters, 1002.page isn't. On the other hand, 1002.page is the version used in the system (by buffer manager, storage manager,...), the other is not.

To load a page from disk into memory, storage manager will fetch the block of (page size) bytes and use one of the page's constructor to build the page from the byte array (byte[]) . Here is the implementation to bring a page into memory:

```
public IThPage loadPage(String pagenum, int size, ThSystem sys)
{
    String physicalID = storageDir+pagenum+".page";
    byte[] inhex = new byte[size];
    try
    {
        DataInputStream inStream = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream(physicalID)));

        int len = inStream.read(inhex);

        if (sys.isNatix())
            return (IThPage) new ThPageInterpreter(len, inhex);

        return (IThPage) new ThPage(len, inhex);
    }
    catch(Exception e)
    {
        e.printStackTrace();
        return null;
    }
}
```

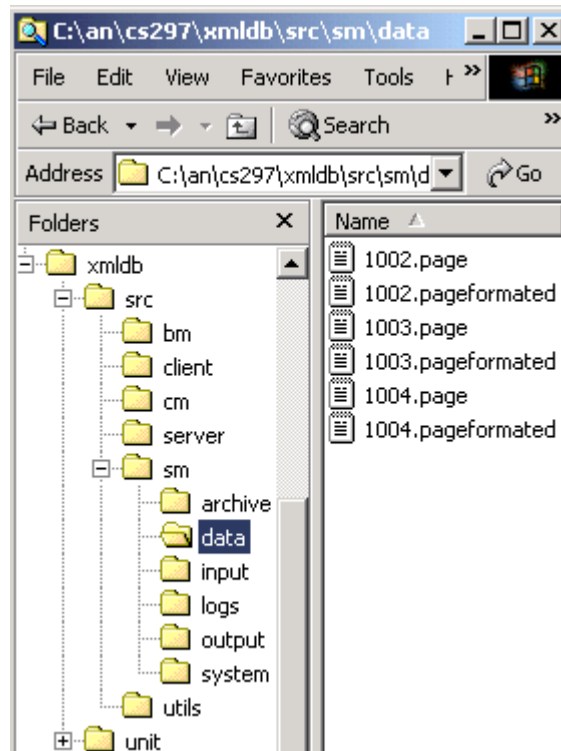


Figure 14 – Different Version of the same Data Page

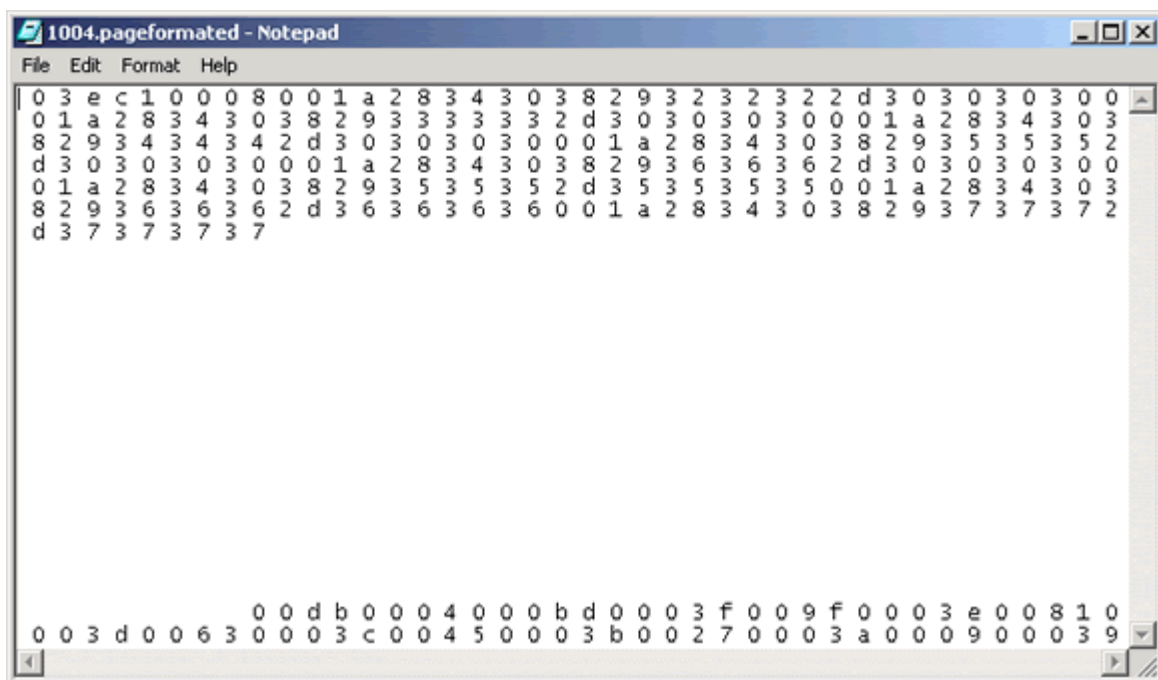


Figure 15 – A Data Page on disk

4.7 Transaction Manager

The transaction manager contains a Hashtable of transactions. A transaction contains its id, a bit indicates its state (whether the transaction is committed or not), and an ArrayList of LSNs. The following is the implementation for a transaction:

```
public class ThTransaction
{
    String iD; //Transaction ID
    boolean committed = false; //transaction current state
    ArrayList lsnRecs; /*chain of log record LSNs of
                                this transaction */
}
```

4.8 Log Manager

The log manager reads and writes log records. Both ARIES and NATIX log managers implement write-ahead logging, which makes sure that log records written to physical storage.

We store log records on the same file: logrecords.txt. Each line is a log record. The following is our pseudo-code implementation for retrieving log records on disk:

```
// loop through all of log records from the forward direction
while ((inline = inStream.readLine())!=null)
{
    tk = new StringTokenizer(inline);
    logtype = tk.nextToken();
    lsn = tk.nextToken();

    // loop through the list of LSNs needed
    while (i < size)
    {
        tmsn = (String)lsnlist.get(i);

        // if the lsn from the log record is match with the
        // lsn that we are looking for,
    }
}
```

```

    if(tmlsn.equals(lsn))
    {

        // if this is a CLR,
        if (logtype.equals("CLR"))
        {
            i++;
            clrln = tk.nextToken();
            tmlsn = (String)lsnlist.get(i);

            //skip to the log record that CLR is pointing to
            while (i < size && !tmlsn.equals(clrln))
            {
                i++;
                tmlsn = (String)lsnlist.get(i);
            }
        }
        else
        {
            logrec = createLogRecord(tk, sys, logtype, lsn);
            logs.add(0, logrec);
            i++;
        }
    } // end while < size
}
}
return logs;

```

4.9 Recovery Manager

As described in Section 1 and 2, the restart recovery goes through three phases. The following is our implementation for the restart recovery:

```

public int restartRecovery(ThSystem sys)
{
    ArrayList redologs = new ArrayList();
    ArrayList undologs = new ArrayList();

    //analyze logs
    analyzeLogs(sys, redologs, undologs);
}

```

```

//Redo for committed transactions
redoS(redologs, sys);

//Undo for un-committed transactions
undoS(undologs, sys);

return 0;
}

