Available online at www.elixirpublishers.com (Elixir International Journal)

Computer Science and Engineering

Elixir Comp. Sci. & Engg. 74 (2014) 27246-27248

ISSN: 2229-712X

Introduction to Query Processing and Optimization

G. R. Bamnote[1] and S. S. Agrawal[2]

[1]Department of Computer Science & Engineering, PRMITR, Badnera, India.
[2]Department of Computer Science & Engineering, COE & T, Akola, India.

ARTICLE INFO

Article history:
Received: 18 April 2013;
Received in revised form:
15 September 2014;
Accepted: 26 September 2014;

Keywords
Query Processing,
Query Optimization,
Database.

ABSTRACT

Query Processing is the scientific art of obtaining the desired information from a database system in a predictable and reliable fashion. Database systems must be able to respond to requests for information from the user i.e. process queries. In large database systems, which may be running on un-predictable and volatile environments, it is difficult to produce efficient database query plans based on information available solely at compile time. Getting the database results in a timely manner deals with the technique of Query Optimization. Efficient processing of queries is an important requirement in many interactive environments that involve massive amounts of data. Efficient query processing in domains such as the Web, multimedia search, and distributed systems has shown a great impact on performance. This paper will introduce the basic concepts of query processing and query optimization in the relational database. We also describe and difference query processing techniques in relational databases.

Introduction

The fundamental part of any DBMS is query processing and optimization. The results of queries must be available in the timeframe needed by the submitting user[1]. Query processing techniques based on multiple design dimensions can be classified as[2]:

1. Query model: Processing techniques are classified according to the query model they assume. Some techniques assume a selection query model, where scores are attached directly to base tuples. Other techniques assume a join query model, where scores are computed over join results. A third category assumes an aggregate query model, where we are interested in ranking groups of tuples.
2. Data access methods: Processing techniques are classified according to the data access methods they assume to be available in the underlying data sources. For example, some techniques assume the availability of random access, while others are restricted to only sorted access.
3. Implementation level: Processing techniques are classified according to their level of integration with database systems. For example, some techniques are implemented in an application layer on top of the database system, while others are implemented as query operators.
4. Data and query uncertainty: Processing techniques are classified based on the uncertainty involved in their data and query models. Some techniques produce exact answers, while others allow for approximate answers, or deal with uncertain data.
5. Ranking function: Processing techniques are classified based on the restrictions they impose on the underlying ranking (scoring) function. Most proposed techniques assume monotone scoring functions.

Query processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

A database query is the vehicle for instructing a DBMS to update or retrieve specific data to/from the physically stored medium. The actual updating and retrieval of data is performed through various "low- level" operations[10]. Examples of such operations for a relational DBMS can be relational algebra operations such as project, join, select, Cartesian product, etc[11]. While the DBMS is designed to process these low -level operations efficiently, it can be quite the burden to a user to submit requests to the DBMS in these formats.

There are three phases [12] that a query passes through during the DBMS' processing of that query:

1. Parsing and translation
2. Optimization
3. Evaluation

The first step in processing a query submitted to a DBMS is to convert the query into a form usable by the query processing engine. High- level query languages such as SQL represent a query as a string, or sequence, of characters.

Certain sequences of characters represent various types of tokens such as keywords, operators, operands, literal strings, etc. Like all languages, there are rules (syntax and grammar) that govern how the tokens can be combined into understandable (i.e. valid) statements.

The primary job of the parser is to extract the tokens from the raw string of characters and translate them into the corresponding internal data elements (i.e. relational algebra operations and operands) and structures (i.e. query tree, query graph). The last job of the parser is to verify the validity and syntax of the original query string.

In second stage, the query processor applies rules to the internal data structures of the query to transform these structures into equivalent, but more efficient representations.

Tele:
E-mail addresses: sachin.s.agrawal@gmail.com

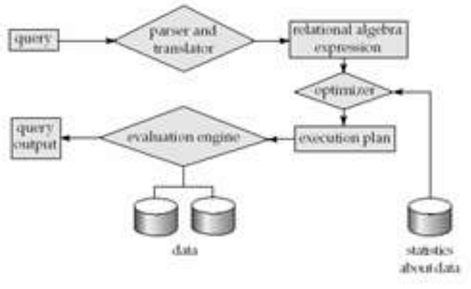

Figure: Steps in query processing.

The rules can be based upon mathematical models of the relational algebra expression and tree (heuristics), upon cost estimates of different algorithms applied to operations or upon the semantics within the query and the relations it involves.

Selecting the proper rules to apply, when to apply them and how they are applied is the function of the query optimization engine.

Figure: Steps in query processing

The final step in processing a query is the evaluation phase. The best evaluation plan candidate generated by the optimization engine is selected and then executed. Note that there can exist multiple methods of executing a query.

