International Journal of Research in Advanced Engineering and Technology
ISSN: 2455-0876; Impact Factor: RJIF 5.44
www.engineeringresearchjournal.com
Volume 1; Issue 2; November 2015; Page No. 115-119

A study on query optimization in generic RDBMS

Gurjeet singh

Asst. Professor, Trai Shatabdi Guru Gobind Singh Khalsa College, Amritsar, India

Abstract
Most personal or industrial data is simply stored in files and accessed via simple programs. This approach works well for small applications but is not generic and does not scale. New applications require new software, and classical software can hardly cope with huge data sets. Database management systems (DBMS) have been built to provide a generic solution to this issue.
Usually, a DBMS enforces a clear distinction between how the data is stored on disk and how it is accessed via queries. When querying a database, one should not be concerned about how and where the data is physically stored, but only with the logical structure of the data. This concept is called the data independence principle. This article highlights the query optimization in generic DBMS.

Keywords: Database, Tuple, Tree

1. Introduction

The relational model is now the most widely used by DBMS. For this reason, and for lack of space, we mainly consider the relational model in this chapter. Only in the last section we shall briefly discuss other major data models. Theoretical database research has covered areas such as the design of query languages, schema design, query evaluation (clustering, indexing, optimization heuristics, etc.), storage and transaction management, and concurrency control to name just a few.
In the early years of databases, when it became clear that file systems are not an adequate solution for storing and processing large amounts of interrelated data, several database models were proposed, including the hierarchical model and the network model. One central idea at the time was that querying a database should not depend on how and where data is actually stored. This is known as the data independence principle.
One of the biggest breakthroughs in computer science came when Codd introduced the relational model in 1970. In this chapter we will focus on the relational model, as it is still dominating the databases industry. Furthermore, most classical results in database theory have been obtained for the relational model. In a nutshell, the basic idea of relational databases is to store data in tables or, seen from a more mathematical point of view, in relations.
The actual content of a relation is given by the set of rows of the table, the tuples of the relation. Each tuple has one entry for each attribute of the relation. We assume that each entry comes from a fixed, infinite domain of potential database elements. Elements in this domain are often called constants or data values. A database schema σ is simply a finite set of relation schemas where no two relations have the same name. For the purpose of this article, we ignore the fact that a database schema usually also includes a set of integrity constraints like key or foreign key constraints.
Relational query languages provide a high-level "declarative" interface to access data stored in relational databases. Over time, SQL has emerged as the standard for relational query languages. Two key components of the query evaluation component of a SQL database system are the query optimizer and the query execution engine. The query execution engine implements a set of physical operators. An operator takes as input one or more data streams and produces an output data stream. Examples of physical operators are (external) sort, sequential scan, index scan, nested loop join, and sort-merge join. We refer to such operators as physical operators since they are not necessarily tied one-to-one with relational operators.
The simplest way to think of physical operators is as pieces of code that are used as building blocks to make possible the execution of SQL queries. An abstract representation of such an execution is a physical operator tree, as illustrated in Figure 1. The edges in an operator tree represent the data flow among the physical operators. We use the terms physical operator tree and execution plan (or, simply plan) interchangeably. The execution engine is responsible for the execution of the plan that results in generating answers to the query. Therefore, the capabilities of the query execution engine determine the structure of the operator trees that are feasible. We refer an overview of query evaluation techniques.

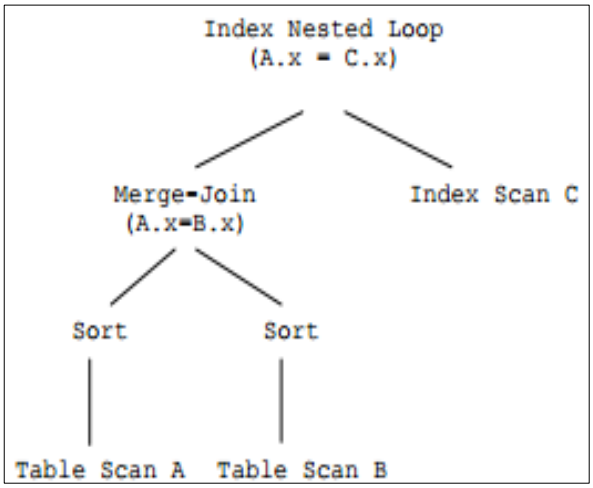

Fig 1

The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and is responsible for generating an efficient execution plan for the given SQL query from the space of possible execution plans.

