2011 35th IEEE Annual Computer Software and Applications Conference

Performance Improvement for Collection Operations Using Join Query Optimization

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

***Abstract*—Programming languages with explicit support for queries over collections allow programmers to express operations on collections more abstractly than relying on their realization in loops or through provided libraries. Join optimization techniques from the field of database technology support efficient realizations of such language constructs. We describe an algorithm that performs run-time query optimization and is effective for single runs of a program. The proposed approach relies on histograms built from the data at run time to estimate the selectivity of joins and predicates in order to construct query plans. Information from earlier executions of the same query during run time can be leveraged during the construction of the query plans, even when the data has changed between these executions. Experimental results demonstrate improvement over earlier approaches, such as JQL.**

Keywords- collection operations; joins; query optimization;

I. INTRODUCTION

A typical program will perform queries over collection data types, such as sets, by examining the collection in a loop. For example, scanning all members of a set for whether they satisfy a certain condition is a low-level way of realizing a select query for that set. Data structure libraries (e.g., STL [4]) or language level operations providing mapping operators simplify the construction of such programs. However, if language were to support queries as first-class language constructs, the level of abstraction of such programs would be raised significantly. Abstract queries over data structures would allow the programmer to filter or join these data structures, and produce new data structures in a straightforward manner. Queries as first-class citizens of the language would allow programmers to decide "which" queries to write instead of focusing on "how" these queries are implemented. A compiler could then optimize the queries into high-performance implementations.

Language level constructs such as set and list comprehensions have been developed early for SETL [14] and are popular in software design languages such as OCL [2]. Programming languages such as Python or LINQ [12] have incorporated first-class query constructs. These query constructs are more concise and readable than their equivalent implementations through (possibly nested) loops.

The Java Query Language (JQL [15, 16]) allows the programmer to write queries explicitly in Java code. JQL supports automatic optimization using join query optimization techniques taken from the database domain: A query plan is constructed at run time after incorporating information concerning the size of relations, etc. JQL performs sampling on a small number of tuples to determine the selectivity of joins and predicates in a query. During the execution of the program, objects may be added, modified, or deleted. Consequentially, the same query may produce different results when invoked multiple times during the execution of the program. JQL handles this issue by generating dynamic join order strategies [11] at run time. However, selectivity estimates based on sampling a small number of tuples does not lead to an efficient ordering of joins and predicates in a query. In addition, query optimization imposes a run-time burden on the program.

In order to reduce this run-time overhead, [13] proposed to perform query optimization [8] at compile time. However, this approach focused on the optimization of multiple subsequent executions of the same program. After executing several sample queries, selectivities were estimated from histograms [1] and future data changes were predicted from update patterns. During each run of the program, histogram and data changes learned were recorded so that an optimal query plan could be selected and executed during the next run of the program.

The focus of this paper is on performing query optimization during a single run of a program. In this situation, estimations concerning the data as well as information regarding changes to the data will not be available until run time. Any learning that happens regarding the data has to be leveraged within a single run, from one execution of a query to the next. Consequentially, query optimization will also be performed at run time.

This approach to query optimization introduces additional run time overhead. However, the cost of building histograms is mitigated, to some degree, by that each data item needs to be scanned only once, in order to place it into the corresponding bucket of the histograms. This cost is further reduced by determining selectivity values through a simple look-up of the histograms. Building histograms at run time has the advantage that multiple scans of data for each evaluation of a query (as in JQL) are avoided. By eliminating multiple scans over the data, our approach incurs less run time overhead than JQL, despite that our approach introduces additional overhead associated with building and maintaining histograms [7].

This paper describes an algorithm for query optimization at run time that is effective even for a single run of a program. We have adapted the approach proposed in [13] by building the histograms from data at run time and using those histograms to determine the selectivity of joins and predicates in a query. After determining the selectivities, we construct the query plan which orders joins and predicates in a query. The query will be executed using that query plan. Experimental evaluation using different types and number of joins demonstrates that our approach results in a reduced run-time overhead compared with that of JQL.

The rest of the paper is organized as follows. In Section II, we present related work on query optimization. Section III provides an overview and motivation for our work. Section IV describes our approach to query optimization for single program runs. Section V presents the performance evaluation of this work, and Section VI provides conclusions and directions for future research.

