Factorizing Complex Predicates in Queries to Exploit Indexes

Surajit Chaudhuri
Microsoft Research

Prasanna Ganesan*
Stanford University

Sunita Sarawagi
IIT Bombay

ABSTRACT

Decision-support applications generate queries with complex predicates. We show how the *factorization* of complex query expressions exposes significant opportunities for exploiting available indexes. We also present a novel idea of relaxing predicates in a complex condition to create possibilities for factoring. Our algorithms are designed for easy integration with existing query optimizers and support multiple optimization levels, providing different trade-offs between plan complexity and optimization time.

1. INTRODUCTION

Modern relational database systems have increasingly come to process queries generated by data-warehousing and data-mining applications. These queries often contain long and complex predicate expressions and execute against large databases, requiring the optimizer to discover good execution plans despite the complexity of the query expressions. The optimization of queries where complexity is due to a large number of joins has received a lot of attention in the database literature, but the optimization of complex selection conditions involving multiple ANDs and ORs has not been widely addressed. In this paper, we concentrate on the problem of optimizing queries with a complex selection condition involving multiple atomic predicates.

While the objects of our optimization are arbitrary SPJ queries, let us first consider a single-table selection query with a complex selection condition. A straightforward way to evaluate the query is to use a sequential scan on the table and evaluate the condition as a filter. However, when the selectivity of one or more conditions is small, and if these conditions can be evaluated by available indexes, it is important to explore the alternative of using a single index seek or scan, as well as that of using multiple indexes together using index-intersection and union (IIU) [17]. Thus, for example, a condition of the form $X\ AND\ Y$ (written as XY) can be evaluated by generating RID lists for each of X and Y and then intersecting the two lists. Similarly, a condition $X\ OR\ Y$ (written as $X + Y$) can be evaluated using the union of the two RID lists. In general, any multi-predicate selection condition can be evaluated by a series of index intersections and unions (IIU), eventually leaving us with a list of RIDs to be fetched from the data table. In some cases, the indexes may provide all the information necessary to answer the query and the actual data need not be accessed at all. Such query plans are called *index-only* plans and can often turn out to be the execution plan of choice if the relation is wide [10]. Although all our discussion in this paper assumes B+-tree or hash indexes on one or more attributes, the technique of factorization is effective for other indexing techniques as well. We omit a detailed discussion for lack of space.

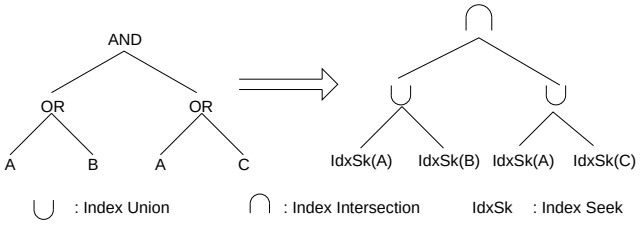

Figure 1: Condition C_1 and its IIU plan

There are many possible IIU plans for a given selection query. Say we have a condition C_1 of the form $(A+B)(A+C)$. The corresponding IIU plan for this condition is shown in Figure 1, consisting of an index access for each of the literals, two index unions and one index intersection. Consider instead the equivalent condition C_2 of the form $A+BC$. The corresponding index-based plan for this condition requires one less index access, and one less index union operation.

While we have shown a fairly trivial example of improving on a query plan, the general problem of identifying better plans for an arbitrarily complex Boolean expression is not easy. Lest we give the impression that representing conditions in *Disjunctive Normal Form(DNF)* will guarantee the the best plan, consider another condition C_3: $AB + AC$ in DNF. The equivalent *Conjunctive Normal Form(CNF)* for C_3 is C_4: $A(B + C)$. It is not immediately obvious as to which of C_3 and C_4 would lead to a better plan. We expect C_4 to be better in several cases since it saves on an index access for A and performs one less index intersection. However C_3 could be better than C_4 if the intermediate RID size of $B+C$ is considerably larger than that of A. C_3 could also be preferred if there is a composite index on AB and a single index on C. This example establishes that neither of the

*Work done while the author was visiting Microsoft Research

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

standard normal forms, CNF and DNF, provides us with a solution. The relative merits of different IIU plans depends on the set of available indexes, selectivities of the different predicates, and the costs of accessing different indexes.

