Query Optimization in the Presence of Top-k Predicates

Lakshmi Priya Mahalingam and K. Selçuk Candan
Computer Science and Engineering Department,
Arizona State University
Email: {priyam,candan}@asu.edu

Abstract

In this paper, we present novel techniques for performing query optimization in databases, such as multimedia and web databases, which rely on top-k predicates. We propose an optimization model that (1) takes into account different binding patterns associated with query predicates and (2) considers the variations in the query result size (or coverage), depending on the execution order. We address the additional complexity and the well-known NP-complete nature of the query optimization problem by *adaptively* reducing the granularity of the search space. For this purpose, unlike the data histograms which capture the data distribution, we propose opt-histograms that capture the distribution of sub-query-plan values over many query optimization tasks.

1 Introduction

Multimedia database queries have different characteristics from queries in traditional databases. For instance, in traditional, such as relational, databases, it is usually the case that the number of results of a query is independent of the query execution order, but the execution cost depends on this order. In multimedia and web databases, however, due to the use of thresholds and top-k predicates and sub-queries [5, 11] the number of query results can also change depending on the query execution order. Due to these inherent differences between traditional databases and multimedia and web databases, we need novel query processing mechanisms. In particular, we need novel query optimization algorithms as the *top-k* natures of the query predicates render existing query optimization algorithms dysfunctional.

1.1 Motivating Example

Let us consider an SQL-like multimedia query that retrieves surveillance images that contain an alert pattern:

```
select location, time, image
from images
where surveillance_scans(location, time, image) and
      location = "entrance_gate" and
      extract_pattern(image, pattern) and
      match_pattern(pattern,"alert.gif").
```

In this application, surveillance images are continuously fed from the cameras into a database and the above query is executed repeatedly to identify the alert condition. Therefore, due to the nature of the application, it is essential that the query is evaluated as quickly as possible. Furthermore, since pattern extraction and matching are fuzzy operations, it is essential that the results of the query have associated *confidence level*s that can be used for deciding when to trigger the alert condition. Note that, in this application, we would like to identify the *best alert candidates* in the *shortest amount of time*.

While processing a query, we need to consider the following three issues characteristic to multimedia databases. Note that some of these, such as overloaded implementation of query predicates, are also seen in other database applications. However, altogether, the following three issues define the characteristics of multimedia databases:

Overloaded implementations of query predicates. Media-related predicates can be implemented by various user-defined functions or indexes corresponding to different ways the predicate can be invoked. For instance, the user defined predicate `extract_pattern(image,pattern)`, can have three different overloaded implementations:

- given an image, one implementation extracts a predetermined number of patterns using a pattern recognition function,
- given a pair of an image and a pattern, another one checks for the presence of the pattern in the image using a neural-net engine, and

- given a pattern, the last implementation retrieves all matching images using a cache of pre-extracted of pattern/image pairs maintained in the form of a multi-dimensional index structure that maps all known patterns into a feature space for quick access.

Each of these implementations can have drastically different behaviors in terms of the number of results returned. These are discussed next.

Thresholds and "top-k" predicates. The solution space in most media-related predicates is very large. For example, in the above example, every given pattern matches every other one to some degree. Therefore, providing all answers to a query is neither feasible nor desirable. Consequently, multimedia functions are usually implemented as *thresholding* or *top-k retrieval* functions. This means that a sub-query or a user defined function returns only a small, but most relevant, portion of all possible results. However, if two overloaded implementations of the same predicate define *relevance* differently, then they can return different portions of the result space.

This may cause certain complications in query optimization. One complication is that a query can result in different numbers of tuples depending on the execution plan chosen by the query optimizer. For example, let us assume that we can execute the multimedia query described earlier in two different ways:

- Find all surveillance images scanned at the "entrance gate"; find the patterns in each image; and return those images that contain a pattern which match "alert.gif".
- Find all patterns that resemble "alert.gif" using an index structure; find all images which contain one of these patterns; and return those images that are scanned at the "entrance gate".

In the last surveillance cycle, let us assume that the cameras scanned and fed into the system 100 images, 75 of which are coming from the camera at the "entrance gate". The image processing engine extracts 10 patterns per image. Due to the overlaps in the surveillance images, the number of unique patterns in all these images is 500, instead of 750 (75×10):

- Using the first execution plan, **(a1)** 100 images are retrieved and out of them 75 are chosen. **(a2)** Then the patterns present in these images are retrieved. If for this purpose, we are using a predicate which performs a top-5 retrieval, then 5 top patterns of each image are extracted, and **(a3)** the resulting 375 patterns are checked to see if they match the given pattern.
- When the second query execution plan is used, however, **(b1)** all patterns that match "alert.gif" (say 50 of them) are returned, **(b2)** all images which contain any one of these patterns are returned using an index structure, and **(b3)** out of these images, those that are located at the "entrance gate" are returned.

Note that there is no reason to expect that these two executions will return the same number of tuples. This difference is due to the fact that, at the second step of the process, different overloaded implementations (or binding patterns) of the predicate `extract_pattern(Image, Pattern)` use different retrieval policies: in **(a2)** the result space is pruned to the top-5 results, whereas in **(b2)** all results are maintained.