II. Related Work

The relational database literature is rich in research on join query optimization. A significant amount of work has been focused on the optimization of queries at compile time, but significant limitations of this approach have been discussed [8]. More recent work on query optimization has focused on postponing optimization decisions to query execution time.

Parametric query optimization [9] identifies multiple execution plans at compile time which are optimal for a subset of all possible values of the run-time parameters. Even though this approach is able to identify the appropriate query plan without run-time overhead, when actual parameters are available, it explores the search space exhaustively and also tends to miss statistical errors in its estimates.

In [6], most of the optimization effort is performed at compile time and only selected optimization decisions are delayed until run time. Choose-plan operators are used to execute the delayed decisions. The dynamic plans remain optimal even if parameters change after the program has been compiled but before it is run. There is an overhead associated with selecting which decisions to delay as well as with the implementation of the choose-plan operator.

In [10], an algorithm has been proposed that detects the sub-optimality of a query execution plan by collecting statistics at significant points during the query execution. These statistics help determine whether to change the execution plan for the remainder of the query.

Dynamic query evaluation [6, 11] and parametric query optimization [3, 9] generate a number of query plans that are optimal for different run-time data. However, the complexity of this approach increases dramatically as the number of unknown run-time data items increases. These approaches rely on randomization when exploring the huge search space or are forced to make simplifying assumptions. There is no run-time overhead, but these approaches have the disadvantage of not detecting statistical errors in estimates.

III. Overview and Motivation

A query as first-class programming concept is an abstract operation over collections. Programmers are able to write queries explicitly rather than expressing their low-level realization, for example, in terms of (nested) loops. The compiler will perform this realization, relying on join optimization techniques from relational database technology. In relational databases, a join is an operation that combines two relations on a common attribute; the result of this operation is the set of matching tuples constructed from the joined relations. Joins are expensive operations in a query. Therefore, database engines attempt to minimize the number of joins performed as well as to minimize the size of the relations that are being joined. A powerful optimization is to construct a sequence of joins to be performed such that the overall cost of the joins is minimized.

Similarly, when using a query as an abstraction in programs, a join is an operation that combines two collections; the output of the join will be a new collection holding the elements matched in the joined collections. Join query optimization here will attempt to avoid constructing unneeded intermediate collections as well as minimize the size of such intermediate collections.

For example, assume a program iterates over two collections in a nested loop to find the matching items in both collections. The nested loop will be transformed to an object query. With queries as first-class language constructs, the two collections are the domain variables and the condition that determines which elements constitute a match comprises the join operation.

In this paper, we propose an approach to optimize the execution of programs containing queries as first-class constructs. From now on, when speaking of queries, query execution, etc., we will refer to queries as first-class constructs in programs, not to database queries.

IV. APPROACH

Joins are the most expensive operations involved in executing a query. Therefore, determining the optimized order of executing joins is a key step in query optimization. Joins should be ordered such that the results of one join cascades fewer tuples as the input for the next join, as the sequence of queries is executed. Using the language of nested loops, join ordering corresponds to the order in which loops are being nested. To obtain a query plan with the proper ordering of join and predicate executions, information about the data manipulated, such as the size of collections, is required.

In our approach for query optimization, we begin by building histograms from the collections. We then compute the selectivity of joins and predicates. Next we generate the optimized query plan and execute the query. After query execution, we determine whether the query should be cached so that its result can be reused later in the program run, using heuristics such as frequency, selectivity and impact of updates on joins. Details for each of these steps are given in the following subsections.

A. Building Histograms at Run time

During the execution of the program, the histograms are built while scanning the data, and each element is mapped into a corresponding bucket of the histogram. If no element is mapped to a particular bucket, the bucket is set to one, or the previous bucket count is incremented by 1. The data elements may change during program execution due to additions, deletions, or modifications of the collections. As the collections change, the frequency counts of the corresponding histogram buckets must be updated. However, updating the frequency counts after every single change may place an undesirable burden on program execution, as subsequently the selectivity values must be recomputed from the histogram also. As mitigation, histograms are only updated when the change in the data is deemed significant, judged by exceeding a specified threshold.