Relational Database Management System

Relational database management systems (RDBMSs) let users specify queries using high level declarative languages such as SQL. Users define their desired results without detailing how such results must be obtained. The query optimizer, a component of the RDBMS, is then responsible for finding an efficient procedure, or execution plan, to evaluate the input query. For that purpose, the optimizer searches a large space of alternative execution plans, and chooses the one that is expected to be evaluated in the least amount of time.

Once the optimizer produces the execution plan estimated to be the best plan, the execution engine of the RDBMS provides an environment in which this plan is evaluated. State-of-the-art query optimizers search the space of alternative execution plans in a cost-based manner. Conceptually, modern optimizers assign each candidate plan its estimated cost (i.e., the expected amount of resources that the execution engine would require to evaluate the candidate plan), and choose the plan with the least expected cost for execution.

Oracle's Query Optimizer

Oracle's query optimizer has been used in more database applications than any other query optimizer, and Oracle's optimizer has continually benefited from real-world input.

Oracle's optimizer consists of four major components:

a) SQL transformations

Oracle transforms SQL statements using a variety of sophisticated techniques during query optimization. The purpose of this phase of query optimization is to transform the original SQL statement into a semantically equivalent SQL statement that can be processed more efficiently.

b) Execution plan selection

For each SQL statements, the optimizer chooses an execution plan (which can be viewed using Oracle's EXPLAIN PLAN facility or via Oracle's "v$sql_plan" views). The execution plan describes all of the steps when the SQL is processed, such as the order in which tables are accessed, how the tables are joined together and whether tables are accessed via indexes. The optimizer considers many possible execution plans for each SQL statement, and chooses the best one.

c) Cost model and statistics

Oracle's optimizer relies upon cost estimates for the individual operations that make up the execution of a SQL statement. In order for the optimizer to choose the best execution plans, the optimizer needs the best possible cost estimates. The cost estimates are based upon in-depth knowledge about the I/O, CPU, and memory resources required by each query operation, statistical information about the database objects (tables, indexes, and materialized views), and performance information regarding the hardware server platform.

The process for gathering these statistics and performance information needs to be both highly efficient and highly automated.

d) Dynamic runtime optimization

Not every aspect of SQL execution can be optimally planned ahead of time. Oracle thus makes dynamic adjustments to its query-processing strategies based on the current database workload. The goal of dynamic optimizations is to achieve optimal performance even when each query may not be able to obtain the ideal amount of CPU or memory resources.

Oracle additionally has a legacy optimizer, the rule-based optimizer (RBO). This optimizer exists in Oracle Database 10g Release 2 solely for backwards compatibility. Beginning with Oracle Database 10g Release 1, the RBO is no longer supported.

The vast majority of Oracle's customers today use the cost based optimizer. All major applications vendors (Oracle Applications, SAP, and People soft, to name a few) and the vast majority of recently built custom applications utilize the cost-based optimizer for enhanced performance, and this paper describes only the cost-based optimizer.

