DATABASE MANAGEMENT SYSTEM IMPLEMENTATION PHASE 2 GROUP - 12

Akhil Jayaraj, Eldo Jacob Mathews, Mahathi Srinivasan,
Meghna Prabhu, Renil Joseph, Vivek Menon

March 18, 2019

Abstract

This project aims at building strong fundamentals in the design and implementation of the intricacies and complexities of a DBMS. Aiming to understand the basics of storage systems, disk-based data structures, buffer managers, query optimization and more, we experiment with the minibase system which is a database management system intended for educational use that was developed by Raghu Ramakrishnan. The of the entire project is to convert this relational database to an XML database. This objective of this phase is to introduce an interval data type to help represent and store XML data in the database. To support queries on the XML data, we modify sort, nestedLoop and sortMerge joins to incorporate interval data types.

Keywords

Database, Minibase, Schema, B+ Tree, Buffer Manager, Heap Files, Sort, SortMerge,NestedLoops, Pattern Trees, Path Expressions.

1 Introduction

Minibase Distribution has a parser, optimizer, buffer pool manager, storage mechanisms (heap files, B+-trees as secondary indexes), and a disk space management system. Besides the DBMS, Minibase contains a graphical user interface and other graphical tools for studying the internal working of a DBMS. [1]

With the rapid growth of the World Wide Web, there arouse a need for a flexible semi-structured model of data formatting. Large business could no longer handle the vast amount of data exchange as the rigid schema they were using was not scalable. This is where XML (eXtensible Markup Langauge) came into play by giving a significant boost to web based and B2B applications. It claimed to handle heterogeneous data representation while acting as a means of data exchange. There now arises a need for efficient and optimized querying processing on the XML data and the need for a formal algebra becomes evident.

We handle the representation of XML by introducing a new interval data type and in turn modifying the tuple, operands and joins to support optimized query processing on these interval based attributes. The primitive tree structured relationships are parent-child and ancestor-descendant, and finding all occurrences of these relationships in an XML database is a core operation for XML query processing.

1.1 Terminology

- *XML* : eXtensible Markup Language that is a flexible semi structured data format used for data exchange and encoding documents.
- *Pattern Tree* : A pair P = (T,F) where T = (V,E) is a node labeled and edge labeled tree such that each node in V has distinct integer as its label, each edge is labeled parent child / ancestor descendant relationship and F is a boolean combination of predicates applicable to nodes. It imposes value and structure based constraints [2, 3]
- *Path Expression* : Starting with a node, following a path using parent child or ancestor descendant relationship while applying predicates to the nodes to select the path.

1.2 Goal Description

The goal of this phase is to implement the building blocks that will eventually transform minibase from a relational DBMS to an XML DBMS.

- We start by defining a new interval data type which consists of 2 values that indicate the node tag and intervalLabel where intervalLabel contains 3 integer values start and end of the interval and the level of the node in the original XML tree.
- The attribute type definitions, tuple and page get and set methods are modified to reflect the addition of this new data type.
- Next, operand definitions are modified to include interval data type. It is to be noted that existing comparison methods were also modified to check for interval containment, enclosure, overlap and non overlap.

- Conditional expressions would now have to include a flag to distinguish overlap and equality. Relational operators like greater than, less than and equal are modified to support interval comparisons.
- Sort is modified such that if the sort attribute is of type interval then all tuples are sorted based on their start values. Similarly, nestedLoopJoins and SortMerge Joins are modified to include new definitions of the conditional expressions.
- We then attempt to parse a tree structured XML file to store all its elements using this interval based representation following the schema, *nodeTable(nodeIntLabel, nodeTag).*
- Next, we implement a program which when given an interval indexed XML database and a pattern tree, identifies matching nodes using the query processing methods that was modified above and returns the results. For every query, we are tasked to implement 3 different query plans which are described in detail in the implementation section.
- Lastly, for every query, we make a note of the number of pages that are requested from the buffer manager.

1.3 Assumptions

- Minibase is installed in and running in a linux environment with JAVA installed on it.
- All processing is done purely on the disk, assuming that large amounts of data will have to be handled.
- Any attributes or plain text content in the XML data is treated as an individual "node", with the up to first 5 characters of the plain text content serving as the tag.
- The XML data file provided will be a single connected tree.
- The pattern tree provided will also be a single connected tree.
- The pattern tree will always have the root node as one of the m nodes.
- XML data and the pattern tree will not have self loops.