B. Incremental Maintenance of Histograms

The data elements in collections may change from one evaluation of a query to the next. These changes in the data will be reflected in the histogram when it is determined that the cost of updating the histogram is lower than the cost of using the current histogram, based on an error estimate. Histograms can be dynamically updated between multiple evaluations of the same query. Histogram maintenance is performed utilizing the split and merge algorithm detailed in [13].

When data has been updated between multiple evaluations of the same query, the estimation error of a histogram can be computed using the formula below, proposed in [13]:

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

$$T_R = \frac{\sum_{i=1}^{n} w_i \mu_i}{\sum_{i=1}^{n} w_i}$$

where

- μ_a is the estimation error for attribute a in the collection R
- 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 in the histogram
- S/N is the query frequency
- B_i is the observed frequency
- T_R is the error estimate for R
- w_i is the weight of attribute i depending on the rate of change of the attribute

If the computed error estimate (T_R) is > 0.5, then the histogram is updated, otherwise the original histogram is used to determine the updated selectivity values. The error estimate is a vital piece of our method as it helps to decide when and if the histograms are updated. This is important because updating the histograms forces the recomputation of the selectivity values. This involves the rescanning of the histograms of the two attributes and re-estimating the count of matching tuples. The error estimation step therefore reduces the overhead of re-computing the histograms and selectivity values.

C. Determination of Selectivity from Histograms

The selectivity of predicates and joins in a query need to be computed in order to construct a query plan. The selectivity of a predicate is defined as the number of tuples in a collection satisfying the predicate. We use the frequency of the buckets in the histogram for the predicate to establish its selectivity. The selectivity of a join is defined as the number of matching tuples in any two collections, divided by the cross product of the size of the collections. It is similarly computed by counting the total number of frequencies of the matching histogram buckets.

For a join of two collections on an attribute, the number of matching tuples cannot be determined without executing the query. We can, however, estimate the matching number of tuples from two histograms that have been built for that attribute with respect to the two collections. The attribute domain is partitioned into equal intervals for buckets in the two histograms. For each interval, we will take the maximum of the two bucket values in that range because there can be at most that many matches in that range. Similarly, for all other buckets in the two histograms, the maximum of the two bucket values in each range will be taken. The sum of these values represents the estimated count of the matching number of tuples. This estimate will be continuously improved as after each execution of a query, we maintain statistics for each join, such as the actual selectivity and frequency of the join.

$$\text{Estimated count} = \sum_{i=1,j=1}^{i=n,j=m} \text{Max}(x_i, y_j)$$

where

- n is the number of buckets in histogram x
- m is the number of buckets in histogram y
- i, j are the indices for the buckets in histograms x and y respectively.

For example, consider the query SelectAll(Student s, Faculty f | s.name.equals(f.name) && s.id.equals(f.id)). We rely on the histograms to estimate selectivity, as follows. For the *Student* and *Faculty* collections, the histograms H_1 and H_2 have been constructed for the attribute *id*, respectively, as shown in Figure 1. Assume that *Student.id* and *Faculty.id* have a range of values from 1 to 14. Assume that the frequencies of these values in the collections have been arranged into buckets of the histogram as shown below.

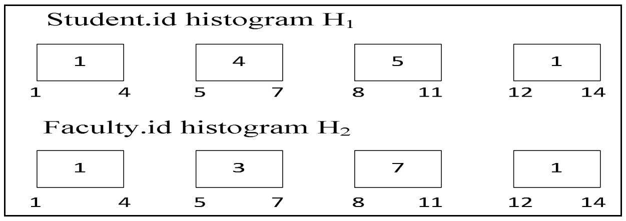

Figure 1. Example of histogram buckets for an attribute

Compute the maximum values in each interval range.

- Interval 1-4 (I_1): max(H_1, H_2)=1
- Interval 5-7 (I_2): max(H_1,H_2)=4
- Interval 8-11(I_3): max(H_1,H_2)=7
- Interval 12-14 (I_4): max(H_1,H_2)=1

The estimated count is the sum of the maximum values from all the intervals.

- Estimated count=I_1+I_2+I_3+I_4=1+4+7+1=13

The selectivity of the join equals the estimated number of matching tuples divided by the product of the sizes of the two relations. Therefore, S.id = 13/11*12 = 0.098. The selectivity is easily computed for all other joins.

