ISSN No. 0976-5697

Volume 3, No. 7, Nov-Dec 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Implementation of Query Optimization for Reducing Run Time Execution

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: compile time; optimal time; histogram; query optimization;

I. 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. 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 [1]. 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.

In Query optimization the optimizer perform poorly often because their compile-time cost models use inaccurate estimates of various parameters. A novel optimization model that assigns the most of the work to compile-time and delays carefully selected optimization decisions until run-time has been explored in . Query plans are incomparable at compile time due to the missing run-time parameter bindings. Those plans are partially ordered by cost at compile-time and they use the choose-plan operator to compare those partially ordered plans at run-time. Compile-time ambiguities are resolved at start-up-time in their approach. During a query execution, values of parameters may be changed during executions. This makes the chosen plan invalid. This issue has been addressed in by proposing to optimize queries as much as possible at compile-time taking into account all possible values that parameters may have at run-time. The techniques earlier use actual parameter values at run-time and choose an optimal plan with no overhead.

A. Related Work:

In Query optimization the optimizer perform poorly often because their compile-time cost models use inaccurate estimates of various parameters. A novel optimization model that assigns the most of the work to compile-time and delays carefully selected optimization decisions until run-time has been explored in [2]. Query plans are incomparable at compile time due to the missing run-time parameter bindings. Those plans are partially ordered by cost at compile-time and they use the choose-plan operator to compare those partially ordered plans at run-time. Compile-time ambiguities are resolved at start-up-time in their approach. During a query execution, values of parameters may be changed during executions. This makes the chosen plan invalid. This issue has been addressed in[3] by proposing to optimize queries as much as possible at compile-time taking into account all possible values that parameters may have at run-time. The techniques earlier use actual parameter values at run-time and choose an optimal plan with no overhead.

A compile-time estimator that provides quantified estimate of [4]the optimizer compile time for given query has also been proposed in they use the number of plans to estimate query compilation time and employ two novel ideas:

a. Reusing an optimizer's join enumerator to obtain actual number of joins, but by passing plan generation to save estimation overhead.
b. Maintaining a small number of "interesting" properties to facilitate counting.

In query optimization approach using a regular query optimizer to generate a single plan, annotated with the expected cost and size statistics at all stages of the plan has been proposed in. During the execution of query, the annotated statistics are compared with the actual statistics and if there is a significant difference then the query execution is suspended and re-optimized using accurate value of parameters. Even though Parametric Query Optimization exhaustively determines the optimal plan in each point of the

parameter space at compile-time, it is not cost effective if the query is executed infrequently or if the query is executed with only a subset of parameters considered during compile-time. This problem has been resolved in [5] by progressively exploring the parameter space and building a parametric plan during several executions of the same query.

All these approaches involve making decision after compile-time. The way they deal with uncertainty is to wait until they have more information. This issue can handle by prefer static query optimization at compile-time over dynamic query optimization because it reduces the query run-time.

II. 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[6,7]. 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.

- ***a. Query Parser –*** Verify validity of the SQL statement. Translate query into an internal structure using relational calculus.
- ***b. Query Optimizer*** – Find the best expression from various different algebraic expressions. Criteria used is 'Cheapness'
- ***c. Code Generator/Interpreter*** – Make calls for the Query processor as a result of the work done by the optimizer.
- ***d. Query Processor*** – Execute the calls obtained from the code generator.

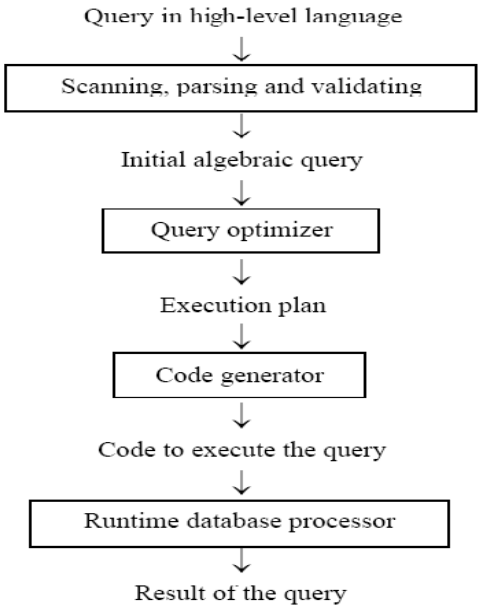


Figure.1 query optimization process

III. STATIC OPTIMIZATION

if optimization is performed once at compiled time. For each invocation at runtime [8], 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.

