Network and Complex Systems www.iiste.org
ISSN 2224-610X (Paper) ISSN 2225-0603 (Online)
Vol.3, No.7, 2013

Implementation of Query Optimization for Reducing Run Time

S.C.Tawalare
Dept of Information Technology,Sipna's College of Engineering &Technology,Amravati (MS) INDIA
swatitawalare18@rediffmail.com
Prof. S.S.Dhande
Dept of computer science &engg., Sipna's College of Engineering &Technology, Amravati (MS) INDIA
dhande_123@rediffmail.com

Abstract
Query optimization is the process of selecting the most efficient query-evaluation plan from many strategies so, In this paper 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. using histograms that are computed from the data and these histograms are used to get the estimate of selectivity for query joins and predicates in a query at compile-time. With these estimates, a query plan is constructed at compile-time and executed it at run-time
Keywords: runtime, query optimization ,compile time histogram

1. Introduction

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. 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. In this paper, 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. The proposed approach uses histograms that are computed from the data and these histograms are used to get the estimate of selectivity for query joins and predicates in a query at compile-time. With these estimates, a query plan is constructed at compile-time and executed it at run-time.

1.1 Query Optimization

Query processing is the process of translating a query expressed in a high-level language such as SQL into low-level data manipulation operations. Query Optimization refers to the process by which the *best* execution strategy for a given query is found from a set of alternatives. Typically query processing involves many steps. The first step is query decomposition in which an SQL query is first scanned, parsed and validate. The scanner identifies the language tokens – such as SQL keywords, attribute names, and relation names – in the text of the query, whereas the parser checks the query Syntax to determine whether it is formulated according to the syntax rules of the query language. The query must also be validated, by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created. A query expressed in relational algebra is usually called *initial algebraic query* and can be represented as a tree data structure called *query tree*. It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes. For a given SQL query, there is more than one possible algebraic query. Some of these algebraic queries are *better* than others. The quality of an algebraic query is defined in terms of expected performance. Therefore, the second step is query optimization step that transforms the initial algebraic query using relational algebra transformations into other algebraic queries until the *best* one is found. A Query Execution Plan (QEP) is then founded which represented as a query tree includes information about the access method available for each relation as well as the algorithms used in computing the relational operations in the tree. The next step is to generate the code for the selected QEP; this code is then executed in either compiled or interpreted mode to produce the query result.(fig.1)

1.1.1 ***Static Optimization***

if optimization is performed once at compiled time. For each invocation at runtime, The plan is activated and executed. If however the state of DBMS changes frequently, it is usually optimized the query a new for each invocation. This compiled time optimization is called as static in that it cannot use the information available at runtime such as current statistics. The common solution is to periodically reoptimize queries with new statistics, based on the pace at which the relevant statistic changes. For instance, describe a schema in which query execution plan generated by an optimizer are re-optimized just before query execution time.

- In static optimization more time available to evaluate larger number of execution strategies
- The runtime overhead is removed.

Static Analysis of queries: Static analysis is a compile time framework of static optimization it performs the following tasks:

1. Type checking: each name and each operator is checked according to the hierarchy.
2. Generating syntax trees of queries which are then modified by queries re-writing methods.

2 Building the Histogram

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. From the data distribution, we build the histogram that contains the frequency of values assigned to different buckets. Frequency distribution for numerical data is straight forward but frequency distribution for alphabetical data is not. Now considering the alphabetical data such as first names, last names, Organization names etc., question arises as to how we can split these into buckets. The idea we propose here is to group the alphabetical data with respect to the letter they start with and alphabets of similar frequency of occurrences grouped into a single bucket. To do this grouping, we make use of statistics from Figure2. that are computed by analysts showing the probable number of occurrences of each alphabet as a starting alphabet of textual data. This grouping avoids the existence of a very high frequency alphabet with a very low frequency alphabet in a bucket.(fig.2)

