ISSN:2320-0790

COMPUSOFT
An International Journal of Advanced Computer Technology

A Knowledge Based Approach for Query Optimization in Preferential Mapping Relational Databases

P.Ranjani[1], B.Murugesakumar[2]

[1]Research Scholar, Dr. SNS.Rajalakshmi College of Arts and Science, Coimbatore, India
[2]HOD, Department of Computer Applications, Dr. SNS.Rajalakshmi College of Arts and Science, Coimbatore, India

Abstract: Relational query databases provide a high level declarative interface to access data stored in relational databases. Two key components of the query evaluation component of a SQL database system are the query optimizer and the query execution engine. System R optimization framework since this was a remarkably elegant approach that helped fuel much of the subsequent work in optimization. Transparent and efficient evaluations of preferential queries are allowed by relational database systems. This results in experimenting extensive evaluation on two real world data sets which illustrates the feasibility and advantages of the framework. Early pruning of results based on score or confidence during query processing are enabled by combining the prefer operator with the rank and rank join operators. During preference evaluation, both the conditional and the scoring part of a preference are used. The conditional part acts as a soft constraint that determines which records are scored without disqualifying any duplicates from the query result. To introduce a preferences mapping relational data model that extends database with profile preferences for query optimizing and an extended algebra that captures the essence of processing queries with ranking method. Based on a set of algebraic properties and a cost model that to propose, to provide several query optimization strategies for extended query plans. To describe a query execution algorithm that blends preference evaluation with query execution, while making effective use of the native query engine.

Keywords: query optimization; relational databases; query plan; preferential databases; query evaluation; query parser; dynamic query optimization algorithm.

I. Introduction

Data mining has attracted a great deal of attention in the information industry and in society as a whole in recent years, due to the wide availability of huge amounts of data and the imminent need for turning such data into useful information and knowledge. Data mining can be viewed as a result of the natural evolution of information technology. Data mining involves an integration of techniques from multiple disciplines such as database and data warehouse technology, statistics, machine learning, high performance computing, pattern recognition, neural net works, data visualization, information retrieval, image and signal processing, and spatial or temporal data analysis. For an algorithm to be scalable, its running time should grow approximately linearly in proportion to the size of the data, given the available system resources such as main memory and disk space. By performing data mining, interesting knowledge, regularities, or high level information can be extracted from databases and viewed or browsed from different angles. The discovered knowledge can be applied to decision making, process control, information management, and query processing. Therefore, data mining is considered one of the most important frontiers in database and information systems and one of the most promising interdisciplinary developments in the information technology.

II. Related Work

The goal of an optimizer as follows, "A plan as cost-effective as possible is looked for as soon as possible". Further observe that the job of a query optimizer is not necessarily to get the cheapest plan (though the cheapest plan would of course be the best). In fact, if a stage is reached where the cost of further optimizing is higher than the resource savings, it is worthwhile to terminate the search.

Optimize the use of memory by appropriately dividing it among the pipelinable sub-expressions in the plan. The amount of memory given to a sub expression determines the cost estimate associated with it. Given the finite value of memory therefore, not all the sub-expressions may be finished. However use the memory division to accommodate as many pipelinable schedules as possible.

Query (and hence sub-expression) scheduling to achieve optimal use of memory. A sub-expression may be removed when it still has other processes to serve so long as it is replaced with one whose benefit is higher than the one removed. With pipeline, the algorithm has to make a decision to pipeline and materialize where the buffer space is not enough. More to that, not all the reused sub-expressions are pipelinable. It is therefore limited to the pipelinable schedules and in case the pipelinable schedules cannot all fit in memory, then the pipelining has to be done in a buffer-conserving manner.

Query optimization begins with a structural representation of the SQL query that is used throughout the lifecycle of optimization. This representation is called the Query Graph Model (QGM). In the QGM, a box represents a query block and labeled arcs between boxes represent table references across blocks. This forces this module to either retain alternatives obtained through rule application or to use the rules in a heuristic way (and thus compromise optimality).

Another interesting approach is to handling queries. When SQL statements are embedded within application programs, predicates usually refer to program variables. This means that compile-time SQL may lead to suboptimal results, as the values of the variables will not be known until execution time. The optimizer will usually make assumptions about the selectivity of these predicates in this case. Unfortunately, this means that accurate estimates of execution parameters, such as the sizes of intermediate result relations, are impossible. Another problem is that the access path is chosen using the state of the database at compile time, and not when the application is run.