```

Here is the detail implementation for the analysis phase:

```

private void analyzeLogs(ThSystem sys, ArrayList redologs, ArrayList undologs)
{
...
    ArrayList redolsns = new ArrayList(),
        undolsns = new ArrayList();
    sys.out("Analyzing Logs.....");

    for (Enumeration e=TM.keys(); e.hasMoreElements();)
    {
... // Get redo and undo LSNs from for all transaction in transaction table
        tx.getLSNsForRedoS(redolsns);
        tx.getLSNsForUndoS(undolsns);
    }

    redologs.addAll(sys.getLM().getLogRecords(redolsns, sys));
    undologs.addAll(sys.getLM().getLogRecords(undolsns, sys));

    //sort the log records
    java.util.Collections.sort(redologs);
    java.util.Collections.sort(undologs);
}

```

Below is the detail implementation for the redo phase:

```

private void redoS(ArrayList logrecs, ThSystem sys)
{
... // Go through each log record in the redo list
    int size = logrecs.size();
    for (int i=size-1; i>=0; i--)

```

```

{
    log = (ThLogRecord)logrecs.get(i);
    String xmlname = ((ThLogRecord)logrecs.get(i)).getXmlName();
    String dbname = sys.getDB(xmlname);
    ThDB db = (ThDB)sys.cataLog.get(dbname);
    xmldoc = (ThXMLDoc)db.docNames.get(xmlname);

    /*re-apply for committed txs */
    xmldoc.redo(log, sys);
}
}

```

The following the detail implementation for the undo phase:

```

public void undoS(ArrayList logrecs, ThSystem sys)
{
... // Go through each log record in the undo list
    int size = logrecs.size();
    for (int i=size-1; i>=0; i--)
    {
        log = (ThLogRecord)logrecs.get(i);
        String xmlname = ((ThLogRecord)logrecs.get(i)).getXmlName();
        String dbname = sys.getDB(xmlname);
        ThDB db = (ThDB)sys.cataLog.get(dbname);
        xmldoc = (ThXMLDoc)db.docNames.get(xmlname);

        /*undo actions for un-committed txs */
        xmldoc.undo(log, sys);
    }
}

```

5 Experiments and Tests Performed with our Toy Database

In this section, we are going to give an overview of our testing system and how our toy DBMS invoked from a JUnit testcase environment. Next, we describe in detail how we tested the DBMS. Finally, we present our experiments with Natix and ARIES.

5.1 Overview of Testing System

We use NetBeans IDE 3.5.1 and JUnit packages to edit and debug for the program. We put test files in a directory and record the directory name in `dbsystem.properties`. A test scenario is a file that contains our defined query language (DDL and DML) and commands for simulating the crash. Appendix A gives detail descriptions of DDL and DML. A JUnit testcase is a class that extends class `Testcase`. In this section, we explain what our unit `testCase` does:

Example 6: A typical JUnit testcase. Please refer to Appendix C for complete listing of class `ThTestExecuteDML`.

```
15. public class ThTestExecuteDML extends TestCase
16. {
    .....
23. public void testExecuteDML()
24. {
27. try
28. {
29.     ThSystem subSys = new ThSystem();
30.     //create an agent to handle execution for a particular DML plan
31.     ThAgent agent = new ThAgent(subSys);
32.     Properties prop = subSys.getProperties();
33.     ThPlan plan = ThParser.errorCheckDML(prop, subSys);
34.     if (plan.getReturnCode().equals(prop.get("EC2")))
```

```

35.      fail("Compiled Errors!");

36.      String rc = agent.executeDMLPlan(plan);
37.      if (!rc.equals((String)(subSys.getProperties().get("EC0"))))
38.          fail("ExecuteDML Error!");

39.      //verify result
40.      ThVerifyResult verify = new ThVerifyResult(prop);
41.      rc = verify.getResult();

42.      if (!rc.equals((String)(subSys.getProperties().get("EC0"))))
43.          fail("Verify Failed!");
44.      else
45.          System.out.println("Verification Success.");
46.      } catch (Exception e){
47.          fail(e.toString());
48.      }
49.  }

.....
54. }

```

Line 29 creates our database sub-system object. Line 31 creates an agent that references the sub-system. Line 32 reads in dbsystem.properties and creates a Properties object. Line 33 invokes ThParser's errorCheckDML method to scan through the test file specified in the properties file and check for queries, commands syntax errors.

Line 34 checks to see if the error code returns from errorCheckingDML is good enough to continue the process. If the error code is greater or equals to "002", the testcase is failed. Otherwise, continue processing the testcase. Line 36 executes the statements in the ThPlan object. Lines 40 to 45 verify the correctness of the output returned from the system.

5.2 Testing that DBMS works

The section below gives detailed explanatory segments from the test20.txt file (the complete listing of this file and data.xml are in Appendix D):

Before running the testcase, we fix the dbsystem.properties to use test file test20.txt and NATIX's recovery method:

```
...  
testFile = c://an//cs297//xmldb//test20.txt  
verifyFile = c://an//cs297//xmldb//unit//verify//verify20.txt  
outputFile = c://an//cs297//xmldb//src//sm//output//output.txt  
resultFile = c://an//cs297//xmldb//unit//verify//result.txt  
IO = NO  
recoverMethode = NATIX  
...
```

In the following input file segment, T1 creates an XMLDOC object in DB1DB. The CREATE XMLDOC statement also triggers an image copy of the database DB1DB. This is the initial state to fall back to in case of disaster.

```
/* Test Restart Recovery with different ARCHIVEs, and rollbacks */  
T1 CREATE DATABASE DB1DB;  
T1 CREATE XMLDOC XML1DOC IN DB1DB;  
  
T2 LOAD "c://an//cs297//xmldb//data2.xml" INTO XML1DOC;  
  
T2 COMMIT;
```

T2 loads the data in file *c://an//cs297//xmldb//data2.xml* into XML1DOC and commit. The load operation does not belongs to either the DDL nor the DML category. It is a utility that we wrote to load an XML document from a data file into a document in our database. The purpose of this is to test page swapping for our buffer management.

```
T1 ARCHIVE DB1DB;

T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>Jane Eyre</L>",
                                                    "<L>3.8</L>");
T1 COMMIT;

T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>392 Lily Ann Way, San Jose CA 95111</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)111-1111</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)222-2222</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)333-3333</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)444-4444</L>");

T2 printTree(XML1DOC);
```

In this segment, the user issues an ARCHIVE statement. This takes an image copy of the database DB1DB. Next, T1 does some inserts into XML1DOC. The result can be viewed using a printTree() statement. Here is the result of printTree(XML1DOC) command:

```
Printing structure of XML Tree: XML1DOC
ID=00000001
  name=CS297
    id=100-23-4567
      Jane Eyre
      3.8
      392 Lily Ann Way, San Jose CA 95111
      (408)111-1111
      (408)222-2222
      (408)333-3333
      (408)444-4444
    id=200-34-5678
      Irene Hugh
      3.0
    id=300-45-6789
      Thomas Lee
      2.8
```

Suppose we add a rollback statement for T1, the result of a printTree command should include the committed insert values and not include un-committed insert values:

```
T1 ROLLBACK;
```

```
T2 printTree(XML1DOC);
```