D. Query Evaluation

During execution of the program, many queries may be evaluated. For each execution of a query, the cheapest query plan needs to be determined so that the cost of executing the queries is minimized. Queries may or may not be repeated during a single execution of the program. If the same query is repeated several times, important information can be learned from its previous execution. Further, the query results may be cached depending on the established cache policy which further reduces the cost of repeated evaluation of the query. The following four cases may occur with respect to the evaluation of a query:

In Case 1, a query occurs for the first time, so there are no cached results and no previous executions of the query are available. In order to construct a query plan for this query execution, we need the selectivity of joins and predicates in the query (which we obtain from the histograms). After determining the selectivities as described in Section IV.C, the query will be executed using the query plan constructed based on selectivity ordering of joins and predicates in the query. Once the query is executed, the selectivity of joins and predicates as well as the join order followed in the query plan is stored.

In Case 2, a query has already been executed before, but its results have not been cached. The join order as well as selectivity of joins and predicates can be determined based on the previous execution of this query. However, the underlying data may have changed since that previous execution. Therefore, we need to determine the significance of these changes by computing the error estimate. If the error estimate exceeds the specified threshold, we update the histograms and recompute the selectivities based on the updated histograms.

In Case 3, a query has been executed before, but only partial results are available from cache. We can immediately use the results that are available from cache, as the cache is incrementally maintained and therefore up-to-date. For the remaining part of the query for which the results are not cached, a query plan is formed that determines the order of execution of the remaining predicates and joins in the query. As the same query has already been executed, we can use the earlier computed join order and selectivities to determine the query plan for the remaining predicates and joins. However, we need to again check if significant changes to the date had occurred (see Case 2).

In Case 4, a query has already been executed and its complete result is available from cache. We can use the results from cache, as discussed above.

E. Learning of Information

We collect the following statistics regarding the execution of a query. For each query, the query frequency and the join order obeyed in the most recent execution of the query are stored. After execution of a query, information regarding the joins contained in that query is also determined. We collect the joins that are contained in that query, the time required to execute each join, the frequency of each join, the selectivity of each join, and the time taken for updating a cached join. These statistics are made available to simplify the construction of the query plan for the next occurrence of a query.

F. Join Ordering

The ordering of joins is the key step in building the query plan. As joins are the most expensive operations performed when executing a query, the joins need to be arranged so that the joins with higher selectivity are executed first which leads to fewer input tuples being passed to the next join in the sequence.

Exhaustive enumeration of all the possible join orders may produce an optimal plan, but the number of possible orders increases as the number of joins in a query increases, rendering such strategy infeasible. Therefore, the maximum selectivity heuristic [16] is used to order the joins: the joins are executing in decreasing order of selectivity. The joins are prepared in order of selectivity to simplify the construction of query plans.

V. PERFORMANCE EVALUATION

We have evaluated the performance of the proposed approach using query optimization techniques for a single run of a program through several experiments. The object size considered for all experiments was fixed at 200 except for the experiment with varying objects reported in Figure 5. All experiments were performed on an Intel Pentium IV running at 3.2 GHz with 1.75 GB RAM. The algorithms were implemented in Java within JQL framework and executed under Eclipse v3.4.0.

We consider four different types of queries with varying numbers of joins. These queries are referred to as "q1", "q2", "q3", and "q4". Details of the four benchmark queries are given in Table 1. Figure 2 shows the execution time for these benchmark queries comparing our approach with JQL. Our approach takes less time than JQL primarily due to the use of histograms to estimate selectivity and to construct the query plan, whereas JQL estimates selectivity by sampling and creates the query plan using an exhaustive join order strategy.

TABLE I. DETAILS OF BENCHMARK QUERIES

Query	Detail
q1	selectAll(Attends a:attendances | a.course == COMP101);
q2	selectAll(Attends a:attendances, Student s:students| a.course == COMP101 && a.student == s);
q3	selectAll(Attends a:attendances, Student s:students, Student t:students| a.course == COMP101 && a.student == s && t.id < s.id);
q4	selectAll(Student s, Faculty f, Attends a, TopStudent t|s.id==t.id and s.department_name==f.department_name and s.course==a.course and s.name==t.name)

Figure 3 compares the execution times of our approach for a single run of the program with the approach presented in [13] for multiple runs, again showing the four benchmark queries. The multiple-run query optimization is faster, because query optimization is shifted from run time to compile time and because selectivity estimation is performed at compile time. Figure 4 shows the difference in compilation time and execution time of our approach considering a mix of queries q1 to q4 and the approach relying on multiple runs of the program [13]. As expected, the time to compile a program is substantially less on the current approach as selectivity is estimated by analyzing information obtained from previous runs.