2 Proposed Solution

2.1 XML Parsing

Given a tree structures XML file, we parse the XML using an event based parser in JAVA with the assumption that any attributes or plain text content can be treated as an individual "node", with the first 5 characters of the plain text content serving as the tag.

Once this is done, a pre order traversal is done on the obtained tree so that we can assign an interval to each node. The interval has 2 integer values a start and end and is obtained as shown below. A level order traversal was also done and stored along with the interval to help in finding parent child relationships easily. So elements in XML were stored with the following schema, ***nodeTable(nodeIntLabel, nodeTag)****, where nodeIntLabel has 3 values start,end and level.* This labelling is also known as the Region Encoding Scheme.

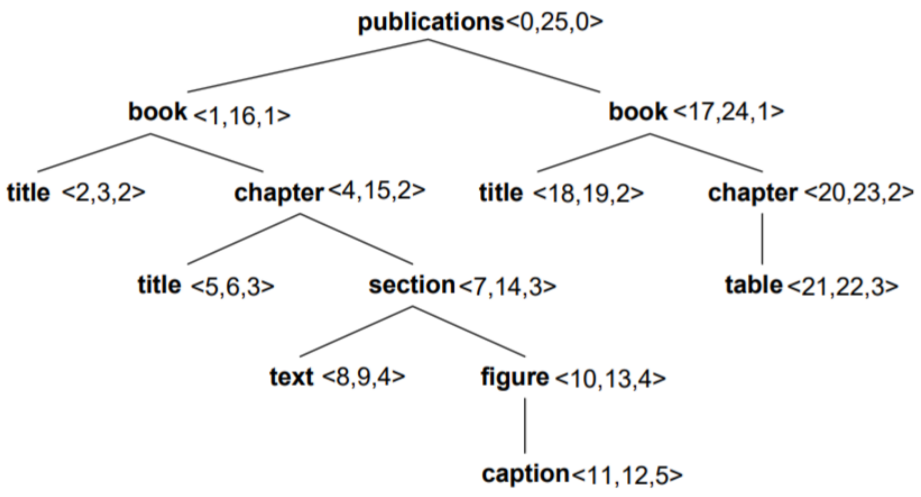

Figure 1: XML labelling scheme [5]

2.2 Tuple Comparison Methods

To support interval type tuple comparison the following have been defined:

1. **Containment**: Given 2 tuples, A and B, B is contained in A, if start(A) < start(B) and end(A) > end(B).

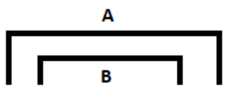

Figure 2: Containment

2. **Enclosure**: Given 2 tuples, A and B, A is enclosed in B, if start(B) < start(A) and end(B) > end(A).

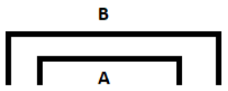

Figure 3: Enclosure

3. **Equality**: Given 2 tuples, A and B, A and B are equal, if start(B) = start(A) and end(B) = end(A).

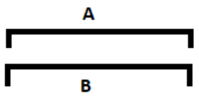

Figure 4: Equality

4. **Other types of Overlap**: This kind and its variants of overlap should ideally not occur in the database and will be an indication of something faulty if it exists.

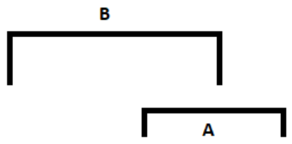

Figure 5: Others

5. **No Overlap**: There is no overlap between the two intervals.

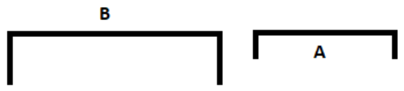

Figure 6: No overlap

2.3 Identifying Ancestor Descendant Relationships

A node which could be any nested descendent sub element of another node is said to satisfy the ancestor descendant relationship. The edge between these two nodes can be indicated with an **ad** (represented as //) implying ancestor descendent relationship between them.

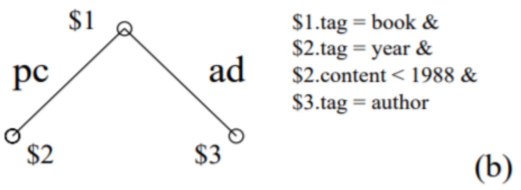

Figure 7: Ancestor Descendant Relationship [2]

In our implementation, an ancestor descendant relationship is identified by performing a structural join. A self join is performed on the node table with containment as a constraint. i.e. if the interval of the outer relation contains the interval of the inner relation.

2.4 Identifying Parent Child Relationships

A node which is a direct sub element of another node is said to satisfy the parent child relationship. The edge between these two nodes can be indicated with a **pc** (represented as /) implying parent child relationship between them.