Printing structure of XML Tree: XML1DOC

ID=00000001

name=CS297

id=100-23-4567

Jane Eyre

3.8

id=200-34-5678

Irene Hugh

3.0

id=300-45-6789

Thomas Lee

2.8

The following is a snapshot of our log records written in the logrecords.txt. Please note that the log record with LSN number 571 points to the log record 565. During the undo and redo phase of recovery, the log manager will not perform undos for Log Records 570, 569, 568, 567, and 566. This is because during analysis phase, log manager sees the CLR written for the rollback of Operations 570, 568, 568, 567, and 566; Thus, it skip these log records. For the complete listing of logrecords.txt, please refer to Appendix E.

```

UNINSERTGROUP 555 DB1DB XML1DOC 1004 T2 2 (ID 00000001)
UNINSERTGROUP 556 DB1DB XML1DOC 1004 T2 3 /G (name CS297)
UNINSERTGROUP 557 DB1DB XML1DOC 1004 T2 4 /G/G (id 100-23-4567)
UNINSERTGROUP 558 DB1DB XML1DOC 1004 T2 5 /G/G (id 200-34-5678) (Irene Hugh) (3.0)
UNINSERTGROUP 559 DB1DB XML1DOC 1004 T2 8 /G/G (id 300-45-6789) (Thomas Lee) (2.8)
UNINSERTGROUP 560 DB1DB XML1DOC 1004 T2 11 /G/G (id 400-56-7890) (Erica Nguyen) (3.1)
UNINSERTLEAVE 562 DB1DB XML1DOC 1005 T2 15 /G/G/L (Shaggy Nguyen)
UNINSERTLEAVE 563 DB1DB XML1DOC 1005 T2 16 /G/G/L (3.1)
UNINSERTGROUP 561 DB1DB XML1DOC 1004 T2 14 /G/G (id 500-23-4567)
UNINSERTLEAVE 564 DB1DB XML1DOC 1005 T1 17 /G/G[1]/L (Jane Eyre)
UNINSERTLEAVE 565 DB1DB XML1DOC 1005 T1 18 /G/G[1]/L (3.8)
UNINSERTLEAVE 566 DB1DB XML1DOC 1005 T1 19 /G/G[1]/L (392 Lily Ann Way, San Jose CA 95111)
UNINSERTLEAVE 567 DB1DB XML1DOC 1005 T1 20 /G/G[1]/L ((408)111-1111)
UNINSERTLEAVE 568 DB1DB XML1DOC 1005 T1 21 /G/G[1]/L ((408)222-2222)
UNINSERTLEAVE 569 DB1DB XML1DOC 1005 T1 22 /G/G[1]/L ((408)333-3333)
UNINSERTLEAVE 570 DB1DB XML1DOC 1005 T1 23 /G/G[1]/L ((408)444-4444)
CLR 571 565
UNINSERTLEAVE 572 DB1DB XML1DOC 1005 T2 24 /G/G[2]/L (123 No Name St., San Jose CA 95112)

```

.....

To continue our experiment, we add more insert statements for both transactions:

```

T1 ARCHIVE DB1DB;

T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>123 No Name St., San Jose CA 95112</L>",
                                                    "<L>(408)111-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)222-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)333-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)444-0000</L>");

T2 COMMIT;

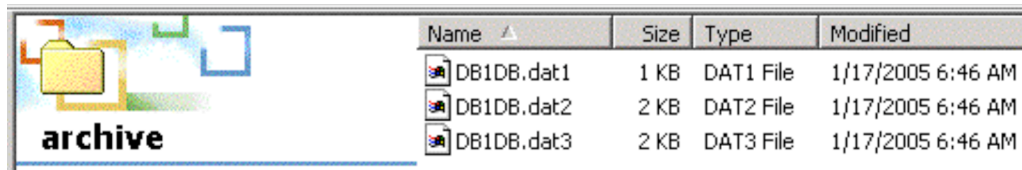
T2 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)555-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)666-0000</L>");

T1 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)555-5555</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)666-6666</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)777-7777</L>");

T2 printTree(XML1DOC);

T1 CRASH;
T2 printTree(XML1DOC);

```



Name	Size	Type	Modified
DB1DB.dat1	1 KB	DAT1 File	1/17/2005 6:46 AM
DB1DB.dat2	2 KB	DAT2 File	1/17/2005 6:46 AM
DB1DB.dat3	2 KB	DAT3 File	1/17/2005 6:46 AM

The first image copy DB1DB.dat1 is taken after the CREATE statement executed. The second and third image copy of DB1DB are taken by the ARCHIVE commands.

The following is the result of XML1DOC before system crash:

```

Printing structure of XML Tree: XML1DOC
ID=00000001
  name=CS297
    id=100-23-4567
      Jane Eyre
      3.8
      (408)555-0000
      (408)666-0000
    id=200-34-5678
      Irene Hugh
      3.0
      123 No Name St., San Jose CA 95112
      (408)111-0000
      (408)222-0000
      (408)333-0000
      (408)444-0000
      (408)555-5555
      (408)666-6666
      (408)777-7777
    id=300-45-6789
      Thomas Lee
      2.8
    id=400-56-7890
      Erica Nguyen
      3.1
    id=500-23-4567
      Shaggy Nguyen
      3.1

```

When the system crashes, the recovery manager takes the following actions to recover for our database:

```
Simulating Crash!
.....
Start Recovery from Crash.....

Analyzing Logs.....

Redo Phase!.....

Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...

Undo Phase!.....
```

During the restart recovery, our system obtains the last image copy of the database from the last checkpoint (in this case, it takes the image copy DB1DB.dat3) and analyzes the log records.

During the redo phase, if a transaction is committed, the recovery manager will use the log records written by this transaction to bring it to its last committed updates (from the image copy).

During the undo phase, if a transaction is not committed, the recovery manager will undo updates performed by this transaction to return to its last consistent state (from the image copy).

By definition, a consistent state can be at a save point, a checkpoint, or the point where the transaction last committed. For our recovery algorithm, we chose to go back to the point where a transaction was committed. Therefore, the output of the printTree of our database after the restart recovery is as follows:

Printing structure of XML Tree: XML1DOC

ID=00000001

name=CS297

id=100-23-4567

Jane Eyre

3.8

id=200-34-5678

Irene Hugh

3.0

123 No Name St., San Jose CA 95112

(408)111-0000

(408)222-0000

(408)333-0000

(408)444-0000

id=300-45-6789

Thomas Lee

2.8

id=400-56-7890

Erica Nguyen

3.1

id=500-23-4567

Shaggy Nguyen

3.1

5.3 The Differences between Natix and ARIES log records

In this section, we explain the differences between Natix and ARIES log records. For a complete listing of the log records files, please refer to Appendix E. With the same test file as the above experiment, we now switch our experiment to ARIES. We edit the property file as follows:

...