Figure 5 shows the comparison between the proposed approach and JQL with respect to varying number of objects and run time execution of a program with q2 types of queries (having two joins). As the number of objects increase, the execution time of a program in JQL increases more rapidly than our approach. This improvement in our approach mainly comes from the advantage of building histograms during run time to compute the selectivity of joins and predicates over JQL's sampling of object values to compute selectivity values.

Figure 6 compares the execution of the benchmark queries between the proposed approach and JQL, each executed 200 times. The additional performance improvement of our approach over JQL with more complex queries results from the impact of caching joins which enables some queries to be evaluated partially or completely based on cached results. In JQL, the complete results of repeated queries are cached thus requiring redundant storage for queries with overlapping results, which reduces the effective size of the cache. With query caching, there will consequentially be more cache misses.

Then, we have performed experiments on three cases of join query types to analyze whether our approach is sensitive to the presence of a particular type of a query. Case 1 contains 65% q1, 15% q2, 10% q3, 10% q4. Whereas case 2 contains 40%q1, 30%q2, 20%q3, 10%q4 and case 3 contains 10% q1, 20% q2, 30% q3 and 40% q4.

As shown in Figure 7, our approach works better when the ratio of q1-type queries is lower. Query q1 is relatively simple; the effectiveness of our approach increases compared to JQL as the complexity of the queries (as reflected in the number of joins) increases.

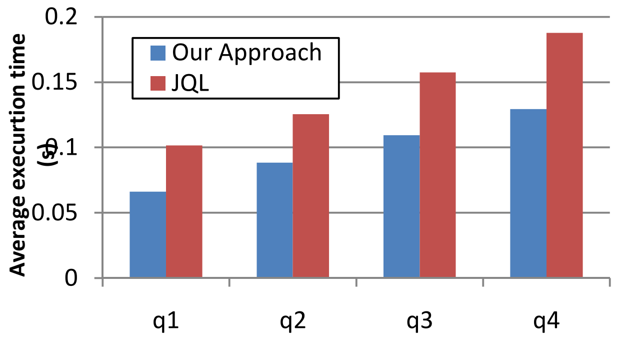

Figure 2. Execution Time for Different Types of Queries: Our Approach Vs. JQL

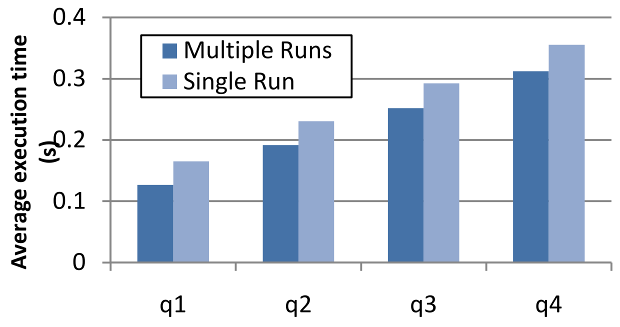

Figure 3. Execution Time for Different Types of Queries: Our Approach (Single Run) Vs. Multiple Run Optimizations

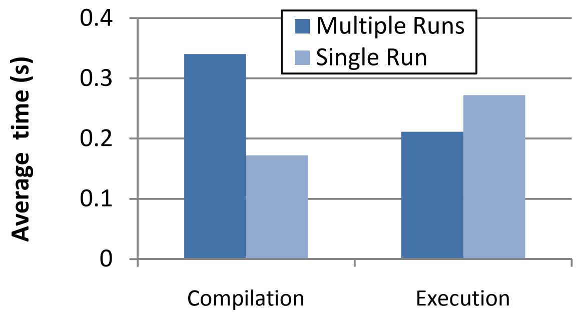

Figure 4. Compilation and Execution Time: Our Approach (Single Run) Vs. Multiple Run Optimizations

