Computer Engineering and Intelligent Systems
ISSN 2222-1719 (Paper) ISSN 2222-2863 (Online)
Vol 3, No.1, 2012

Query Optimization to Improve Performance of the Code Execution

Swati Tawalare * S.S Dhande

Dept of CSE, SIPNA's College of Engineering & Technology, Amravati, INDIA

* E-mail of the corresponding author : swatitawalare18@rediffmail.com

Abstract

Object-Oriented Programming (OOP) is one of the most successful techniques for abstraction. Bundling together objects into collections of objects, and then operating on these collections, is a fundamental part of main stream object-oriented programming languages. Object querying is an abstraction of operations over collections, whereas manual implementations are performed at low level which forces the developers to specify how a task must be done. Some object-oriented languages allow the programmers to express queries explicitly in the code, which are optimized using the query optimization techniques from the database domain. In this regard, we have developed a technique that performs query optimization at compile-time to reduce the burden of optimization at run-time to improve the performance of the code execution.

Keywords- *Querying; joins; compile time; run-time; histograms; query optimization*

Introduction

Query processing is the sequence of actions that takes as input a query formulated in the user language and delivers as result the data asked for. Query processing involves query transformation and query execution. Query transformation is the mapping of queries and query results back and forth through the different levels of the DBMS. Query execution is the actual data retrieval according to some access plan, i.e. a sequence of operations in the physical access language. An important task in query processing is query optimization. Usually, user languages are high-level, declarative languages allowing to state what data should be retrieved, not how to retrieve them. For each user query, many different execution plans exist, each having its own associated costs. The task of query optimization ideally is to find the best execution plan, i.e. the execution plan that costs the least, according to some performance measure. Usually, one has to accept just feasible execution plans, because the number of semantically equivalent plans is to large to allow for enumerative search A query is an expression that describes information that one wants to search for in a database.

For example, query optimizers select the most efficient access plan for a query based on the estimated costs of competing plans. These costs are in turn based on estimates of intermediate result sizes. Sophisticated user interfaces also use estimates of result sizes as feedback to users before a query is actually executed. Such feedback helps to detect errors in queries or misconceptions about the database. Query result sizes are usually estimated using a variety of statistics that are maintained for relations in the database. These statistics merely approximate the distribution of data values in attributes of the relations. Consequently, they represent an inaccurate picture of the actual contents of the database. The resulting size-estimation errors may undermine the validity of the optimizer's decisions or render the user interface application unreliable. Earlier work has shown that errors in query result size estimates may increase exponentially with the number of joins. This result, in conjunction with the increasing complexity of queries, demonstrates the critical importance of accurate estimation .Several techniques have been proposed in the literature to estimate query result sizes, including histograms, sampling, and parametric techniques. Of these, histograms approximate the frequency distribution of an attribute by grouping attribute values into

"buckets" (subsets) and approximating true attribute values and their frequencies in the data based on summary statistics maintained in each bucket. The main advantages of histograms over other techniques are that they incur almost no run-time overhead, they do not require the data to fit a probability distribution or a polynomial and, for most real-world databases, there exist histograms that produce low-error estimates while occupying reasonably small space Although histograms are used in many systems, the histograms proposed in earlier works are not always effective or practical. For example, *equidepth* histograms work well for range queries only when the data distribution has low skew, while *serial* histograms have only been proven optimal for equality joins and selections when a list of all the attribute values in each bucket is maintained. Implementing operations over these collections with conventional techniques severely lacks in abstraction. Step-by-step instructions must be provided as to how to iterate over the collection, select elements, and operate on the elements. Manually specifying the implementation of these operations fixes how they are to be evaluated, rendering it impossible to accommodate changes in the state of the program that could make another approach superior. This is especially problematic when combining two collections of objects together, frequently an expensive operation. Object querying is a way to select an object, or set of objects, from a collection or set of collections.