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

- ***a. Static Analysis of queries:*** Static analysis is a compile time framework of static optimization it performs the following tasks:
  - i. Type checking: each name and each operator is checked according to the hierarchy.
  - ii. Generating syntax trees of queries which are then modified by queries re-writing methods.

A. Dynamic Optimization:

Dynamic optimization is a mix of compile time and runtime optimization i.e. the plan produced at compile time includes plan alternatives and then actual plan to be executed is chosen at runtime. If only a part of it is rendered before query execution and the rest is made during evaluation (i.e. at runtime), then it is referred to as dynamic due to the incomparability of cost at compile time alternative plans are ordered only partially, and the final choice is delayed until state of time, when all runtime bindings can be instantiated.

Then cost calculation and comparison become feasible and the optimal plan can be chosen and evaluated.

Most of the approaches available for optimization the query involve making decision after compile time i.e. at runtime. During optimization performed at runtime a query optimizer has access to statistics that are relevant to the environment in which a given query is to be evaluated. Dynamic optimization can usually take the full advantage of statistics, thus such optimization is more effective then static but this optimization has one serious flaw, since a part it is performed at runtime and user is interactively waiting for the query result it must me very efficient itself. This means that often a query optimizer cannot consider as much optimization technique it should and the advantage of having access to current statistics cannot be fully utilized. So we prefer The static query optimization at compile time over dynamic query optimization because it reduces the query at runtime.

Static optimization does not have the flaws, but its disadvantage is that it usually can use only some estimate of environment in which queries to be evaluated. Usually such an estimate is updated only periodically i.e. not updated every time the state of database changes. However, practice has proved that statistically this advantage is not critical specially for database with stable states besides, if the estimate of statistics start to be irrelevant, a query can be re-compiled for new, more up to date estimate. Therefore, current DBMS apply mainly static optimization.

During static analysis we simulate runtime query evaluation to gather information that we need to optimize the queries. The general architecture of query processing is shown below a parser of queries and program takes a query source is input makes syntactic analysis and returns a query/program syntactic tree. A program/query syntactic tree is a data structure which keeps the abstract syntax in a well structured form allowing for easy manipulation. Each node of the tree contains a free space for writing various query optimization information.

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

There are various method available for query optimization; we proposed a method 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, so the proposed method reduce the run time

Implementation of proposed work of query optimization takes place through following ways

a) Query Parsing
b) Query Optimizing
c) Query compilation &execution

B. Analysis:

During a query execution, values of parameters may be changed during executions. This makes the chosen plan invalid. This issue has been addressed in by proposing to optimize queries as much as possible at compile-time taking into account all possible values that parameters may have at run-time. The proposed techniques earlier use actual parameter values at run-time and choose an optimal plan with no overhead. Query optimizers often make poor decisions because their compile-time cost models use inaccurate estimates of various parameters. There have been several efforts in the past to address this issue, which can be categorized as – strategies that make decisions at the start of query execution and strategies that make decisions during query execution. There are some parameters, like memory availability, whose value cannot be predicted at compile-time, but are accurately known at the start of execution. Assuming that the values of these parameters remain constant for the duration of the execution The way they deal with uncertainty is to wait until they have more information. Therefore, we propose to use histograms to estimate selectivities of joins and predicates in a query at compile-time.

In this method, we prefer static query optimization at compile-time over dynamic query optimization because it reduces the query run-time. To achieve this, 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 better query plans. For such estimations and predictions we need to have information such as sizes of relations, sizes of intermediate results. 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.

Analyze query using compiler techniques

a. Verify relations and attributes exists.
b. Verify operations are appropriate for object type
c. Transform the query into some internal representation.

