# Advance Run-Time Implementation By Using Query Optimisation

Prof. Swati C. Tawalare[1], Prof. Dhiraj S.kalyankar[2]

*Abstract:* Object querying is an abstraction of operations over collections, whereas manual implementations are performed at a low level, forcing 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 optimisation techniques from the database domain. In this regard, we have developed a method that performs query optimisation at compile-time to reduce the burden of optimisation at run-time to improve the performance of the code Implementation

*Keywords:* Querying; Histogram; query optimisation; joins.

## I. INTRODUCTION

Query processing is the sequence of actions that input a query formulated in the user language and delivers. As a result, the data asked for. Query processing involves query transformation and query Implementation. Query transformation is the mapping of queries and query results back and forth through the different levels of the DBMS. Query Implementation is the actual data retrieval according to some access plan, i.e., a sequence of physical access language operations. An essential task in query processing is query optimisation. Usually, user languages are high-level, declarative languages that state what data should be retrieved, not how to retrieve them. For each user query, many different Implementation plans exist, each having its associated costs. Ideally, the task of query optimisation is to find the best Implementation plan, i.e., the Implementation plan that costs the least, according to some performance measure. Usually, one has to accept feasible Implementation plans because the number of semantically equivalent programs is too large to allow for enumerative search. A query is an expression that describes the information that one wants to search for in a database. Query optimizers select the most efficient access plan for a question based on competing plans' estimated costs. 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 executed.

Such feedback helps to detect errors in queries or misconceptions about the database. Query result sizes are usually estimated using various statistics that are maintained for relations in the database. These statistics merely approximate the distribution of data values in attributes of the connections. Consequently, they represent an inaccurate picture of the actual contents of the database. The resulting size-estimation errors may undermine the validity of the optimizer's decisions or render the user interface application unreliable. Earlier work has shown that mistakes in query result size estimates may increase exponentially with the number of joins. In conjunction with the increasing complexity of queries, this result demonstrates the critical importance of accurate estimation. Several techniques have been proposed in the literature to estimate query result sizes, including histograms, sampling, and parametric approaches. Of these,[1]histograms approximate the frequency distribution of an attribute by grouping attribute values into "buckets" (subsets) and comparing actual attribute values and their frequencies in the data based on summary statistics maintained in each bucket. Implementing operations over these collections with conventional techniques severely lacks abstraction. Step-by-step instructions must be provided on how to iterate over the array, select elements, and operate on the details.

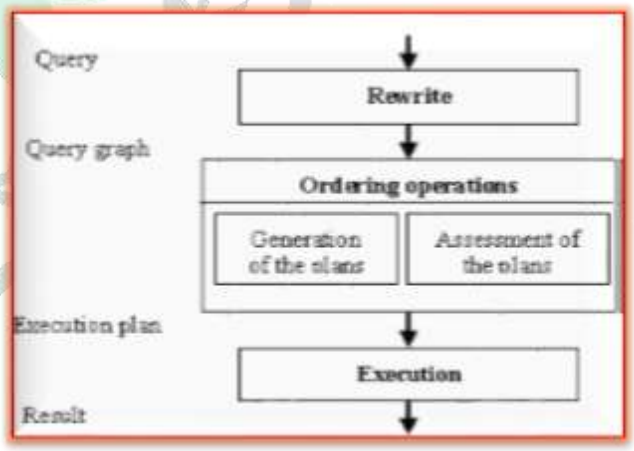



Figure.1 Optimisation process

JQL is an addition to Java that provides the capability for querying collections of objects.[6] These queries can be applied on objects in groups in the program or used to check expressions on all instances of specific types at run-time. Questions allow the query engine to take up the task of implementation details by providing abstractions to handle sets of objects, making the code smaller and allowing the query evaluator to choose the optimisation approaches dynamically even though the situation changes run-time. The Java code and the JQL query will give the same set of results, but the JQL code is elegant, brief, and abstracts away the accurate method of finding the matches. Java Query Language (JQL) by generating the dynamic join ordering strategies. Queries can be evaluated

