2010 34th Annual IEEE Computer Software and Applications Conference

Exploring Query Optimization in Programming Codes by Reducing Run-Time Execution

Venkata Krishna Suhas Nerella, Swetha Surapaneni, Sanjay Kumar Madria and Thomas Weigert
Department of Computer Science, Missouri University of Science and Technology, Rolla, MO
{(vnhh4, ssfz2, madrias, weigert)@mst.edu}

***Abstract*—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, Java Query Language has been developed that allows object querying and performs the query optimization at run-time. Therefore, only one problem is how to reduce the task of query optimization at run-time as much as possible within the Java Query Language system. 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. The experimental trials show that our method performs better in terms of run time comparisons than the existing query optimization techniques used in the Java Query Language.**

Keywords- selectivity; joins; compile time; run-time; histograms; query optimization

I. INTRODUCTION

First class query constructs are being introduced in many object-oriented programming languages. These constructs help in increasing the legibility of the programs and capability of the programmers. Various constructs such as C# LINQ [1], Python comprehensions [2] and Java Query Language [5] allow queries to be written in a concrete manner. These query constructs have various benefits over representing queries implicitly. Explicit queries can be more concise and clear than the queries written by making use of other APIs. These query constructs also allow developers to be more productive and work at a higher level of abstraction. LINQ provides uniformity by giving a set of query operators that operate over various data sources like objects, XML, and relational data.

JQL [5] 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. For example, if there is a nested loop iterating on two collections in a code then the loop is executed by iterating over both the collections, whereas in JQL, the query evaluator will select a method for joining two collections together by making use of join query optimization techniques that a developer may think too complex or time-consuming to write. 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.

Object collections are mutated in object-oriented languages as objects will be added or removed during the execution of a program. Therefore, same query evaluation at two different places in the program may produce different results. The issue of updates to underlying data does not arise in list and set comprehensions in functional languages. However, this issue has been handled in Java Query Language (JQL) by generating the dynamic join ordering strategies.

In this paper, we address the problem of reducing the burden of query optimization to the query optimizer at run-time in Java Query Language to a significant extent. Our key concept is to perform the task of query optimization at compile-time as much as possible. We improvise over JQL on the issue of handling updates to data using histograms discussed later in this section.

The main issues addressed in this paper are:

1. How to shift most of the work of query optimization from run-time to compile-time so that least amount of work is left to be done at 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.

2. What information is needed to allow the prediction and the code to work in a static fashion? Given a join query, its selectivity needs to be estimated to design better query plans. For such estimations and predictions we

IEEE computer society

need to have information such as sizes of relations, sizes of intermediate results, etc. To accomplish this task, we propose the following:

- Perform some sample query executions.
- Have an estimate of pattern of database changes.
- From the results of sample queries, estimate the selectivities using histograms.
- Record change of data periodically before compile time, estimate delta change and pattern of changes so that the histograms are adaptable for data additions.

This paper describes a method using histograms to get the estimates of selectivity. In JQL [5], 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.

II. Related Work

Query optimizers 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 [9]. 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 [4] 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.

An 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 [12]. 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 [20] by progressively exploring the parameter space and building a parametric plan during several executions of the same query.

A compile-time estimator that provides quantified estimate of the optimizer compile time for given query has also been proposed in [7]. They use the number of plans to estimate query compilation time and employ two novel ideas: (i) reusing an optimizer's join enumerator to obtain actual number of joins, but bypassing plan generation to save estimation overhead; (ii) maintaining a small number of "interesting" properties to facilitate counting.

Algorithms for compile-time regular path expression expansion in the context of Lorel query language for semi-structured data have been explored in [6]. They expand regular path expressions at compile-time using the structural summary and thus, reducing the run-time overhead of database exploration.

All these approaches involve making decision after compile-time. 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 our research, we prefer static query optimization at compile-time over dynamic query optimization because it reduces the query run-time.

III. Estimating Selectivity Using Histograms

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 form T.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. A detailed description of how the histograms are built is given in the following section.

A. 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. These sample executions consume a small amount of the available resources._From the results of these queries, we will estimate frequencies for the histogram. However, the underlying data can tend to undergo changes. Thus we need to have an estimate for the pattern of data changes. For this, we record changes in data, estimate delta change and pattern of change which can be inferred as the executions proceed so that the histograms are adaptable for data additions.

B. Incremental Maintenance of Histograms

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 following equations.