The optimization of a query is in centralized database management systems. The process of query optimizations goes by two key stages are the rewriting stage and the planning stage. Various query optimizer components are then explored in these stages. The rewriter module in the rewriting stage performs transformations for a given query and produces an efficient query. Planner which is the basic module of planning strategy performs various search strategies mechanism. It explores plans identified by the algebraic space. Method-structure space modules evaluate these plans using the cost derived from the Size-Distribution Estimator and Cost Model module. Author has focused on histogram method where each attributes values are distributed into chunks or buckets. However, there are several issues in the field of query optimizations that necessitates making the query optimizers architecture in a generalized way to handle every type of query either simple or complex.

III. Proposed Work

Searches for the web pages of a person with a given name constitute a notable fraction of queries to web search engines. A query would normally return web pages related to several namesakes, who happened to have the queried name, leaving the burden of disambiguating and collecting pages relevant to a particular word (from among the namesakes) on the user. This can lead to some search results being polluted, with more relevant links being pushed down in the result list. Several searches have been proposed, which allow increasing information retrieval accuracy by exploiting a key content of Semantic resources that is relations. However, in order to rank results, most of the existing solutions need to work on the whole annotated knowledge base. Algorithms and techniques also need a new concept to evolve into quick results and more effective experimental results. Queries optimized needs more accuracy and efficiency for relational databases where enormous data is involved. Hence, existing systems require a new link or approach for the betterment of mining large amount of data in database systems.

A. Preference Aware Mechanism

Poor mathematical model estimates and uncaught correlation are one of the main reasons why query optimizers pick poor query plans. This is one reason why a database administrator should regularly update the database statistics, especially after major data loads/unloads. For query check, the optimizer considers combining each pair of relations for which a join condition exists. For each pair, the optimizer will consider the available join algorithms implemented by the databases. Cost query plan relies heavily on estimates of the number of tuples, flowing through each edge in a query plan.

To proposed a new technique to group preferences based on maximum number of executions made. Queries can be grouped based on specific criteria into preference and can be ranked based on optimization. For processing a query with preferences, to follow a hybrid approach with respect to plug-in and native approaches. First construct an extended query plan that contains all operators that comprise a query and we optimize it. Then, for processing the optimized query plan, the general strategy is to blend query execution with preference evaluation and leverage the native query engine to process parts of the query that do not involve a prefer operator.

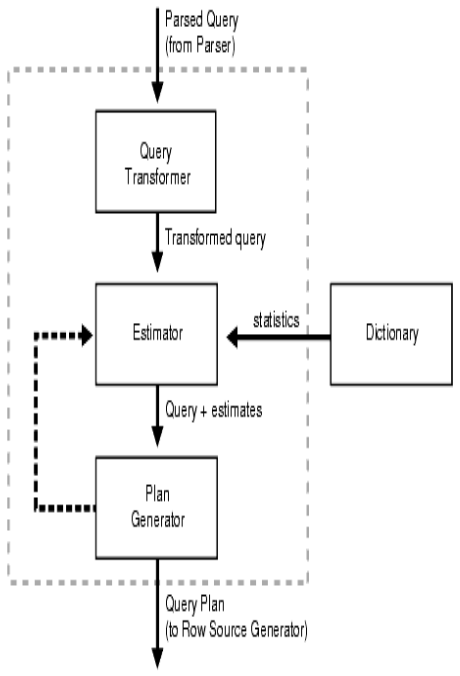

Figure 1. Query Optimization Architecture

The concept of preference aware query processing appears in many applications, where there is a matter of choice among alternatives, including query personalization, recommendations and multi-criteria decision making.

A. Qualitative Approach

In the qualitative approach, preferences are specified using binary predicates called preference relations. In quantitative approaches, preferences are expressed as scores assigned to tuples or query conditions. Existing works have studied various types of preferences including likes and dislikes, multi-granular preferences that involve many attributes and context-dependent preferences. In the latter case, the context can be dictated by the data or it can be external to the database.

B. Quantitative Approach

A quantitative approach covers prior works with respect to different preference types. In this model, preference scores are assigned to tuples in a context-dependent way. Context is related to the data as in but it is defined in a quantitative way. In addition, each preference score carries a confidence value that captures how certain a preference is. Using scores, context, and confidences allows not only expressing several types of preferences; it also enables the formulation of different types of queries with preferences where the expected answer may be specified based on any combination of scores, confidences and context.