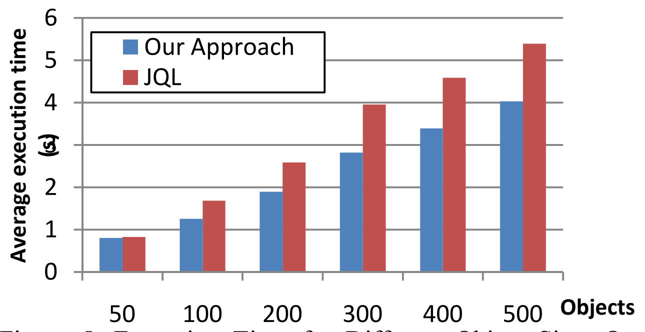

Figure 5. Execution Time for Different Object Size: Our Approach Vs. JQL

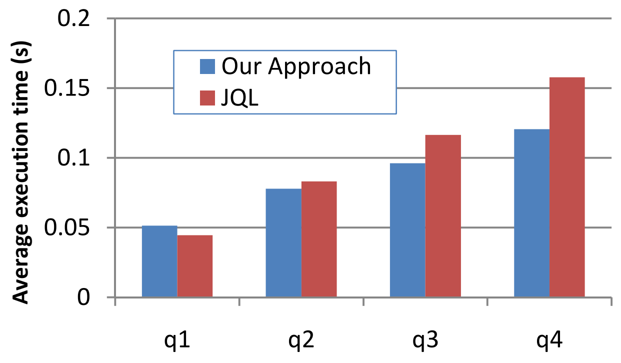

Figure 6. Execution Time for Different Types of Queries: Our Approach Vs. JQL

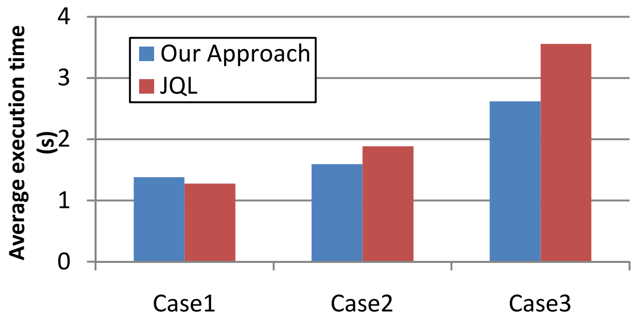

Figure 7. Execution Time for Different Cases: Our Approach Vs. JQL

VI. Conclusion and Future Work

In this paper, we propose an algorithm for improving the execution time for single runs of programs written using queries as first-class constructs. We perform query optimization at run time by first constructing histograms from data and then estimating the selectivity of joins and predicates from the histograms. Finally, a query plan is constructed by ordering the joins and predicates using the maximum selectivity heuristics. If a query is executed repeatedly during a run of the program, information regarding the join order and selectivity of the joins can be obtained from preceding executions. In addition, joins may be cached, providing further performance improvement. Experimental evaluation shows that our approach performs better than JQL for complex queries during single runs of programs. We plan to further improve on this work by moving part of the construction of the query plan to compile time, thus further reducing the overhead incurred by query optimization at run time, while preserving the advantages of run-time query plan selection and construction.

References

[1] 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.

[2] Object Management Group, Object Constraint Language, 2.0, formal/2006-05-01, 2006.

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

[4] G. Bowden Wise, “An overview of the standard template library,” ACM SIGPLAN Notices, Vol. 31, Issue 4, 1996.

[5] 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.

[6] 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.

[7] 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.

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

[9] Yannis E. Ioannidis, Raymond Ng, Kyuseok Shim, Timos K. Sellis, “Parametric Query Optimization”, In Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), pp. 103-114, 1992.

[10] 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.

[11] 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.

[12] E.Meijer, B.Beckman, G.M. Bierman,“LINQ: reconciling object,relations and XML in the .NET framework,” SIGMOD, 2006.

[13] Venkata Krishna Suhas Nerella, Swetha Surapaneni, Sanjay Kumar Madria, Thomas Weigert, “Exploring Query Optimization in Programming Codes by Reducing Run-Time Execution,” IEEE 34th Annual Computer Software and Applications Conference, pp.407-412, 2010.

[14] J.Schwartz, R. Dewar, E. Dubinsky, E. Schonberg,”Programming with Sets:An Introduction to SETL,” Springer-Verlag, 1986.

[15] 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.

[16] 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.