testFile = c://an//cs297//xmldb//test20.txt

verifyFile = c://an//cs297//xmldb//unit//verify//verify20.txt

outputFile = c://an//cs297//xmldb/src//sm//output//output.txt

resultFile = c://an//cs297//xmldb//unit//verify//result.txt

IO = NO

recoverMethode = ARIES

The log records bellow were written during a forward phase from our program when we use the ARIES's recovery method. There are nine log records:

```
UINSERTGROUP 558 DB1DB XML1DOC 1004 T2 5 /G/G (id 200-34-5678)
UINSERTLEAVE 559 DB1DB XML1DOC 1004 T2 6 /G/G/L (Irene Hugh)
UINSERTLEAVE 560 DB1DB XML1DOC 1004 T2 7 /G/G/L (3.0)
UINSERTGROUP 561 DB1DB XML1DOC 1004 T2 8 /G/G (id 300-45-6789)
UINSERTLEAVE 562 DB1DB XML1DOC 1004 T2 9 /G/G/L (Thomas Lee)
UINSERTLEAVE 563 DB1DB XML1DOC 1004 T2 10 /G/G/L (2.8)
UINSERTGROUP 564 DB1DB XML1DOC 1004 T2 11 /G/G (id 400-56-7890)
UINSERTLEAVE 565 DB1DB XML1DOC 1004 T2 12 /G/G/L (Erica Nguyen)
UINSERTLEAVE 566 DB1DB XML1DOC 1004 T2 13 /G/G/L (3.1)
```

On the other hand, there are only three log records for the same updates from our program when we use the NATIX's recovery method:

```
UINSERTGROUP 558 DB1DB XML1DOC 1004 T2 5 /G/G (id 200-34-5678) (Irene Hugh) (3.0)
UINSERTGROUP 559 DB1DB XML1DOC 1004 T2 8 /G/G (id 300-45-6789) (Thomas Lee) (2.8)
UINSERTGROUP 560 DB1DB XML1DOC 1004 T2 11 /G/G (id 400-56-7890) (Erica Nguyen) (3.1)
```

This is because when leave nodes are inserted into a group node that created by the same transaction, Natix modifies its local log records so that it logs only a sub-tree.

5.4 Performance Test

We conducted an experiment, using a computer with 800 MHz CPU and 256 MB RAM, to compare the performance of our DBMS when uses ARIES and when uses Natix. We used two types of tree structures: one has a large structure (lots of nodes) with small data nodes, and one has small structure (small number of nodes) but with large data nodes. We used the class ThRandomTestCreator to randomly generate operations to be performs on the trees. Input parameters that ThRandomTestCreate uses to generate data are in dbsystem.properties file.

Experimentors can modify the value of the key “paragraph” in the dbsystem.properties file to reflex the data size for a node to be generated. The number of nodes to be generated can be adjusted via key “numberOfNodes”. For example, for a large structure tree, the user can assign numberOfNodes to 2000. Please refer to Appendix D for the listing of the source code of ThRandomTestCreator class.

5.4.1 Case Study with Small Structure, Large Data

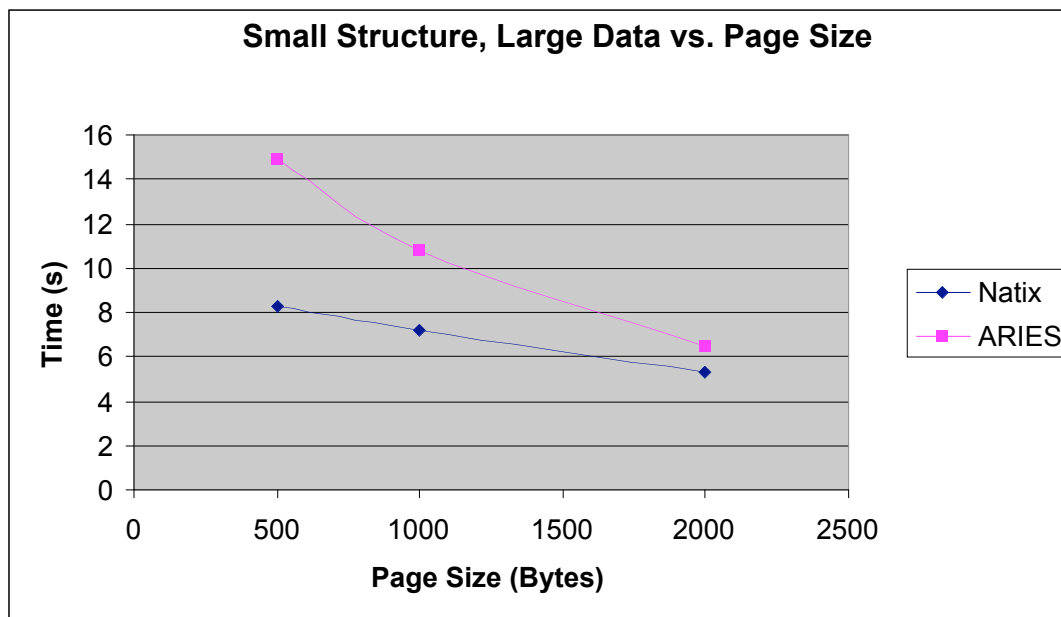
With the small structure and large data, we used ThRandomTestCreator to generated 500 insert and update statements with 70 percent insert statements and 30 percent update statements. For simplicity, we added four delete statements at the end of the random generator class. We ran the generated test file ten times. Each time, we experiment with a different page size with respect to 500 bytes, 1000 bytes, and 2000bytes. The following are our performance matrixes:

Page Size (Bytes)	ARIES - Small Structure, Large Data (Sec.)											Stdev
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	Average	
500	14.461	14.751	15.122	14.691	15.011	14.681	14.831	14.28	15.823	15.823	14.95	0.493834
1000	11.567	10.205	10.836	10.443	11.667	10.034	10.736	9.905	10.912	11.374	10.77	0.595512
2000	6.45	6.509	6.459	6.68	6.298	6.508	6.269	6.219	7.06	6.249	6.47	0.240315

Page Size (Bytes)	Natix - Small Structure, Large Data (Sec.)											
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	Average	Stdev
500	8.5	8.442	8.212	8.382	8.402	8.011	8.281	8.151	8.312	8.21	8.29	0.14086
1000	7.651	7.21	7.02	6.7	6.879	7.19	7.261	7.309	7.691	7.122	7.20	0.29183
2000	5.188	5.247	5.217	5.148	5.077	5.388	5.167	5.248	5.418	5.979	5.31	0.24445

The following is our performance plot that shows the performance of ARIES and Natix when using small structure, large data nodes against page size:

Page size (Bytes)	Small Structure, Large Data (Sec.)	
	Natix	ARIES
500	8.29	14.95
1000	7.2	10.77
2000	5.31	6.47



From the matrixes and graphs above, we noticed that Natix does better than ARIES in all of the page size variations. There are several factors that could lead the above result. First, during the redo phase, ARIES designed to use log records to bring dirty pages of both un-committed and committed transactions to the point of system crash before rolls back the changes for

uncommitted transactions (repeat history). Natix, on the other hand, redo only for committed transactions. The reason that ARIES has to “repeat history” was because it is designed to work for relational databases where it allows page and row locking. Natix is designed for XML databases where it enforces page locking. Secondly, ARIES writes log records for every update transaction onto the disk. Natix, on the other hand, optimizes their log records before writes them to the disk.

From the graph, we noticed that when the page size is 500 bytes, the performance of Natix over ARIES is significantly improved. However, when the page size is 2000 bytes, the performance of Natix over ARIES is not significantly improved. This is because when we increase the page size, more nodes can be stored in a page. Therefore, the system uses less number of page to store the same number of nodes. Thus, reduce the number of page swapping operations and improves the performance.