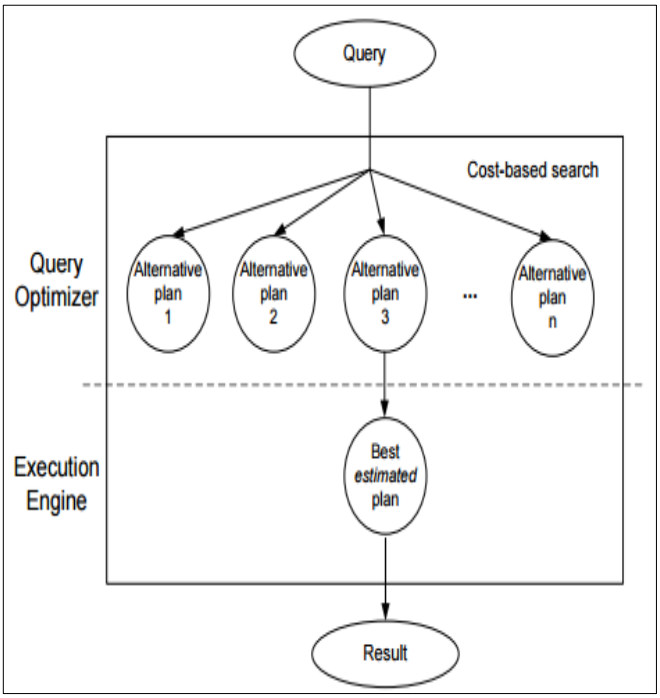

Fig 2: The RDBMS chooses efficient execution plans to evaluate input queries.

About all residents in states that have a gross product exceeding 500 billion dollars:

Select Page, P gender,
From Population P, States S
Where
P. state id = S. state id
And
S. gross product > 500,000,000,000

Note that the query above only indicates the structure and relationships that the output tuples must satisfy, without specifying how to obtain the required information. In general, there are many possible execution plans to obtain the same result. For instance, we can first join tables Population and States on state id and then filter out the joined tuples that do not satisfy the condition S.gross product > 500 billion.

Alternatively, we can use an index on States. gross product to quickly identify the states with gross product exceeding 500

billion and then join each resulting state with all the matching tuples in Population. Which alternative plan is the most efficient depends on the actual data distribution of tables States and Population.

Executing SQL Queries in a Relational Database System

Relational query languages provide a high-level declarative interface to access data stored in relational database systems. With a declarative language, users (or applications acting as users) write queries stating what they want, but without specifying step-by-step instructions on how to obtain such results. In turn, the RDBMS internally determines the best way to evaluate the input query and obtains the desired result.

Structured Query Language, or SQL has become the most widely used relational database language.

1. The input query, treated as a string of characters, is parsed and transformed into an algebraic tree that represents the structure of the query. This step performs both syntactic and semantic checks over the input query, rejecting all invalid requests.
2. The algebraic tree is optimized and turned into a query execution plan. A query execution plan indicates not only the operations required to evaluate the input query, but also the order in which they are performed, the algorithm used to perform each step, and the way in which stored data is obtained and processed.
3. The query execution plan is evaluated and results are passed back to the user in the form of a relational table.

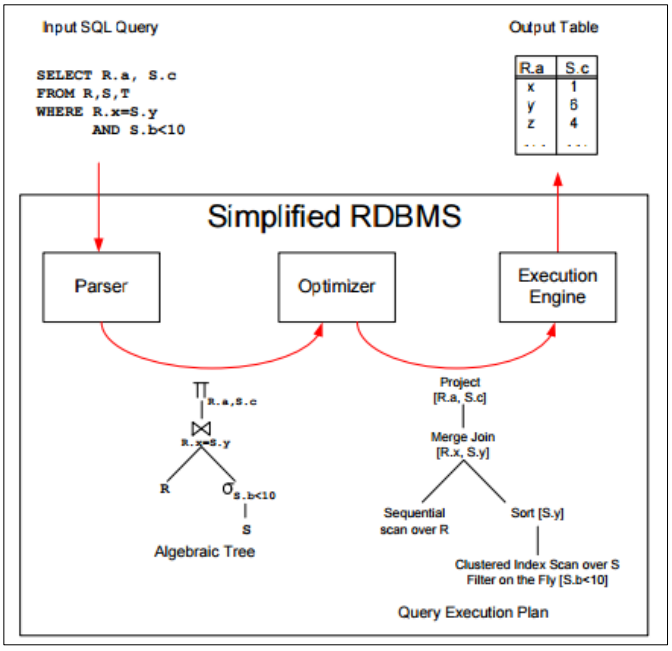

Fig 3: Executing SQL queries in a relational database system.