Besides processing a query in a simple sequential manner, some of a query's individual operations can be processed in parallel either as independent processes or as interdependent pipelines of processes or threads. Regardless of the method chosen, the actual results should be same

Consider for example:
select *balance*
from *account*
where *balance* < 2500

This can be translated into either of the following relational algebra expressions:

- $\sigma_{balance<2500}\ (\Pi_{balance}(account))$
- $\Pi_{balance}(\sigma_{balance<2500}(account))$

Which can also be represented as either of the following query trees:

$$\begin{tikzcd}[row sep=small] \sigma_{balance<2500} \arrow[d, dash] & \Pi_{balance} \arrow[d, dash] \\ \Pi_{balance} \arrow[d, dash] & \sigma_{balance<2500} \arrow[d, dash] \\ account & account \end{tikzcd}$$

Figure: A query-evaluation plan.

Masures of query cost

The cost of query evaluation can be measured in terms of a number of different resources, *including disk accesses*, *CPU time to execute a query*, and, in a *distributed or parallel database system*, the *cost of communication*.

The *response time* for a query-evaluation plan, assuming no other activity is going on the computer, would account for all these costs, and could be used as a good measure of the cost of the plan. In large database systems, however, disk accesses are usually the most important cost, since disk accesses are slow compared to in-memory operations.

Also CPU speeds have been improving much faster than have disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query. Finally, estimating the CPU time is relatively hard, compared to estimating the disk-access cost. Therefore, most people consider the disk-access cost a reasonable measure of the cost of a query-evaluation plan.

Query algorithms

Queries are ultimately reduced to a number of file scan operations on the underlying physical file structures[3, 4]. For each relational operation, there can exist several different access paths to the particular records needed.

The query execution engine can have a multitude of specialized algorithms designed to process particular relational operation and access path combinations.

Selection Algorithms

The *Select* operation must search through the data files for records meeting the selection criteria. Following are some examples of simple (one attribute) selection algorithms [13]:

1. Linear search: Every record from the file is read and compared to the selection criteria. The execution cost for searching on a non-key attribute is b_r, where b_r is the number of blocks in the file representing relation r. On a key attribute, the average cost is $b_r/2$,with a worst case of b_r.
2. Binary search: A binary search, on equality, performed on a primary key attribute has a worst-case cost of $\lceil \log (b_r) \rceil$. This can be considerably more efficient than the linear search, for a large number of records.
3. Search using a primary index on equality: With a B^+-tree index, an equality comparison on a key attribute will have a worst -case cost of the height of the tree plus one to retrieve the record from the data file. An equality comparison on a non-key attribute will be the same except that multiple records may meet the condition, in which case, we add the number of blocks containing the records to the cost.
4. Search using a primary index on comparison: When the comparison operators ($<$, $\leq$, $>$, $\geq$) are used to retrieve multiple records from a file sorted by the search attribute, the first record satisfying the condition is located and the total blocks before ($<$, $\leq$) or after ($>$, $\geq$) is added to the cost of locating the first record.
5. Search using a secondary index on equality: Retrieve one record with an equality comparison on a key attribute; or retrieve a set of records on a non-key attribute[6]. For a single record, the cost will be equal to the cost of locating the search key in the index file plus one for retrieving the data record. For multiple records, the cost will be equal to the cost of locating the search key in the index file plus one block access for each data record retrieval, since the data file is not ordered on the search attribute.

Join Algorithms

The join algorithm can be implemented in a different ways. In terms of disk accesses, the join operations can be very expensive, so implementing and utilizing efficient join algorithms is important in minimizing a query's execution time[8]. The following are 4 well - known types of join algorithms:

1. Nested-Loop Join: It consists of a inner for loop nested within an outer for loop [12].

2. Index Nested-Loop Join: This algorithm is the same as the Nested-Loop Join, except an index file on the inner relation's join attribute is used versus a data-file scan on each index lookup in the inner loop is essentially an equality selection for utilizing one of the selection algorithms Let c be the cost for the lookup, then the worst -case cost for joining rand sis $b_r + n_r * c$.

3. Sort –Merge Join: This algorithm can be used to perform natural joins and equi-joins and requires that each relation be sorted by the common attributes between them [5]

4. Hash Join: The hash join algorithm can be used to perform natural joins and equi-joins. The hash join utilizes two hash table file structures (one for each relation) to partition each relation's records into sets containing identical hash values on the join attributes. Each relation is scanned and its corresponding hash table on the join attribute values is built.

Indexes Role

The execution time of various operations such as select and join can be reduced by using indexes[7]. Let us review some of the types of index file structures and the roles they play in reducing execution time and overhead:

1. Dense Index: Data-file is ordered by the search key and every search key value has a separate index record. This structure requires only a single seek to find the first occurrence of a set of contiguous record s with the desired search value[9].

2. Sparse Index: Data-file is ordered by the index search key and only some of the search key values have corresponding index records. Each index record's data-file pointer points to the first data-file record with the search key value. While this structure can be less efficient than a dense index to find the desired records, it requires less storage space and less overhead during insertion and deletion operations.