Existing query optimizers approach this problem in three ways. The first approach has been to resort to a sequential scan followed by a filtered evaluation of the condition, particularly when the selection condition is very large and contains several disjuncts and conjuncts. In addition to the I/O cost, even the CPU cost of evaluating a complex condition can be quite high. A number of methods have been proposed to optimize the order of evaluating complex filter conditions to reduce CPU cost [11, 14]. The second approach has been to generate an IIU plan directly from the form in which the condition is represented in the query without searching the space of IIU plans for the best plan. The third approach has been to rewrite the query in CNF (as in System R[21]) or DNF[8]. We have already shown that neither of these forms might be optimal in all cases.

As a crude approximation, we can imagine that the best form of the Boolean expression would be the one using the fewest number of literals. A well-known technique to minimize the total number of literals in a Boolean expression is *factorization*. Factorization has been used to optimize complex expressions in a variety of fields, including compilers, VLSI design and databases [20, 4, 14]. Since our problem actually needs to deal with various index-availability patterns and varying costs associated with index scans, index intersections and unions, minimizing the total number of literals does not always lead to the best plan in our case.

This paper addresses the problem of identifying the best IIU plan for a query using the idea of factorization, but with modifications to deal with real cost functions. We illustrate two ways in which factorization may be used effectively in query optimization.

First, complex conditions often contain atomic predicates repeated across many disjuncts. This becomes even more likely when predicates are blindly normalized to a standard DNF format. We introduce *exact factoring* to exploit such repetition and reduce access cost. We discuss how the exact-factoring problem can be solved efficiently for various expression formats in Section 3.

Next, predicates in a Boolean expression might share similar subexpressions even if they don't have a strict common subexpression. We propose *Approximate Factoring* that exploits a novel condition-relaxation technique to derive a predicate expression (*approximate factor*) that must be satisfied for the given Boolean expression to be true. The approximate factor might help achieve lower data access cost even though it still requires evaluating the given Boolean expression subsequently. Thus, this technique expands the scope of factorization where syntactic factorization is not possible. Consider an example E_1: *(x BETWEEN 1 AND 5)A + (x BETWEEN 2 AND 6)B* where x is a numerical attribute and A and B refer to some other logic expressions. We can relax both predicates involving x to (*x BETWEEN 1 AND 6*) and factor it out to produce E_2: *(x BETWEEN 1 AND 6)(A+B)*. The factored expression might have considerably lower data access cost even though it requires an additional filter pass to evaluate the exact expression E_1. In Section 4, we describe a locally-optimal algorithm for performing such factoring.

The output of our optimizations is a factorized Boolean expression with a corresponding IIU plan, whose RID list will be used to fetch the data pages on which any remaining part of the expression will be evaluated. Note that our optimization technique will automatically identify an index-only plan, if such a plan is applicable and more efficient. Finally, our design takes into account the need for easy integration with existing optimizers as well as the goal of minimizing query-compilation time. We discuss the framework for the integration of our solution into a real optimizer and to optimize arbitrary SPJ queries in Section 2. Section 5 describes the experimental evaluation of our algorithms on a commercial DBMS, using both real and synthetic workloads. Related work and conclusions appear in Sections 6 and 7 respectively.

2. ARCHITECTURAL FRAMEWORK

2.1 Integration with existing framework

We discuss the role of factorization in optimization in the context of SPJ queries. Factorization plays a critical role for join ordering as well as for choosing access paths. While exploring a particular join order, it is important to be able to extract the relevant join conditions when the query expression is complex and not in CNF. We highlight the role of factorization in this context in Section 2.2.

