DATABASE MANAGEMENT SYSTEM IMPLEMENTATION PHASE 3 GROUP - 12

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

April 29, 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 goal of the project is to transform this relational database to an XML database. The objective of this phase is add on to the previous phase by introducing indexing and algebraic XML Operators to enable efficient query processing. Additionally we maintain statistics to enable query optimization.

Keywords

Interval Indexing, Algebraic XML Operations, Query Processing, Query Optimization.

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.

Efficient query processing and query optimization strategies are of paramount importance to any database system design. Algebraic simplification of queries can reduce the size of intermediate results and hence reduce cost associated with the operation. Efficient XML query processing is done by introducing tag and interval based indexing. Modifying Minibase's physical operators helps support Select, Cartesian Product, Join, Sorting and Grouping of witness trees. We leverage the use of statistics to help optimize queries by eliminating the worst or high cost query plans.

1.1 Terminology

- *Select*: Outputs every node that satisfies a predicate for every input tree. Implemented by a fileScan or an indexScan.
- *Join*: A common operation to join one or more XML trees. Note that even though relational joins commute and hence can be evaluated in any order. However, in XML algebra, changing the order will produce different results.
- *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]

1.2 Goal Description

The goal of this phase is to building on the previous phase by introducing indexing and efficient query processing techniques.

- We start by ensuring that our design allows for submitting multiple queries to the XML database while also reporting reasonable page read and write counts.
- We also show every query plans used for the particular query along with the pattern tree results.
- We then create a B+ tree index on tag name at the time of XML database creation.
- Next we modify our output to include query plan along with the amount of buffer frames used to implement each step of the plan.
- At this point, we introduce support for insertion and interval based search operations by creating an interval index tree.
- To ensure performance of the system we leverage sorting, tag-based B+ tree indexing, and interval-based indexing, including sort-merge and index-nested loop joins, as potential pattern-tree query processing strategies.
- We then implement physical operators that correspond to select, Cartesian product, join, sort and grouping of witness trees and the results along with the query plan, number of disk accesses and buffer frames used in each step are returned.

1.3 Assumptions

- Minibase is installed 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.
- Statistics have been used with the assumption that the interval range for each node tags are uniformly distributed.
- Interval indexing outperforms node tag indexing for * queries.
- Enough buffers are present to incorporate both the pattern trees for task 4.
- Page size is always greater than tuple size. For the Group By operator, the tuple size for a witness tree could exceed page size.
- 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 Interval Indexing

- In certain situations, tag based indexing is not beneficial and results in too many comparisons when performing a join and in turn increases the cost. For e.g. there are tags like entry, org and so on and there are many duplicates in the tags. So in a tag based join, for each outer tuple it will have to join with many inner tuples. Additionally if the query is not an index only query then for each index look up for the outer tuple a random access will have to be performed.

- To avoid this we use interval indexing with the order being the start value of the index. This type of ordering helps in retrieving ancestor descendant relationships easily.
- For each join we use statistics to determine if a tag based or interval based indexing needs to be used. For this we maintain the total number of tags for every node and the average range of its interval. If the total count in inner relation > average range of interval for the current tuple's tag name in outer relation then we use interval indexing. Else we say that it is better to use a tag based indexing scheme. For e.g. Interval indexing is not used on the root as the interval of the root will be larger than the total count of inner relation and it will be a huge overhead using interval indexing possibly even worse than a file scan.
- To ensure index only queries in interval indexing we store tag names along with their interval values. This is similar to a composite index but the ordering is based only on the interval values whereas the tag values are not used for ordering.
- After getting the tuples from the index, the only conditions we have to check are for PC relationship and the current rule's inner tag name.

2.2 Composite Indexing

- Composite index has been created on tag name, interval with ordering based on tag names and interval start values.
- This ensures an index only retrieval and since tuples are ordered based on tag name and interval, we can filter the rule's tags and find AD relations by just using the index.
- Composite index has been used as a replacement to node tag indexing, since it should perform much faster than node based indexing as it is an index only query. The only downfall is that the number of pages used for indexing will increase.
- After getting the tuples from the index, the only condition we have to check is for PC relationship.

