QUERY OPTIMIZATION FOR DATABASE MANAGEMENT SYSTEM BY APPLYING DYNAMIC PROGRAMMING ALGORITHM

Wisnu Adityo – NIM 13506029

Information Technology Department Institut Teknologi Bandung
Jalan Ganesha 10
e-mail: if16029@students.if.itb.ac.id

ABSTRACT

This paper was made to inform my research about implementation of dynamic programming algorithm. Discussion of this paper is emphasized on optimizing query for Database Management System.

DBMS (*Database Management System*) is an application that use query to execute command (to access database), with SQL (Structured Query Language).

Given a query, there are many plans that a database management system (DBMS) can follow to process it and produce its answer. All plans are equivalent in terms of their final output but vary in their cost, i.e., the amount of time that they need to run. What is the plan that needs the least amount of time?

The cost difference between two alternatives can be enormous. For it is given five solution to execute an execution plan, each with their own running time, that varies from 0.32 seconds to more than a whole day. Subsequently, Such query optimization is absolutely necessary in a DBMS, to prevent the unwanted solution and select the right, efficient solution.

With the implementation of dynamic programming algorithm, the query optimization can be done by filterng those plans and produce the most efficient plan to be executed.

Keyword **:** dynamic programming, DBMS, query optimization.

1. INTRODUCTION

Dynamic programming was first proposed as a query optimization search strategy in the context of System R by Selinger et al. Commercial systems have then used it in various forms and with various extensions. We present this algorithm pretty much in its original form, only ignoring details that do not arise in at SQL queries, which are our focus.

2. DYNAMIC PROGRAMMING ALGORITHM FOR QUERY OPTIMIZATION

In this chapter, I will explain how dynamic programming could make all plans that are going to be executed by DBMS, efficient. But, I will first explain how queries are executed and query optimization for DBMS.

2.1 Steps of query processing

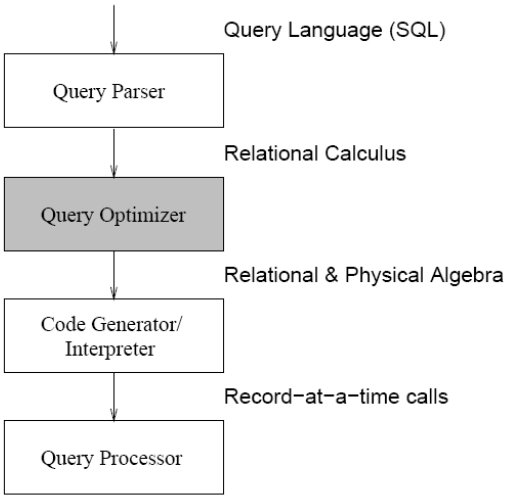

Figure 1. Query flow through DBMS

The path that a query traverses through a DBMS until its answer is generated is shown in Figure 1. The system modules through which it moves have the following functionality:

- The Query Parser checks the validity of the query and then translates it into an internal form, usually a relational calculus expression or something equivalent.
- The Query Optimizer examines all algebraic expressions that are equivalent to the given query

and chooses the one that is estimated to be the cheapest.

- The Code Generator or the Interpreter transforms the access plan generated by the optimizer into calls to the query processor.
- The Query Processor actually executes the query.

2.2 Steps of query optimisator

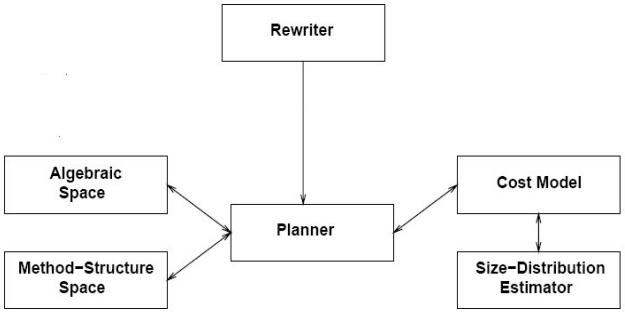

Figure 2. Query optimizator architecture

The path of query optimization is shown in Figure 2. Ech modules have the following functionality :

- Rewriter: This module applies transformations to a given query and produces equivalent queries that are hopefully more effcient, e.g., replacement of views with their de_nition, attening out of
  nested queries, etc

- Planner: This is the main module of the ordering stage. It examines all possible execution plans for each query produced in the previous stage and selects the overall cheapest one to be usedto generate the answer of the original query.