The necessity of new optimization techniques was noted in [2], where authors provide a novel similarity-based algebra (with new selection and join operators) and develop transformation that can be used in optimizing a query tree. We, on the other hand, use a query model where both selections and joins are modeled as predicates and, consequently, are able to state the query optimization problem as a join-ordering problem. We have studied semantics of various fuzzy conjunction, disjunction, and negation operators in [1]. Note that once an appropriate join order, which obeys the binding patterns of the predicates, is found some of the predicates will act as selections and others will act as join operators; implicitly achieving a rewritten query.

2 Optimization in the Presence of Top-k Predicates

Multimedia queries are generally processed in two ways: **(1)** using regular query processing techniques within database management systems and **(2)** using special merge techniques that benefit from the fact that results returned by most multimedia user-defined functions are ranked based on their scores [3, 6, 10, 11, 14]. In this paper, we develop a model which can lend itself to query optimization in the traditional sense instead of applying a merge strategy.

An optimization model assigns a value (to be minimized) to all partial or complete plans in the search space. It also determines the output size of the data stream for every operator and predicate in the plan. This plan must respect the available binding patterns of each query predicate. A binding pattern of a predicate specifies which attributes of a relation must be bound to access it. For example, a predicate $r(x, y)$, with two parameters can have four binding patterns:

- ff (or, alternatively, $\emptyset$) when both parameters are free,
- bf (or, alternatively, $\{x\}$) when x is bound and y is free,

- fb (or, alternatively, $\{y\}$) when x is free and y is bound, and
- bb (or, alternatively, $\{x, y\}$) when both parameters are bound.

Each binding pattern abstracts an overloaded implementation of a predicate in terms of an external function or an index structure. Therefore, given a multimedia query predicate, each binding pattern of predicate has different *cost*, and *size* parameters associated with it. This research aims at finding an optimal query execution plan, given a set of predicates, and the associated binding patterns, each with different execution cost and size characteristics. Therefore, we need an *optimization* model which captures not only the execution cost, but also the result size.

2.1 Query Execution Cost and Result Size (Fanout)

In this subsection, we introduce two parameters involved in the optimization model we propose, instead of the *table scan cost* and *selectivity* parameters traditionally used for optimizing queries:

- $cost$ and $fanout$.

These parameters have the following properties:

Property I: The $cost$ and $fanout$ parameters are defined *per binding pattern.* For example, given a predicate $r_1(x, y)$,

$$cost^{bf}(r_1(x, y))$$

denotes the expected cost of accessing r_1 by binding the parameter x to a constant and retrieving all matching values for y. Note that this definition is similar to the cost of a limited access pattern as used in [13].

Property II: The $cost$ parameter is defined as a function of the number of inputs to the bound parameters of the predicate. If a predicate, say $r_2(y, z)$ is executed after another one, say $r_1(x, y)$, during the processing of a join query

$$r_1(x, y) \wedge r_2(y, z),$$

then there will be a set, $\mathcal{Y}$, of y values that are already identified before r_2 is processed. Depending on the implementation of the query predicate, either each y value can be processed separately by r_2 to identify the corresponding z values or the entire set of y values can be passed into the corresponding media function at once to identify the corresponding $\langle y, z\rangle$ pairs. These two options would require different cost calculations, each of which is a function of the number of y values. For example, let us consider the two predicates introduced in the Introduction:

$$extract\_pattern(``image1.gif'', pattern) \wedge match\_pattern(pattern, ''alert.gif'').$$

This query can be processed, by finding all patterns in the given image and, then, either (1) by matching each pattern separately with “alert.gif” or (2) by using a search structure, which contains all extracted patterns, for identifying those patterns that are close to ”alert.gif”. Note that the two above conditions are sometimes observed in traditional databases as well. For instance, the two query execution options described above and the difference between the corresponding cost calculations is analogous to difference between nested-loop and hash-joins. The next, and most significant, property however (at least currently) is more common in multimedia and web applications and we are not aware of any attempt to address it within the realm of query optimization.

Property III: The $fanout$ parameter denotes the expected number of outputs as a function of the number of inputs to the bound parameters of the predicate. This definition is inherently different from the definition of *selectivities* used in query optimization literature [12, 15]. For example,

$$fanout^{bf}(r_1(x, y))$$

denotes the expected number of $\langle x, y\rangle$ pairs that the predicate returns per each bound value of x (consequently, the fanout parameter can be larger than 1.0). The reason why we have to define a new parameter called $fanout$, instead of using the standard concept of selectivities is that,

- a top-k retrieval predicate (which may either be a user defined function or a top-k retrieval subquery) will have a predetermined number of results (k) independent of the value of x or the size of the actual result space. For example the predicate `extract_pattern(inputimage, outputpattern)` described in the Introduction extracts at most 5 patterns per input image, independent of the complexity of the image.

- in general, since we are not referring to a table with a known number of tuples, the definition of selectivity does not apply. For example the predicate `extract_pattern(inputimage, outputpattern)` does not perform a selection on a table of $\langle image, pattern \rangle$ pairs. Instead, it extracts whatever patterns are available in a given image.