5.4.2 Case Study with Large Structure, Small Data

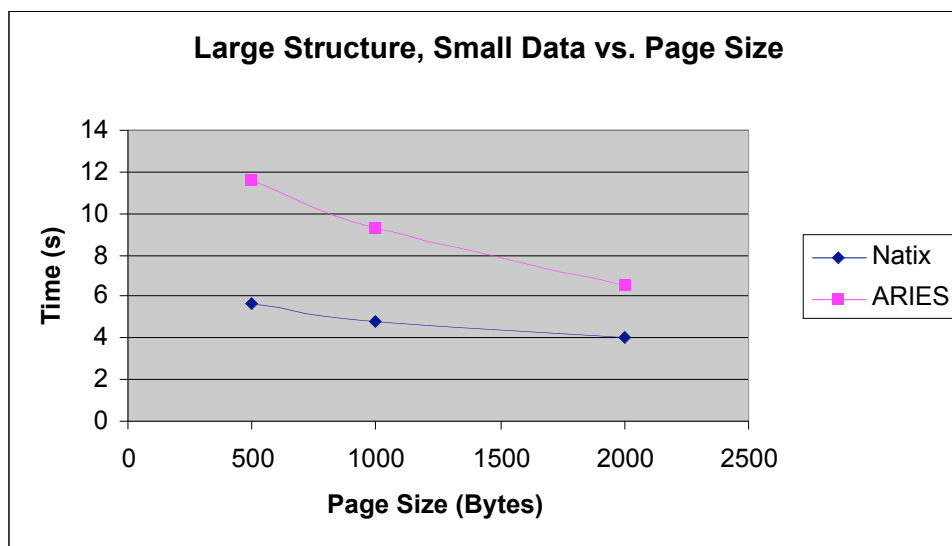
With the large structure and small data, we used a paragraph of text as the input for ThRandomtestCreator to generate node’s contents. It generate 2000 insert and update statements. with 70 percent insert statements and 30 percent update statements. We run the generated test file ten times with three different page sizes: 500 bytes, 1000 bytes, and 2000 bytes. The following are our performance matrixes:

Page Size (Bytes)	ARIES - Large Structure, Small Data (Sec.)											Stdev
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	Average	
500	11.567	11.386	11.596	12.107	12.398	11.587	11.496	11.637	11.543	11.086	11.64	0.347032
1000	9.764	9.173	9.774	9.864	8.802	8.67	8.76	8.99	9.64	9.774	9.32	0.462885
2000	5.879	5.898	6.64	6.359	5.961	7.541	5.798	7.19	5.969	7.711	6.49	0.698486

Page Size (Bytes)	Natix - Large Structure, Small Data (Sec.)											
	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	Average	Stdev
500	5.357	5.528	6.809	5.207	5.457	6.42	5.498	5.915	5.418	5.378	5.70	0.496773
1000	5.198	4.776	4.901	4.678	4.716	4.396	4.627	4.457	5.858	4.606	4.82	0.406740
2000	3.08	3.024	4.115	4.356	5.017	3.916	4.106	4.096	4.126	4.176	4.00	0.552048

The following is our performance plot that shows the performance of ARIES and Natix when using small structure, large data nodes against page size:

Page size (Bytes)	Large Structure, Small Data (Sec.)	
	Natix	ARIES
500	5.7	11.64
1000	4.82	9.32
2000	4	6.49



From the matrixes and graphs above, Natix also does better than ARIES in all of the cases. The explanation would be the same as in the case study with small structure and large data.

6 Conclusion

In this project, we developed a toy XML database management system (DBMS) useful for studying recovery and buffer management issues. Our system is capable of storing XML document that span over pages and supports both ARIES and Natix recovery methods.

We also established an experimental environment where we can test our DBMS with different parameters such as page size, page number, input test file, verify file, recovery method, etc.... In addition, we built several other components from scratch: i.e., storage management, log management, transaction management, and query engine. We assumed that updates by different transactions are serializable. Thus, we omit the lock management sub-component.

Our DBMS supports operations and commands such as create, insert, update, delete, rollback, commit, load, and printTree, etc.... However, it limits the number of XML documents it can store and manage to one document per database. We did this because our main emphasis was not on storage management but on data integrity.

We conducted an experiment, using a computer with 800 MHz CPU and 256 MB RAM, to compare the performance of our DBMS using ARIES and using Natix. The overall result shows that Natix's recovery management does better.

The project was challenged yet interesting and exciting. Some of the challenges are buffer management implementation, especially for Structure-Map Page. It is interesting to learn new

and old design pattern with Professor Pollett to design the structure, the flow for our project. It is exciting to see the design works.

For future development, this project can be use to model latency involved with lock management. It can also be extended to model client-server environment. For recovery management area, fast log apply and defer recovery should be study to improve the performance and availability for the database system.

7 Bibliography

[Kan02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. *Anatomy of a native XML base management system*. VLDB Journal, 11(4):292--314, 2002.

[W3C] “Namespaces in XML”, World Wide Web Consortium 14-January-1999.

[Mohan92] C. Mohan et al. *ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging*. ACM Transactions on Database Systems, 17(1):94--162, March 1992.

[Carey 92] M. Franklin, M. Zwillig, C. Tan, M. Carey, and D. DeWitt. *Crash Recovery in Client-Server EXODUS*. In Proceedings of the ACM-SIGMOD Conference, San Diego, CA, June 1992.

[Seshadri97] PREDATOR: An OR-DBMS with Enhanced Data Types. P. Seshadri, M. Paskin. SIGMOD, 1997, p. 568-571.

[Witt94] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. *Client-Server Paradise*. In VLDB, pages 558--569, Santiago, Chile, 1994.

[Carey 94] M. J. Carey et al. *Shoring up persistent applications*. ACM SIGMOD International Conference on Management of Data. 1994.

[Jag 02] H.V.Jagadish, S.Al-Khalifa, A.Chapman, L.V.S.Lakshmanan, A.Nierman, S.Paparizos, J.Patel, D.Srivastava, N.Wiwatwattana, Y.Wu, and C.Yu. *Timber:a native xml database*. VLDB Journal, 11(4), 2002.

[Mohan99] C. Mohan. *Repeating history beyond aries*. In VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999.

[Lahiri01] Fast-Start: Quick Fault Recovery in Oracle. T. Lahiri, A. Ganesh, R. Weiss, A. Joshi. ACM SIGMOD, 2001.

Appendix A - Glossary

XML	Extensible Markup Language – designed to meet the challenges of large-scale electronic publishing [W3C].
XML Database	Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
DDL	Data Definition Language - define databases and their components.
DML	Data Manipulation Language - a language for the manipulation of data in a database. For example, language that contains statements such as INSERT, UPDATE, and DELETE.
CLR	Compensation Log Record – written during forward phase for rollback operations to avoid complications during redo recovery phase.
WAL	Write-Ahead Logging - an update operation cannot be completed unless all log records for the operation has been written to disk.
CHECK POINT	A consistent point where system take an image copy of the database.

SAVE POINT A consistent point in time defined by user application.

Appendix B - Query and Commands Syntax

I. Data Definition Language (DDL):

It is use to define databases and their components. The following are the description DDL statements:

1. **CREATE DATABASE** *database-name*;

The above statement creates a database called *database-name*.

database-name must start with a character or an underscore.