When a query expression is complex and selective, factorization provides a powerful tool for the access-path selection module of the optimizer. Our factorization routines for identifying the best IIU plans for a given condition build upon the existing single-table access-path selection routine found in many optimizers that already is capable of determining index-intersection (and index-only) plans for a conjunctive expression [10]. Since we refer to this access-selection routine for conjunctive conditions repeatedly in this paper, we introduce some notation here. Let f be a conjunctive condition to be satisfied by retrieved tuples of a table T and let A be the set of attributes of T that needs to be retrieved. We use $\mathbf{AccessPath}(f, A)$ to denote the best access path for retrieving all RIDs and attributes in A that satisfy f. The plan returned could involve a table or an index scan, a single index seek or an intersection of multiple index seeks or scans. Some of the index seeks could be on multi-column indexes and cover multiple atomic conditions in f. There may also be a data-lookup step if all the conditions in f aren't covered by a corresponding index seek. Most optimizers use a cost-based search in coming up with such a plan. The estimated cost for the access path selected is denoted by $\mathbf{AccessPath}(f, A)\mathbf{.cost}$ in the rest of this paper. During factorization we build upon existing access path selection routines, and thus do not need to bother about details of specific indices and cost functions.

Note that access-path selection and join ordering may interact in complex ways. For example, if a nested-loops-with-index join method is chosen, the inner table is accessed using an index on the join condition. But the quality of the access path available for that table may also, in fact, determine the join order. This interaction between join ordering and access-path selection is, however, orthogonal to the location of our factorization routines. It is also important to note that the optimization for access-path selection (as well as for join ordering) is influenced by the optimization level that trades off query compilation time and the quality of plans. While describing the role of factorization, we also demon-

strate different ways in which factorization may be applied to achieve such trade-offs.

2.2 The Search Space for Factorization

We design our factorization routines to accept a selection condition on a single table as input. The selection condition can be visualized as a general AND/OR tree with the atomic predicates forming the leafs. The indexes on the table cover any number (zero, one or more) of the leaf nodes. The condition itself may be in any one of three forms: DNF, CNF or an arbitrary Boolean expression. As one may expect, we can exploit the fact of the condition being in CNF or DNF to design more efficient factorization algorithms than for the general case. The output is a plan for evaluating the selection condition on the table, involving various index intersection and union operations and data lookup steps. If index-only access or sequential scan is found more efficient, such plans would be chosen. The factorization routines are designed to produce plans at three levels of complexity. The complexity level chosen is directly tied to the optimization level at which the access-path selection module is invoked.

At the lowest complexity level, only single-index and index-intersection-based plans are considered. Such plans, if available for a query, can be identified extremely fast and prove useful for a large fraction of queries. For example, given the expression $AB + AC$, with an index available for condition A, we can produce a plan that performs an index seek with condition A, followed by a data seek and filter using condition $B+C$. In most cases where single-index or index-intersection plans exist, it is usually not necessary to explore the search space further to look out for more complex plans. A simple variant of this routine is also used to identify join conditions. Say, a plan first joins three tables T_1, T_2 and T_3, and we now wish to join the result with another table T_4. The join conditions for this join are simply all (atomic) conjunctive factors of the WHERE clause that involves an attribute from T_4 on one side of the atomic condition, and an attribute from one of T_1, T_2 or T_3 on the other side. If no such condition exists, the join is, in fact, a cross-product.

At the next level of complexity, we consider access paths that include a single index union over index intersections. For example, the condition $AC + BC + AD$, with individual indexes available for A, B and C, requires the index-union operator in order to exploit any of the indexes. One plan for this condition could be obtained from rewriting it as $A(C+D)+BC$, and performing index-intersection for B and C, followed by an index-union with the index corresponding to A, and finally a filter pass using the original condition. Such plans are among the most common for queries which do not have a simple index-intersection plan.