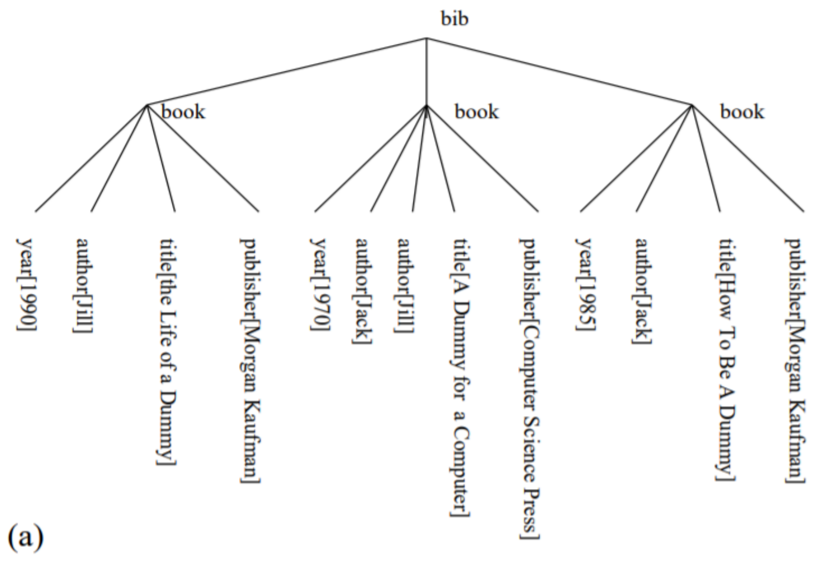
bib
book
book
book
year[1990]
author[Jill]
title[the Life of a Dummy]
publisher[Morgan Kaufman]
year[1970]
author[Jack]
author[Jill]
title[A Dummy for a Computer]
publisher[Computer Science Press]
year[1985]
author[Jack]
title[How To Be A Dummy]
publisher[Morgan Kaufman]
(a)

Figure 8: One Tree XML Database [2]

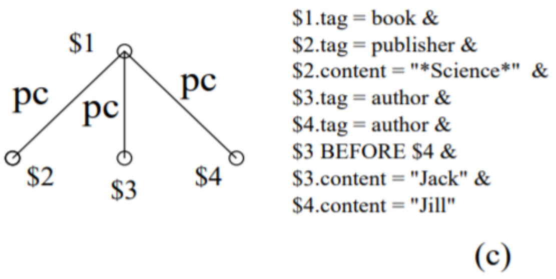

Figure 9: Parent Child Relationship [2]

In our implementation, an parent child relationship is identified in the same way as the ancestor descendant but with an additional constraint - the level of inner relation exceeds the level of the outer relation by 1.

2.5 Query Plans

A join is one of the most expensive physical operations when evaluating a query be it relational / XML. A join in the relational context is usually a value based equality join. In the XML context, even though there are value-based joins, structural joins occur much more frequently. A structural join focuses on the containment (ancestor descendant or parent child) relationship of the XML elements to be joined. The join condition is specified not on the value of the XML elements, but on their relative positions (intervals) in the XML document. [4]

- First query plan recursively applies all rules one by one using nested loop join to form the required projections.
- Second query plan iteratively computes a pattern tree for all rules using nested loop join resulting in relations of 4 columns (tag1, interval 1, tag 2, interval 2). All these relations

are joined using sortMerge,(where the tuples are sorted in ascending order of their intervals) in a pair-wise fashion, on a common column which is determined by the equal intervals (as described in 3) in the original tree. The rules are joined based on the level of the tag on the left side of the rule.

- The third query plan is almost same as the second query plan with the difference being that the sort in sortMerge, orders the tuples in a descending fashion of their intervals.

3 Implementation

3.1 Interval Data Type Definition

1. We start by setting page size to 256 bytes.
2. Then we add a new class IntervalType which has three integer values start,end (indicating interval) and level of the tree.
3. The minimum and maximum values for the interval have been set as -100000 and 100000.
4. Attribute Type definitions have been modified to include attrInteval.
5. Tuple field and page get and set methods are modified to include support for interval data types.
6. Next we modify the operand definitions to include operands of type interval. We include the following operands - containment, enclosure, other types of overlap and no overlap having definitions as described in 2.2.
7. To support range based conditions on interval data types we modify the condition expressions and Eval as below:
    - Conditional Expression flag will be set to a non negative integer if operands are of type attrInterval.
    - operator aopGT should return true, if the first operand contains the second operand
    - operator aopLT should return true, if the first operand is contained within the second operand.
    - operator aopEQ should return true
        - if flag = 0, if the two operands are equal
        - if flag = 1, if the two operands overlap (for ancestor descendant relations)
        - if flag = 2, if the two operands overlap and if the level of the outer relation is one more than the level of the inner relation (for parent-child relations)
    - operator aopNE should return true
        - if flag = 0, if the two operands are not equal
        - if flag = 1, if the two operands do not overlap
    - operators aopGE are aopLE are not defined if operands are of type attrInterval