- Algebraic Space: This module determines the action execution orders that are to be considered by the Planner for each query sent to it.

- Method-Structure Space: This module determines the implementation choices that exist for the execution of each ordered series of actions specified by the Algebraic Space.

- Cost Model: This module specifies the arithmetic formulas that are used to estimate the cost of execution plans.

- Size-Distribution Estimator: This module specifies how the sizes (and possibly frequency distributions of attribute values) of database relations and indices as well as (sub)query results are estimated.

All such series of actions produce the same query answer, but usually differ in performance. They are usually represented in relational algebra as formulas or in tree form.

Planner employs a search strategy, which examines the space of execution plans in a particular fashion. This space is determined by two other modules of the optimizer, the Algebraic Space and the Method-Structure Space. For the most part, these two modules and the search strategy determine the cost, i.e., running time, of the optimizer itself, which should be as low as possible. The execution plans examined by the Planner are compared based on estimates of their cost so that the cheapest may be chosen. These costs are derived by the last two modules of the optimizer, the Cost Model and the Size-Distribution Estimator.

2.3 Typical Space Restriction

A flat SQL query corresponds to a select-project-join query in relational algebra. Typically, such an algebraic query is represented by a query tree whose leaves are database relations and non-leaf nodes are algebraic operators like selections (denoted by σ), projections (denoted by π), and joins (denoted by ∞). An intermediate node indicates the application of the corresponding operator on the relations generated by its children, the result of which is then sent further up. Thus, the edges of a tree represent data ow from bottom to top, i.e., from the leaves, which correspond to data in the database, to the root, which is the final operator producing the query answer

Consider the following database schema, which will be used throughout this paper:
emp(name,age,sal,dno)
dept(dno,dname,oor,budget,mgr,ano)
acnt(ano,type,balance,bno)
bank(bno,bname,address)

the schema above initiate and execute four tables (emp. dept, acnt and bank) consist of each own category (emp has four category, which is name, age, sal, dno).

Figure 3 gives three examples of query trees for the query
select name, floor
from emp, dept
where emp.dno=dept.dno and sal>100K .

the query above select the name and floor category from emp and dept table. It also restrict the emp.dno to be identical with dept.dno and demand sal above 100 K.