2.1 The Split & Merge Algorithm

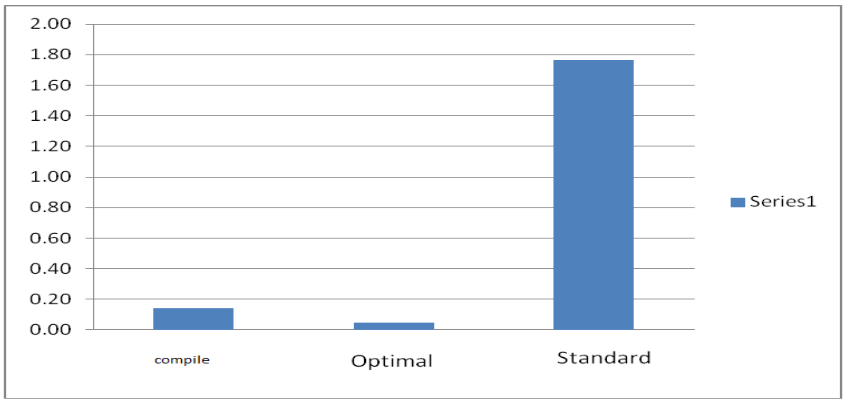
The split and merge algorithm helps reduce the cost of building and maintaining histograms for large tables. The algorithm is as follows: When a bucket count reaches the threshold, T, we split the bucket into two halves instead of recomputing the entire histogram from the data .To maintain the number 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 samples new tuple 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, a 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 predicate ordering at compile-time.

2.2 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. 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.

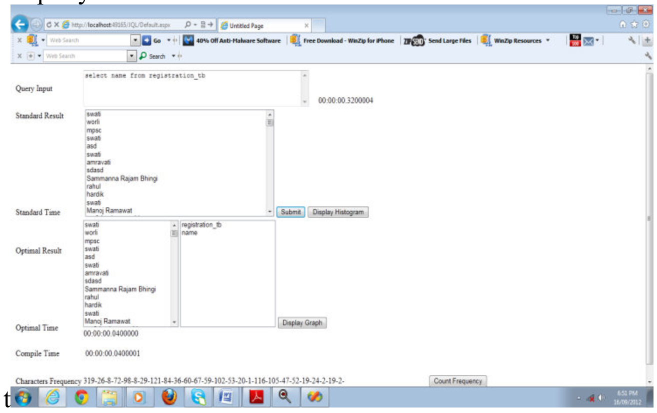
3 experimental setup and Results

The experimental trials show that our method performs better in terms of run time comparisons than the existing query optimization as showing the difference between standard result and optimized optimal result, in terms of compile time and runtime



Screenshot1: compile ,optimal and standard time

Screenshot shows main GUI consist of Query Input in which if we insert a query and then click submit button it will get Execution time of query



Query Input

```
select name from registration_tb
```

00:00:00.3200004

Standard Result

swati
worli
mpsc
swati
asd
swati
amravati
sdasd
Sammanna Rajam Bhingi
rahul
hardik
swati
Manoj Ramawat

Standard Time

Optimal Result

swati
worli
mpsc
swati
asd
swati
amravati
sdasd
Sammanna Rajam Bhingi
rahul
hardik
swati
Manoj Ramawat