3.2 Joins with Interval Data Types

3.2.1 Processing Rules

- Pattern trees describe rules that are applied on the database.
- The rules are created via a level based traversal of pattern tree, this means that the list contains all rules associated with the top level node followed by the second level and so on.
- It is to be noted that we are trimming the pattern trees also.

3.2.2 Path Expressions

The join condition for aopEQ returns True for both ancestor descendant and parent child relations however, this happens in 2 different scenarios, that is for parent child relations an extra condition where level of outer relation is one more than level of inner relation needs to be satisfied. This is determined by the value of flag. So a parent child relation is implied by a flag = 2 whereas ancestor descendant is implied when the flag = 1.

3.2.3 Query Plan 1

- We perform a fileScan on the node table. We provide a filter as an argument to the fileScan. The filter prunes out all the rows whose tag name is not equal to the outer tag name of the first rule.
- This fileScan object is used to perform a self join with the nodeTable based on the first rule. For the nodeTable relation in the above case, we include the inner tag of the first rule as a filter before joining. This creates a relation of 4 columns.
- For the next rule , we open the heap file and create a scan object on it while providing a filter argument which is the inner tag (right) of the current next rule. This scan object is then joined with the 4 column relation obtained above. The joining condition is on the containment relation of the outer tag of the rule's interval. The joined table will now have 6 columns.
- We are able to do this since we have ordered the rules in a level order fashion.
- This process is repeated for all the rules.
- The final result will contain the required matching sub-trees in their respective rows.
- Each row contains in the result contains the tag name and interval for all matching sub-tree nodes.
- NOTE: if the rule has a star then we wont apply a condition.

3.2.4 Query Plan 2 & 3

- For each rule we perform a self join on the node table with the condition on the containment of the rule's outer tag interval with inner tag interval. The flag values are changed as described in 3.2.2.
- Each of these nested loops results are joined using sortMerge join.
- For sortMerge Join, we first sort the tuples in ascending order of the relations based on the start value of the interval column we are going to join.

- Each of the tuples are compared and if they are equal a 0 is returned, if the left tuple is lesser than right then a -1 is returned and in the opposite case a 1 is returned. The columns are then merged according to the regular sortMerge condition.
- The first rule of the nested join result is joined with the second rule's nested join result, the join condition being equality 3 of the leftmost tag's interval in the second nested join's result and the corresponding tag's interval from the first nested join result.
- This process is repeated for all rules.
- We are able to do this since we have ordered the rules in a level order fashion.
- Each row contains in the result contains the tag name and interval for all matching sub-tree nodes.
- For query plan 3 we have used the same approach with the only difference being that the sort in the sortMerge join is done in a descending fashion.
- NOTE: if the rule has a star then we wont apply a condition.

3.3 Parsing and Storing XML data in the Database

- Given a tree structured XML file, we parse the XML using an event based parser in JAVA as described in 2.1.
- The result of the above step will be a table with columns nodeTag and nodeLabel where nodeLabel consists of 3 values, start, end and level.
- Next given a file containing a pattern tree, we identify matching nodes to this query using the operands for interval type.
- Pattern trees have the following format:
  - m -> number of nodes
  - tag1 -> tag of node 1
  - tag2 -> tag of node 2 and so on
  - i j AD -> node i is ancestor of node j
  - k l PC -> node k is parent of l
- A query will be defined using the above format and the goal is to match this pattern tree with the Database and return all results that match this tree.
- It is to be noted that the root will always be provided as one of the m nodes in the pattern tree.
- Every query is executed using 3 distinct query plans and the results are shown below.
- We also modify Minibase's buffer manager to count the number of pages that are requested from the buffer and return the total number of pages accessed with the results.
- This is done in the buffer manager which has 2 functions read page and write page. A static variable counter has been declared such that it can be accessed for every query run. When a page is requested by the query, if the page is not in the buffer, read page or write page function will be called, this will in turn increment the counter by 1. Once a query is complete, this counter is reset to 0. At the end, we remove all frames from the buffer by calling the

flush function.