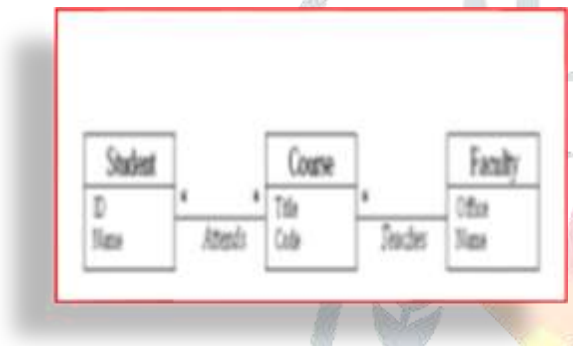
over one collection of objects or many groups, allowing the inspection of relationships between objects in these collections. The Java Query Language (JQL) provides first-class object querying for Java.

**II. Query Implementation**

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

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

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




In database terms, this code joins the name field for the allFaculty and allStudent collections[4]. The code is cumbersome and can be Replaced with the following object query, which is more succinct and, potentially, more efficient:

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

This gives the same set of results as the loop code. The selectAll primitive returns a list of tuples containing all possible instantiations of the domain variables (i.e., those declared before the colon) where the query expression holds (i.e., after the colon). The domain variables determine the set of objects which the query ranges over: they can be initialized from a collection (as above); or left uninitialized to range over the entire extent set (i.e., the set of all instantiated objects) of their type. Queries can define as many domain variables as necessary and can use the usual array of expression constructs found in Java. One difference from regular Java expressions is that Boolean operators, such as && and ||, do not imply any order of Implementation for their operands. It allows flexibility in the order they are evaluated, potentially leading to greater efficiency. As well as its simplicity, there are other advantages to using this query in place of the loop code. The query evaluator can apply well-known optimisations which the programmer might have missed. By leaving the decision of which optimisation to use until run-time, it can make a more informed decision based upon the data's dynamic properties (such as the relative size of input sets), something that is, at best, challenging for a programmer to do.

A good example, which applies in this case, is the so-called hash-join. The idea is to avoid enumerating all of all Faculty × all Students when there are few matches. A hash-map is constructing from the largest of the two collections, which maps the value being joined upon (in this case name) back to its objects. This still requires O(SF) time in the worst-case, wheres = |all Students| and f = |all Faculty|, but in practice is likely to be linear in the number of matches (contrasting with a nested loop which always takes O(SF) time). We have prototyped a Java Query Language system (JQL), which permits queries over object extents and collections in Java. The implementation consists of three main components: a compiler, a query evaluator, and a run-time system for tracking all active objects in the program. The latter enables the query evaluator to range over the extent sets of all classes. Our purpose in doing this is twofold: firstly, to assess the performance impact of such a system and provide a platform for experimenting with the idea of using queries as a first-class language construct.

**III. Evaluation Pipeline**

The JQL evaluator evaluates a query by pushing tuples through a staged pipeline[4]. Each stage, known as a join in the databases' language, corresponds to a condition in the query. Only tuples matching a join's condition are allowed to pass through to the next. Those tuples which make it through to the end are added to the result set. Each join accepts two lists of tuples, L(left) and R(right), and combines them, producing a single list. We enforce the restriction that, for each intermediate join, either input comes from the previous stage or one comes directly from an input collection, and the other comes from the last step. It is known as a linear processing tree, and it simplifies the query evaluator, although it can lead to inefficiency in some cases.

**IV. JQL Query Evaluator**

The core component of the JQL system is the query evaluator. This is responsible for applying whatever optimisations it can to evaluate queries efficiently[4]. The evaluator is a call at run-time with a tree representation of the question (called the query

tree). The tree itself is either constructed by the JQL Compiler (for static queries) or by the user (for dynamic questions).

