Journal of Advances and Scholarly Researches in Allied Education

Vol. VII, Issue No. XIII, January-2014, ISSN 2230-7540

QUERY OPTIMIZATION: COST-BASED OPTIMIZATION

AN INTERNATIONALLY INDEXED PEER REVIEWED & REFEREED JOURNAL

www.ignited.in

Journal of Advances and Scholarly Researches in Allied Education
Vol. VII, Issue No. XIII, January-2014, ISSN 2230-7540

Query Optimization: Cost-based Optimization

Deepti Khanna[1] Priyanka Jha[2] Prof (Dr.) V B Aggarwal[3]

[1]Associate Professor, JIMS

[2]Assistant Professor, JIMS

[3]Director, JIMS

Abstract – Distributed query processing is fast becoming a reality. With the new emerging applications such as the grid applications, distributed data processing becomes a complex undertaking due to the changes coming from both underlying networks and the requirements of grid-enabled databases. Recent database research has demonstrated that memory access is more and more becoming a significant—if not the major—cost component of database operations

In this article, we propose a generic technique to create accurate cost functions for database operations. The method of optimizing the query by choosing a strategy that results in minimum cost i.e. Cost based Query Optimization. We calculate the cost of executing the different alternatives .The cost of executing a query include the following components: secondary storage access cost, storage cost, computation cost, memory usage cost, communication cost.

We identify a few basic memory access patterns and provide cost functions that estimate their access costs for each level of the memory hierarchy. The cost functions are parameterized to accommodate various hardware characteristics appropriately. Database processes queries, Processing Selection Queries, Processing Projection Queries and Eliminating Duplicates, Processing Join Queries: Two plans have the same cost through Improvement - block nested loops join, indexed nested loops join, sort-merge join, hash join, Query Plans and Query Optimization for Complex Relational Expression.

Keywords: Cost Function, Linear search, Binary Search.

----------X----------

INTRODUCTION

Query optimization is a function of many relational database management systems in which multiple query plans for satisfying a query are examined and a good query plan is identified. This may or not be the absolute best strategy because there are many ways of doing plans. There is a tradeoff between the amount of time spent figuring out the best plan and the amount running the plan. Different qualities of database management systems have different ways of balancing these two. Cost based query optimizers evaluate the resource footprint of various query plans and use this as the basis for plan selection. Typically the resources which are cost are CPU path length, amount of disk buffer space, disk storage service time, and interconnect usage between units of parallelism. The set of query plans examined is formed by examining possible access paths (e.g., primary index access, secondary index access, full file scan) and various relational table join techniques (e.g., merge join, hash join, product join). The search space can become quite large depending on the complexity of the SQL query. There are two types of optimization. These consist of logical optimization which generates a sequence of relational algebra to solve the query. In addition there is physical optimization which is used to determine the means of carrying out each operation.

The query optimizer is the component of a database management system that attempts to determine the most efficient way to execute a query. The optimizer considers the possible query plans for a given input query, and attempts to determine which of those plans will be the most efficient. Cost-based query optimizers assign an estimated "cost" to each possible query plan, and choose the plan with the smallest cost. Costs are used to estimate the runtime cost of evaluating the query, in terms of the number of I/O operations required, the CPU requirements, and other factors determined from the data dictionary. The set of query plans examined is formed by examining the possible access paths (e.g. index scan, sequential scan) and join algorithms (e.g. sort-merge join, hash join, nested loops). The search space can become quite large depending on the complexity of the SQL query.

www.ignited.in

Generally, the query optimizer cannot be accessed directly by users: once queries are submitted to database server, and parsed by the parser, they are then passed to the query optimizer where optimization occurs. However, some database engines allow guiding the query optimizer with hints

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?

Such query optimization is absolutely necessary in a DBMS. The cost difference between two alternatives can be enormous. For example, consider the following database schema, which will be used throughout this article:

emp(name,age,sal,dno)

dept(dno,dname,oor,budget,mgr,ano)

acnt(ano,type,balance,bno)

bank(bno,bname,address)

Further, consider the following very simple SQL query:

select e.name, e.floor from emp e, dept d where e.dno=d.dno and e.sal>100K and d.dname='hr'.