Note that, whenever they are already available, it is possible to map selectivities to fanouts. For example, if we are given a table with schema $t(x, y)$ which has $num(t(x, y))$ rows and selectivity $sel^{bf}(t(x, y))$, then we can calculate the corresponding fanout as

$$fanout^{bf}(t(x, y)) = sel^{bf}(t(x, y)) \times num(t(x, y)).$$

In other words, this table will return on average $sel^{bf}(t(x, y)) \times num(t(x, y))$ pairs of $\langle x, y \rangle$ per each bound value of x. The reverse of this, however, is not true; that is, given a fanout, it is not always possible to map it into a selectivity as we are not always referring to a table with a known number of rows.

2.2 Modeling the Desirability of a Query Plan

The above semantics of predicate cost and fanout necessitate a corresponding model to evaluate the *desirability* [1] of a query plan. In general, given

- a set of predicates $\mathcal{P} = p_1, \ldots p_n$,
- a set of attributes $\mathcal{A} = a_1, \ldots, a_m$,
- a join query

$$q(a_1, \ldots, a_m) = p_1(a_1, \ldots, a_m) \wedge \ldots \wedge p_n(a_1, \ldots, a_m),$$

- an input binding pattern, $in\_bound \subseteq \mathcal{A}$, which denotes the set of attributes that are bound in the beginning ($in\_free = \mathcal{A} - in\_bound$),
- an output binding pattern, $out\_bound \subseteq \mathcal{A}$, which denotes the set of attributes that are expected to be bound at the end of the process ($in\_bound \subset out\_bound$ and $out\_free = \mathcal{A} - out\_bound$), and
- for each predicate $p_i \in \mathcal{P}$, a binding function

  $bind_i : 2^{\mathcal{A}} \rightarrow \{2^{\mathcal{A}} \cup \perp\}$, such that given an input binding pattern, in, we have $bind_i(in) \supset in$ if the input binding pattern is allowed for this predicate and $bind_i(in) = \perp$ if it is not

  which describes the input/output binding relationships acceptable for this predicate,

we can define the cost and the fanout of the query as

$$\begin{array}{l}
\langle\ cost^{out_bound}(q)\ ,\ fanout^{out_bound}(q)\ \rangle\ =\ \textbf{best_of}\ \{\ \langle cost, fanout \rangle\quad \textbf{such that} \\
\qquad cost = join_cost(\langle cost^{pat_1}(p_1), fanout^{pat_1}(p_1)\rangle, \ldots, \langle cost^{pat_n}(p_n), fanout^{pat_n}(p_n)\rangle)\ \wedge \\
\qquad fanout = join_fanout(fanout^{pat_1}(p_1) \ldots, fanout^{pat_n}(p_n))\quad \wedge \\
\qquad (\mathcal{A} - pat_1) \cup (\mathcal{A} - pat_2) \cup \ldots \cup (\mathcal{A} - pat_n) = in_free\quad \wedge \\
\qquad bind_1(pat_1) \cup bind_2(pat_2) \cup \ldots \cup bind_n(pat_n) = out_bound \\
\quad \},
\end{array}$$

where $join\_cost$ is the combined cost of the join as a function of the predicate costs and fanouts and $join\_fanout$ is the combined fanout as a function of the input fanouts. The last two conditions guarantee that input/output binding constraints are observed. Note that, we do not specify how the $join\_cost$ and $join\_fanout$ are computed, as these depend on the semantics of the join operator. The following example describes one way to compute $join\_cost$ and $join\_fanout$ assuming a nested-loop join.

Example 2.1 Using a nested loop join operator, the best way of computing $q(x, y, z) = r_1(x, y) \bowtie r_2(y, z)$ can be found by selecting the *best* among the following three options:

$$\begin{array}{l}
\langle cost^{bbb}(q(x, y, z)), fanout^{bbb}(q(x, y, z))\rangle = \ \textbf{best_of}\ \{ \\
\quad \langle cost^{ff}(r_1(x, y)) + fanout^{ff}(r_1(x, y)) \times cost^{bf}(r_2(y, z)), fanout^{ff}(r_1(x, y)) \times fanout^{bf}(r_2(y, z))\rangle, \\
\quad \langle cost^{ff}(r_2(y, z)) + fanout^{ff}(r_2(y, z)) \times cost^{fb}(r_1(x, y)), fanout^{ff}(r_2(y, z)) \times fanout^{fb}(r_1(x, y))\rangle, \\
\quad \langle cost^{ff}(r_1(x, y)) + cost^{ff}(r_2(y, z)) + fanout^{ff}(r_1(x, y)) \times fanout^{ff}(r_2(y, z)), fanout^{ff}(r_1(x, y)) \times fanout^{ff}(r_2(y, z))\}.
\end{array}$$

[1] Here, we use the term desirability instead of optimality as, in the literature, optimality is generally used to mean minimal cost.