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

$$T_i = \frac{w_1\mu_1 + w_2\mu_2 + \cdots + w_n\mu_n}{w_1 + w_2 + \cdots + w_n} \qquad (2)$$

where μ_a 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
- f_i is the frequency of bucket i as in the histogram
- q_f=S/N is the query frequency
- B_i is the observed frequency
- T_i is the error estimate for each individual table
- W_i are the weights with respect to every attribute depending on the rate of change

If the calculated error (T_i) is > 0.5 then we update the histogram. Otherwise we use the same old histogram to give the selectivity estimate. Next, we scan the database and update buckets. If some buckets exceed a fixed threshold then we use split and merge algorithm. However the issues are how and when we know that the underlying database has been updated. For this, a heuristic that can be used is to consider popular queries. A popular query is a query that has high frequency of occurrence. These popular queries can help in reporting data changes.

We can constantly keep track of the result set of a popular query. When the results of consecutive executions of this query do not match, it indicates a database update and thus we can compute the error and decide whether to recompute the histogram or to continue with the existing histogram. However, we do not want to recompute a histogram for a table that is not often accessed. Thus, we make use of the frequency of access of a particular table to decide when and when not to compute the histogram. If the access frequency is getting higher then it increase its probability and the corresponding histogram needs to be maintained up-to-date. Access Frequency represents the number of tuples accessed by a query.

When the access frequency is high, and the tuples are accessed more often, we need to recompute the histogram. When the access frequency is low, the tuples are not accessed frequently and therefore, there is no need to recompute the histogram even in case of a data change.

When building a histogram, we need to assign the values to 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 Figure 1 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.

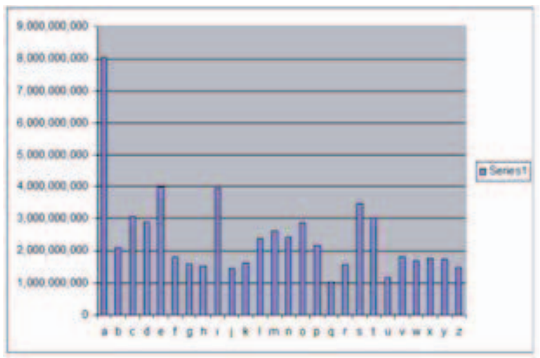

Figure 1. Frequencies of alphabets [22]

C. 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. If that weighted average is greater than a certain threshold (say 0.5) then the table's histogram must be updated. Here is the approach for the error estimates using the equations from Section III-B:

- For every selection on the histogram attribute, we compute the approximation error (T_i). For each table, we compute error estimate for all the attributes (μ_a) in that table. Then for each table, we take a weighted average of all the attribute errors. If that computed error (T_i) is greater than a threshold, and then we update histogram otherwise we need not update the histogram. This is shown in the Figure 2 (Lines 1-7).
- If error (T_i) > 0.5 then we scan the database and update buckets.
- If some buckets exceed a threshold then we use split and merge algorithms.

```
Function  ErrorEstimate
Inputs : H "Histogram"
Outputs : $H_{new}$ "Updated Histogram based on estimate of the error"
1:    for T in TableList
2:    begin
3:        for a in T.attlist
4:        begin
5:            calculate $\mu_a$
6:        end
7:        calculate $T_i$

8:         if $T_i > 0.5$ then
9:             $H_{new}$=Update_Histogram(H);
10:            return $H_{new}$;
11:        else
12:            return H;
13:        end
14:   end
```

Figure 2. Error Estimate Algorithm

D. *Query Evaluation*

Given a query Q, we use the histogram H to get the estimate of the selectivity of the query predicates and the selectivities of the joins. Now we have the join order and predicate order in a query which will be used to construct a query plan. Below we discuss the possible cases.

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. The algorithm for query evaluation is presented in Figure 3. Firstly, we check if the query is present in the log (Line 2) 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 recompute the histogram because we have assumed the data may be modified within a pre specified time interval (Lines 3-5). If the time period difference is less than the threshold, we first compute the error through error estimate function discussed in Section III-B and then based on the error estimate we decide whether to recomputed the histogram or not (Lines 7-12). 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.

```
Function Evaluate Query
Input: q "Query to be processed" ,Th "Time period Threshold", H "Initial Histogram", $T_{cur}$ "Current Time period of the query", $T_{prev}$ "Previous Time period of execution of query"
Output: rs "result set of a query"
1:  rs:=NIL
2: if  log_contains(q)=true then
3: if $T_{cur}$-$T_{prev}$> Th then
4:          $H_{new}$=Update_Histogram(H);
5:          rs=exec(q,$H_{new}$);
6:              else
7:        if check_DBupdate()=true then
8:                $H_{new}$=Update_Histogram(H);
9:                rs=exec(q,$H_{new}$);
10:        else
11:                $H_{new}$=H;
12:                rs=exec(q,$H_{new}$);
13:        end
14: end
15: else if log_contains(q)=false then
16: rs=exec(q,H);
17: end
  18:   return rs;
```

Figure 3. Evaluate Query Algorithm

E. *The Split & Merge Algorithm*

The split and merge algorithm [2] 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 recompute 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 recompute 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. We present the experimental results of how our approach for various types of queries in the next Section IV.