At the highest level of complexity, we consider plans permitting an arbitrary mixture of index-intersections and unions. A plan corresponding to the condition $A(B + C)$ might require an index-union for satisfying $B + C$ followed by an index-intersection with the index for A. Although there are many queries which might require such sequencing of intersections above unions, and even multiple levels of intersections and unions, the cost of executing such plans is normally higher than plans of the earlier two kinds. The presence of index-union lower down in the tree means that the input to the intersection operator is likely to be large which, in turn, drives up the cost of index intersection. Thus, such complex trees often end up more expensive than a simple table scan, limiting their utility to a smaller fraction of the queries normally encountered.

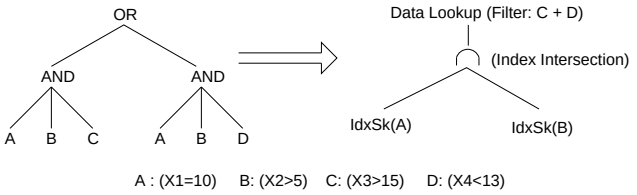

Figure 2: An Example Query Expression and the Optimized Plan

3. EXACT FACTORING

In this section, we discuss the use of exact factoring to find efficient index-based plans. As discussed in Section 2.2 we can produce plans at three different levels of complexity. We now consider each level individually.

3.1 No-Union Plans

We describe a simple algorithm to generate an output plan consisting solely of index seeks and optional index-intersection and data-lookup steps. The optimal algorithm consists of two stages, first identifying the greatest common factor f over the given AND/OR expression, and then invoking the AccessPath procedure above to find the best plan. For any arbitrary AND/OR input expression E, the greatest common factor f can be found using a simple bottom-up traversal of the query-expression tree as shown below:

Algorithm NoUnion(E)

1: **for** each node n visited in any bottom-up order of E **do**
2: If n is a leaf, initialize $n.factor = n$.
3: If n=*AND*, $n.factor = \bigcup_{c \in Children(n)} c.factor$.
4: If n=*OR*, $n.factor = \bigcap_{c \in Children(n)} c.factor$.
5: **end for**
6: f = *rootNode.factor* // $E = f.Q$ for some Q.
7: A = all attributes in E
8: return AccessPath(f, A)

The above algorithm is extremely efficient because it does not involve any expensive rewrites of the expression. It is optimal subject to the optimality of the AccessPath(f, A) routine introduced in Section 2.

Example: Consider the query "SELECT * FROM T WHERE ((X1=10) AND (X2>5) AND (X3 > 15)) OR ((X1=10) AND (X2>5) AND (X4<13))". Figure 2 shows the query expression associated with the WHERE clause of this query. Applying the above factorization step, we identify the maximal conjunctive factor f to be (X1=10) AND (X2>5). We then invoke AccessPath((X1=10) AND (X2>5), {X3, X4}). Let us say that table T has single-column indexes on both X1 and X2. Procedure AccessPath can then produce a plan involving zero, one or both these indexes. The figure shows one possible output plan produced by the procedure.

3.2 Single Union over Intersections (UoverI)

In this case, the output plan is allowed to have any number of index-intersection nodes followed by a single index-union node that combines the output of all the intersection nodes. We will first concentrate on the case where the input tree is in DNF. We then proceed to handle the case where the input is in CNF and, finally, an arbitrary AND/OR tree. These cases could be handled by just converting the expression to DNF, but such conversion could be expensive especially if