registration_tb
name

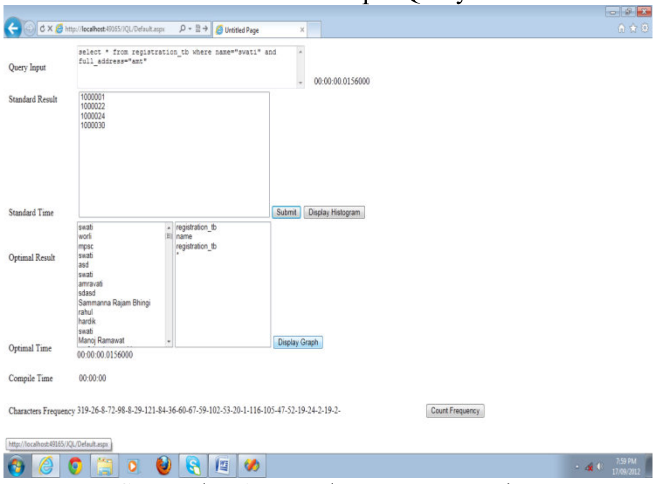
Optimal Time 00:00:00.0400000

Compile Time 00:00:00.0400001

Characters Frequency 319-26-8-72-98-8-29-121-84-36-60-67-59-102-53-20-1-116-105-47-52-19-24-2-19-2-

t

Screenshot 2: simple Query execution



Query Input

```
select * from registration_tb where name="swati" and
full_address="amt"
```

00:00:00.0156000

Standard Result

1000001
1000022
1000024
1000030

Standard Time

Optimal Result

swati
worli
mpsc
swati
asd
swati
amravati
sdasd
Sammanna Rajam Bhingi
rahul
hardik
swati
Manoj Ramawat

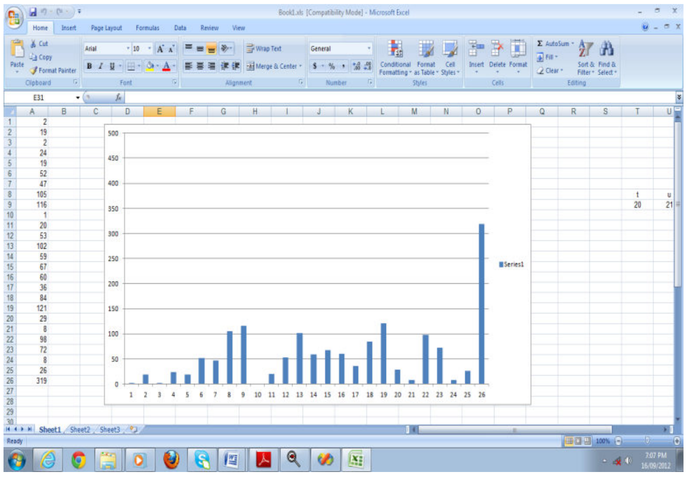
registration_tb
name
registration_tb
*

Optimal Time 00:00:00.0156000

Compile Time 00:00:00

Characters Frequency 319-26-8-72-98-8-29-121-84-36-60-67-59-102-53-20-1-116-105-47-52-19-24-2-19-2-

Screenshot 3: complex query execution



	A
1	2
2	19
3	2
4	24
5	19
6	52
7	47
8	105
9	116
10	1
11	20
12	53
13	102
14	59
15	67
16	60
17	36
18	84
19	121
20	29
21	8
22	98
23	72
24	8
25	26
26	319

Screenshot 4: histogram

Screenshot 4 shows the data of the database whenever the changes occur in database it will show in the form of histogram

References

[1] Venkata Krishna,Suhas Nerella,Swetha Surapaneni,Sanjay Kumar Madria and Thomas Weigert Department of Computer Science, Missouri University of Science and Technology, Rolla, MO Exploring QueryOptimization in Programming Codes by ReducingRun-TimeExecution,0730-3157/10©2010IEEEDOI 10.1109/COMPSAC.2010.48

[2]"Query Optimization in distributed databases" by Dilşat ABDULLAH reference Ibaraki, T. and T. Kameda. (1984). "On the Optimal Nesting Order for Computing N-Relational Joins." ACM Transactions on Data Bases 9, 482–541

[3] Chaudhuri, Surajit. *An Overview f Query Optimization in Relational Systems*. New York, USA. 1998.[4] Ioannidis, Yannis E. and Cha Kang, Younkyung. *Left-deep vs bushy trees: an analysis of strategy space and its implications for query optimization*. ACM Press, New York, USA. 1991

[5] 17. Ioannidis, Yannis E. *Query Optimization*, ACM Press, New York, USA. 1996.