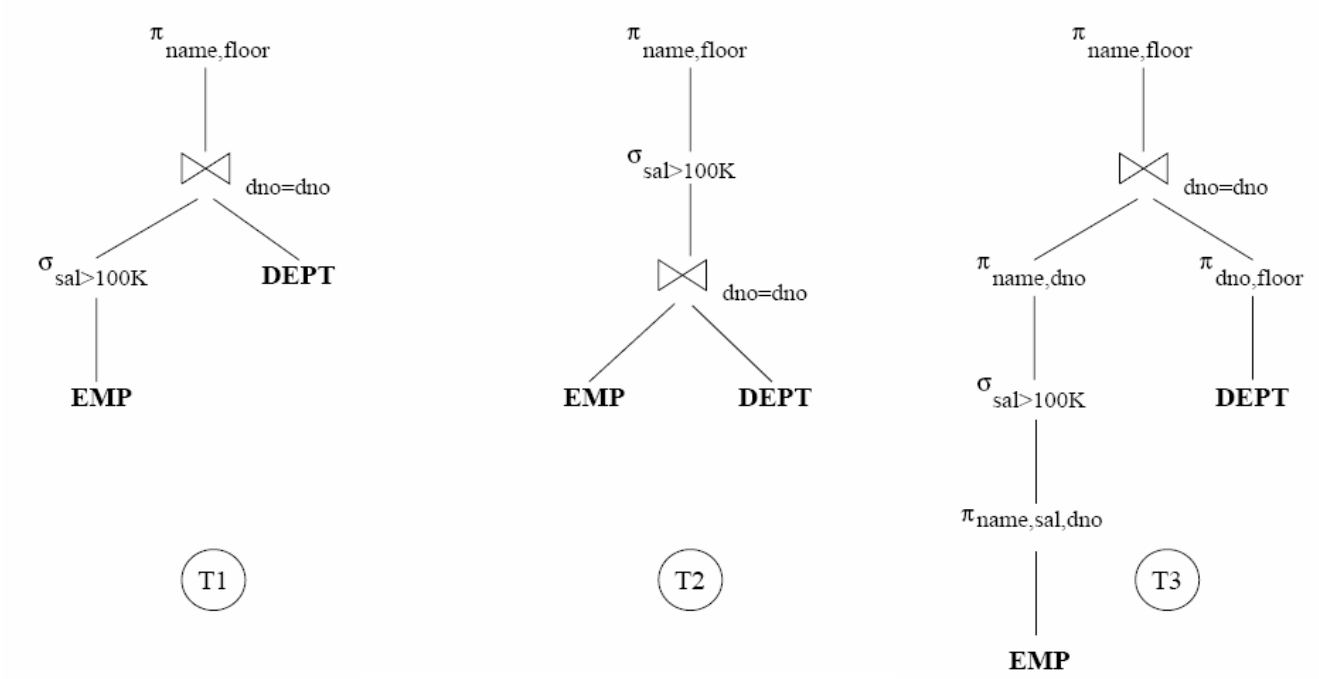

Figure 3 examples of general query trees

For a complicated query, the number of all query trees may be enormous. To reduce the size of the space that the search strategy has to explore, DBMSs usually restrict the space in several ways. The first typical restriction deals with selections and projections:

R1 : Selections and projections are processed on the fly and almost never generate intermediate relations. Selections are processed as relations are accessed for the first time.Projections are processed as the results of other operators are generated.

In particular, the second typical restriction deals with cross products:

R2 : Cross products are never formed, unless the query itself asks for them. Relations are combined always through joins in the query.

In particular, the third typical restriction deals with the shape of join trees :

R3 : The inner operand of each join is a database relation, never an intermediate result.

Typical query optimizers make restrictions R1, R2, and R3 to reduce the size of the space they explore.

2.4 Applying Dynamic Programming Algorithm In Planner's Search Strategy

The algorithm is essentially a dynamically pruning, exhaustive search algorithm. It constructs all alternative join trees (that satisfy restrictions R1-R3) by iterating on the number of relations joined so far, always pruning trees that are known to be suboptimal. Before we present the algorithm in detail, we need to discuss the issue of interesting order. One of the join methods that is usually specified by the Method-Structure Space module is merge scan. Merge scan first sorts the two input relations on the corresponding join attributes and then merges them with a synchronized scan. If any of the input relations, however, is already sorted on its join attribute (e.g., because of earlier use of a B+-tree index or sorting as part of an earlier merge-scan join), the sorting step can be skipped for the relation. Hence, given two partial plans during query optimization, one cannot compare them based on their cost only and prune the more expensive one; one has to also take into account the sorted order (if any) in which their result comes out. One of the plans may be more expensive but may generate its result sorted on an attribute that will save a sort in a subsequent merge-scan execution of a join. To take into account these possibilities, given a query, one defines its interesting orders to be orders of intermediate results on any relation attributes that participate in joins. (For more general SQL queries, attributes in order-by and group-by clauses give rise to interesting orders as well.) For example, in the query of , orders on the attributes emp.dno, dept.dno, dept.ano, acnt.ano, acnt.bno, and bank.bno are interesting. During optimization of this query, if any intermediate result comes out sorted on any of these attributes, then the partial plan that gave this result must be treated specially.

Using the above, we give below a detailed English description of the dynamic programming algorithm optimizing a query of N relations:

Step 1 For each relation in the query, all possible ways to access it, i.e., via all existing indices and including the simple sequential scan, are obtained. (Accessing an index takes into account any query selection on the index key attribute.) These partial (single-relation) plans are partitioned into equivalence classes based on any interesting order in which they produce their result. An additional equivalence class is formed by the partial plans whose results are in no interesting order. Estimates of the costs of all plans are obtained from the Cost Model module, and the cheapest plan in each equivalence class is retained for further consideration. However, the cheapest plan of the no-order equivalence class is not retained if it is not cheaper than all other plans.

Step 2 For each pair of relations joined in the query, all possible ways to evaluate their join using all relation access plans retained after Step 1 are obtained. Partitioning and pruning of these partial (two-relation) plans proceeds as above.
: :

Step i For each set of i - 1 relations joined in the query, the cheapest plans to join them for each interesting order are known from the previous step. In this step, for each such set, all possible ways to join one more relation with it without creating a cross product are evaluated. For each set of i relations, all generated (partial) plans are partitioned and pruned as before.
: : :

Step N All possible plans to answer the query (the unique set of N relations joined in the query) are generated from the plans retained in the previous step. The cheapest plan is the final output of the optimizer, to be used to process the query. For a given query, the above algorithm is guaranteed to find the optimal plan among those satisfying restrictions R1-R3. It often avoids enumerating all plans in the space by being able to dynamically prune suboptimal parts of the space as partial plans are generated. In fact, although in general still exponential, there are query forms for which it only generates O(N3) plans.