Modern relational query optimizers are complex pieces of code and typically represent 40 to 50 developer-years of effort. As stated before, the role of the optimizer in a database system is to identify an efficient execution plan to evaluate the input query. To that end, optimizers usually examine a large number of possible query plans and choose the one that is expected to result in the fastest execution.

Architecture of a Query Optimizer

Several query optimization frameworks have been proposed in the literature and most modern optimizers rely on the concepts introduced in these references. Although implementation details vary among specific systems, virtually all optimizers share the same basic structure, shown in Figure 4.

For each input query, the optimizer considers a multiplicity of alternative plans. For that purpose, an enumeration engine navigates through the space of candidate execution plans by applying rules. Some optimizers have a fixed set of rules to enumerate alternative plans, for example, System Analysis Codes, while others implement extensible transformational rules to navigate through the search space, for example, Hierarchical Low-level Programs.

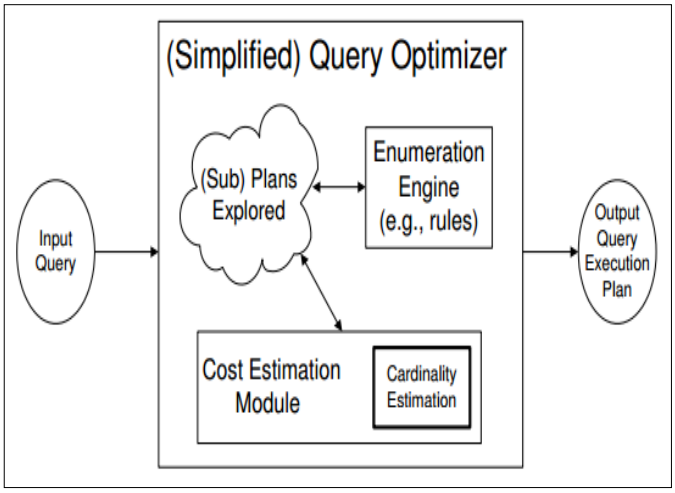

Fig 4: Simplified architecture of the query optimizer in a database system.

Query flow through a DBMS

The expressive power, the complexity of evaluation, and static analysis are correlated properties of a query language. More expressive power usually increases the complexity of query evaluation and static analysis. But even if two query languages have the same expressive power, they may vastly differ in terms of the complexity of static analysis and query evaluation.

The reason for this is that even if every query of one language can be translated into an equivalent query of the other language, the translation may turn a short query into a huge one. Thus, apart from expressive power and complexity, also the succinctness of queries is a natural measure for comparing query languages. Although well-investigated for languages used in specification and automated verification, a systematic study of the succinctness of database query languages has started only recently.

An interactive (ad hoc) query goes through the entire path shown in Figure 5.

On the other hand, an embedded query goes through the first three steps only once, when the program in which it is embedded is compiled (compile time). 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).