Assume the characteristics below for the database contents, structure, and run-time environment:

Parameter Description	Parameter Value
Number of emp pages	20000
Number of emp tuples	100000
Number of emp tuples with sal>	1000K 5
Number of dept pages	10
Number of dept tuples	100
Number of buffer pages	3
Cost of one disk page access	20ms

Consider the following three different plans:

P1 : Through the B+-tree and all tuples of emp that satisfy the selection on e.sal and d.dname is HR. For each one, use the hashing index to find the corresponding dept tuples.

P2 : For each dept page, scan the entire emp relation. If an emp tuple agrees on the dno and dname attributes with a tuple on the dept page and satisfies the selection on e.sal, then the emp-dept tuple pair appears in the result. Join should be included.

P3: For each dept tuple, scan the entire emp relation and store all emp-dept tuple pairs.

Then, scan this set of pairs and, for each one, check if it has the same values in the two dno attributes and satisfies the selection on e.sal and d.dname is HR.

Optimizing parameters under multiple constraints and negotiating compromises between different objectives has a long history in economic problems. Though simplifying approaches often reduce business decisions to 'maximize profits', common problems often deal with non-monetary intangibles like product quality, public image, tradition, corporate identity or ethics like environmental concerns or safety features. But apart from mere business problems multi-objective optimization also plays a role in many areas of computer science:

- Multi-objective agents negotiate compromises on behalf of different users or interest groups

- Decision support systems try to integrate various interests to recommend strategic decisions

- Trade-offs in e-commerce environments e.g. between price, efficiency and quality of certain products have to be assessed

- Personal preferences of users requesting a Web service for a complex task have to be evaluated to select most appropriate services

Recent advances in computing technology have led to the production of a new class of computing devices: the wireless, battery-powered, smart sensor. Traditional sensors deployed throughout buildings, labs, and equipment is passive devices that simply modulate a voltage on the basis of some environmental parameter. These new sensors are active, full-fledged computers, capable of not only sampling real-world phenomena but also filtering, sharing, and combining sensor readings with each other and nearby Internet-equipped end points.

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

The code produced by the Code Generator is stored in the database and is simply invoked and executed by the Query Processor whenever control reaches that query during the program execution (run time). Thus, independent of the number of times an embedded query needs to be executed, optimization is not repeated until database updates make the access plan invalid (e.g., index deletion) or highly suboptimal (e.g., extensive changes in database contents).

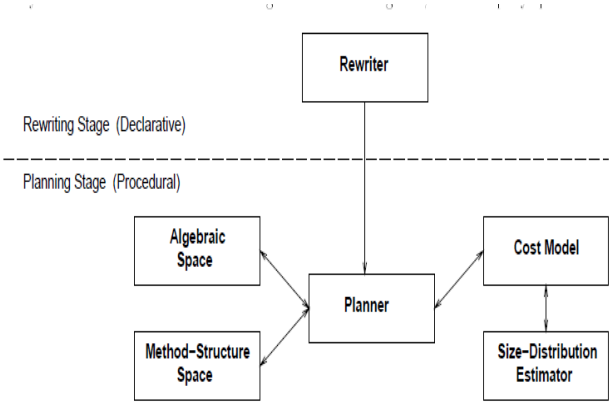
The area of query optimization is very large within the database field. It has been studied in a great variety of contexts and from many different angles, giving rise to several diverse solutions in each case.

The purpose of this chapter is to primarily discuss the core problems in query optimization and their solutions, and only touch upon the wealth of results that exist beyond that. More specifically, we concentrate on optimizing a single at SQL query with `and' as the only Boolean connective in its qualification (also known as conjunctive query, select-project-join query, or nonrecursive Horn clause) in a centralized relational DBMS, assuming that full knowledge of the run-time environment exists at compile time.

II. QUERY OPTIMIZER ARCHITECTURE

We provide an abstraction of the query optimization process in a DBMS. Given a database and a query on it, several execution plans exist that can be employed to answer the query. In principle, all the alternatives need to be considered so that the one with the best estimated performance is chosen.

An abstraction of the process of generating and testing these alternatives is shown in Fig, which is essentially a modular architecture of a query optimizer. Although one could build an optimizer based on this architecture, in real systems, the modules shown do not always have so clear-cut boundaries as in Fig. Based on Fig, the entire query optimization process can be seen as having two stages: rewriting and planning. There is only one module in the first stage, the Rewriter, whereas all other modules are in the second stage. The functionality of each of the modules in Fig is analyzed below.