C. Optimization Strategy:

a. In Compile time optimization The query is parsed, validated, and optimized once.
b. In this implementation We Optimize query Q, store the plan, and run it whenever Query is executed.

IV. ESTIMATING SELECTIVITY USING HISTOGRAM

The selectivity of a predicate in a query is a decisive aspect for a query plan generation [9]. 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. A detailed of how the histograms are built is given in the following section.

V. 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[9].

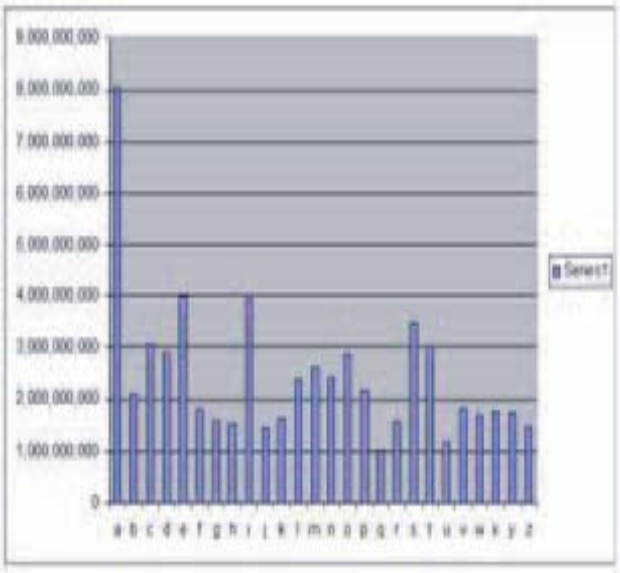


Figure.2 Frequency distribution of alphabet

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.

VI. THE SPLIT & MERGE ALGORITHM

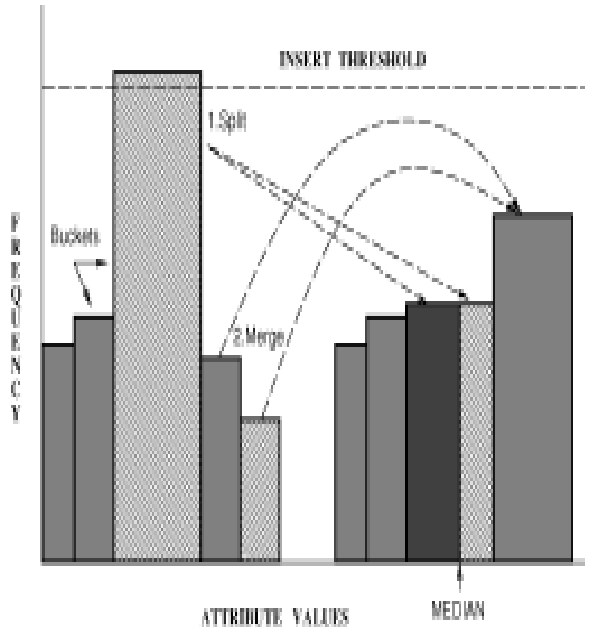


Figure .3 The Split & Merge Algorithm

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 [9].

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.

VII. EXPERIMENTAL SETUP AND RESULTS

a. 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 we have obtained the query plan at compile-time and then we executed the query plan at run-time.
b. Our approach reduces run-time execution less than the existing code's run-time due to our approach of optimizing the query and handling data updates using histogram.
c. The work of query optimization is performed in compile time by generating compiler or parser which generate flexible plan for input query, so that the least amount work is left to be done in runtime thus at runtime, the time required for plan construction is omitted and reduce a runtime of query.

The comparative analysis of query optimization for relational queries that it will get, Optimal time (i.e. optimized time) which is less than the standard time (i.e. without optimization strategy).

The optimization performed at is compile time so, the action entirely performed before query is executed, hence query optimization process itself does not burden the performance. Rewriting requires performing special phase called static analysis. During the static analysis we simulate runtime query evaluation to gather all information that we need to optimize queries.