The statement has to end with a semicolon.

2. **CREATE XMLDOC** *xml-document-name* **IN** *database-name*;

This statement creates an *xml-document-name* document in *database-name*.

xml-doc-name must start with a character or an underscore.

database-name must be already exist in the system.

II. Data Manipulation Language (DML):

It is a language for the manipulation of data in a database. For example, language that contains statements such as INSERT, UPDATE, and DELETE. Here are the description DML statements:

1. **INSERT INTO** *xml-document-name* **PATH**("location-path") **VALUES**("<L>leave-text-data-1</L>", ..., "<L>leave-text-data-n</L>");

This statement inserts n leave text nodes into specified location-path.

At this time, we support only absolute location-path. An absolute location-path starts with a '/' and followed by one or more location steps separated by a '/'. A location path can be used as an expression. Evaluation of the expression returns a set of nodes selected by the path.

Each location step can contain a filter predicate. A filter predicate is put inside open and close brackets ([]). It filters the set of nodes into a new set of nodes.

2. **DELETE FROM** *xml-document-name* **WHERE PATH**("location-path");

The statement will delete the node returned from evaluation of the location-path.

3. **UPDATE** *xml-document-name* **SET VALUE**("<L>text-data-1</L>") **WHERE PATH**("location-path");

This statement updates the node returned by the location-path with value inside **VALUES**("...",...")

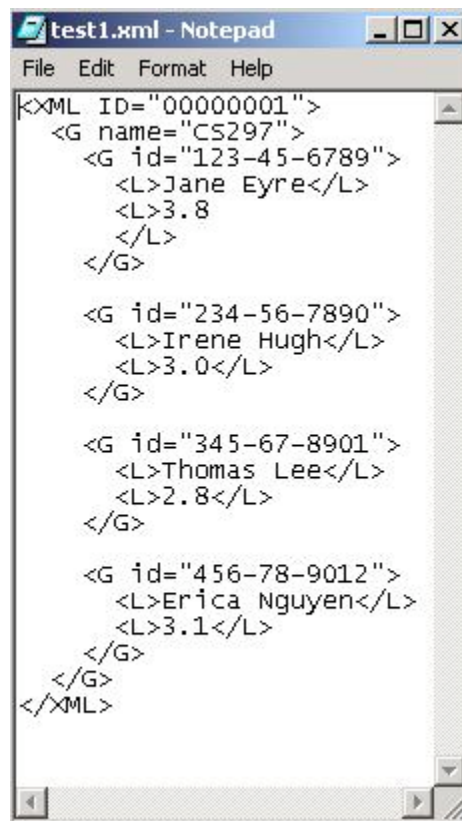


Figure 17 - Our XML Document Example

4. **LOAD** "file-name" **INTO** xml-document-name;

This statement loads data from a flat file file-name into create xml document xml-document-name. Format of the flat file could be as below, in which there are 3 types of tags:

a. <XML> </XML>: these tags indicate the beginning and the ending of an XML document.

<XML can be followed by an attribute and attribute value, separated by an equal sign.

b. <G> </G>: these tags are inside XML tags. <G> </G> tags can be nested with <G></G> and/or <L></L> tags. <G> can have attribute and attribute value separate by an equal sign (like XML tag).

c. <L> </L>: these tags are the lowest level in our xml document. These tags start and end text data.

5. **DROP XMLDOC** *xml-document-name*;

This statement drops xml-document-name from current database system.

6. **DROP DATABASE** *database-name*;

This statement drops the current database-name from the database system.

7. **COMMIT**;

This statement commits the changes by the current transaction. The more often the user commit, the faster for the system to recover during restart recovery.

8. **ROLLBACK**;

This statement is use to undo the changes since the last commit or changes made by current transaction.

9. **/***

This is use for comments. For comment that spans over several lines, “/*” has to be placed at the beginning of every line to be commented.

10. **CRASH**;

This statement is use to simulate the database's crash. When this command is execute, recovery manager will be triggered and execute.

Appendix C- JUnit Testcase

```
1. package cs297.xmlldb.unit.server;

2. import junit.framework.*;
3. import java.util.*;
4. import java.lang.*;
5. import java.io.*;
6. import cs297.xmlldb.src.server.*;
7. import cs297.xmlldb.src.cm.*;
8. import cs297.xmlldb.src.sm.*;
9. import cs297.xmlldb.src.bm.*;

10. /**
11.  Unit Test execution of statements in specify in test scenario file.
12.  @Date: Spring 2004
13.  @Author: Thien An Nguyen
14.  */

15. public class ThTestExecuteDML extends TestCase
16. {
17.     public ThTestExecuteDML(String name)
18.     {
19.         super(name);
20.     }

21.     protected void setUp()
22.     { }

23.     public void testExecuteDML()
24.     {
25.         try
26.         {
27.             ThSystem subSys = new ThSystem();

28.             //create an agent to handle execution for a particular DML plan
29.             ThAgent agent = new ThAgent(subSys);
30.             Properties prop = subSys.getProperties();

31.             ThPlan plan = ThParser.errorCheckDML(prop, subSys);

32.             if (plan.getReturnCode().equals(prop.get("EC2")))
33.                 fail("Compiled Errors!");
```

```

36.    String rc = agent.executeDMLPlan(plan);
37.    if (!rc.equals((String)(subSys.getProperties().get("EC0"))))
38.        fail("ExecuteDML Error!");
39.    //verify result
40.    ThVerifyResult verify = new ThVerifyResult(prop);
41.    rc = verify.getResult();

42.    if (!rc.equals((String)(subSys.getProperties().get("EC0"))))
43.        fail("Verify Failed!");
44.    else
45.        System.out.println("Verification Success.");
46.    } catch (Exception e){
47.        fail(e.toString());
48.    }
49. }

50. public static void main(String args[])
51. {
52.     junit.textui.TestRunner.run(ThTestExecuteDML.class);
53. }
54. }

```

Appendix D – Scenarios/Input Files, Data Files

Listing of test20.txt:

```
/* Test Restart Recovery with different ARCHIVEs, and rollbacks */
T1 CREATE DATABASE DB1DB;
T1 CREATE XMLDOC XML1DOC IN DB1DB;

T2 LOAD "c://an//cs297//xmldb//data2.xml"
    INTO XML1DOC;

T2 COMMIT;

T1 ARCHIVE DB1DB;

T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>Jane Eyre</L>",
    "<L>3.8</L>");
T1 COMMIT;

T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>392 Lily Ann Way, San Jose CA 95111</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)111-1111</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)222-2222</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)333-3333</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)444-4444</L>");

T2 printTree(XML1DOC);

T1 ROLLBACK;

T2 printTree(XML1DOC);

T1 ARCHIVE DB1DB;

T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>123 No Name St., San Jose CA 95112</L>",
    "<L>(408)111-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)222-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)333-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)444-0000</L>");

T2 COMMIT;

T2 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)555-0000</L>");
T2 INSERT INTO XML1DOC PATH("/G/G[1]/L") VALUES("<L>(408)666-0000</L>");

T1 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)555-5555</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)666-6666</L>");
T1 INSERT INTO XML1DOC PATH("/G/G[2]/L") VALUES("<L>(408)777-7777</L>");

T2 printTree(XML1DOC);

T1 cRASH;
T2 printTree(XML1DOC);

T1 DROP XMLDOC XML1DOC;
T1 DROP DATABASE DB1DB;
```