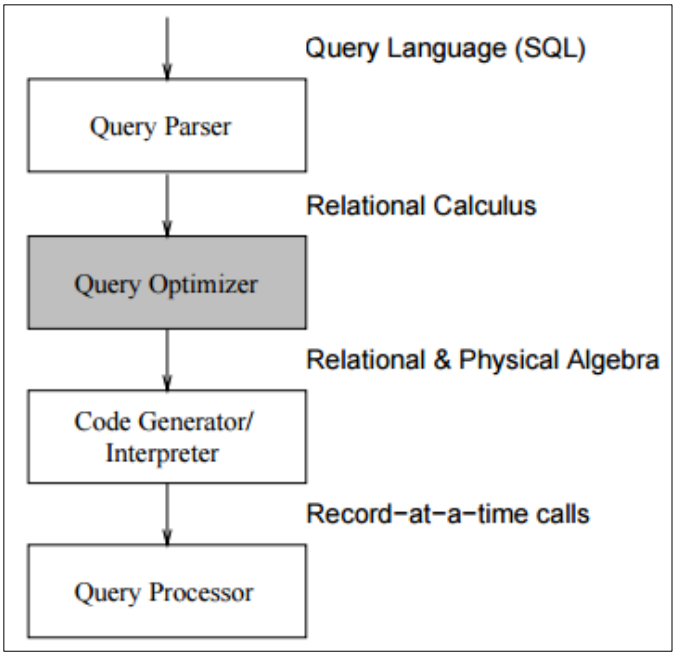

Fig 5: Query flow through a DBMS.

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).

General Query Tree

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 non-recursive Horn clause) in a centralized relational DBMS, assuming that full knowledge of the run-time environment exists at compile time. Likewise, we make no attempt to provide a complete survey of the literature, in most cases providing only a few example references. More extensive surveys can be found elsewhere.

As mentioned above, 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 1).

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.

Figure 6 gives three examples of query trees for the query.

Select name, floor from emp, dept where emp.dno=dept.dno and sal>100K.

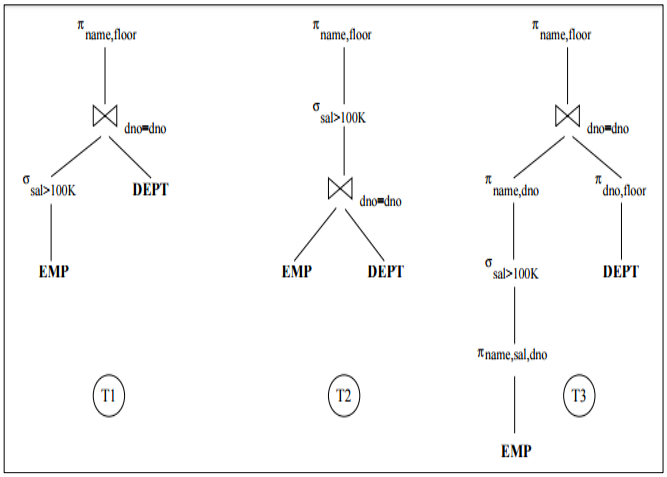

Fig 6: Examples of general query trees.

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 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.

For example, plan P1 of Section 1 satisfies restriction R1: the index scan of emp finds emp tuples that satisfy the selection on emp.sal on the y and attempts to join only those; furthermore, the projection on the result attributes occurs as the join tuples are generated. For queries with no join, R1 is moot.

In traditional execution of a SPJ query with group-by, the evaluation of the SPJ component of the query precedes the group by. The set of transformations described in this section enable the group by operation to precede a join. These transformations are applicable to queries with SELECT DISTINCT since the latter is a special case of group-by.

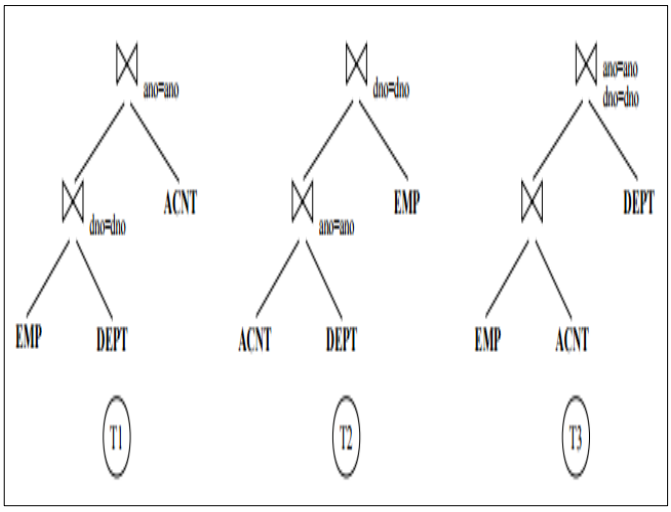

Fig 7: Examples of join trees; T3 has a cross product.

