Optimization in Quark System

Fan Yang
12th April 2005

Computer Science Department
Cornell University
Copyright (C) 2003-2005 Cornell University. All Rights Reserved.

Contents
1 Introduction 1
2 General Optimization process 2
3 Storage Query Unit (SQU) 2
4 Optimizer using SQU 3

1 Introduction

This document describes the implementation and design of the Optimizer component of the Quark System. The main design goal of the optimization part is to be flexible enough to easily incorporate new storage, index and query primitives.

Unlike in relational database management systems, query processing techniques in native XML databases are far from mature. Many native XML database systems (e.g., Lore, Tamino, Natix, Timber and ToX) emerge in recent years in academic and industrial communities and along with them come new storage, indexing, and query optimization techniques. The booming of these techniques not only introduce benefits but also problems to compare and integrate them. As a very complicated component, query optimizer appears to be the hardest part to modify to incorporate new techniques. Optimizers of those systems differ dramatically with each other and only work with techniques introduced by those particular systems. Since the algebra, cost model, access methods etc. of XML query processing are still not well understood, we expect the technique included
in those optimizers will keep changing, which makes an extensible query optimizer more
critical to incorporate and compare new techniques.

We consider that the optimization framework, among other things, should be extensible
in four aspects as follows. The technique of those aspects shouldn’t be hard-coded in the
optimizer. 1) Storage format. XML data can be stored in various formats (e.g. relational
table, native or document). Every format has their own strength and are good at answering
certain types of queries. For example, Native is good at navigation and update and bad at
complex twig matching compared with path index in form of relational table. We expect
different format coexists in one system and the same data can be replicated and stored in
different formats. 2) Index. Unlike for relational table, indices for XML data need to keep
track of structure as well as value information. Different indices (e.g. Rootpath, Datapath)
will also result in very different plans to do twig matching. 3) Query primitives. Up to
now, every XML database has their own set of logical operations. Unlike relational model,
there is no consensus of an algebra up to now. We expect new type of query primitives (e.g.
structural join, holistic join) will present themselves in the near future. 4) Materialized view.
Materialized views if made use of properly can speed up query processing dramatically.
Optimizer should easily detect and use them whenever possible.

The optimizer component is composed of several directories: value_format, format_transformer,
value_format_xml_std, optimizer, optimizer_squ, squ and squ_default.

2 General Optimization process

The optimization process is like below graph. First the xquery will be translated into YQGM
graph, then we good at each subgraph from bottom up, call SQU (Storage Query Unit) to
provide candidate execution plan for each subgraph along with the approximate cost for
that plan. Optimizer will then choose the optimal plan for each subgraph using a dynamic
programming algorithm and the final optimal plan for the whole query is got by stitching
together plans chosen for subgraphs.

3 Storage Query Unit (SQU)

SQU encapsulate methods and information about storage format, query primitive and index
built on a fragment of XML data.

The SQU supports public interface

- virtual list<SubPlan*> getCandidates(MemoryContext* YQGMMemCtx, Node* currentNODE, IDGenerator* idGen);

It takes in the memory context of execution, current node we are looking at and a ID
generator for yqgm graph.

For each node (root of a subtree in yqgm graph), the squ will return a list of interesting
execution plans based on the techniques this SQU introduced into the system. So to add
new storage format, index or query primitive, we add a new subclass of SQU and override this function.

4 Optimizer using SQU

The optimizer use a DP algorithm to visit each node (and the subgraph under this node) in the yqgm graph, call squ to provide possible execution plan and chose the best one based on cost.

Figure 2 and 3 shows the algorithm. Storage Query Unit (SQU) is in charge of providing subplans which can answer part of the query. It supports the interface getCandidatePlans(nodeToCover, subPlanWeHave) and transformID(inputType, outputType). getCandidatePlans takes nodeToCover which is the root of subtree to be covered and subPlanWeHave which are subplans generated up to now as argument and generate subplans which covers the subtree rooted at nodeToCover. transformID provides the subplans which transform value of type inputType to type outputType. The optimizer visits every operators in YQGM bottom up in reversed topological order. When visiting an operators, all operators serve as the input of this operator have all been visited and covered by optimal SubQuery. Since different indices and primitives work with different value format (e.g. DIL works with dewey ID and structural joins need startend ID), value format transformation is needed. We have a default SQU which can provide plans cover every single operator. So our algorithm

3
can guarantee to produce plans covering the whole YQGM.
procedure GetOptPlan(InXQGMGraph, SQUList, IDTypeList) return plan
//InXQGMGraph: the XQGM of the query
//SQUList: all SQU in the system
//IDTypeList: ID types of the output we consider

//initialize the list of SubPlan generated
plansGenerated=empty

//get the list of operators in reverse topological order
nodeList=ReverseTopologicalSort(InXQGMGraph)

//when we visit operator currentNode, operators in //the subtree rooted at currentNode have been visited.
foreach curOp in nodeList
  //initialize the list of SubPlan newly generated
  plansNewlyGenerated=empty

  foreach SQU in SQUList
    plansNewlyGenerated+=SQU.getCandidate(currentNode, plansGenerated)
  end for

//prune the plans provided by graph transformer
plansPruned=prune(plansNewlyGenerated)
optCosts=MaximinNums;
optPlan=empty
foreach newSubPlan in plansPruned
  //idType transformation
  IDTypeTransform(newSubPlan, SQUList)
  costForSubTree=newSubPlan.cost + newSubPlan.inputCost
  if (costForSubTree<optCost[newSubPlan.outputIDtype])
    optCost[newSubPlan.outputIDtype]=costForSubTree
    optSub[newSubPlan.outputIDtype]=newSubPlan
  end if
end for //loop over pruned plans

if (currentNode is the top Operator)
  return OptPlans;
end if
foreach IDType in IDTypeList
  plansNewlyGenerated+=OptPlans[IDType]
end for //loop over all operators in XQGM

Figure 2: DP Algorithm for Optimization
procedure IDTransform(plan, SQUList)
//plan: the subplan we want to transform input ID type
//SQUList: all SQU in the system.

//get input plan list and ID type list
inputIDTypeList = newSubPlan.getInputIDType()
inputPlanList = newSubPlan.getInputPlan()
inNum = newSubPlan.getInputNum()

for all i such that 0 ≤ i < inNum
    if (inputPlanList[i].outputIDType != inputIDTypeList[i]){
        optTransformCost = MaximumNumber
        optTransformPlan = NULL;
        foreach SQU in SQUList
            transformPlan = SQU.transform(inputPlanList[i].outputIDType, inputIDTypeList[i])
            if (transformPlan != NULL && transformPlan.cost < optTransformCost)
                optTransformCost = transformPlan.cost()
                optTransformPlan = transformPlan
        end if
    end for //loop over list of SQU

//set the input
optTransformPlan.setInputPlan(inputPlanList[i], 0)
//set the ith input
newSubPlan.setInputPlan(optTransformPlan, i)
newSubPlan.inputCost = optTransformCost
end if //if need transformation
end for //loop over all inputs

Figure 3: Algorithm for value format transformation

process.ps