MODULE FUNCTIONALITY

Rewriter: This module applies transformations to a given query and produces equivalent queries that are hopefully more efficient, e.g., replacement of views with their definition, attending out of nested queries, etc. The transformations performed by the Rewriter depend only on the declarative, i.e., static, characteristics of queries and do not take into account the actual query costs for the specific DBMS and database concerned. If the rewriting is known or assumed to always be beneficial, the original query is discarded; otherwise, it is sent to the next stage as well. By the nature of the rewriting transformations, this stage operates at the declarative level.

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 used to generate the answer of the original query. It 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.

Algebraic Space: This module determines the action execution orders that are to be considered by the Planner for each query sent to it. 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. Because of the algorithmic nature of the objects generated by this module and sent to the Planner, the

overall planning stage is characterized as operating at the procedural level.

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. This choice is related to the available join methods for each join. if supporting data structures are built on the y, if/when duplicates are eliminated, and other implementation characteristics of this sort, which are predetermined by the DBMS implementation.

This choice is also related to the available indices for accessing each relation, which is determined by the physical schema of each database stored in its catalogs. Given an algebraic formula or tree from the Algebraic Space, this module produces all corresponding complete execution plans, which specify the implementation of each algebraic operator and the use of any indices.

Cost Model: This module specifies the arithmetic formulas that are used to estimate the cost of execution plans. For every different join method, for every different index type access, and in general for every distinct kind of step that can be found in an execution plan, there is a formula that gives its cost. Given the complexity of many of these steps, most of these formulas are simple approximations of what the system actually does and are based on certain assumptions regarding issues like buffer management, disk-cpu overlap, sequential vs. random I/O, etc. The most important input parameters to a formula are the size of the buffer pool used by the corresponding step, the sizes of relations or indices accessed, and possibly various distributions of values in these relations. While the first one is determined by the DBMS for each query, the other two are estimated by the Size-Distribution Estimator.

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. As mentioned above, these estimates are needed by the Cost Model.

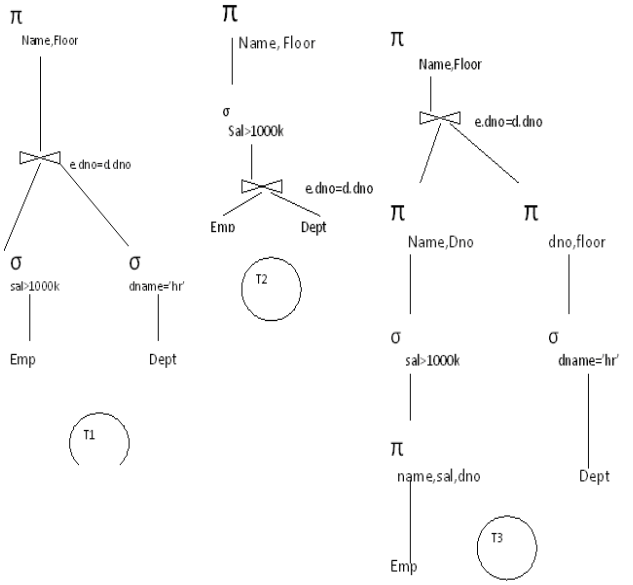
DESCRIPTION FOCUS

Most of the transformations normally performed by this module are considered an advanced form of query optimization, and not part of the core (planning) process. The Method-Structure Space specifies alternatives regarding join methods, indices, etc., which are based on decisions made outside the development of the query optimizer and do not really affect much of the rest of it. For the Cost

Model, for each alternative join method, index access, etc., offered by the Method-Structure Space, either there is a standard straightforward formula that people have devised by simple accounting of the corresponding actions (e.g., the formula for tuple-level nested loops join) or there are numerous variations of formulas that people have proposed and used to approximate these actions (e.g., formulas for finding the tuples in a relation having a random value in an attribute). In either case, the derivation of these formulas is not considered an intrinsic part of the query optimization field.

ALGEBRAIC SPACE

As mentioned above, a at 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 ∂), projections (denoted by ∏), and joins1 (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 flow 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. Fig gives three examples of query trees for the query select name, floor from emp, dept where emp.dno=dept.dno and sal>1000K .



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 y 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.