3. Primary Index: The data file is ordered by the attribute that is also the search key in the index file. Primary indices can be dense or sparse. This is also referred to as an Index-Sequential File [5].

4. Secondary Index: The data file is ordered by an attribute that is different from the search key in the index file. Secondary indices must be dense.

5. Multi-Level Index: An index structure consisting of 2 or more tier s of records where an upper tier's records point to associated index records of the tier below. The bottom tier's index records contain the pointers to the data-file records. Multi-level indices can be used, for instance, to reduce the number of disk block reads needed during a binary search.

6. Clustering Index: A two-level index structure where the records in the first level contain the clustering field value in one field and a second field pointing to a block [of 2nd level records] in the second level. The records in the second level have one field that points to an actual data file record or to another 2nd level block.

7. B^+-tree Index: Multi- level index with a balanced-tree structure. Finding a search key value in a B^+-tree is proportional to the height of the tree maximum number of seeks required is $\lceil \log(\mathit{height}) \rceil$. While this, on average, is more than a single - level, dense index that requires only one seek, the B^+-tree structure has a distinct advantage in that it does not require reorganization, it is self-optimizing because the tree is kept balanced during insertions and deletions.

Choice of evaluation plans

The query optimization engine generates a set of candidate evaluation plans. Some will, in heuristic theory, produce a faster, more efficient execution. Others may, by prior historical results, be more efficient than the theoretical models, this can very well be the case for queries dependent on the semantic nature of the data to be processed. Still others can be more efficient due to "outside agencies" such as network congestion, competing applications on the same CPU, etc.

Conclusion

One of the most critical functional requirements of a DBMS is its ability to process queries in a timely manner. This is particularly true for very large, mission critical applications such as weather forecasting, banking systems and aeronautical applications, which can contain millions and even trillions of records. The need for faster and faster, "immediate" results never ceases.

Thus, a great deal of research and resources is spent on creating smarter, highly efficient query optimization engines. Some of the basic techniques of query processing and optimization have been presented in this paper.

References

[1] Henk Ernst Blok, Djoerd Hiemstra and Sunil Choenni, Franciska de Jong, Henk M. Blanken and Peter M.G. Apers. Predicting the cost-quality trade-off for information retrieval queries: Facilitatiing database design and query optimization. Proceedings of the tenth international conference on Information and knowledge management, Pages 207 - 214.

[2] D. Calvanese, G. DeGiacomo, M. Lenzerini and M. Y. Vardi. Reasoning on Regular Path Queries. ACM SIGMOD Record , Vol. 32, No. 4, December 2003.

[3] Andrew Eisenberg and Jim Melton. Advancements in SQL/XML. ACM SIGMOD Record ,Vol. 33, No. 3, September 2004.

[4] Andrew Eis enberg and Jim Melton. An Early Look at XQuery API for Java™ (XQJ). ACM SIGMOD Record ,Vol. 33, No. 2

[5] Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database Systems, second edition. Addison-Wesley Publishing Company,

[6] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A new Class of Query Optimization Algorithms. ACM Transactions on Database Systems, Vol. 25, No. 1, March 2000, Pages 43- 82.

[7] Chiang Lee, Chi - Sheng Shih and Yaw - Huei Chen. A Graph-theoritic model for optimizing queries involving methods. The VLDB Journal —The International Journal on Very Large Data Bases, Vol. 9,Issue 4, Pages 327 -343.

[8] Hsiao-Fei Liu, Ya - Hui Chang and Kun-Mao Chao. An Optimal Algorithm for Querying Tree Structures and its Applications in Bioinformatics. ACM SIGMOD Record Vol. 33, No. 2, June 2004.

[9] Reza Sadri, Carlo Zaniolo, Amir Zarkesh and Jafar Adibi. Expressing and Optimizing Transactions on Database Systems, Vol . 29, Issue 2, Pages 282 - 318.

[10] Reza Sadri, Carlo Zaniolo, Amir Zarkesh and Jafar Adibi. Optimization of Sequence Queries in Database Systems. In Proceedings of the twentieth ACM SIGMOD -SIGACT-SIGART symposium on Principles of database systems, May 2001, Pages 71 -81.

[11] Thomas Schwentick. XPath Query Containment. ACM SIGMOD Record , Vol. 33, No. 1, March 2004.

[12] Avi Silbershatz, Hank Korth and S. Sudarshan. Database System Concepts, 7th Edition. McGraw - Hill.

[13] Dimitri Theodoratos and Wugang Xu. Constructing Search Spaces for Materialized View Selection. Proceedings of the 7th ACM international workshop on Data warehousing and OLAP, Pages 112 - 121.

[14] Jingren Zhou and Kenneth A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. Proceedings of the 2004 ACM SIGMOD international conference on Management of data, June 2004, Pages 191 - 202.