Listing of data2.xml

```
<XML ID="00000001">
  <G name="CS297">
    <G id="100-23-4567">
      </G>

    <G id="200-34-5678">
      <L>Irene Hugh</L>
      <L>3.0</L>
    </G>

    <G id="300-45-6789">
      <L>Thomas Lee</L>
      <L>2.8</L>
    </G>

    <G id="400-56-7890">
      <L>Erica Nguyen</L>
      <L>3.1</L>
    </G>

    <G id="500-23-4567">
      <L>Shaggy Nguyen</L>
      <L>3.1</L>
    </G>

  </G>
</XML>
```

```

package cs297.xmlldb.src.utils;
import java.util.*;
import java.math.*;
import java.io.*;
import cs297.xmlldb.src.server.*;
import cs297.xmlldb.src.cm.*;
import cs297.xmlldb.src.sm.*;
import cs297.xmlldb.src.bm.*;

/**
This class is used as an experimental input generator.
It generates operations such as insert, update randomly.
The data for the node it generating should be specified in
the file dbsystem.properties file.
@Date: Spring 2004
@author: Thien An Nguyen
*/
public class ThRandomTestCreator
{
    public ThRandomTestCreator()
    {
        try {
            /* Initialization */
            Random rand = new Random();
            Properties prop = new Properties();
            prop.load(new FileInputStream("c://an//cs297//xmlldb//dbsystem.properties"));
            int num = Integer.parseInt(prop.getProperty("numberOfNodes"));
            String para = prop.getProperty("paragraph");
            String outfile = prop.getProperty("randomOutFile");
            int delcount = Integer.parseInt(prop.getProperty("deleteCount"));
            int insertweight = Integer.parseInt(prop.getProperty("insertWeight"));
            int updateweight = Integer.parseInt(prop.getProperty("updateWeight"));

            /* Open Output file */
            DataOutputStream outStream = new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream(outfile)));
            /* Insert some DDL and Util statements first */
            outStream.writeBytes("T1 CREATE DATABASE DB1DB;\r\n"
                + "T1 CREATE XMLDOC XML1DOC IN DB1DB;\r\n\r\n"
                + "T1 LOAD "
                + "\"c://an//cs297//xmlldb//src//sm//input//test1.xml\""
                + " INTO XML1DOC;\r\n\r\n"
                + "T1 COMMIT;\r\n\r\n"
                + "T1 ARCHIVE DB1DB;\r\n");
            outStream.writeBytes("\r\n");
        }
    }
}

```

```

/* Generating operations */
int possi;
for (int i = 0; i < num; i++)
{
    possi = rand.nextInt(10);
    if (possi < insertweight)
    { /* insert statement */
        outputStream.writeBytes("T1 INSERT INTO XML1DOC PATH(\"/G/G["
            +(i%10+1)
            +"]/L\") "
            + "VALUES(\"<L>"
            + i
            + para
            + "</L>\");\r\n");
    }
    else
    { /* update statement */
        outputStream.writeBytes("T1 UPDATE XML1DOC SET VALUE(\"<L>"
            + i
            + para
            + "</L>\")"
            + " WHERE PATH(\"/G/G[1]/L[3]\");"
            + "\r\n");
    }

    outputStream.writeBytes("\r\n");
}

for (int i = 1; i <= delletecount; i++)
    outputStream.writeBytes("T1 DELETE FROM XML1DOC WHERE PATH(\"/G/G[6]\");"
        + "\r\n\r\n");

/* crash system */
outputStream.writeBytes("T1 CRASH;\r\n\r\n");
outputStream.writeBytes("T1 DROP XMLDOC XML1DOC;\r\n\r\n");
outputStream.writeBytes("T1 DROP DATABASE DB1DB;\r\n\r\n");
outputStream.flush();
} catch (Exception e) {
    System.out.println(e.toString());
}
}
}
public static void main(String[] args) {
    ThRandomTestCreator ran = new ThRandomTestCreator();
}
}

```

Appendix E – Output Results

Output from running test20.txt:

```
Sub-system is up
Checking DDL/DML error(s) ...
Executing DML Plan
DB1DB created
XML1DOC added in DB1DB
Opening file: c://an//cs297//xmldb//src//sm//input//data2.xml
Loading into XMLDoc...
Archive DB1DB
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Printing structure of XML Tree: XML1DOC
ID=00000001
  name=CS297
    id=100-23-4567
      Jane Eyre
      3.8
      392 Lily Ann Way, San Jose CA 95111
      (408)111-1111
      (408)222-2222
      (408)333-3333
      (408)444-4444
    id=200-34-5678
      Irene Hugh
      3.0
    id=300-45-6789
      Thomas Lee
      2.8
    id=400-56-7890
      Erica Nguyen
      3.1
    id=500-23-4567
      Shaggy Nguyen
      3.1
Rollback!
Printing structure of XML Tree: XML1DOC
ID=00000001
```

name=CS297
id=100-23-4567
Jane Eyre
3.8
id=200-34-5678
Irene Hugh
3.0
id=300-45-6789
Thomas Lee
2.8
id=400-56-7890
Erica Nguyen
3.1
id=500-23-4567
Shaggy Nguyen
3.1

Archive DB1DB

Inserting into XMLDoc...

Inserting into XMLDoc...

Inserting into XMLDoc...

Inserting into XMLDoc...

Inserting into XMLDoc...

Inserting into XMLDoc...

Inserting into XMLDoc...

Inserting into XMLDoc...

Inserting into XMLDoc...

Printing structure of XML Tree: XML1DOC

ID=00000001

name=CS297

id=100-23-4567

Jane Eyre

3.8

(408)555-0000

(408)666-0000

id=200-34-5678

Irene Hugh

3.0

123 No Name St., San Jose CA 95112

(408)111-0000

(408)222-0000

(408)333-0000

(408)444-0000

(408)555-5555

(408)666-6666

(408)777-7777

id=300-45-6789

Thomas Lee
2.8
id=400-56-7890
Erica Nguyen
3.1
id=500-23-4567
Shaggy Nguyen
3.1

Simulating Crash!

.....
Start Recovery from Crash.....

Analyzing Logs.....

Redo Phase!.....

Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...
Inserting into XMLDoc...

Undo Phase!.....