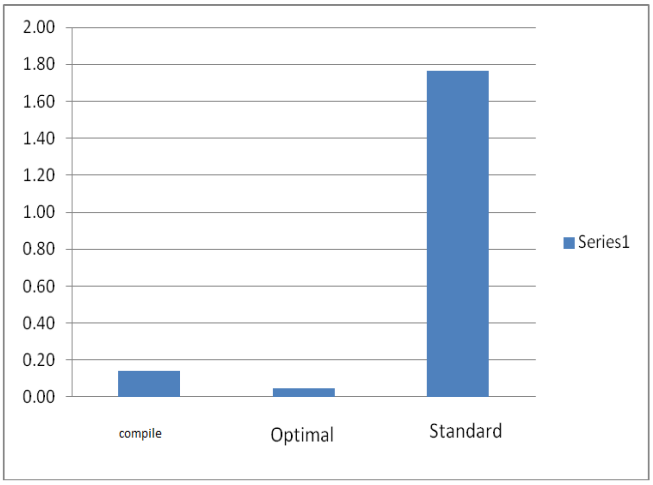
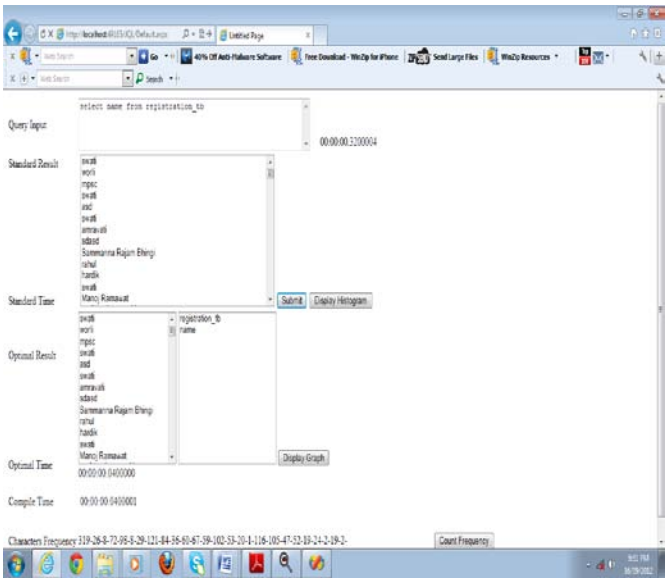