The framework allows processing in a uniform way all these different query and preference types. In terms of preference integration and processing, one approach is to translate preferences into conventional queries and execute them over the DBMS.

C. Preferential Database Approach

PrefDB (Preferential Database) takes user profile along with preferences and stores in the database. In earlier approach, to run queries and optimize them, operators are injected into the query engine to execute the queries along with the preferences. Also many plans need to address before optimizing or executing any query. It consumes more time and also for scoring part it checks whether it involves an attribute for each preference. This check makes lots of matching issues with existing preferences and thus leads to loss of preferential queries. When this happens to large amount of data, data loss cannot be accepted. Also it is not cost effective since it retrieves every plan from a native query optimizer and also to make necessary changes every time it needs valid time to join the order for query executions. Also it does not satisfy user interests or search constraints of large data.

Several efficient algorithms have been proposed for processing different types of queries, including top-k queries and skylines. These algorithms as well as query translation methods are typically implemented outside the DBMS. Thus, they can only apply coarse-grained query optimizations, such as reducing the number of queries sent to the DBMS. Further, with the existing approach, plug-in methods do not scale well when faced with multi-join queries or queries involving many preferences.

This system provides a personalization framework that facilitates the enrichment of queries with preference semantics such that query results match the specified preferences. Implemented framework and methods in a prototype system allows transparent and efficient evaluation of preferential queries on top of relational databases.

The extensive experimental evaluation on two real world datasets demonstrates the feasibility and advantages of the framework. The proposed approach aligns with an algorithm and uses the sorting technique with the help of following tasks in query optimization:

Manage user preferences, group them into profiles and select which ones and how will be used in queries.

Build and execute queries, select among a set of available execution strategies and configure various query parameters such as the expected number and type of results, score and confidence thresholds, and so forth.

Inspect the query execution through a built-in console, explore the preference-aware query plan followed, and browse statistics and profiling information available for their queries.

In this way, preference mapping separates evaluation from query filtering. This separation is a distinguishing feature of the work with respect to previous works. It allows to define the algebraic properties of the prefer operator and build generic query optimization and processing strategies

that are applicable regardless of the type of preference specified in a query or the expected type of answer. Further, to extend all query optimization to preferences of heterogeneous relational databases.

D. Query Plan

Every node represents either an attribute in the conjunctive query (i.e., a service invocation), or a join, or a selection operation. Every arc indicates data flow and parameter passing from outputs of one service to inputs of another service. Atoms are partitioned into exact and search services. Exact services are distinguished between proliferative and selective and may be chunked, while search services are always proliferative and chunked.

An exact service is selective if it produces in average less than one tuple per invocation (and therefore, in average, fewer output tuples than input tuples). Selection nodes express selection or join predicates which cannot be performed either by calling services or by using connection patterns. Each predicate is independently evaluated on tuples representing intermediate or final query results, immediately after the service call that makes the selection or join predicates valuable.

Two explicit nodes represent the query input (i.e., the process of reading input variables, mapping onto the arguments of services and joins, and starting query execution) and output (i.e., returning tuples to the query interface).

The graphical syntax of representing query plans is represented in figure.

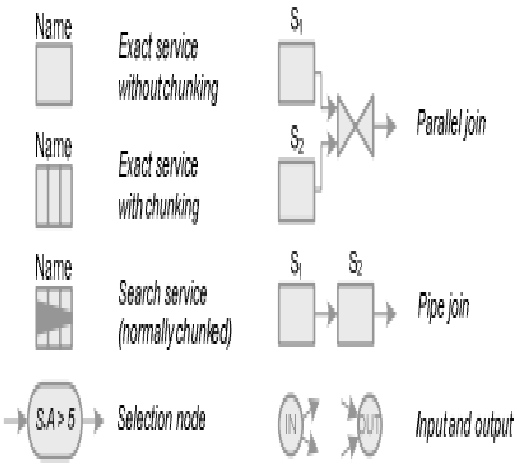

Figure 2. Elements of Query Plan

To assume that services are independent of each other and that at each service call the values are uniformly distributed over the domains associated to their input and output fields. These assumptions allow obtaining estimates for predicate selectivity and sizes of results returned by each service call. Cost models use estimates of the average result size of exact services and of chunk sizes.