Printing structure of XML Tree: XML1DOC

ID=00000001
name=CS297
id=100-23-4567
Jane Eyre
3.8
id=200-34-5678
Irene Hugh
3.0
123 No Name St., San Jose CA 95112
(408)111-0000
(408)222-0000
(408)333-0000
(408)444-0000
id=300-45-6789
Thomas Lee
2.8
id=400-56-7890
Erica Nguyen
3.1
id=500-23-4567

Shaggy Nguyen
3.1
DB1DB dropped

Output Log Records from running test20.txt (using Natix recovery mode):

UINSERTGROUP 555 DB1DB XML1DOC 1004 T2 2 . (ID 00000001)
UINSERTGROUP 556 DB1DB XML1DOC 1004 T2 3 /G (name CS297)
UINSERTGROUP 557 DB1DB XML1DOC 1004 T2 4 /G/G (id 100-23-4567)
UINSERTGROUP 558 DB1DB XML1DOC 1004 T2 5 /G/G (id 200-34-5678) (Irene Hugh) (3.0)
UINSERTGROUP 559 DB1DB XML1DOC 1004 T2 8 /G/G (id 300-45-6789) (Thomas Lee) (2.8)
UINSERTGROUP 560 DB1DB XML1DOC 1004 T2 11 /G/G (id 400-56-7890) (Erica Nguyen) (3.1)
UINSERTLEAVE 562 DB1DB XML1DOC 1005 T2 15 /G/G/L (Shaggy Nguyen)
UINSERTLEAVE 563 DB1DB XML1DOC 1005 T2 16 /G/G/L (3.1)
UINSERTGROUP 561 DB1DB XML1DOC 1004 T2 14 /G/G (id 500-23-4567)
UINSERTLEAVE 564 DB1DB XML1DOC 1005 T1 17 /G/G[1]/L (Jane Eyre)
UINSERTLEAVE 565 DB1DB XML1DOC 1005 T1 18 /G/G[1]/L (3.8)
UINSERTLEAVE 566 DB1DB XML1DOC 1005 T1 19 /G/G[1]/L (392 Lily Ann Way, San Jose CA 95111)
UINSERTLEAVE 567 DB1DB XML1DOC 1005 T1 20 /G/G[1]/L ((408)111-1111)
UINSERTLEAVE 568 DB1DB XML1DOC 1005 T1 21 /G/G[1]/L ((408)222-2222)
UINSERTLEAVE 569 DB1DB XML1DOC 1005 T1 22 /G/G[1]/L ((408)333-3333)
UINSERTLEAVE 570 DB1DB XML1DOC 1005 T1 23 /G/G[1]/L ((408)444-4444)
CLR 571 565
UINSERTLEAVE 572 DB1DB XML1DOC 1005 T2 24 /G/G[2]/L (123 No Name St., San Jose CA 95112)
UINSERTLEAVE 574 DB1DB XML1DOC 1006 T2 26 /G/G[2]/L ((408)222-0000)
UINSERTLEAVE 575 DB1DB XML1DOC 1006 T2 27 /G/G[2]/L ((408)333-0000)
UINSERTLEAVE 573 DB1DB XML1DOC 1005 T2 25 /G/G[2]/L ((408)111-0000)
UINSERTLEAVE 576 DB1DB XML1DOC 1006 T2 28 /G/G[2]/L ((408)444-0000)
UINSERTLEAVE 577 DB1DB XML1DOC 1006 T2 29 /G/G[1]/L ((408)555-0000)
UINSERTLEAVE 578 DB1DB XML1DOC 1006 T2 30 /G/G[1]/L ((408)666-0000)
UINSERTLEAVE 579 DB1DB XML1DOC 1006 T1 31 /G/G[2]/L ((408)555-5555)
UINSERTLEAVE 580 DB1DB XML1DOC 1006 T1 32 /G/G[2]/L ((408)666-6666)
UINSERTLEAVE 581 DB1DB XML1DOC 1006 T1 33 /G/G[2]/L ((408)777-7777)

Output Log Records from running test20.txt (using ARIES recovery mode):

UINSERTGROUP 555 DB1DB XML1DOC 1004 T2 2 . (ID 00000001)
UINSERTGROUP 556 DB1DB XML1DOC 1004 T2 3 /G (name CS297)
UINSERTGROUP 557 DB1DB XML1DOC 1004 T2 4 /G/G (id 100-23-4567)
UINSERTGROUP 558 DB1DB XML1DOC 1004 T2 5 /G/G (id 200-34-5678)
UINSERTLEAVE 559 DB1DB XML1DOC 1004 T2 6 /G/G/L (Irene Hugh)
UINSERTLEAVE 560 DB1DB XML1DOC 1004 T2 7 /G/G/L (3.0)
UINSERTGROUP 561 DB1DB XML1DOC 1004 T2 8 /G/G (id 300-45-6789)
UINSERTLEAVE 562 DB1DB XML1DOC 1004 T2 9 /G/G/L (Thomas Lee)
UINSERTLEAVE 563 DB1DB XML1DOC 1004 T2 10 /G/G/L (2.8)
UINSERTGROUP 564 DB1DB XML1DOC 1004 T2 11 /G/G (id 400-56-7890)
UINSERTLEAVE 565 DB1DB XML1DOC 1004 T2 12 /G/G/L (Erica Nguyen)
UINSERTLEAVE 566 DB1DB XML1DOC 1004 T2 13 /G/G/L (3.1)
UINSERTGROUP 567 DB1DB XML1DOC 1004 T2 14 /G/G (id 500-23-4567)
UINSERTLEAVE 568 DB1DB XML1DOC 1005 T2 15 /G/G/L (Shaggy Nguyen)
UINSERTLEAVE 569 DB1DB XML1DOC 1005 T2 16 /G/G/L (3.1)
UINSERTLEAVE 570 DB1DB XML1DOC 1005 T1 17 /G/G[1]/L (Jane Eyre)
UINSERTLEAVE 571 DB1DB XML1DOC 1005 T1 18 /G/G[1]/L (3.8)
UINSERTLEAVE 572 DB1DB XML1DOC 1005 T1 19 /G/G[1]/L (392 Lily Ann Way, San Jose CA 95111)
UINSERTLEAVE 573 DB1DB XML1DOC 1005 T1 20 /G/G[1]/L ((408)111-1111)
UINSERTLEAVE 574 DB1DB XML1DOC 1005 T1 21 /G/G[1]/L ((408)222-2222)
UINSERTLEAVE 575 DB1DB XML1DOC 1005 T1 22 /G/G[1]/L ((408)333-3333)
UINSERTLEAVE 576 DB1DB XML1DOC 1005 T1 23 /G/G[1]/L ((408)444-4444)
CLR 577 571
UINSERTLEAVE 578 DB1DB XML1DOC 1005 T2 24 /G/G[2]/L (123 No Name St., San Jose CA 95112)
UINSERTLEAVE 579 DB1DB XML1DOC 1005 T2 25 /G/G[2]/L ((408)111-0000)
UINSERTLEAVE 580 DB1DB XML1DOC 1006 T2 26 /G/G[2]/L ((408)222-0000)
UINSERTLEAVE 581 DB1DB XML1DOC 1006 T2 27 /G/G[2]/L ((408)333-0000)
UINSERTLEAVE 582 DB1DB XML1DOC 1006 T2 28 /G/G[2]/L ((408)444-0000)
UINSERTLEAVE 583 DB1DB XML1DOC 1006 T2 29 /G/G[1]/L ((408)555-0000)
UINSERTLEAVE 584 DB1DB XML1DOC 1006 T2 30 /G/G[1]/L ((408)666-0000)
UINSERTLEAVE 585 DB1DB XML1DOC 1006 T1 31 /G/G[2]/L ((408)555-5555)
UINSERTLEAVE 586 DB1DB XML1DOC 1006 T1 32 /G/G[2]/L ((408)666-6666)
UINSERTLEAVE 587 DB1DB XML1DOC 1006 T1 33 /G/G[2]/L ((408)777-7777)

Appendix F – JAVA Program Source Code