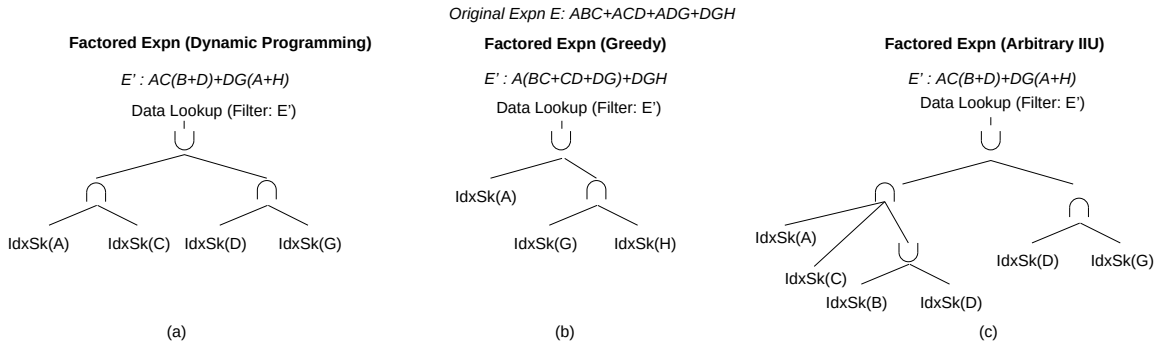

Figure 3: Example Plans from DNF Expressions

the expression is nearly in CNF.

3.2.1 Input: DNF

Let the input expression E be of the form $e_1 + e_2 + \ldots e_n$ where each e_i is a conjunction of atomic predicates. If there are no predicates in common between the disjuncts, then the only option for a UoverI plan is to evaluate each disjunct as an index intersection and add a single union above them. When various subsets of the n disjuncts have common factors, several options arise for rewriting them to obtain better plans. Consider, for example, the query expression $E = ABC + ACD + ADG + DGH$ shown in Figure 3 where A, B, C, D, G and H all stand for atomic predicates. The first two terms share AC, the first three share A, the last two share DG and so on. Factorization using one or more of these shared predicates could produce an improved plan, depending on the available indexes.

We next present an optimal dynamic programming algorithm, UoverI_dnf, for finding the best UoverI plan from an input DNF expression. We first explain the algorithm ignoring union costs. The basic idea is to consider all possible ways of factorizing a DNF expression E as $f.Q + R$, where f stands for a conjunctive factor of some of the terms in E and R is the remainder, i.e., those terms which don't have f as a factor. The cost of the access path for E is then given by the equation:

$$\text{UoverI\_dnf}(E).cost = \min_{E=f.Q+R}(\text{AccessPath}(f,Q).cost + \text{UoverI\_dnf}(R).cost) \quad (1)$$

The AccessPath function returns an index-intersection plan on f that needs to be unioned with the union-over-intersection plan returned on R. Note that AccessPath could return a table scan as the best plan for f. In this case, the entire expression will be evaluated using a sequential scan.

Equation 1 can be immediately translated into a dynamic-programming solution where we simply consider all subsets of the original expression E in increasing order of size. We can optimize the implementation of the above dynamic-programming formulation by only considering those sets of terms that have a common factor. Consider, once again, the expression E in Figure 3. We proceed bottom-up, first initializing the cost for all singleton terms. We then consider all pairs with a common factor, namely ABC and ACD, ABC and ADG, ACD and ADG, ACD and DGH, and ADG and DGH. Once the costs have been computed for all these pairs, we consider all triplets of terms with a common factor, and finally, all possible factorizations for the entire expression. For example, the best option could be to factorize two different pairs, to create the final decomposition $AC(B+D)+DG(A+H)$, with the corresponding plan shown in Figure 3(a).

It is easy to also incorporate union costs into the above equation. The topmost union node, if any, of the plan for evaluating R can be collapsed with the binary union between fQ and R to get a single multi-ary union plan. For most union implementations, the cost of the union depends only on the sum of the lengths of its inputs and the length of the output, which is the same for all possible ways of factorization. Equation 1 is easily modified to account for the union cost by adding a new term unionCost(l_{fQ}) where l_{fQ} is the number of RIDs that satisfy the expression fQ. The only situation when the unionCost term will not be added is when R is empty and E is the top-level expression. In this case we do not have a union-based plan.

The above formulation returns the optimal single-union plan under the assumption that there is no gain to be achieved by combining data lookups of the union-ed RID lists. This assumption does not hold when there is a significant overlap of RIDs among the union-ed lists. Unfortunately, there normally aren't enough statistics maintained for the optimizer to accurately estimate the amount of this overlap. For this reason, we exploit RID overlap, not by incorporating it analytically into our dynamic programming formulation, but through heuristic post-processing steps. We omit details in this paper.