E. Query Evaluation

In the proposed system, multi-block queries are considered for optimization which is converted to single block query and then moved for further processing. If tuple iteration semantics are used to answer the query, then the inner query is evaluated for each tuple of the Dept relation once.

An obvious optimization applies when the inner query block contains no variables from the outer query block (uncorrelated). In such cases, the inner query block needs to be evaluated only once. However, when there is indeed a variable from the outer block, to say that the query blocks are correlated.

F. Query Model

The path that a query traverses through a database until its answer is generated. The system modules through which it moves have the following functionality:

The Query Parser checks the validity of the query and then translates it into an internal form, usually a relational calculus expression or something equivalent.

The Query Optimizer examines all algebraic expressions that are equivalent to the given query and chooses the one that is estimated to be the cheapest.

The Code Generator or the Interpreter transforms the access plan generated by the optimizer into calls to the query processor.

The Query Processor actually executes the query.

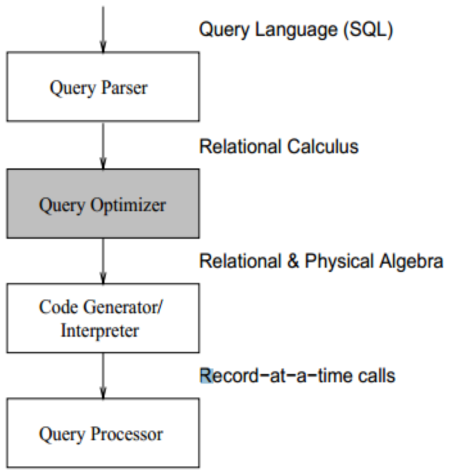

Figure 3. Query Flow through Databases

There are three main typical restrictions dealt in this model.

Restriction R1: Selections and projections are processed on the fly and almost never generate inter- mediate relations. Selections are processed as relations are accessed for the first time. Projections are processed as the results of other operators are generated. Restriction R1 eliminates only suboptimal query trees, since separate processing of selections and projections incurs additional costs.

Restriction R2: Cross products are never formed, unless the query itself asks for them. Relations are combined always through joins in the query.

Restriction R3: The inner operand of each join is a database relation, never an intermediate result.

G. Dynamic Query Optimization Algorithm

The algorithm is essentially a dynamically pruning, exhaustive search algorithm. It constructs all alternative join trees (that satisfy restrictions R1-R3) by iterating on the number of relations joined so far, always pruning trees that are known to be suboptimal.

Before present the algorithm in detail, need to discuss the issue of interesting order. One of the join methods that are usually specified by the query model is sorting. Merge sort first sorts the two input relations on the corresponding join attributes and then merges them with a synchronized join attribute.
If any of the input relations, however, is already sorted on its join attribute (e.g., because of earlier use of a B+ tree index or sorting as part of an earlier merge-sort join), the sorting step can be skipped for the relation. Hence, given two partial plans during query optimization, one cannot compare them based on their cost only and prune the more expensive one; one has to also take into account the sorted order in which their result comes out.

Sort order can avoid a redundant sort operation later on in processing the query. Also a particular sort order can speed up a subsequent query with its profile and manage user preferences (interests) because it clusters the data in a particular way.

Here to assign each query with a node descriptor and then match those for further ranking process. Starts by sorting small sub files (runs) of the main file and then merges the sorted runs, creating larger sorted sub files that are merged in turn.

Sorting phase: $n_R = \lceil (b/n_B) \rceil$ (1)

Merging phase: $d_M = \text{Min}(n_B\text{-}1, n_R)$; $n_P = \lceil (\log_{dM}(n_R)) \rceil$ (2)

n_R: number of initial runs; b: number of file blocks;

n_B: available buffer space; d_M: degree of merging;

n_P: number of passes.

```
set   i ← 1;
      j ← b;                    {size of the file in blocks}
      k ← n_B;                  {size of buffer in blocks}
      m ← ⌈(j/k)⌉;

{Sort Phase}
while (i ≤ m)
do {
      read next k blocks of the file into the buffer or if there are less than k blocks
         remaining, then read in the remaining blocks;
      sort the records in the buffer and write as a temporary subfile;
      i ← i + 1;
}

{Merge Phase: merge subfiles until only 1 remains}
set   i ← 1;
      p ← ⌈log_{k−1} m⌉;   {p is the number of passes for the merging phase}
      j ← m;
while (i ≤ p)
do {
      n ← 1;
      q ← ⌈(j/(k−1))⌉;    {number of subfiles to write in this pass}
      while (n ≤ q)
      do {
            read next k−1 subfiles or remaining subfiles (from previous pass)
               one block at a time;
            merge and write as new subfile one block at a time;
            n ← n + 1;
      }
      j ← q;
      i ← i + 1;
}
```