4 System requirements & Execution Steps

- Install Linux system with an appropriate linux distribution such as Ubuntu.
- Download and extract the minjava.tar.gz file
- A makefile has been for our project.
- make db has been modified to include even our project for the build.
- Run make db
- java project .Phase1. This runs a test file called Phase 1 which provides interface to specify file that contains the query.
- Once query is processed we can enter another one or quit.

5 Results

The test query took too long to execute and hence the results shown below are based of the XML created by the team.

```
XML:
<A>
<B><E><F></F></E></B>
<B><E><F></F></E><D></D></B>
<B></B>
</A>

Query 1:
6
A
B
C
D
E
F
1 2 PC
2 6 AD

Query 2:
6
A
B
C
D
E
F
```

```
1 2 PC
2 5 PC
5 6 PC

Result
Number of page accessed before read= 5
Enter input filename for query
/home/eldo/Documents/CSE510Candan/Project/query.txt
QUERY PLAN---1
PCADResult 1:
[A, [1 18], B, [2 7], F, [4  5]]
Result 2:
[A, [1 18], B, [8 15], F, [10  11]]
Number of page accessed = 0
QUERY PLAN---2
Result 1:
[A, [1 18], B, [2 7], F, [4  5]]
Result 2:
[A, [1 18], B, [8 15], F, [10  11]]
Number of page accessed = 0
QUERY PLAN---3
Result 1:
[A, [1 18], B, [2 7], F, [4  5]]
Result 2:
[A, [1 18], B, [8 15], F, [10  11]]
Number of page accessed = 0
Press N to stop
y
yEnter input filename for query
/home/eldo/Documents/CSE510Candan/Project/query1.txt
QUERY PLAN---1
PCPCPCResult 1:
[A, [1 18], B, [2 7], E, [3 6], F, [4  5]]
Result 2:
[A, [1 18], B, [8 15], E, [9 12], F, [10  11]]
Number of page accessed = 0
QUERY PLAN---2
Result 1:
[A, [1 18], B, [2 7], E, [3 6], F, [4  5]]
Result 2:
[A, [1 18], B, [8 15], E, [9 12], F, [10  11]]
Number of page accessed = 0
QUERY PLAN---3
Result 1:
[A, [1 18], B, [2 7], E, [3 6], F, [4  5]]
Result 2:
[A, [1 18], B, [8 15], E, [9 12], F, [10  11]]
```

```
Number of page accessed = 14
Press N to stop
```

6 Conclusion

The following are the key takeaways from this phase of the project:

- Familiarity was achieved with the different modules of a DBMS.
- Understood how an XML can be parsed and stored in the database using interval representation.
- Gained knowledge on how attribute type, tuple fields, conditional expressions and operands need to be modified to handle interval type data.
- Learnt how structural joins are implemented and the benefits of parent child and ancestor descendant relationships in query processing.
- Reasoned about the use of various query plans using sort, nestedJoin and sortMerge joins.
- Nested Loop Joins are a very inefficient way to carry out joins as although it works well for small xml database , it does not scale well for large datasets.
- In a XML database containing around 180000 nodes, a query takes over 60 minutes to execute.

References

[1] http://research.cs.wisc.edu/coral/mini_doc/minibase.html

[2] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, et al. TAX: A Tree Algebra for XML. Proceedings of DBPL01 TAX: A Tree Algebra for XML - Database Research Group - University http://dbgroup.eecs.umich.edu/timber/files/tax_full.pdf

[3] https://hive.asu.edu/minc/images/originalphotos/groupphotos/4022/4024/xml_S19.pdf

[4] Yuqing Wu,Jignesh M. Patel,H. V. Jagadish Structural Join Order Selection for XML Query Optimization

[5] SU-CHENG HAW AND CHIEN-SING LEE, TwigX-Guide: An Efficient Twig Pattern Matching System Extending DataGuide Indexing and Region Encoding Labeling , JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 25, 603-617 (2009)

7 Appendix

Member Contribution

- Akhil Jayaraj M - Nested Loop Join and Query Plan 1
- Eldo Jacob Mathews - XML Parsing and Nested Loop Join
- Mahathi Srinivasan - Initial set up and Query Plan 1
- Meghna Prabhu - Sorted Merge and Query Plan 2,3
- Renil Joseph - Initial set up and XML parsing, buffer management
- Vivek Menon - Sorted Merge and Query Plan 2, 3