The worst case complexity of the memoized bottom-up dynamic programming algorithm is $O(3^n)$ where n is the number of disjuncts in the input DNF. However, in practice the cost is significantly smaller because we only need to consider those subsets of disjuncts that have at least one *indexed* factor in common. The dynamic programming solution can still prove to be expensive in some cases, leading us to also consider greedy alternatives. One possible greedy algorithm is to modify Equation 1 to choose the factor f that yields the greatest reduction in cost by factorizing E as $f.Q+R$. In our example expression $E = ABC + ACD + ADG + DGH$, the greedy algorithm might find that factoring out A is more beneficial than factoring AC or DG. Thus we would end up factorizing the expression as $A(BC + CD + DG) + DGH$. The corresponding plan is shown in Figure 3(b). The algorithm presented below performs such a greedy rewriting, which can then be used to call AccessPath() to identify the best UoverI plan.

UoverI_dnf_Greedy (E):

for each factor f (set of one or more base literals) **do**

Benefit(f) = ($\sum_{e_i=f.x_i}$ AccessPath(e_i, e_i).*cost*)− AccessPath(f, Q).*cost*;

end for

Choose f_m with the maximum benefit

Rewrite $E = f_m.Q + R_m$.

Invoke UoverI_dnf_Greedy(R_m).

3.2.2 Input: CNF

We now consider the case where the input expression is in CNF, of the form $c_1.c_2.\ldots c_n$ and the output is restricted to be a UoverI plan. We avoid converting the CNF into DNF and applying the earlier solution because of the exponentially large number of terms the conversion might generate. Instead, we propose a solution that does simple opportunistic conversion into DNFs of only a subset of terms as needed.

Assume each c_i is of the form $(l_{i1}+l_{i2}\ldots+l_{ik_i})$ where each l_{ij} term refers to a atomic condition. The c_is with only one literal ($k_i = 1$) are amenable to slightly better optimization but we do not discuss them due to lack of space. A simple choice for the output plan is one of the c_is. Any c_i with a suitable set of corresponding indexes can be evaluated using a single index-union node. The rest of the terms c_js ($j \neq i$) would be evaluated as a filter.

We next consider solutions that combine terms. It is not useful to combine two terms c_i and c_j that do not share any literal. For example expanding out $(A + B)(C + D)$ as $AC + AD + BC + BD$ does not offer any opportunity for better factorization or any improvement in index access for the cost functions we normally encounter. However, when two terms share one or more literals, converting them into DNF could be useful, as when $(A + B)(A + D)$ is rewritten as $A + BD$.

We propose a recursive procedure called UoverI_cnf that uses these ideas to find a single-union plan. Assume that $c_1, c_2 \ldots c_k$ are k terms with a common set f of one or more literals. Thus, E could be rewritten as $(f + Q)R$ where $Q = (c_1 - f)(c_2 - f)\ldots(c_k - f)$, and $R = c_{k+1}...c_n$. The best such rewriting is given by the equation:

$$\begin{aligned} \text{UoverI\_cnf}(E).cost &= \min_{E=(f+Q)R}(\text{UoverI\_dnf}(f).cost \\ &+ \text{UoverI\_cnf}(Q).cost) \end{aligned} \quad (2)$$

The algorithm UoverI_cnf() evaluates the above equation by letting f span over all terms c_i and all common factors of more than one term. When f is one of the terms c_i, the Q set is empty. This corresponds to using c_i for generating the best UoverI_dnf(f) plan of Section 3.2.1 and evaluating the rest of the terms as a filter. In general, f would span more than one term.