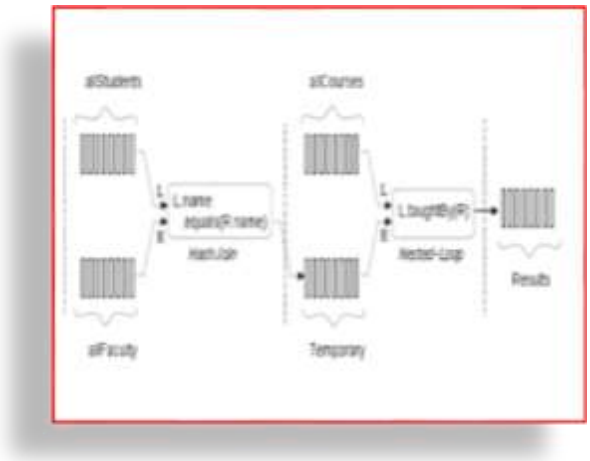


Figure 2. Illustrating a Query Pipeline

Given query Q, we use the histogram H to estimate the query predicates' selectivity and the joins' selectivities, which is used to construct a query plan.

The first Implementation of Query Q uses the histogram H1 to estimate the selectivity. Then the result of the query is computed. But for the subsequent Implementation 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 is updated between the first and the second Implementations of the same query. Firstly, we check if the question is present in the log. The period difference between consecutive Implementations of the same query Q from the query log is compute. If that value is greater than a pre-specified time interval, we directly recomputed the histogram because we assumed the data is modified within a pre-specified time interval. We first compute the error through the error estimate function, and then, based on the error estimate, we decide whether to recomputed the histogram or not. If the query is not present in the log, then we execute the query based upon the initial histogram that reduces the overhead cost of incremental maintenance of histogram the experimental results of how our approach various types of questions the comparison of run-times of our approach and the

In this method, they are using selectivity estimate based on sampling some tuples, but that does not lead to efficient ordering of joins and predicates in a query. Therefore, we propose using the forecasts 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 question. After we get the query plan at compile-time, we execute that plan at run-time to reduce the Implementation time. JQL code's run-time due to our approach of optimizing the query and handling data updates using histograms.

### *A. Estimating Selectivity Using Histogram*

A predicate's selectivity in a query is a decisive aspect for query plan generation[6]. 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 all the query predicates' selectivities. 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 concerning its window. This histogram approach would help us estimate 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.

### *B. Building the Histogram*

A histogram is one of the essential quality tools. It is used to graphically summarize and display the distribution[1] and variation of a process data set. A frequency distribution shows how often each different value in a set of data occurs. The primary purpose of a histogram is to clarify the presentation of data. 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 recomputed the histogram even in case of a data change. When building a histogram, we need to assign the values to buckets. The frequency distribution for numerical data is straightforward, but the frequency distribution for alphabetical information is not. Considering the alphabetical data such as first names, last names, Organization names, etc., the question arises as to how we can split these into buckets. We propose here to group the alphabetical data concerning the letter they start with and alphabets of similar frequency of occurrences grouped into a single bucket. To do this grouping, we use statistics from Figure that are computed by analysts showing the probable number of circumstances 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.