This formulation describes the three alternative ways to join the given predicates from an input binding pattern $\emptyset$ (all attributes are free) to an output binding pattern $\{x, y, z\}$ (all attributes are bound). It further associates an execution cost (*cost*) and result size ($fanout$) for each option:

- In the first case, x and y have been instantiated by calling r_1 with free variables, and then these values have been validated by accessing r_2 with y bound. The cost is the access cost of r_1 when its attributes are free, plus the total access cost of r_2 when the common attribute, y, is bound for different possible values obtained from r_1. *The size of the results is the product of the expected result sizes associated with the corresponding binding patterns.*
- The second case is similar to the first case, except that the orders of r_1 and r_2 are exchanged.
- In the third case, both relations are independently accessed using both attributes free, and the results have been joined together. □

The **best_of** function is used to choose the best query plan among all the alternatives. In traditional models, it is relatively straightforward to choose the **best** plan among the alternative query processing plans. In the above example, traditional cost models would recognize that there are different costs associated with different ways to join these two relations. However, all possible executions would result in the same number of resulting tuples. Therefore, irrespective of the plan chosen, the final size, $fanout(r_1(x, y) \bowtie r_2(y, z))$, for the above example would be the same and we could choose the *best* plan by identifying the plan with the smallest $cost$. The proposed model, on the other hand, differs from the traditional models in that it has to account for different result sizes for different execution orders, a characteristic of multimedia and web databases as discussed in the Introduction. Hence, we can not choose the *best* plan in such a straight forward manner.

2.3 Alternative Ways to Define the Desirability of a Query Plan

In general, **best_of** will be a function of the cost and the fanout. We see three main alternative ways to define this measure in multimedia and web database applications:

Alternative I: best_of = min_cost. When the goal is to process the query as quickly as possible, the minimum *overall cost* (independent of the number of resulting tuples), we can define **best_of** as **min_cost**. In this model, given a set of $\langle cost, fanout \rangle$ pairs, we would choose the pair with the smallest cost.

This is similar to the *cost-based* optimization commonly used in databases. One major difference (and disadvantage) of this formulation is that, the principle of optimality may not be preserved. In other words, optimal (in terms of cost) plans may not have optimal (in terms of cost) subplans. This is due to the fact that logically equivalent sub-plans may have different fanouts, and when the cost parameter is used to select among alternative subplans, we may not reach an optimal overall plan. Consequently, a **min_cost** based model can not be applied recursively to sub-query plans and we can not use any recursively structured algorithm, such as the popular dynamic programming which assumes that all alternative ways to execute a sub-plan will result in the same number of tuples, to optimize queries.

Alternative II: best_of = min_unit_cost. When we are aiming to generate as many results as possible with the smallest cost, we can define the **best_of** as **min_unit_cost**, where *unit_cost* is defined as

$$unit\_cost = \frac{cost}{fanout}.$$

When the number of results may change based on the way query is processed, *unit_cost* may be a better performance indicator. One desirable property of a *unit_cost*-based model is that, under certain conditions, it may preserve the principle of optimality. For instance, if the cost and fanout is defined as in Example 2.1, then the principle holds.

Note that the concept of *unit-cost* is, in a sense, similar to the concept of *rank* [7, 12] used for dealing with expensive predicates. A *rank* is associated with every predicate in a query and these predicate ranks are used as heuristics to improve query plans. In the model we present, however, *unit-cost*s are associated with alternative sub-plans, not with predicates themselves. Furthermore, even when *unit-cost*s are associated with sub-plans that consist of only one predicate, they are functions of the corresponding binding patterns. In contrast, each predicate has one and only one *rank* associated with it [7, 12].

Alternative III: best_of = min_fanout. When the user wants to find a plan which returns the minimum number of results, we can define **best_of** as the plan with the smallest fanout. Note that, **min_fanout** is especially important when there is an access penalty associated with results (as it is the case when transmission of large media objects from a remote repository to a user is required). The **min_fanout**-based model, under certain conditions, may also preserve the principle of optimality. For instance, if the fanout is defined in a multiplicative way as in Example 2.1, then the principle holds.

2.4 Query Optimization Algorithms based on Desirability

When the **best_of** measure is defined as **min_unit_cost**, since it shows a recursive structure, we can benefit from traditional query processing algorithms, such as dynamic programming.

When the **min_cost** is required, however, we can no longer benefit from these algorithms. In this case, one solution is to use the **min_unit_cost** as a heuristic to reduce the search space at every level of a dynamic programming algorithm [2]. In other words, given a dynamic-programming based optimization algorithm, at each level, we can **(1)** rank sub-plans based on their *unit-cost*s, **(2)** prune-away sub-plans with large *unit-cost*s, and **(3)** consider only those plans with small *unit-costs*. The amount of pruning can be controlled to achieve different levels of optimization speed and optimality. There are three main options:

- *No pruning:* This approach will result in the most optimal plan as it exhaustively searches the whole search space. However, the drawback is that the search space is large and execution times are high.
- *Prune to the maximum level:* This essentially means that every stage the model prunes all the sub-plans except the one with the lowest rank. Though the approach increases execution speed largely, the error introduced is potentially very high.
- *Prune to a DBA-defined level:* In this approach, for each sub-goal, the algorithm considers the best k ranked subplans for further joins and others are neglected. These k sub-plans are used for further cost evaluations in the subsequent stages.