Group-By and Join

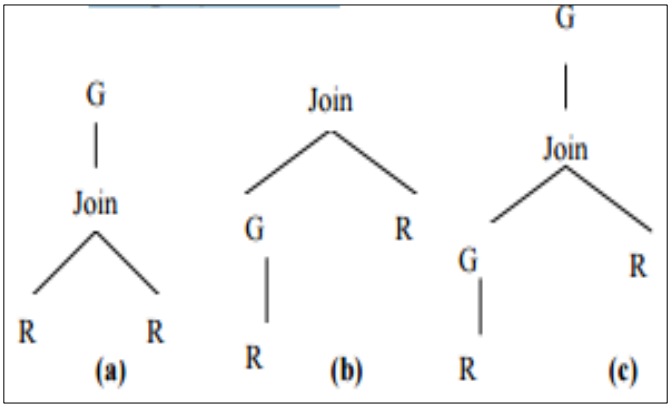

Fig 8: Group By and Join

Evaluation of a group-by operator can potentially result in a significant reduction in the number of tuples, since only one tuple is generated for every partition of the relation induced by the group-by operator. Therefore, in some cases, by first doing the group-by, the cost of the join may be significantly reduced. Moreover, in the presence of an appropriate index, a group-by operation may be evaluated inexpensively. A dual of such transformations corresponds to the case where a group-by operator may be pulled up past a join.

References

1. Astrahan MM *et al.* System R: A relational approach to data management. ACM Transactions on Database Systems. 2011; 1(2):97-137.
2. Antoshenkov G. Dynamic query optimization in Rdb/VMS. In Proc. IEEE Int. Conference on Data Engineering, Vienna, Austria. 2013, 538-547.
3. Bennett K, Ferris MC, Ioannidis Y. A genetic algorithm for database query optimization. In Proc. 4th Int. Conference on Genetic Algorithms. San Diego, CA. 2011, 400-407.
4. Bernstein PA, Goodman N, Wong E, Reeve CL, Rothnie JB. Query processing in a system for distributed databases (SDD-1). ACM TODS. 2011; 6(4):602-625.
5. Cole R, Graefe G. Optimization of dynamic query evaluation plans. In Proc. ACM-SIGMOD Conference on the Management of Data, Minneapolis, MN. 2010, 150-160.
6. Christodoulakis S. Implications of certain assumptions in database performance evaluation. ACM TODS. 2010; 9(2):163-186.
7. Christodoulakis S. On the estimation and use of selectivities in database performance evaluation. Research Report CS, Dept. of Computer Science, University of Waterloo.2009, 89-24.
8. Graefe, DeWitt D. The exodus optimizer generator. In Proc. ACM-SIGMOD Conf. on the Management of Data, San Francisco, CA. 2012, 160-172.
9. Galindo-Legaria C, Pellenkoft A, Kersten M. Fast, randomized join-order selection - why use transformations? In Proc. 20th Int. VLDB Conference, Santiago, Chile. 2010, 85-95.
10. Graefe G, McKenna B. The Volcano optimizer generator: Extensibility and efficient search. In Proc. IEEE Data Engineering Conf., Vienna, Austria. 2013.
11. Goldberg DE. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA. 2009.
12. Graefe G, Ward K. Dynamic query evaluation plans. In Proc. ACM-SIGMOD Conference on the Management of Data, Portland, OR.2009, 358-366.
13. Haas L *et al.* Starburst mid light: As the dust clears. IEEE Transactions on Know ledge and Data Engineering. 2010; 2(1):143-160.
14. Hong W, Stonebraker M. Optimization of parallel query execution plans in xprs. In Proc. 1st Int. PDIS Conference, Miami, FL. 2011, 218-225.