3 Implementation

3.1 Efficient Insertion into Heap Files

- The insertion to heap file was done by saving a pointer to last inserted data page.
- This avoided the need to search the pages which are already full, thus significantly reducing the time taken to insert

3.2 Interval Indexing

- The interval indexing was created using the data file which was already sorted based on interval start value.
- For interval indexing as opposed to string indexing we have to reinitialize the indexScan object with new CondExpr for every outer tuple.

- The CondExpr would be used to set the low and high key of the indexScan.
- The low key will be set as outer tuple's start interval and the high key will be the outer tuple's end interval.
- To ensure an index only scan we store the tag name along with the interval values in the IntervalKey class. All the required classes mentioned in the specification have been modified accordingly to incorporate interval indexing.
- This is similar to composite indexing with the difference being that only interval start values are used in keyCompare function for ordering.

3.3 Composite Indexing

- Composite index was created after sorting the data file based on the tag name.
- As mentioned before we order the index using the tag name and interval value.
- In the keyCompare function first the tag names are compared and then if they are equal the interval start values are compared.
- This ensures that the index is ordered by tag name and interval start values.

3.4 Query Plan 1: Nested Loop Join using Composite Index

We have optimized the query plans developed in the last phase 8.1 by leveraging indexes.

- We have implemented this using by cascading nested loop join similar to the approach in the previous phase.8.1.1
- The difference is that we have developed a composite index on tag name, interval as described in 2.1
- For the first rule, the outer tag name is retrieved using a string index scan.

3.5 Query Plan 2: Indexed Nested Loop Join using Statistics

- This is also similar to the cascading nested loop join approach adopted in phase 2 8.1.1
- The choice of which index to use is decided through statistics as mentioned in 2.1
- For the first rule, the outer tag name is retrieved using a string index scan.

3.6 Query Plan 3: Sort Merge with Nested Loop Join using Composite Index

- This approach is similar to approach used in 8.1.2
- The individual rules are joined separately using nested loop joins.
- Nested Loop Joins are then joined using sort merge with an equality condition.
- For nested loop joins we use composite index (tag name, interval) scans for both relations before joining. This in turned helps us derive interesting order of the intervals as the intervals withing each tag names are sorted inside the index. Because of this when applying sort merge,

we do not have to perform an additional sort and can perform the merge operation directly. This ensures a non blocking implementation of sort merge.

- Using either interval indexing or node based indexing will not produce an interesting order, thus adding an extra cost of sorting the relations. This will further block the process as well.

3.7 Index Nested Loop Join

- Minibase by default uses a heap file for the inner tuple. We have modified the function to use index files for the inner tuple.
- Since composite index outperforms interval and string indexes in most cases, we have used composite indexes in index nested loop joins for the fourth task to implement the complex operators.
- For query plan 2 explained above, the statistics wont work in this case because * is not a valid tag. So if a * comes in the outer side (left side) of a rule, we use interval indexing. Using a composite index (or node tag index) won't help here either if the * is present on the outer side of the rule, since we don't have the node tag value. If the * is present on the inner side (right side), we can leverage composite indexing.

3.8 Implementation of Complex Operators

For implementation of complex operators we use composite index nested loop join. In order to implement the operations we create an iterator on the results of one pattern tree and treat the results of the second pattern tree as an inner relation. The results of the second pattern tree is written to a heap file which is traversed using a file scan, The inner relation is traversed using a heap scan.

- Cartesian Product
  - Results of the first pattern tree is combined with the results of the second pattern tree.
  - Implemented using a nested loop join with joining condition set to NULL.
  - Nested loop join is performed using the iterator for pattern tree 1 and file scan for results of pattern tree 2.
  - This gives us all possible combinations of the results.
- Tag join and node join
  - Tag join i j: results of the first pattern tree joins with the results of the second pattern tree based on the tags of i^{th} and j^{th} nodes of the two trees respectively.
  - Both are implemented using nested loop join with condition expression as right filter. The two conditions of the filter are the column numbers of the left and right relations.
  - So in compareTupleTotuple, we compare the value of left and right column number and returns 0 if tags of these two nodes are same.
  - If this returns, we create a temp root and then call projection.join on the two tuples and the total number of fields between the two tuples which is then returned.
  - A printTreeFormat is used to print the results in a tree format. To identify parent child relationships we use the stack provided by this function