JQL is an addition to Java that provides the capability for querying collections of objects. These queries can be applied on objects in collections in the program or can be used for checking expressions on all instances of specific types at run-time. Queries allow the query engine to take up the task of implementation details by providing abstractions to handle sets of objects thus making the code smaller and permitting the query evaluator to choose the optimization approaches dynamically even though the situation changes at run-time. The Java code and the JQL query will give the same set of results but the JQL code is elegant, brief, and abstracts away the accurate method of finding the matches. Java Query Language (JQL) by generating the dynamic join ordering strategies. The Java Query Language provides straightforward object querying for Java. JQL designed as an extension to Java, and implemented a prototype JQL system, in order to investigate the performance and usability characteristics of object querying. Queries can be evaluated over one collection of objects, or over many collections, allowing the inspection of relationships between objects in these collections. The Java Query Language (JQL) provides first-class object querying for Java.

For example A difficulty with this decomposition is representing students who are also teachers. One solution is to have separate Student and Teacher objects, which are related by name.

The following code can then be used to identify students who are teachers:

```
List<Tuple2<Faculty, Student>>
  matches = new Array List<..>();
  for(Faculty f : all Faculty) { for(Student s : all Students) {
if(s.name.equals(f.name)) {
matches. add(new Tuple2<Faculty, Student>(f,s));
}}}
```

In database terms, this code is performing a join on the name field for the allFaculty and allStudent collections. The code is cumbersome and can be Replaced with the following object query, which is more succinct and, potentially, more efficient:

```
List<Tuple2<Faculty, Student>> matches;
Matches = selectAll (Faculty f=allFaculty, Student s=allStudents: f.name.equals (s.name));
```

This gives exactly the same set of results as the loop code. The selectAll primitive returns a list of tuples containing all possible instantiations of the domain variables (i.e. those declared before the colon) where the query expression holds (i.e. that after the colon). The domain variables determine the set of objects which the query ranges over: they can be initialized from a collection (as above); or, left uninitialized to range over the entire extent set (i.e. the set of all instantiated objects) of their type. Queries can define as

many domain variables as necessary and can make use of the usual array of expression constructs found in Java. One difference from normal Java expressions is that Boolean operators, such as && and ||, do not imply an order of execution for their operands. This allows flexibility in the order they are evaluated, potentially leading to greater efficiency. As well as its simplicity, there are other advantages to using this query in place of the loop code. The query evaluator can apply well-known optimizations which the programmer might have missed. By leaving the decision of which optimization to apply until runtime, it can make a more informed decision based upon dynamic properties of the data itself (such as the relative size of input sets) something that is, at best, difficult for a programmer to do. A good example, which applies in this case, is the so-called hash-join The idea is to avoid enumerating all of all Faculty × all Students when there are few matches. A hash-map is constructed from the largest of the two collections which maps the value being joined upon (in this case name) back to its objects. This still requires O(sf) time in the worst-case, where s = |all Students| and f = |all Faculty|, but in practice is likely to be linear in the number of matches (contrasting with a nested loop which always takes O(sf) time).

Implementation

We have prototyped a system, called the Java Query Language (JQL), which permits queries over object extents and collections in Java. The implementation consists of three main components: a compiler, a query evaluator and a runtime system for tracking all active objects in the program. The latter enables the query evaluator to range over the extent sets of all classes. Our purpose in doing this is twofold: firstly, to assess the performance impact of such a system; secondly, to provide a platform for experimenting with the idea of using queries as first-class language constructs.

JQL Query Evaluator

The core component of the JQL system is the query evaluator. This is responsible for applying whatever optimizations it can to evaluate queries efficiently. The evaluator is called at runtime with a tree representation of the query (called the query tree). The tree itself is either constructed by the JQL Compiler (for static queries) or by the user (for dynamic queries).

Evaluation Pipeline.

The JQL evaluator evaluates a query by pushing tuples through a staged pipeline. Each stage, known as a join in the language of databases, corresponds to a condition in the query. Only tuples matching a join's condition are allowed to pass through to the next. Those tuples which make it through to the end are added to the result set. Each join accepts two lists of tuples, L(eft) and R(ight), and combines them together producing a single list. We enforce the restriction that, for each intermediate join, either both inputs come from the previous stage or one comes directly from an input collection and the other comes from the previous stage. This is known as a linear processing tree and it simplifies the query evaluator, although it can lead to inefficiency in some cases. The way in which a join combines tuples depends upon its type and the operation (e.g. ==, < etc) it represents. JQL currently supports two join types:

nested-loop join and hash join. A nested-loop join is a two-level nested loop which iterates each of L × R and checks for a match. A hash join builds a temporary hash table which it uses to check for matches. This provides the best performance, but can be used only when the join operator is == or equals(). Future implementations may take advantage of B-Trees, for scanning sorted ranges of a collection.