Since, **min_cost** is an important measure even in multimedia databases, it is important that the proposed heuristic works well and efficiently. Therefore, in the experiments section, we will evaluate the effectiveness of this approach.

3 Opt-Histograms: Adaptive Reduction of Search Space Granularity

Optimization is an NP-complete problem. The problem is rendered more drastic by the fact that, as discussed in Section 2, when we optimize for cost, we can not benefit from dynamic-programming based algorithms. As a solution, in Section 2, we proposed a sub-plan-rank based pruning approach to reduce the search space largely. Still, when k is large, the savings achieved in search space and execution time may not be very significant. Therefore, in addition to pruning the search space as described in Section 2, we propose to reduce the granularity of the search space to further bring down the complexity of query execution.

The number of sub-plans to be considered can be significantly reduced by approximating the costs of the sub-plans and choosing only the best of them. The desired level of approximation needed is described as the granularity of the distribution. If the granularity is small, there potentially is a considerable reduction in optimization time, but this introduces an error in the optimization.

We divide the whole search space into buckets, whose sizes are determined by the granularity desired. All sub-plans that belong to a particular interval of values are placed in the same bucket. In each step of query-optimization, instead of using all sub-plans, only one sub-plan from every bucket is used. Due to this approximation, the search-space size is controlled.

Example 3.1 Let us have eight alternative sub-plans that we have to consider,each with the following associated costs:

Sub-plan 1 = 120,	Sub-plan 2 = 130,	Sub-plan 3 = 240,	Sub-plan 4 = 520,
Sub-plan 5 = 650,	Sub-plan 6 = 110,	Sub-plan 7 = 1000,	Sub-plan 8 = 620.

Let us also assume that we use a bucket width of 100 unit costs to reduce the granularity of the search space. Instead of maintaining all eight alternatives, we need to maintain only five alternatives:

Sub-plan 6 = 110,	Sub-plan 3 = 240,	Sub-plan 4 = 520,	Sub-plan 8 = 620,	Sub-plan 7 = 1000.

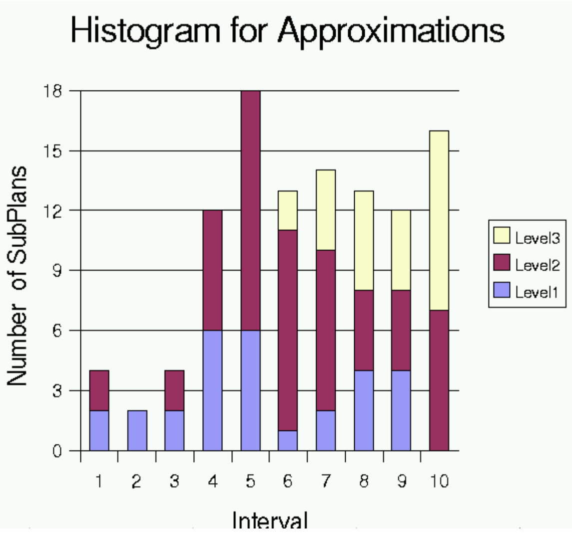
□

Note that in the above example, the buckets are placed uniformly across the search space starting from cost 0. However, in general, identifying the boundaries and widths of the buckets is not trivial. First of all, since the execution cost is likely to increase with the number of predicates in the query, at different levels of query optimization, different bucket boundaries may need to be used. Secondly, since the search space has two dimensions, *cost* and *size*, we need to identify different bucketing strategies for each dimension. In this section, we introduce *opt-histograms* that can be used for intelligently choosing the granularity (places of bucket boundaries) of the search space. Opt-histograms describe the optimization statistics accumulated by observing similar queries optimized over a period of time.

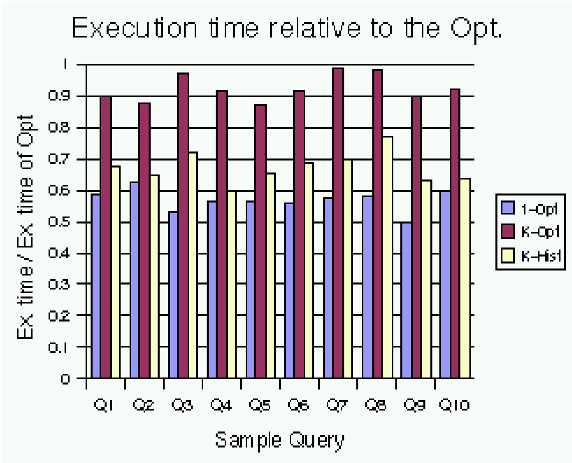
3.1 Opt-Histograms

An opt-histogram is generated by analyzing the query optimization history. Figure 1(a) shows an example opt-histogram generated using a set of similar queries. The opt-histogram shows the distribution of the estimated costs of sub-plans that belong to different

[2] Since dynamic programming-based query optimization algorithms are common and well-understood, we do not provide the pseudo-code of such an algorithm in this paper



(a)



(b)

Figure 1: (a) An example opt-histogram, which shows that the relevant space of costs varies with the query optimization stages. (b) Average optimization time relative to the optimization time required by the optimal-cost algorithm

level of join ordering. This histogram was generated over 1000 optimization tasks using a 4-relation query. The levels in the figure denote the 2-, 3-, 4- way join stages of the query optimization process. As shown in Figure 1(a), we observed a consistent behavior of sub-plan execution costs in different query optimization stages. Note that, according to this figure, relevant regions of the cost-space shifts at different stages of the query optimization process. Therefore, different bucket boundaries should be used at different stages of query optimization.