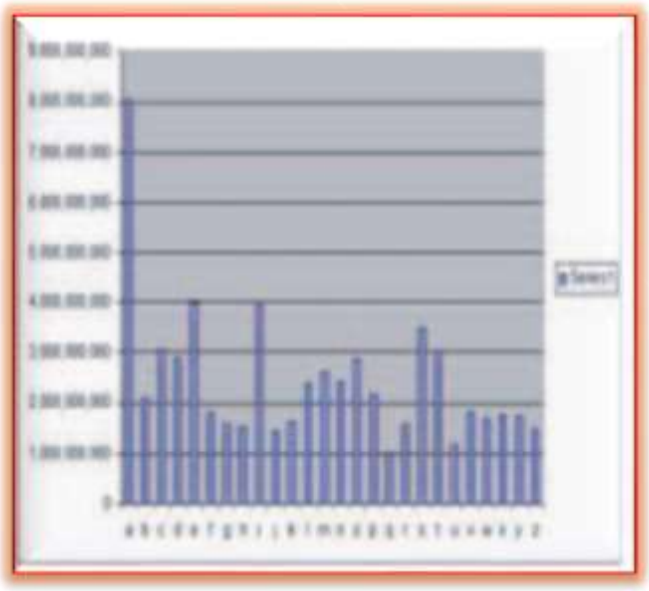


Figure 3. Frequencies of alphabets

### C. *Incremental Maintenance of Histograms*

[1]we propose an *incremental* technique, which maintains approximate histograms within specified error bounds at all times with high probability and never accesses the underlying relations for this purpose. There are two components to our incremental approach: (i) maintaining a backing sample and (ii) a framework for maintaining an approximate histogram that performs a few program instructions in response to each update to the database and detects when the histogram requires an adjustment of one or more of its bucket boundaries. Such adjustments make use of the backing sample. There is a fundamental distinction between the backing sample and the histogram it supports: the histogram is accessed more frequently than the sample and uses less memory, and hence it can be stored in the main memory while the sample is likely stored on disk. A *backing sample* is a uniform random sample of the tuples in a relation that is kept up a to- date in the presence of updates. For each tuple, the sample contains the unique row id and one or more attribute values. We argue that maintaining a backing sample is helpful for histogram computation, selectivity estimation, etc.[6]In most sampling-based estimation techniques, whenever a sample of size _ is needed, either the entire relation is scanned to extract the sample, or several random disk blocks are read. In the latter case, the tuples in a disk block may be highly correlated, and hence to obtain a truly random sample, _ disk blocks must be read. A backing sample can be stored in consecutive disk blocks and can therefore be scanned by reading sequential disk blocks. Moreover, for each tuple in the sample, only the unique row id and the attribute(s) of interest are retained. Thus the entire sample can be stored in only a small number of disk blocks for even faster retrieval. Finally, an indexing structure for the sample can be created, maintained, and stored; the index enables quick access to sample values within any desired range. 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 Implementations, if the database is updated, we compute the histogram's estimation error by using the following equations.

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

$$T_i = \frac{w_1\mu_1 + w_2\mu_2 + \cdots + w_n\mu_n}{w_1 + w_2 + \cdots + w_n}$$

where μa is the estimation error for every attribute

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

If the calculated error (Ti) is > 0.5, then we update the histogram.[6] 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 the 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 a 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 question. When the consecutive Implementations 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 continue with the existing histogram. However, we do not want to recompute a histogram for a table that is not often accessed. Thus, we use the frequency of access to a particular table to decide when and when not to compute the histogram. If the access frequency is getting higher, it increases 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.

### *D. Method Outline for Error Estimation*

We compute the error estimate for each attribute in the database table by using the 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 attribute error estimates. If that weighted average is more significant than a certain threshold, then the table's histogram must be updated.

For every selection on the histogram attribute, we compute the approximation error (Ti). We calculate the error estimate for all the details (μa) in the table for each table. Then for each table, we take a weighted average of all the attribute errors. If that computed error (Ti) is greater than a threshold, we update the histogram; otherwise, we need not update it. If error (Ti) > 0.5, then we scan the database and update buckets. If some buckets exceed a threshold, then we use split and merge algorithms.

### *E. 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) 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. 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 breaking the buckets starts with iterating through a list of buckets, 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 most negligible value and whose count is less than a certain threshold. Then we link those two buckets. If we fail to find any pair of buckets to merge, then we recomputed the histogram from the 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 a join and predicates' selectivity, we get the join and predicate ordering at compile-time.