JQL Implementation

Our implementation of JQL has two main components. The first is a frontend, which compiles JQL expressions into Java code. The second component is the query evaluation back-end, which is responsible for evaluating the query and choosing the most optimal evaluation strategy it can.

JQL Compiler

The JQL compiler is relatively straightforward — it compiles JQL expressions into equivalent Java code, which can then be compiled by a normal Java compiler. This Java code binds domain variables to

collections, and builds a tree representation of the query expression, which the evaluator uses at runtime. The compiler 'inlines' queries which only use one domain variable; they are compiled into normal loops, but with hooks to allow for query caching. Inlined queries cannot have their stages dynamically ordered, but dynamic ordering is of almost no importance to single-variable queries (whilst different orderings may 'short-circuit' evaluation for individual objects more quickly, the gain is minute compared to the performance gain of inlining the query). Using the compiler also allows for checking of the well-formedness of the query — for example, that query expressions only use domain variables that are declared for that query.

Query Evaluator

The core of query evaluation is carried out through the query's 'Query Pipeline. A query pipeline consists of a series of stages, each corresponding to a part of the query expression. Each stage can take either one or two input sets of tuples, and for all stages except the very first, one of these input sets is the result set of the previous stage in the pipeline. This structure is known, in the database community, as a 'left-branching tree', and we impose it to simplify the problem of ordering stages. In each stage, if there is more than one input set, the input tuples are combined (we describe the join techniques used for this combination shortly), and the stage's expression is evaluated using the values in the combined tuple in the place of their corresponding domain variables. If the expression evaluates to true, the combined tuple is added to the stage's result set. The set of tuples which pass the final stage are returned to the program as the results of the query

Literature Review & Related Work

The Program Query Language (PQL) (S. Chiba 1995) is a similar system which allows the programmer to express queries capturing erroneous behavior over the program trace . A key difference from other systems is that static analysis was used in an effort to answer some queries without needing to run the program. As a fallback, queries which could not be resolved statically are compiled into the program's byte code and checked at runtime. **(**C. Hobatr & B. A. Malloy 2001)Hobatr and Malloy present a query-based debugger for C++ that uses the Open++ Meta-Object Protocol and the Object Constraint Language(OCL) (Ihab F. Ilyas et al. 2003) This system consists of a frontend for compiling OCL queries to C++, and a backend that uses Open C++ to generate the instrumentation code necessary for evaluating the queries.JQL is the recent development of Microsoft's Language Integrated Query (LINQ) project LINQ aims to add first class querying support to .NET languages (in particular Visual Basic .NET and C#)

(Y.E. Ioannidis,R. Ng, K. Shim & T.K. Selis 1992) LINQ operates by translating queries into additional methods on collections of objects, which then perform filtering and mapping on the collection. LINQ's scope is wider than JQL's, providing integrated querying for object collections, XML structures and SQL databases. It is unclear at present what optimizations LINQ provides for object querying, or if it provides incrementalized caching Finally, there are a number of APIs available for Java which provide access to SQL databases. These are quite different from the approach we have presented in this work, as they do not support querying over collections and/or object extents. Furthermore, they do not perform any query optimizations, instead relying on the database back end to do this.(Darren Willis & David J. Pearce, James Noble 2008),Our main contribution is the design and implementation of JQL, a prototype object querying system that allows developers to easily write object queries in a Java-like language. This prototype provides automatic optimization of operations that join multiple collections together.