If the two records R and S are physically sorted (ordered) by the value of join attributes are A and B respectively. Both files are sorted in order of the join attributes, matching the records that have the same values for A and B. In this method, the records of each file are scanned only once each for matching with the other file unless both A and B are non-key attributes, in which case the method needs to be modified slightly. Ranking functions may be assigned prior to query execution, either at query definition time or at query presentation time. They can also be altered dynamically through the query interface, yielding to changes in the query execution strategy. Only ranking functions defined at query definition time can be used for query optimization.

Sorting could be done by redistributing all tuples in the relation using range partitioning.

Sort the collection of employee tuples by salary whose values are in a certain range.

For N processors each processor gets the tuples which lie in range assigned to it. Like processor 1 contains all tuples in range 10 to 20 and so on.

Each processor has a sorted version of the tuples which can then be combined by traversing and collecting the tuples in the order on the processors (according to the range assigned)

IV. Experimental Analysis

To evaluate the relative performance of SQL queries vs. optimized plans with sorting technique have conducted an experimental study. The system provides experimental results to explore the validity of our rate-based cost model. We focus our attention on two questions:

1. Does the cost model correctly estimate individual plan performance?

2. Is the framework capable of providing correct decisions regarding the best choice among a set of plans?

As is the case with traditional cardinality based optimization, it would be unrealistic to expect the optimizer to be accurate to the granularity of seconds. To expect it, however, that is to be correct in terms of identifying points of interest in an execution plan. For instance, if two plans "cross" in terms of which is best at some point, the optimizer should predict such a crossing point and roughly identify where it occurs.

The experiments involved queries built out of four equi join predicates, with selectivity ranging from 10-5 to 5·10-3 presents the specifics of sources, while presents the four join predicates and their respective selectivity's. The join predicate A⋈C as an expensive one, assigning to it an additional transmission delay, while for the rest of the predicates and their costs are equal to the cost of the evaluation algorithm. Because of its natural fit with streaming environments, so have to use symmetric hash join as the evaluation algorithm for all join predicates.

A. Query Generator

In order to perform experiments involving IDP1ccp and the ML algorithms it is necessary for generate a set of queries in join graph form. However, cannot produce queries without having relations to refer to. Therefore have to generate a system catalogue before generating queries. Consider the experimental case where it has a distributed setting consisting of the execution sites and where the maximum number of relations involved in a query is n. The minimum number of relation entries required to be present in the catalogue must therefore be n. The next step is to populate the relation entry with fields. Each relation entry is assigned between 5 and 10 fields according to a uniform probability distribution. As with to assign each field a domain according to the probabilities in. The size of each domain and the size in bytes can also be seen. The final addition to each relation entry is the resident sites of the relation. First choose randomly how many sites the relation should be available at using 1+U where U is a discrete random variable taking values from 0 to 1. Then proceed by picking the required number of sites at random and allocating to the given relation entry.

Introduce the concept of computation reusing with several intuitive examples on structural pattern matching. Given a simple query, A→B, C→B, one of the possible plans is to first execute A→B and C→B separately and then perform a distributed hash join on B. But one should note that matching A→B and C→B is the same exact match (despite the different variable names). They are both simply selecting all edges in the graph where the originating vertex is different from the destination vertex, and can be represented as V1 → V2. Consequently, the two input relations of the hash join on each machine are identical. Therefore, instead of generating the same intermediate result set twice, to generate it just once, and do a self-join. The resulting query plan is shown as Q.

Another example query could be: B→A, C→B, D→B. Similar to above, to get the set of all edges V1→V2. In this case use 3 copies of the intermediate result. One copy stays put (since it is already partitioned by V1), and the other two copies are repartitioned by V2. At this point, the three sets of intermediate results: V1→V2, V2→V1, and V2→V1 can all be joined (locally on each node) on the partitioning vertex of each intermediate result set. Note that as an optimization, do a self-join of V2→V1 instead of making a third copy.