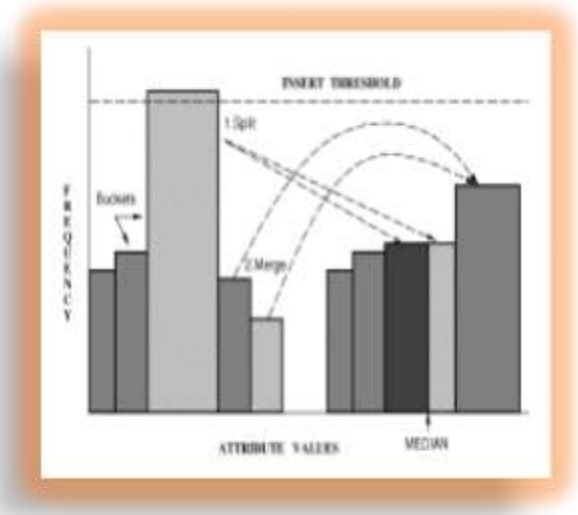



Figure 4. split and merge algorithm

## V. Implication

- Our approach reduces run-time Implementation less than the existing JQL code's run-time due to our policy of optimizing the query and handling data updates using histograms.
- We proposed a technique for query optimisation at compile-time by reducing the burden of optimisation at run-time. We suggested using histograms to estimate the selectivity of joins and predicates in a query and then, based on those estimates, to order query joins and predicates in a question. We have obtained the query plan at compile-time from the join and predicate order, and then we executed the query plan at run-time. Error estimate and split merge algorithms are efficient and maintain the histograms accurately
- The comparison of run-times of our approach and the JQL approach for all the benchmark queries. The difference in run-times has occurred because in our process, we have estimated selectivities using histograms, and these histograms are incrementally maintained at compile time which provides the optimal join order strategy most of the times faster than the exhaustive join order strategy used by JQL

## VI. CONCLUSION

I have shown the query optimisation strategies from the database domain that can improve the run time Implementations in the programming language. We proposed a technique for query optimisation at compile-time by reducing the burden of optimisation at run-time. I suggested 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 question.

## VII. 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]S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 285–299. ACM Press, 1995.

[3] C. Hobart 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.

[4] 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.

[5] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations, and XML in the .NET framework. In *Proceedings of the ACM Symposium on Principles Database Systems*, 2006.

[6] Venkata Krishna, "Exploring Query Optimisation in Programming Codes by Reducing Run-Time Implementation," Department of Computer Science, Missouri University of Science and Technology, Rolla, MO,2010

[8]Ihab F. Ilyas et al. (2003), "Estimating Compilation Time of a Query Optimizer," Proceedings of the 2003 ACM SIGMOD international conference on data management, pp 373 –384, 2003.

[9] S. Goldsmith & R. O'Callahan (2005) A. Aiken. Relational queries over program traces. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 385–402. ACM Press, 2005

[10] YE Ioannidis, R. Ng, K. Shim & TK. Selis(1992) .Parametric Query Optimisation, In Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), pp. 103-114, 1992.

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

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

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

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

[15] Surajit Chaudhuri," An overview of query optimisation in relational systems," Proceedings of the seventeenth ACM SIGACT-SIGMODSIGART symposium on Principles of database systems, pp. 34-43, 1998.

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

[17] Michael Steinbrunn, Guido Moerkotte, Alfons Kemper, "Heuristic and randomized optimisation for the join ordering problem," VLDB Journal, vol. 6, pp. 191-208, 1997.

[18] 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.

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

[20] YE Ioannidis, Younkyung Kang, "Randomized algorithms for optimizing large join queries," ACM SIGMOD Record, vol. 19, pp. 312-321, 1990.

[21]PedroBizarro, NicolasBruno, David J. DeWitt, "Progressive Parametric Query Optimisation," IEEE Transactions on Knowledge and Data Engineering, vol. 21, pp. 582-594, 2009.

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