IV. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the proposed query evaluation and the histogram maintenance algorithms with several experiments. The algorithms are implemented in Java. For all the experiments, we have used an Intel Pentium IV 3.2 GHz, with 1.75 GB RAM running Eclipse v3.4.0.

We have considered four queries of differing complexity based on the number of joins to explore the effect of query size on performance. The four queries are described in Table 1. We have tested our algorithms performance using these benchmark queries.

TABLE I. DETAILS OF BENCHMARK QUERIES

Query1: selectAll(Student s, Faculty f, Course c| s.id=2); This query requires only the estimate of predicate which is directly made from the histogram of the student attribute id.
Query2: selectAll(Student s, Faculty f| s.name=f.name); This query has only one join, so no need of ordering, which can be also estimated easily from the student and faculty name attribute sizes.
Query3: selectAll(Student s, Faculty f| s.name=f.name and s.id=2); This query requires ordering of join and predicate. Predicate estimate is made from the histogram of student attribute id. And the selectivity of the join is made from the estimate of the student and faculty name attribute sizes.
Query4: selectAll(Student s, Faculty f, Course c| s.name=f.name and f.name=c.fname); This query requires two joins and can be optimized by hash join rather than nested loop join.

A. Observations

We have conducted several experiments on these benchmark queries. In each experiment, we have taken a query and executed it using the JQL optimization strategy and our approach. We measured the run-time of both the approaches for the query execution. The average run-time was taken over 50 runs for both the approaches. Similarly, we have performed experiments for all the benchmark queries and plotted the graphs in Figure 4 and Figure 5.

Figure 4 and Figure 5 show 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.

And from the Figure 5, we can see that as the number of joins increase in a query, JQL's approach becomes more expensive and our approach performs much better than the exhaustive join ordering strategy of JQL.

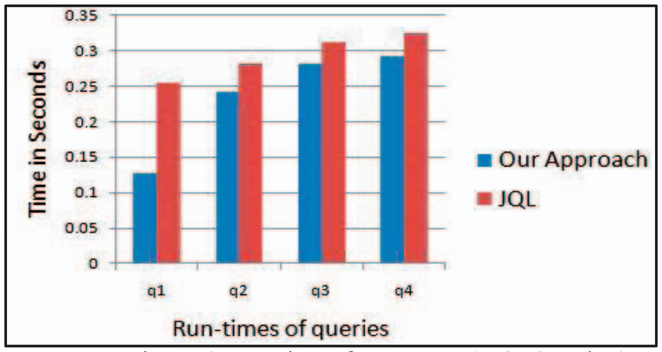

Figure 4. Run times for queres q1,q2,q3 and q4

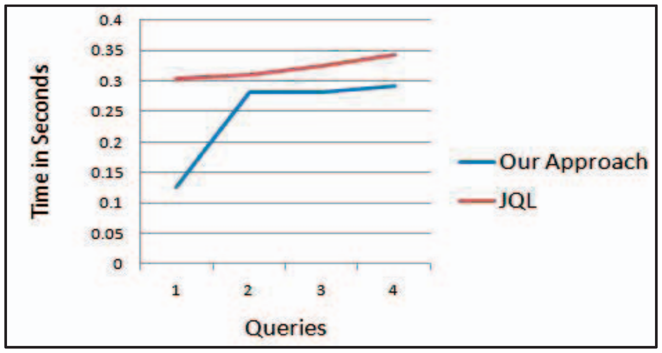

Figure 5. Run-times comparison

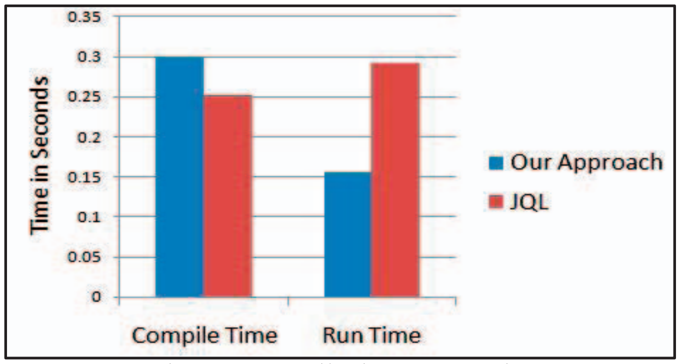

Figure 6. Compile-time vs. Run-time

Figure 6 shows the difference achieved in run-time and compile-time for execution of query q1. We can clearly see that our approach has decreased the run-time for the execution of query. However, the compile-time for our approach is slightly higher than the JQL because we are generating query plan at compile-time.