Analysis of Problem

In our research, we prefer static query optimization at compile-time over dynamic query optimization because it reduces the query run-time. In this method they are using selectivity estimate based on sampling some number of tuples, but that does not lead to efficient ordering of joins and predicates in a query. Therefore, we propose using the estimates of selectivities of joins and the predicates from histograms to provide us an efficient ordering of joins and predicates in a query. Once we collect this information, we can form the query plan by having the order of joins and predicates in a query. After we get the query plan at

compile-time, we execute that plan at run-time to reduce the execution time. Experimental results indicate that our approach reduces run-time execution less than the existing JQL code's run-time due to our approach of optimizing the query and handling data updates using histograms.

Given a query Q,

1. we use the histogram H to get the estimate of the selectivity of the query predicates and the selectivities of the joins
2. We have the join order and predicate order in a query which will be used to construct a query plan.

A.The first execution of Query Q uses the histogram H1 to estimate the selectivity. Then the result of the query is computed. But for the subsequent execution of the same query Q after a time T, the same histogram H can be left invalid. This situation arises because there is a possibility that the underlying data has been updated between the first and the second executions of the same query.

B. Firstly, we check if the query is present in the log then the time period difference between consecutive executions of the same query Q from the query log is computed and if that value is greater than a pre specified time interval then we directly recomputed the histogram because we have assumed the data may be modified within a pre specified time interval.

C. we first compute the error through error estimate function and then based on the error estimate we decide whether to recomputed the histogram or not . If the query is not present in the log, then we execute the query based upon the initial histogram that reduces the overhead cost of incremental maintenance of histogram the experimental results of how our approach various types of queries the comparison of run-times of our approach and the JQL approach for all the four benchmark queries. The difference in run-times has occurred because in our approach, we have estimated selectivities using histograms and these histograms are incrementally maintained at compile time which provide the optimal join order strategy most of the times faster than the exhaustive join order strategy used by JQL. we can see that as the number of increase in a query, JQL's approach becomes more expensive and our approach performs much better than the exhaustive join ordering strategy of JQL.

Proposed Work

We can form the query plan by having the order of joins and predicates in a query. After we get the query plan at compile time, we execute that plan at run-time to reduce the execution time. Experimental results indicate that our approach reduces run-time execution less than the existing JQL code's run-time due to our approach of optimizing the query and handling data updates using histograms. We intend to have the query plans generated at compile time Query plans are a step by step ordered procedure describing the order in which the query predicates need to be executed. Thus, at run- time, the time required for plan construction is omitted. So we need to have the code working in static mode, i.e., without knowing the inputs at compile-time, we need to be able to derive some information about inputs like sizes of relations by estimating them to generate the query plan. Given a join query ,its selectivity needs to be estimated to design plans. A histogram is one of the basic quality tools. It is used to graphically summarize and display the distribution and variation of a process data set. A frequency distribution shows how often each different value in a set of data occurs. The main purpose of a histogram is to clarify the presentation of data. You can present the same information in a table; however, the graphic presentation format usually makes it easier to see relationships. It is a useful tool for breaking out process data into regions or bins for determining frequencies of certain events or categories of data. The approach followed for maintenance in nearly all commercial systems is to recomputed histograms periodically (e.g., every night), regardless of the number of updates performed on the database. This approach has two disadvantages. First, any significant updates to the data since the last recomputation could result in poor estimations by the optimizer. Second, because the histograms are recomputed from scratch by discarding the old histograms, the recomputation phase for the entire database can be computationally very intensive and may have to be performed when the

system is lightly loaded

4.1. The Split & Merge Algorithm

The split and merge algorithm helps reduce the cost of building and maintaining histograms for large tables. The algorithm is as follows: When a bucket count reaches the threshold, T, we split the bucket into two halves instead of recomputing the entire histogram from the data. To maintain the number of buckets (β) which is fixed, we merge two adjacent buckets whose total count is least and does not exceed threshold T, if such a pair of buckets can be found. Only when a merge is not possible, we recomputed the histogram from data. The operation of merging two adjacent buckets merely involves adding the counts of the two buckets and disposing of the boundary between them. To split a bucket, an approximate median value in the bucket is selected to serve as the bucket boundary between the two new buckets using the backing sample. As new tuples are added, we increment the counts of appropriate buckets. When a count exceeds the threshold T, the entire histogram is recomputed or, using split merge, we split and merge the buckets. The algorithm for splitting the buckets starts with iterating through a list of buckets, and splitting the buckets which exceed the threshold and finally returning the new set of buckets.After splitting is done, we try to merge any two buckets that add up to the least value and whose count is less than a certain threshold. Then we merge those two buckets. If we fail to find any pair of buckets to merge then we recomputed the histogram from data. Finally, we return the set of buckets at the end of the algorithm. Thus, the problem of incrementally maintaining the histograms has been resolved. Having estimated the selectivity of a join and predicates, we get the join and the predicate ordering at compile-time