Note that, one property of the opt-histograms that is special to the multimedia and web databases, is that sizes show different characteristics than the cost. Sizes can be defined multiplicatively (unlike the additively defined cost). Therefore, they require non-uniform granularity reductions of the space. In this paper, we do not discuss the details of the opt-histogram creation for size.

In Section 2, we mentioned that a dynamic-programming based algorithm can be implemented by selecting the best k ranked sub-plans for every two-way join, when we are dealing with the **best_cost** metric which does not preserve optimality for sub-plans. We can further reduce the search space, by coalescing the sub-plans using a low granularity search space. All plans that fall into a particular bucket are approximated as being equivalent and this reduces the search space further. The salient features of the described approach are that it

- approximates the sub-plans by reducing the granularity of search space,
- uses an optimization histogram to reduce the granularity of the space adaptively, and
- generates this histogram by observing the behavior of the algorithm, for similar queries, at different levels of join ordering.

Next, we describe experiment results to justify that highlight the issues raised in this paper and that evaluate the effectiveness of the proposed solutions.

4 Experimental Results

In order to evaluate the effectiveness of the techniques presented in this paper, we ran a set of experiments. In order to observe the behavior of the proposed query optimization algorithm under very different conditions, we opted to use synthetically generated data. For this purpose, we used a set of join-queries, each with different number of joins and joining attributes. In different runs, each predicate got associated with randomly generated data that follow random *cost* and *size* patterns. These patterns are generated keeping in mind the characteristics of multimedia and web predicates:

- *Cost of predicates:* The cost of predicates varied greatly. In fact, the cost of different binding patterns of individual predicates also varied. There are expensive binding patterns which correspond to user-defined functions which require media processing and there are cheaper binding patterns which correspond to use of appropriate index structures.

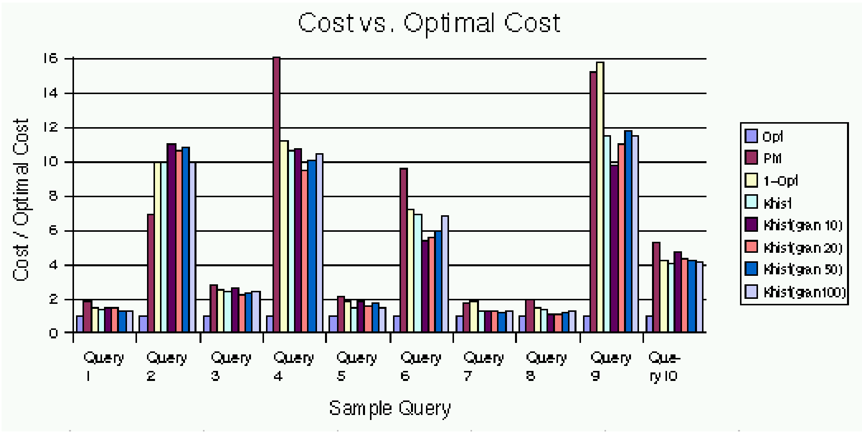


Figure 2: Relative optimality of various algorithms

- *Size:* The size factor of predicates and their binding patterns also varied to reflect the top-k retrieval functions which return a predetermined number of results for each input as well as thresholding functions which returns a portion of the database determined by the threshold value as well as the inherent distribution of the data.

‘ We have varied these expected cost and size values of each predicate across different runs to observe the performance of the proposed optimization algorithm under different input parameters. In order to compare the performances (in terms of time as well as the closeness to optimality) of various approaches, we have implemented the following algorithms:

- *Optimal:* This algorithm performs an exhaustive search. It enumerates and expands all possible plans with no pruning or approximations.
- *Predicate migration:* We have also implemented the predicate migration algorithm [12]. Note that predicate migration uses only one rank per predicate. It is essentially a heuristic when applied to the model presented in this paper. Therefore, to evaluate its performance conservatively, for each predicate we used the worst rank among all its binding patterns.
- *1-Optimal:* This algorithm prunes the search space choosing only one best plan for each sub-goal.
- *K-Optimal:* This algorithm keeps k best plans for each sub-goal (3 in these experiments).
- *K-Hist:* This algorithm approximates the top k ranked sub-plans using a reduced granularity search space.

The following sections discuss how these algorithms perform with respect to cost and size factors.