2.5 Case Study

An example that shows dynamic programming in its full detail takes too much space. We illustrate its basic mechanism by showing how it would proceed on the simple query below:
select name, mgr
from emp, dept
where emp.dno=dept.dno and sal>30K and oor=2

Assume that there is a B+-tree index on emp.sal, a B+-tree index on emp.dno, and a hashing index on dept.oor. Also assume that the DBMS supports two join methods, nested loops and merge scan. (Both types of information should be specifed in the Method-Structure Space module.) Note that, based on the defnition, potential interesting orders are those on emp.dno and dept.dno, since these are the only join attributes in the query. The algorithm proceeds as follows:

Step 1: All possible ways to access emp and dept are found. The only interesting order arises from accessing emp via the B+-tree on emp.dno, which generates the emp tuples sorted and ready for the join with dept. The entire set of alternatives, appropriately partitioned are shown in the table below.

Relation	Interesting Order	Plan Description	Cost
emp	emp.dno	Access through B+-tree on emp.dno.	700
	–	Access through B+-tree on emp.sal.	200
		Sequential scan.	600
dept	–	Access through hashing on dept.floor.	50
		Sequential scan.	200

Each partial plan is associated with some hypothetical cost; in reality, these costs are obtained from the Cost Model module. Within each equivalence class,

only the cheapest plan is retained for the next step, as indicated by the boxes surrounding the corresponding costs in the table.

Step 2: Since the query has two relations, this is the last step of the algorithm. All possible ways to join emp and dept are found, using both supported join methods and all partial plans for individual relation access retained from Step 1. For the nested loops method, which relation is inner and which is outer is also specifed. Since this is the last step of the algorithm, there is no issue of interesting orders. The entire set of alternatives is shown in the table below in a way similar to Step 1. Based on hypothetical costs for each of the plans, the optimizer produces as output the plan indicated by the box surrounding the corresponding cost in the table.

Join Method	Outer/Inner	Plan Description	Cost
nested loops	emp/dept	• For each emp tuple obtained through the B+-tree on emp.sal, scan dept through the hashing index on dept.floor to find tuples matching on dno.	1800
		• For each emp tuple obtained through the B+-tree on emp.dno and satisfying the selection on emp.sal, scan dept through the hashing index on dept.floor to find tuples matching on dno.	3000
	dept/emp	• For each dept tuple obtained through the hashing index on dept.floor, scan emp through the B+-tree on emp.sal to find tuples matching on dno.	2500
		• For each dept tuple obtained through the hashing index on dept.floor, probe emp through the B+-tree on emp.dno using the value in dept.dno to find tuples satisfying the selection on emp.sal.	1500
merge scan	–	• Sort the emp tuples resulting from accessing the B+-tree on emp.sal into L_1. • Sort the dept tuples resulting from accessing the hashing index on dept.floor into L_2. • Merge L_1 and L_2.	2300
		• Sort the dept tuples resulting from accessing the hashing index on dept.floor into L_2. • Merge L_2 and the emp tuples resulting from accessing the B+-tree on emp.dno and satisfying the selection on emp.sal.	2000

3. SUMMARY

As the above example illustrates, the choices offered by the Method-Structure Space in addition to those of the Algebraic Space result in an extraordinary number of alternatives that the optimizer must search through. The memory requirements and running time of dynamic programming grow exponentially with query size (i.e., number of joins) in the worst case since all viable partial plans generated in each step must be stored to be used in the next one. In fact, many modern systems place a limit on the size of queries that can be submitted (usually around fifteen joins), because for larger queries the optimizer crashes due to its very high memory requirements. Nevertheless, most queries seen in practice involve less than ten joins, and the algorithm has proved to be very effective in such contexts. So, it is considered successful in query optimizing search strategies for DBMS, thus optimize the query itself.

REFERENCE

[1] http://blogs.msdn.com/ericlippert/default.aspx accessed 17 May 2008 18.45

[2] http://www.cs.nyu.edu/courses/spring06/queryopt.ppt accessed 17 Mei 2008 19.24

[3] http://www.cs.rutgers.edu/%7Emuthu/vishy.ppt accessed 17 May 2008 22.09