4.2 Incremental Maintenance of Histograms

we propose an *incremental* technique, which maintains approximate histograms within specified errors bounds at all times with high probability and never accesses the underlying relations for this purpose. There are two components to our incremental approach: (i) maintaining a backing sample, and (ii) a framework for maintaining an approximate histogram that performs a few program instructions in response to each update to the database, and detects when the histogram is in need of an adjustment of one or more of its bucket boundaries. Such adjustments make use of the backing sample. There is a fundamental distinction between the backing sample and the histogram it supports: the histogram is accessed more frequently than the sample and uses less memory, and hence it can be stored in main memory while the sample is likely stored on disk. A *backing sample* is a uniform random sample of the tuples in a relation that is kept up to-date in the presence of updates to the relation. For each tuple, the sample contains the unique row id and one or more attribute values. We argue that maintaining a backing sample is useful for histogram computation, selectivity estimation, etc. In most sampling-based estimation techniques, whenever a sample of size _ is needed, either the entire relation is scanned to extract the sample, or several random disk blocks are read. In the latter case, the tuples in a disk block may be highly correlated, and hence to obtain a truly random sample, _ disk blocks must be read. In contrast, a backing sample can be stored in consecutive disk blocks, and can therefore be scanned by reading sequential disk blocks. Moreover, for each tuple in the sample, only the unique row id and the attribute(s) of interest are retained. Thus the entire sample can be stored in only a small number of disk blocks, for even faster retrieval. Finally, an indexing structure for the sample can be created, maintained and stored; the index enables quick access to sample values within any desired range.

4.3 Estimating Selectivity Using Histogram

The selectivity of a predicate in a query is a decisive aspect for a query plan generation. The ordering of predicates can considerably affect the time needed to process a join query. To have the query plan ready at compile-time, we need to have the selectivities of all the query predicates. To calculate these selectivities, we use histograms. The histograms are built using the number of times an object is called. For this, we partition the domain of the predicate into intervals called windows. With the help of past queries, the selectivity of a predicate is derived with respect to its window. That is, if a table T has 100,000 rows and a query contains a selection predicate of the formT.a=10 and a histogram shows that the selectivity of T.a=10

is 10% then the cardinality estimate for the fraction of rows of T that must be considered by the query is 10% x 100,000= 10,000. This histogram approach would help us in Estimating the selectivity of a join and hence decide on the order in which the joins have to be executed. So, we get the join ordering and the predicate ordering in the query expression at compile-time itself. Thus, from this available information, we can construct a query plan.

4.4 Building the Histogram

From the data distribution, we build the histogram that contains the frequency of values assigned to different buckets. If the data is numerical, we can easily assign some ranges and assign the values to buckets accordingly. If the data is categorical then we have to partition the data into ranges with respect to the letter they start with and assign the appropriate values to buckets. Next, we perform some sample query executions.

4.5 Method Outline for Error Estimation

For each attribute in the database table, we compute the error estimate by using standard deviation between updated data values and old data values in the histogram buckets. Then, for every table, we have error estimates for all the attributes. Then, we take a weighted average of all the attributes error estimates. The underlying data could be mutable. For such mutable data, we need a technique by which we can restructure the histograms accordingly. Thus, in between multiple query executions if the database is updated, then we compute the estimation error of the histogram by using the equations (1).

Conclusion

In this paper we have presented the query optimization strategies from database domain can be used in improving the run time executions in programming language .We proposed a technique for query optimization at compile-time by reducing the burden of optimization at run-time. We proposed using histograms to get the estimates of selectivity of joins and predicates in a query and then based on those estimates, to order query joins and predicates in a query. From the join and predicate order, we have obtained the query plan at compile-time and then we executed the query plan at run-time from this we can improve the performance of the code execution.