As a first step towards validating optimization framework, have to evaluate the performance of a three-way join query containing the predicates A⋈B and A⋈C. Then to explored two execution plans: (A⋈B) ⋈C and (A⋈C) ⋈B. Assigned an inter-arrival delay to each stream, with stream B being the fastest, having an inter-arrival delay of 2 milliseconds, while streams A and C were considerably slower with inter-arrival delays of 20 and 10 milliseconds. Then fed each plan's parameters into an estimator developed using the rate-based optimization framework as the plan evaluation criterion. The issue was to estimate the performance of each plan as a plot of output size vs. time.

I. Parameter in Estimator for Plan

Source	Number of Tuples	Size
A	5,000	0.7 MB
B	10,000	1.5 MB
C	20,000	1.8 MB
D	50,000	5.9 MB
E	100,000	9.3 MB

II. Parameters for Join Predicate in Queries

Predicate	Selectivity	Handling cost
A,B	2.10^{-3}	-
A,C	5.10^{-3}	5 ms
B,D	10^{-4}	-
D,E	10^{-5}	-

After predicting the performance of the plans, ran them through the execution engine, keeping track of the time at which each result tuple appeared. Although not exactly matching the predictions (the actual performance curve was more ragged than the estimated curve) the general behaviour of each plan was similar to the prediction.

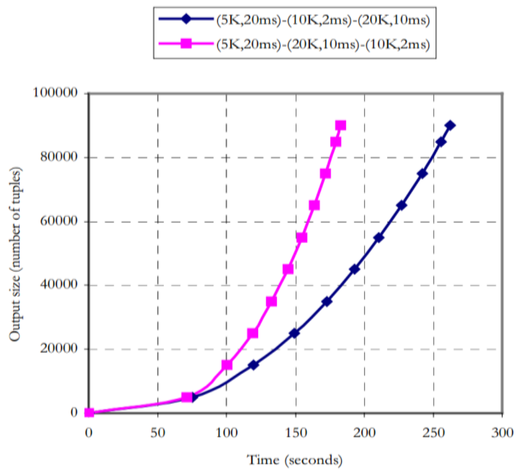

Figure 4. Estimated Plan Performance until the Last Result Tuple

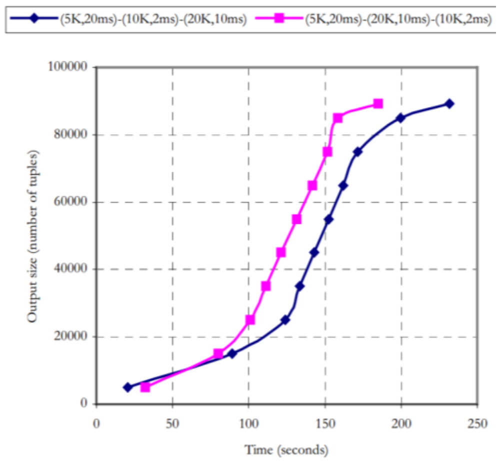

Figure 5. Measure Plan Performance until the Last Result Tuple

The optimizer's predictions, as far as ordering the plans and crossing points are concerned, were again correct. The plan performance estimations were that Left Deep would be the slowest plan, while at the same time it would have comparable performance to Fast Leaves. Evenly Spread on the other hand, would clearly outperform the other two, starting to do so from the initial stages of execution have marginally better performance than Fast Leaves for the first 20,000 tuples, which is the actual case as depicted.

III. COMPARISON BETWEEN THE TRADITIONAL AND THE RATE BASED COST MODEL

Plan	Traditional Estimation	Rate-Based Estimation
Left Deep	10^4	1.3
Fast Leaves	2.10^3	9.7
Evenly Spread	5.10^3	8.8

To compare, and asked the rate-based estimator to cost the plans in terms of final result output performance, i.e., time needed until complete results are produced. Above table summarizes the results. From that table it is clear that the rate-based estimator could distinguish between the plans, predicting which would be the first to reach the final result size.

The traditional estimator, on the other hand, although it successfully managed to identify Left Deep as the most expensive plan, it failed to distinguish between the two bushy plans, costing Fast Leaves as the cheapest one. In this case, the reason why the cost-based optimize orders the plans incorrectly is that it assumes all of its input is present when execution commences.

