CSE 232B : Database System and Implementation

Project Report : XQuery Processor

Team Members: Dhruv Sharma, Swapnil Taneja (Team #19)

XQuery Rewriter

We implemented the XQuery Rewriter as concrete child class extending the XQueryBaseVisitor (the abstract visitor class for our grammar) separate from the xquery evaluation class. The disjoint optimization and evaluation of query resulted in clean and easy to understand implementation in code.

Most rule implementations (such as xquery, absolute path, filters etc.) in the Rewriter class are trivial in nature and simply emit back a formatted string with the same elements as contained in the parsed query text. Majority of the rewriting logic, which detects and converts join-equivalent-for loop to explicit hash joins, resides in the rule implementation of the for clause (i.e. loop variable definitions, where clause and return statement).

We now discuss in detail, the implementation of join rewriting logic in the Rewriter. The class maintains a VariableTree instance that tracks the dependencies of the variables defined in the for statement along with the xqueries the variables bind during iterations. This data structure helps us in grouping the inter-dependent variables together under the same sub-tree. Each variable's node is attached as a child to the node of the variable it depends on its xquery expression. The independent variables, i.e. the ones binding to an absolute path, are attached as a child to root node that stands for the new '$tuple' variable.

Once the variables have been arranged in the tree hierarchy based on their dependencies, we know that variables under each sub-tree of the root represent one end of a join. We exploit the VariableTree to build another data structure 'HyperGraph' that helps us computing a join order between these groups of variables used in building join subexpressions of the rewritten query.

The HyperGraph structure build an undirected graph between HyperGraphNodes connected by HyperGraphEdges. Each HyperGraphNode contains the variables in a single independent variable group in VariableTree, whereas each HyperGraphEdge

maintains a list of join conditions between the two endpoint HyperGraphNodes of that edge with each join condition having one variable each belonging to the HyperGraphNodes. Any sub-conditions in the original where clause (such as the ones having one constant operand) are added as local conditions to HyperGraphNode which they are associated with.

Now, to compute a join order for these nodes in HyperGraph, we implemented a slightly modified version of Prim's minimum spanning tree algorithm. While for our use case, the HyperGraphEdges are unweighted so the algorithm results in some spanning tree of the HyperGraph. However, in real world these edges would be weighted by cost of join operation and the algorithm would require only a minor tweak to pick smallest edge first, giving us the minimum spanning tree i.e. cheapest join order - an interestingly desirable trait.

As in the Prim's algorithm, a HyperCluster starts with just one HyperGraphNode and expands along the neighboring HyperGraphEdges of the HyperCluster, merging and growing by one HyperGraphNode neighbor at a time (representing a single join operation) until it includes all HyperGraphNodes of the HyperGraph (i.e. full multi-way join). We also cover the case when we have disconnected HyperGraphNodes by adding such nodes to the HyperCluster. Such HyperNodes result in a cartesian product as they are missing join conditions (represented by HyperGraphEdge). The HyperCluster keeps updating an internal string representation for the join as it grows (i.e. each time neighbor HyperGraphNode and it's associated HyperGraphEdge merge with it). Finally, a simple toString() call to the HyperCluster returns the explicit join expression.

Hash Join Implementation

For optimizing joins in XQuery processor, we implemented a hashing mechanism for trees . We go ahead with the approach that hashing is done on the tuple nodes that have equality determined by the first variable in the join expression . For example, in the expression " join (xq1, xq2, [a1, b1] , [a2, b2]) " the nodes are by default hashed on the variables a1 and a2.

Now, in order to develop a hashing mechanism that guarantees uniqueness of the tree rooted at the node being hashed , we implemented a levelOrderHash . In this approach , we go through the nodes of the tree in a level Order Fashion and store three values corresponding to each node in a String. The three values are NodeName , NodeType and NodeValue . In addition, we also added levelMarkers which identify the beginning and end of a level (tree level). This approach guarantees that two trees having different structure will not hash into same hash value.

In order to implement multi way join we had to introduce recursive behavior of join keyword, hence we added that into our grammar . We, then implemented the join that recursively visits xq1 and xq2 to perform computations. Once the NodeLists of xq1 and xq2 are retrieved , we hash all the nodes of xq1 based on variable a1 and then join each node of xq2 . If the node of xq2 doesn't hash into an already existing hash, then that node is dropped as it will not contribute to the join. Hence, we end up saving a lot of computation.

In some of the queries time improvement was in the orders of magnitude .
For example, we ran the demo query –

```
for $act in doc("j_caesar.xml")//ACT,
    $title in $act/SCENE/TITLE,
    $speaker1 in $act//SPEAKER/text(),
    $speaker2 in $act//SPEAKER/text(),
    $pg in doc("j_caesar.xml")//PGROUP,
    $persona1 in $pg/PERSONA/text(),
    $persona2 in $pg/PERSONA/text()
where $speaker1 eq $persona1 and $speaker2 eq $persona2
return <tuple>{ <scene>{$title/text()}</scene>, <persona1>{$persona1}</persona1>,
<persona2>{$persona2}</persona2> }</tuple>
```

With Hash Join our query ran in 367 milliseconds whereas from the normal nested loop join the query ran for 446 milliseconds .