References

C. Hobatr and B. A. Malloy(2001), “Using OCL-queries for debugging C++” ,In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 839–840.IEEE Computer Society Press, 2001.

Darren Willis ,David J. Pearce & James Noble(2008), “Caching and Incrementalisation in the Java Query Language”, Proceedings of the 2008 ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 1-18, 2008.

Ihab F. Ilyas et al. (2003), “Estimating Compilation Time of a Query Optimizer”, Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 373 –384, 2003.

S. Chiba(1995), “A meta object protocol for C++”,In *Proceedings of the ACM conference on Object-Oriente Programming, Systems, Languages and Applications (OOPSLA)*, pages 285–299. ACM Press, 1995.

Y.E. Ioannidis,R. Ng, K. Shim & T.K. Selis(1992), “Parametric Query Optimization”, In Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), pp. 103-114, 1992.

Venkata Krishna et al.(2010), "Exploring Query Optimization in Programming Codes by Reducing Run-Time Execution" ,Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 07303157/10©2010IEEEDOI10.1109/COMPSAC.2010.48, NY, USA, 2006), ACM Press, pp. 706–706.

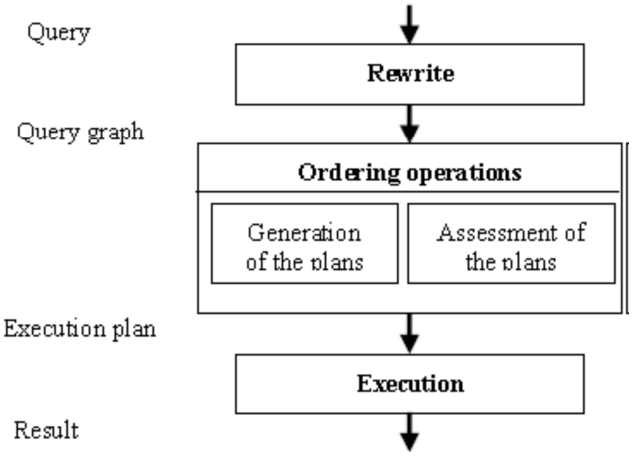

Fig. 1.Optimization process

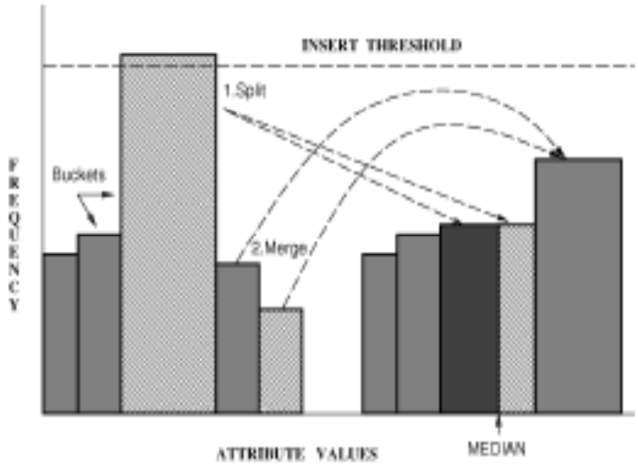

Fig 2. split and merge algorithm

$$\mu_a = \frac{\beta\sqrt{\frac{S}{N\beta}\sum_{i=1}^{\beta}(f_i - B_i)^2}}{N}$$

$$T_i = \frac{w_1\mu_1 + w_2\mu_2 + \cdots + w_n\mu_n}{w_1 + w_2 + \cdots + w_n} \quad \text{----------(1)}$$

where compute the estimation error of the histogram by using the equations..

μa is the estimation error for every attribute

- β is the number of buckets ,
- N is the number of tuples in R
- S is the number of selected tuples
- Fi is the frequency of bucket i as in the histogram
- qf= S/N is the query frequency
- Bi is the observed frequency
- Ti is the error estimate for each individual table
- Wi are the weights with respect to every attribute depending on the rate of change