This is however, is not the case in proposed scheme. The size of the input is time-dependent, which is essentially what the rate based optimization framework captures by optimizing for output rate.

V. CONCLUSION

In this system, propose rate based optimization as a way to enable query optimizers to work with infinite input streams. For more traditional applications, rate-based optimization may be useful because it allows optimization for specific points in time during query evaluation. To evaluate the framework, compared the predications an optimizer would make using the framework with measured execution times in a prototype version of the SQL Query Engine. The results of this experiment indicate that rate based optimization is indeed a potentially viable approach, worthy of further exploration.

The need for a query optimization technique for providing the best response time and the best throughput in query processing is very important. Considering that nested queries are the most complex and expensive operators, this system have focused on providing an effective optimization technique by using Hints. The chosen technique is precisely a manual tuning that requires users to insert additional comments into an SQL statement.

A great deal of room for future work exists in fact, to think that this initial work raises as many questions as it answers. In one direction, our cost models are quite simple, with rough heuristics to approximate integrals and naïve assumptions about the costs of various operators as a function of their inputs. Clearly these can be refined. In another direction, and the one perhaps find most interesting, there are potentially powerful synergies between the rate-based approach and previous work on adaptive or dynamic query processing and re optimization. Plan to explore both directions in future work.

The future work will include evaluating Hints as an effective optimization technique for nested queries using aggregate function and object relational database management system (ORDBMS) queries, since join queries usually include nested queries with a lot of aggregate functions for processing real world query data that cost complex processing time which may impact on the system's performance. Moreover, join queries in ORDBMS also require a good optimization technique, as the new database design that uses the REF data structure operates differently from the conventional database.

REFERENCES

[1] Hong, W. Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays. Ph.D. Thesis, University of California, Berkeley, 1992.

[2] Bizarro, P., Bruno, N., De Witt, D.J.: Progressive Parametric Query Optimization. IEEE Transactions on Knowledge and Data Engineering 21(4), 582 – 594 (2009).

[3] Kießling W., Hafenrichter B. (2002): Optimizing Preference Queries for Personalized Web Services. In Proceedings of the IASTED International Conference, Communications, Internet and Information Technology (CIIT 2002), St. Thomas, Virgin Islands, USA, 461 - 466.

[4] Kießling W., Hafenrichter B. (2003): Algebraic Optimization of Relational Preference Queries, Technical Report 2003-1, University of Augsburg, Germany.

[5] Kießling W., Köstler G. (2002): Preference SQL Design, Implementation, Experiences. Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, 990 - 1001.

[6] Köstler G., Kießling W., Thöne H.,Güntzer U.(1995): Fixpoint Iteration with Subsumption in Deductive Databases. Journal of Intelligent Information Systems, 4(2): 123 - 148.

[7] Kossmann D., Ramsak F., Rost S. (2002): Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, 275 - 286.

[8] Lacroix M., Lavency P. (1987): Preferences: Putting More Knowledge into Queries. Proceedings of 13th International Conference on Very Large Data Bases, Brighton, UK, 217 - 225.

[9] Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., Price T.G. (1979): Access Path Selection in a Relational Database Management System. Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, USA, 23 - 34.

[10] Tan K.-L., Eng P.-K., Ooi B. C. (2001): Efficient Progressive Skyline Computation. Proceedings of 27the International Conference on Very Large Data Bases, Rome, Italy, 301 - 310.

[11] Minyar Sass [MA05]i, and Amel Grissa-Touzi "Contribution to the Query Optimization in the Object-Oriented Databases" World Academy of Science, Engineering and Technology 11 2005.

[12] Nikose M.C. Dhande.S.S, Dr. G. R. Bamnote Query "Optimization in Object Oriented Databases through Detecting Independent Sub queries".

[13] Navta Kumari "Query Optimization Techniques-Tips for writing Efficient Queries", International Journal of Scientific and Research Publications, Volume 2, Issue 6, June 2012.

[14] Spaccapietra.S and Parent.C, "A step forward in solving structural conflicts," IEEE Transactions on Knowledge 5and Data Engineering, vol. 6, no. 2, 1998.

[15] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A.,Price T.G. Access Path Selection in a Relational Database System. In Readings in Database Systems. Morgan Kaufman.