4.1 Complexity

Figure 1(b) shows execution cost of various algorithms for different queries. Since the optimal algorithm performs an exhaustive search, it requires the highest execution time. The predicate migration considers only one plan per level of join and hence the search space is very small. Therefore, it performs the fastest optimization (not shown in the figure). 1-opt also performs optimization quickly (in the same order of the predicate migration) when compared to others as it prunes the search space to the greatest level possible and thus time for enumerating the entire search space is reduced. K-opt searches a larger search space and hence performs slower than 1-opt and faster than optimal algorithms. The k-hist algorithm reduces the search space by reducing the granularity of the search space and approximating the results at every stage. Thus, a considerable saving in execution times is achieved.

4.2 Degree of Optimality

Figure 2 shows the relative behavior of different algorithms in terms of finding the best query execution plan. The experiments were performed for a small sample of queries with different predicate costs and sizes [3]. The x-axis shows a sample of queries and y-axis shows the ratio of the cost of each resulting plan to the cost of the cheapest plan (found by the optimal algorithm). According to this figure, 1-opt over-prunes the search space and the costs deviate from the optimal largely. The k-opt algorithm

[3] Since in this paper we do not concentrate on the question of whether different algorithms works better for different queries, we do not list the actual queries used in this example. But, it is important to note that the relative patterns is more or less the same across different queries.

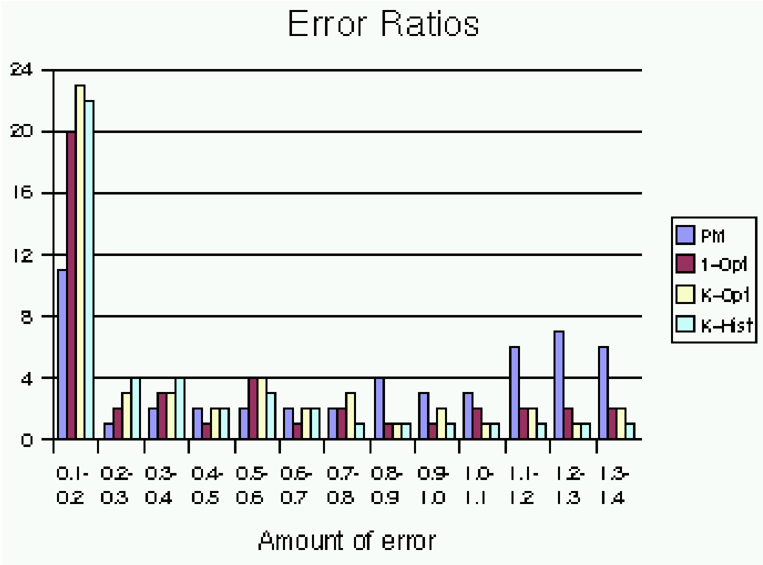


Figure 3: Error distribution (error boundaries are selected in 0.1 increments (i.e., 0.0-to-0.1, 0.1-to-0.2, etc.)

performs relatively well and k-hist (shown with different granularities) approximates the results of k-opt well. Note that, as the granularity increases, k-hist resembles the k-opt algorithm. Note that, since it is essentially a heuristic when applied to the novel model, predicate migration behaves less predictably.

Note that although it shows the relative behavior of the different optimization algorithms, this small sample does not represent the degree of optimality of these algorithms. A better way to observe the degree of optimality is to plot an histogram of their error relative to the optimal result:

$$Error = \frac{Cost - Optimal\_cost}{Optimal\_cost},$$

over a large number of queries. The experimental results are plotted in Figure 3. In this figure, each bucket on the x-axis corresponds to an error range of size 0.1 (that is intervals corresponds to the error ranges 0.0-0.1, 0.1-0.2 etc.) and the y-axis corresponds to the number queries which falls in each bucket (or error range). Note that according to this figure, k-opt algorithm has the highest number of correct estimations (error range within the 0.0-0.1 bucket). Note, on the other hand, that surprisingly, k-hist algorithm seems to cause a smaller number of large errors than the k-opt algorithm. Although, this is interesting, we do not believe that it is significant and generalizable.

5 Related Work

The cost model used by most researchers does not consider the effect of varying result sizes. Selinger *et al.* [15] find the cost of a join assuming a constant product of cardinalities independent of query execution order. Recently, there has been efforts realizing the fuzzy and probabilistic nature of new database applications (such as [2, 3, 10, 11, 14]) and benefit from this nature in top-k retrieval [5]. Most of the existing work in this area concentrate on identification of efficient query processing techniques for queries with fuzzy predicates. In this paper, we concentrate on optimizing queries; i.e., identifying efficient query plans when there are various alternatives. In [6], Chaudhuri and Gravano discuss cost-based query optimization techniques for such *filter* queries. In essence they convert top-k retrieval queries to thresholding (filtering) queries that can be processed using more traditional database processors. In [9] Donjerkovic and Ramakrishnan looked at the effect of top-k evaluations on the query optimization task. Their focus was to minimize the overall query execution cost by trying to minimize the effect of restarts due to too little answers. In [3], we presented an approximate query evaluation algorithm that builds on [10, 11] to address the existence of non-progressive fuzzy predicates. This algorithm uses the statistical properties of the minimizes the unnecessary accesses to non-progressive predicates, while providing error-bounds on the top-k retrieval results.