  - Node join i j : results of the first pattern tree joins with the results of the second pattern tree based on the IDs of i$^{\text{th}}$ and j$^{\text{th}}$ nodes of the two trees respectively
  - The same function is used as above but the arguments to the right filter are node number * 2 -1 which gives the column for the relation for tag name.
  - BaseTable attribute type and Rtypes are used to store the attribute type of each column of the relation.

- Sort
  - Results of the pattern tree are sorted based on the tag of i$^{\text{th}}$ node
  - Nested loop join with composite index is applied on the input pattern tree to get the iterator which is passed to sort physical operator along with node number.
  - Here we call the sort function where we pass number of fields of results of pattern tree and node number * 2 - 1 (to identify the tag as we sort by tag).
  - The result of the sort is an iterator of sort class and we call getNext for each tree as a result.
  - Here also getNext uses the printTreeFormat along with arguments - Basetable Attribute Types and number of fields of output tree to display results in the form of a tree.

- Group By
  - We use nested loop join with composite index function to retrieve the results using an iterator.
  - Implementation contains two functions groupPhysicalOperator and getNextGroup.
  - GroupPhysicalOperator: we use the number of fields of results of the iterator to create Basetable attribute types and the stringSizes array.
  - These parameters are used to call sort function along with arguments - iterator and column number calculated as node number * 2 - 1.
  - Result of the sort is an iterator that is stored in a class iterator variable and we call getNextGroup to fetch each group.
  - getNextGroup uses IoBuff class to store all tuples as we traverse through iterator to identify number of trees with similar tags.
  - We traverse this iterator to identify the number matching trees with same tag and first dissimilar one is put in a temp tuple.
  - getNextGroup is called and we check if temp tuple is not null and if not null we start grouping from that tuple.
  - Iterator.getNext returns a tree as a tuple. We check if this tree is same as the previous tree's tags and if so we push it into the buffer. We also get the number of field present in the resulting tuples returned by the iterator.
  - We traverse this iterator to identify the number matching trees with same tag and first dissimilar one is put in a temp tuple.
  - getNextGroup is called and we check if temp tuple is not null and if not null we start grouping from that tuple on wards.

  - To find the total number of fields of the resulting tuple we multiply the number of similar trees and number of fields in resulting tree.
  - We allocate 2 extra nodes for storing the header nodes to form a group structure.
  - Group root and column tag are the columns of the resulting tree.
  - All trees are inserted into IoBuff after which we perform a get on Iobuff which results in a tuple and values of each tuple are put into a Jtuple.
  - Jtuple.printTreeFormat is called to display results as a tree,
  - We initialize IoBuff by passing a 2D byte array, the size of tuple to be stored, a temporary heap file and number of files. Data is pushed to the heap file when the buffer is filled.

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 results have been attached in a separate file.

6 Conclusion

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

- Implementation of indexes have improved performance significantly as it reduced page size and processing time.
- Since composite indexes made up of tag name and interval is observed to prune more data than the other indexes and hence performs better.
- XML was parsed to obtain data in an ordered manner which prevented the need for sort before performing the sort merge join. Additionally it provided a non blocking version of sort merge.
- We found that sorting the results was leveraged in not only the sort complex operation but also other operators like group by.

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

7.1 Member Contribution

- Akhil Jayaraj M - Node tag Indexing and Phase 2 Completion
- Eldo Jacob Mathews - Composite Indexing and Complex Operators
- Mahathi Srinivasan - Phase 2 Completion and Buffer Management
- Meghna Prabhu - Composite Indexing and Complex Operators
- Renil Joseph - Interval Indexing and Complex Operators
- Vivek Menon - Phase 2 Completion and Complex Operators

8 Implementation - Phase 2

8.1 Query Processing Implementation

8.1.1 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.

8.1.2 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 **??**.
- 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 **??** 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.