[6] J. Plodzien, "Optimization of Object query Language",Ph.D.Thesis, Institute Of Computer Science,Polish Academy Of Science,2000.

[7]"Object Oriented Database System-Servey-Caixue Lin,April 2003 reference SIGMOD record 19,IEEE

[8] Ashraf Aboulnaga,Surajit Chaudhuri,"Self-tuning Histograms: building histograms without looking at data", Proceedings of the 1999 ACM SIGMOD international conference on Management of data, pp. 181-292, 1999

[9] C. Hobatr and B. A. Malloy, "The design of an OCL query based debugger for C++", In Proceedings of the ACM Symposium on Applied Computing (SAC), pages 658–662.ACM Press, 2001.

Table 1: execution time of query

Compile time	0.14
Optimal time	0.05
Standard time	1.76

Table1: shows the difference of compile time, standard time, optimal time of a query

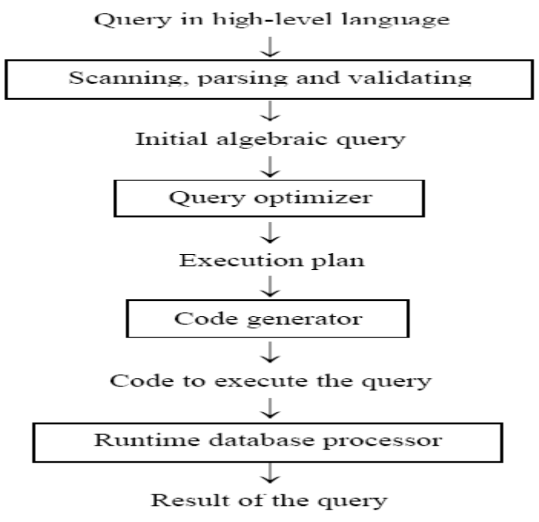

Query in high-level language
Scanning, parsing and validating
Initial algebraic query
Query optimizer
Execution plan
Code generator
Code to execute the query
Runtime database processor
Result of the query

Figure.1 query optimization process

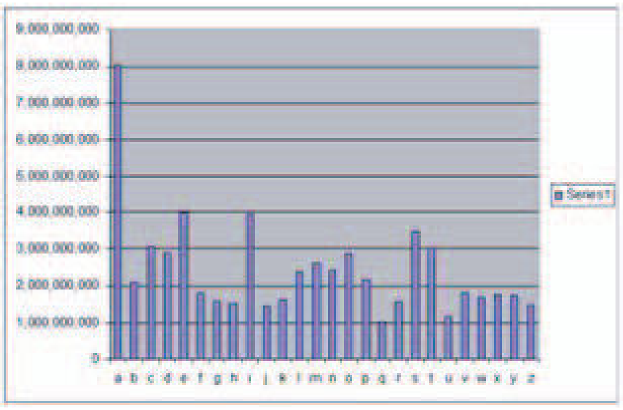


Figure.2 Frequency distribution of alphabet

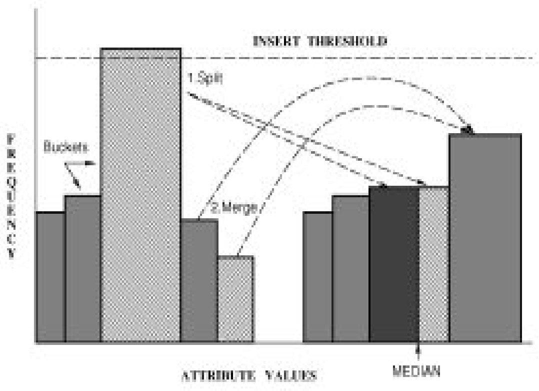


Figure .3 The Split & Merge Algorithm