For example, consider an input CNF expression E: $(A + B)(A+C+G)(C+G+D)$. The set of factors f the algorithm will consider are the original terms $(A + B), (A + C + G)$ and $(C + G + D)$, along with the common factors A and $(C + G)$. If $(C + G)$ turns out to be the winning factor, the expression $(C+G+AD)$ would be used to generate a union-over-intersection plan and the conjunct $A + B$ evaluated as a filter.

3.2.3 Input: General AND/OR expression

When the input is a general AND/OR expression and the output is still restricted to be intersection followed by union, we traverse the tree bottom-up and construct the solution incrementally. Each AND/OR node takes as input the best UoverI plan of each of its children and combines them to output another UoverI plan. In each UoverI plan, the predicates used in index intersection and unions can be directly translated to a DNF expression.

For example, consider the expression $(ABC + ABD + CD)((C + F)(D + F) + I(G + H))$. At the lowest level of the tree, we first apply the UoverI_cnf() procedure on the CNFs inside the second term, to obtain DNF expressions corresponding to their UoverI plans. The first CNF $(C + F)(D + F)$ may be converted into $CD + F$, while $I(G + H)$ may be evaluated by a single index seek for I, meaning that the corresponding DNF is simply I. Now, we consider the expression to be $(ABC + ABD + CD)(CD + F + I)$, and apply UoverI_dnf() on each of the two conjunct terms. The first term may be rewritten as $AB(C + D) + CD$, which may lead to a UoverI plan where C and D are intersected, followed by union with A and a filter. The corresponding DNF for this plan is $A + CD$. The second term $CD + F + I$ might have a UoverI plan directly corresponding to it, and thus remains unchanged. Finally, we apply UoverI_cnf() to the resulting expression $(A + CD)(CD + F + I)$ to get our final UoverI plan. This plan might just be to work with the first term, that is, intersect C and D, union with A, and follow it up with a data lookup with a filtered evaluation of the original condition. The algorithm is formally stated below.

UoverI(E):

If E a pure DNF, return UoverI_dnf(E)

If E a pure CNF, return UoverI_cnf(E)

for each child of c_i of the root node of E **do**

p_i = the terms appearing as index union/intersection in UoverI(c_i) expressed as a DNF.

end for

If root node of E is AND, return UoverI_cnf($p_1.p_2 \ldots p_n$)

If root node of E is OR, return UoverI_dnf($p_1+p_2+\ldots p_n$)

3.3 Arbitrary IIU Plans

When the output is allowed to be any arbitrary nesting of index union and intersection nodes, the search space becomes much larger and it gets harder to produce efficient algorithms that provide guarantees as to the output plan quality. We sketch how our algorithms can be modified to output general IIU plans when the input is either a pure DNF or CNF. We do not discuss the case of general AND/OR inputs in this paper.

3.3.1 Input: DNF

We modify UoverI_dnf to IIU_dnf by invoking UoverI_dnf recursively on the quotient Q instead of just on R when exploring rewrites of the form $E = fQ + R$. Thus, the modified equation is:

$$\begin{aligned} \text{IIU\_dnf}(E).cost &= \min_{E=f.Q+R}(\text{AccessPath}(f).cost \\ &+ \text{IIU\_dnf}(Q).cost + \text{IIU\_dnf}(R).cost) \end{aligned}$$

Procedure IIU_dnf(Q) could return a plan involving one or more index seeks whose results could be intersected with the index seek on f. Figure 3(c) shows how the IIU_dnf(E) procedure could have modified the plan produced by UoverI_dnf(E) so that $B + D$ is evaluated as a index union and then intersected with AC instead of being evaluated as a filter.