Hellerstein [12] presents various techniques for efficiently optimizing declarative queries that contain time consuming methods. The approach used for optimization in this work is construction of a plan tree as a first step with each sub-plan represented as a separate stream. Then each of the streams are sorted based on a rank[4]. Then a differential cost function prunes the results by pulling the top ranked predicates. Hellerstein discusses four main optimization strategies, namely pushdown with rank ordering, pull up costly sub-plans, pull the costly ranked predicates while performing a join based on join ranks rather than predicate ranks, and predicate migration, which considers the least cost plan at every stage of join. LDL [8] treats predicates as relations. One major drawback of the optimization approach used in [8] is over eager pull up decisions. The algorithm makes decisions looking at only the immediate sub-plans. Another drawback of this approach is that the complecity is exponential in terms of number of relations and number of expensive predicates. [4, 7] provide a comparison of LDL and predicate migration approaches. Their research focuses on pruning by ranks where $Rank = \frac{Cost}{(1-Selectivity)}$. Chaudhuri and Shim's algorithm [7] performs well with the assumption that predicates with smaller ranks are always applied earlier than others. The assumption however holds for restrictive predicates only (predicates where all the parameters are bound), which ceases to be the case when varying binding patterns are used. Another assumption of their work is that there is only one rank value per predicate which again may not always hold. The complexity is polynomial in the number of user-defined functions. The heuristic is very successful in pruning unwanted results. The push down heuristic pushes all expensive predicates down or defers their evaluation until the last join.

6 Conclusion

In this paper, we presented a new query optimization model for multimedia and web databases, which use top-k predicates. Unlike the previous work, we considered non-restrictive predicates in query optimization. This model takes into account different bindings for every predicate, and it recognizes the fact that a query may have different number of results (or coverages), depending on the query execution plan. We used an efficient pruning criteria that prunes, based on cost and size factors, in the sub-plan level. The proposed approach addresses the well-known NP-complete nature of the query optimization problem by *adaptively* reducing the granularity of the search space. For this purpose, unlike the data histograms which capture the data distribution, we use opt-histograms that capture the distribution of sub-query-plan values over many query optimization tasks.

Acknowledgements

We thank to Chris Mayer for his helpful comments.

References

[1] K.S. Candan and W.-S. Li. *On Similarity Measures for Multimedia Database Applications*, KAIS journal, 3(1), pp. 30-51, 2001.

[2] S. Adalı, P. Bonatti, M.L. Sapino, and V.S. Subrahmanian. *A Multi-Similarity Algebra.* SIGMOD 98.

[3] K.S. Candan, W.-S. Li, and M.L. Priya. *Similarity-based Ranking and Query Processing in Multimedia Databases.* DKE 35(3): 259-298, 2000.

[4] Surajit Chaudhuri, *An Overview of Query Optimization in Relational Systems*, Principles Of Database Systems, 1998.

[5] Surajit Chaudhuri and Luis Gravano, *Evaluating Top-k Selection Queries*, VLDB 1999, pp 397-410, 1999.

[6] Surajit Chaudhuri and Luis Gravano, *Optimizing Queries over Multimedia Repositories*, SIGMOD 1996, pp. 91-102.

[7] Surajit Chaudhuri and Kyuseok Shim, *Optimization of Queries with User-Defined Predicates,* VLDB 96, pp. 87-98.

[8] D. Chimenti, R. Gamboa, and R. Krishnamurthy. *Towards on Open Architecture for LDL,* VLDB 1989, pp. 195-203, 1989.

[9] Donko Donjerkovic and Raghu Ramakrishnan, *Probabilistic Optimization of Top n Queries.* VLDB, pp. 411-422. 1999.

[10] R. Fagin. *Combining Fuzzy Information from Multiple Systems.* 15^{th} ACM Symposium on Principles of Database Systems, pp. 216-226, June 1996.

[11] Ronald Fagin. *Fuzzy Queries in Multimedia Database Systems*, Principles of Database Systems, Seattle, WA, 1998.

[12] Joseph M. Hellerstein, *Optimization Techniques for Queries with Expensive Methods,* Association of Computing Machinery, Transactions on Database Systems, Vol.23, No.2, June 1998, pp 113-157.

[13] Alon Levy, Ioana Manolesu, Dan Suciu, Daniela Florescu, *Query Optimization in the Presence of Limited Access Patterns*, SIGMOD 1999, pp. 311-322, 1999.

[14] M. Ortega, Y. Rui, K. Chakrabarti, K. Porkaew, S. Mehrotra, and T.S. Huang, *Supporting Ranked Boolean Similarity Queries in MARS*, TKDE, (10)6, pp. 905-925, 1998.

[15] P. Griffiths Selinger, D.D. Chamberlin, R.A Lorie, T.G. Price, *Access Path Selection in a Relational Database Management System*, SIGMOD 1979, pp.23-34, 1979.

[4] The definition of rank of a predicate is $Rank = \frac{(Selectivity-1)}{Cost}$