Figure. 5 compile ,optimal and standard time

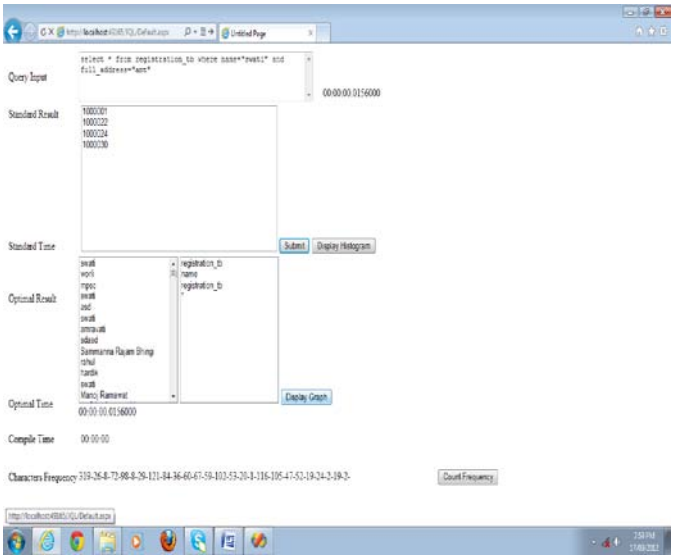
Table 1: execution time of query

Compile time	0.14
Optimal time	0.05
Standard time	1.76

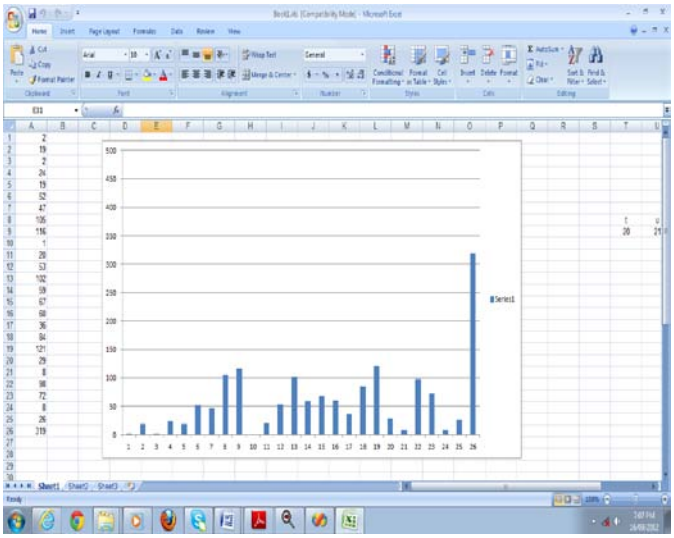
Screenshot shows main GUI consist of Query Input in which if we insert a query and then click submit button it will get
Execution time of query.



Screenshot 1: simple Query execution

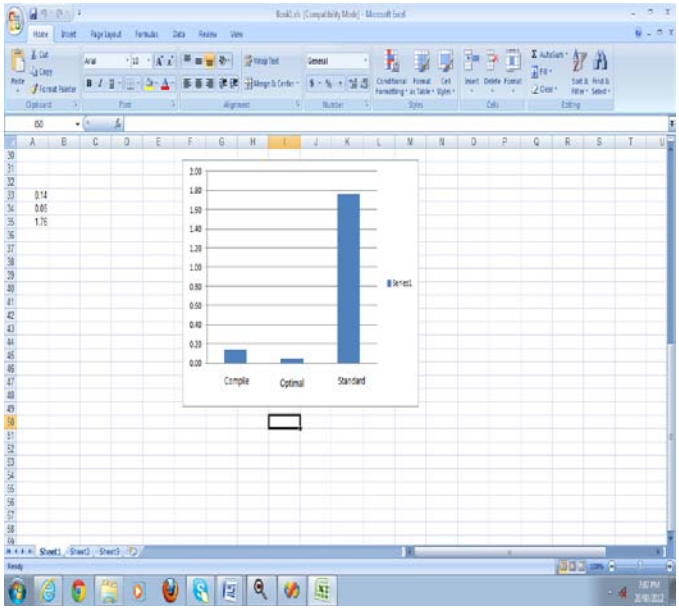


Screenshot 2: complex query execution



Screenshot 3: histogram

Screenshot 3 shows the data of the database whenever the changes occur in database it will show in the form of histogram



Screenshot 4: comparison of compile, Optimal, Standard time

VIII. CONCLUSION

This work is motivated by the fact that the query optimization strategies from database domain can be used in improving the run time executions in programming languages. We have implemented a technique for query optimization at compile-time by reducing the burden of optimization at run-time, to improve the performance of the code execution, to optimize queries at compile-time taking into all possible values that parameters may have at run-time. To achieve this, we intend to have the query plans generated at compile time. Estimate of selectivity for query joins and predicates in a query at compile-time. Using histograms that are computed from the data and these histograms are used to get the frequency of alphabet of the database. Furthermore, our query evaluation performs well for different types of queries as we have shown in our experimental results.

IX. REFERENCES

[1]. Ioannidis, Yannis E. "Query Optimization", ACM Press, New York, USA. 1996.

[2]. Ramakrishnan, R. and Gehrke, J. (2000). Database Management Systems Third Edition. McGraw Hill.

[3]. Ihab F. Ilyas, Jun Rao, Guy Lohman, Dengfeng Gao, Eileen Lin, "Estimating Compilation Time of a Query Optimizer," Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 373 – 384, 2003.

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

[5]. Pedro Bizarro, Nicolas Bruno, David J. DeWitt, " Progressive Parametric Query Optimization," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

[6]. "Object Oriented Database System-Servey-Caixue Lin,April 2003 reference SIGMOD record 19,IEEE computer,33,No.8:16-19,2000

[7]. "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

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

[9]. "Query Optimization in Programming Codes by ReducingRun-TimeExecution", 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 0730-3157/10©2010IEEEDOI 10.1109/COMPSAC.2010.4