In Figure 7, we have shown the graph for error that the approaches give when the data is changing in between the query executions. As we can see from the graph, our approach gives an almost steady error even when objects are being added to the collections, whereas in JQL, as the number of objects in the collections is increasing, the error is also increasing.

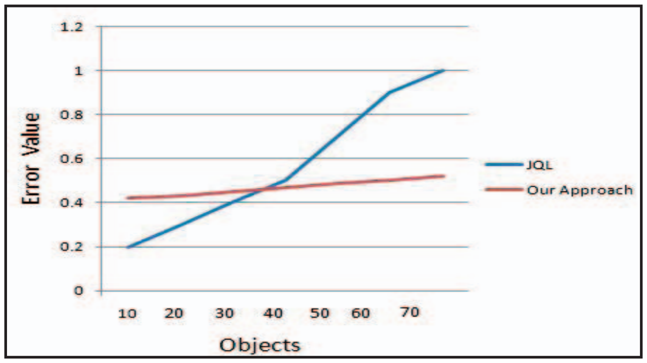

Figure 7. Error with change of data

V. CONCLUSION AND FUTURE WORK

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. In this paper, 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. Experimental results showed that error estimate and split merge algorithms are efficient and maintain the histograms accurately. Furthermore, our query evaluation algorithm performs well for different types of queries as we have shown in our experimental results.

REFERENCES

[1] Phillip B. Gibbons, Yossi Matias, Viswanath Poosala, “Fast Incremental Maintenance of Approximate Histograms,” ACM Trans actions on Database Systems, vol. 27, pp. 261-298, 2002.

[2] 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.

[3] Gennady Antoshenkov, Mohamed Ziauddin, “Query processing and optimization in Oracle Rdb,” VLDB Journal, vol. 5, Issue 4,pp. 229-337, 1996.

[4] Yannis E. Ioannidis,”Query optimization,” ACM Computing Surveys, vol. 28, pp. 121-123, 1996.

[5] Darren Willis, David J. Pearce, James Noble, “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.

[6] Jason Mchugh, Jennifer Widom, “Compile Time path expansion in lore,” In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, 1999.

[7] 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.

[8] Darren Willis, David J. Pearce and James Noble, “Efficient Object Querying in Java,” In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2006.

[9] 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.

[10] Richard L. Cole, Goetz Graefe, “Optimization of dynamic query evaluation plans,” Proceedings of the 1994 ACM SIGMOD international conference on Management of data, pp. 150-160, 1994.

[11] P.G. Selinger, “Access path selection in relational database systems,” Proceedings of 1979 ACM SIGMOD International Conference on Management of Data.

[12] Navin Kabra, David J. DeWitt, “Efficient mid-query re-optimization of sub-optimal query execution plans,” ACM SIGMOD Record, vol. 27, pp. 106-117,1998.

[13] Francis Chu, Joseph Y. Halpen, Praveen Seshadri, “Least expected cost query optimization: an exercise in utility,” Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 138-147, 1999.

[14] Surajit Chaudhuri,”An overview of query optimization in relational systems,” Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pp. 34-43, 1998.

[15] Donald Kossmann, Konrad Stocker, “Iterative dynamic programming: a new class of query optimization algorithms,” ACM Transactions on Database Systems, vol. 25, pp. 43-82, 2000.

[16] Michael Steinbrunn, Guido Moerkotte, Alfons Kemper, “Heuristic and randomized optimization for the join ordering problem,”VLDB Journal, vol. 6, pp. 191-208, 1997.

[17] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Arun N. Swami, Balakrishna R. Iyer, “A Polynomial Time Algorithm for Optimizing Join Queries,” Proceedings of the Ninth International Conference on Data Engineering, pp. 345-354, 1993.

[18] Joseph M. Hellerstein, Michael Stonebraker, “Predicate migration: optimizing queries with expensive predicates,” ACM SIGMOD Record, vol. 22, pp. 267-276, 1993.

[19] Y.E. Ioannidis, Younkyung Kang, “Randomized algorithms for optimizing large join queries,” ACM SIGMOD Record, vol. 19, pp. 312-321, 1990.

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

[21] K.D. Seppi, J.W. Barnes, C.N. Morris, “A Bayesian approach to database query optimization”, ORSA Journal on Computing, pp. 410-419, 1993.

[22] Frequencies of alphabets.
http://4.bp.blogspot.com/_0UyEDiS7nik/RkdqEytnyil/AAAAAAAAACY/7y7Jo5VAk1A/s1600/Slide1.JPG.