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

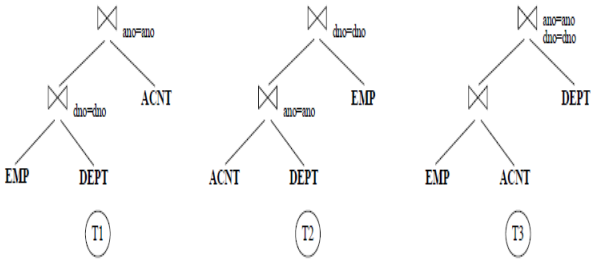
www.ignited.in

through joins in the query. For example, consider the following query:

Select name, floor, balance from emp, dept, acnt where emp.dno=dept.dno and dept.ano=acnt.ano

Fig shows the three possible join trees (modulo join commutativity) that can be used to

Combine the emp, dept, and acnt relations to answer the query. Of the three trees in the figure, tree T3 has a cross product, since its lower join involves relations emp and acnt, which are not 1. Explicitly joined in the query. Restriction R2 almost always eliminates suboptimal join trees, due to the large size of the results typically generated by cross products. The exceptions are very few and are cases where the relations forming cross products are extremely small. Hence, the Algebraic Space module specifies alternative join trees that involve no cross product.



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

For example, consider the following query:

Select name, floor, balance, address from emp, dept, acnt, bank where emp.dno=dept.dno and dept.ano=acnt.ano and acnt.bno=bank.bno

The typical arguments used are two:

¥ Having original database relations as inners increases the use of any preexisting indices.

¥ Having intermediate relations as outers allows sequences of nested loops joins to be executed in a pipelined fashion

Both index usage and pipelining reduce the cost of join trees. Moreover, restriction R3 significantly reduces the number of alternative join trees. Hence, the Algebraic Space module of the typical query optimizer only specifies join trees that are left-deep. Summary, typical query optimizers make restrictions R1, R2, and R3 to reduce the size of the space they explore. Hence, unless otherwise noted, our descriptions follow these restrictions as well.

III. CONCLUSION

The study is being taking into consideration that the different parameters which is been mentioned in the objective of my study. It includes both descriptive and empirical study and the conclusions drawn are having far reaching implications. On the basis of the study we had concluded:

- The cost of first_row and all_rows is same in all two versions of oracle.
- In case of estimated statistics the exact the match has the different number of rows and bytes.
- In oracle 10g, estimated statistics and computed statistics shows the different results.
- In oracle 10g, the cost is less than the oracle 9i but slightly greater than that of oracle 8i.
- In case of non-indexed table, CPU time and elapsed time in oracle 10g is lesser than that of oracle 8i and oracle 9i but the CPU time is inverse.
- In case of indexed table, CPU and elapsed time in oracle 8i and oracle 9i is lesser than that of oracle 10g.
- In case of bitmapped indexed table in oracle 9i, elapsed time is lesser than that of normal indexed table but the CPU time is inverse.

REFERENCES

1. Ioannidis, Yannis (March 1996). "Query optimization". ACM Computing Surveys 28 (1): 121–123. doi:10.1145/234313.234367. http://citeseer.ist.psu.edu/487912.html.

2. Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G. (1979), "Access Path Selection in a Relational Database Management System", Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, pp. 23-34, doi:10.1145/582095.582099

3. *Chaudhuri, Surajit (1998). "An Overview of Query Optimization in Relational Systems". Proceedings of the ACM Symposium on Principles of Database Systems: pages 34–43. doi:10.1145/275487.275492.*

*4. Ioannidis, Yannis (March 1996). "Query optimization". ACM Computing Surveys 28 (1): 121–123. doi:*10.1145/234313.234367. http://citeseer.ist.psu.edu/487912.html.

5. Selinger, P. Ge.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G. (1979), "Access Path Selection in a Relational Database Management System", Proceedings of the 1979 ACM

*SIGMOD International Conference on Management of Data, pp. 23-34, doi:*10.1145/582095.582099

6. [DB01] Desai, b c, An Introduction to database system, 2000

7. [EN01] Elmasri, E and Navathe, Fundamental of Database System, 2004

8. [AE01] Aronoff, eyal,et, Oracle 8 Advance Tuning and Administration, Oracle press.

9. http://www.oracle.com

10. http://www.dba-oracle.com/art-otn-cbo.htm

11. http://www.oracle-base.com

12. http://www.courses.csus.edu

13. http://whitepapers.techrepublic.com