3.3.2 Input: CNF

We modify UoverI_cnf to IIU_cnf by allowing recursive invocation of IIU_cnf on R when exploring rewrites of the form $E = (f+Q)R$. Thus, an index-intersection node could be added over the IIU plan produced on each of the two conjuncts.

$$\begin{aligned} \text{IIU\_cnf}(E).cost &= \min_{E=(f+Q)R}(\text{UoverI\_dnf}(f).cost \\ &+ \text{IIU\_cnf}(Q).cost + \text{IIU\_cnf}(R).cost) \end{aligned}$$

4. APPROXIMATE FACTORING

In this section, we show how to expand our search space to generate better IIU plans by using relaxations of the original condition when factoring. Say we have an expression E of the form $XA+YB$. Then the expression $E' = (X+Y)(A+B)$ is a relaxation of the original expression E. We might prefer E' to E if predicates X and Y overlap and enable more efficient evaluation of $X+Y$ than evaluating X and Y individually. The factored predicate $E' = (X+Y)(A+B)$ can be used to find a relaxed set of RIDs, which can then be filtered by an exact evaluation of E. We have already seen in Section 3 that there is a potential advantage if X and Y are identical. In this section, we explore the case where X and Y are not identical.

We limit the space of possible merges to consider by focussing only on those predicates which, when merged, will yield another atomic predicate that can be evaluated by a single index scan. Examples of such predicates are multi-dimensional range predicates. We revisit our example from Section 1, comparing the ease of evaluation of E_1: *(x BETWEEN 1 AND 5)A + (x BETWEEN 2 AND 6)B*, as compared to E_2: *(x BETWEEN 1 AND 6)(A+B)*, obtained by approximate factorization of E_1. Here, we have relaxed both predicates involving x to obtain a single common range predicate on x. We can identify which of E_1 and E_2 is better by using our cost functions.

In a sense, there is nothing more to be done in order to incorporate condition relaxation into our algorithms for identifying No-Union plans, Single-Union plans or general IIU plans. For identifying No-Union plans, we now also consider a relaxed predicate on a column as a possible conjunctive factor, if such a predicate exists. The dynamic programming and greedy algorithms for identifying Single-Union plans can be similarly modified to consider all possible merged predicates, but unfortunately, the number of different ways in which we could choose a set of predicates to merge could be exponential.

We therefore try to solve the problem of *selecting* the set of predicates to merge so as to maximize the gain from factorizing the expression. The space of possibilities we need to consider has 2 degrees of freedom: the attribute whose predicates we should relax, and the set of predicates to relax for that attribute. The first degree of freedom can be handled by exhaustive linear search—we identify the best solution for each attribute and pick the best among these. The second degree of freedom needs to be handled with more care, since there are an exponential number of possibilities.

There is no alternative to exhaustive search if we were to treat our cost functions as a black box. However, for many cost functions that we actually encounter in this situation, we can use an efficient polynomial-time algorithm to identify the exact predicates to relax, which is guaranteed to be optimal. The exact characterization of the cost functions for which the algorithm is optimal is somewhat complex and we omit the characterization in this paper. Instead, we simply note that for complex cost functions, this algorithm can still be used as a heuristic means of identifying the best merged predicate. We sketch the algorithm for dealing with DNF input, while noting that CNF input can be handled analogously, albeit with different cost functions.

For ease of explanation, we demonstrate the algorithm and prove its optimality with respect to a simple cost function. The actual cost functions we used were a more complex variant for which the algorithm continues to remain optimal.

4.1 The Bene£t from merging predicates

Let $a_1, a_2, ...a_n$ be atomic predicates along the same index whose merge is the atomic predicate a. We want to estimate the benefit from factorizing the expression $a_1x_1 + a_2x_2 + ...a_nx_n$ into $a(x_1+x_2+...x_n)$. We should bear in mind that x_1,x_2,...x_n could themselves be conjunctions of literals. The benefit from factorizing the expression, when considering index-only plans, can then be estimated by the formula:

$$G(a) = (k_1+1)(\sum_{i=1}^{n} l_{a_i} - l_a) - \sum_{i=1}^{n}(l_{x_i} - l_{a_i x_i}) \qquad (3)$$

In this formula, l_c refers to the number of tuples satisfying condition c, and k_1 is a scale factor between the cost of RID intersection, and the cost of accessing tuples off the disk. However, if all of x_1,x_2,.....x_n are just the constant $True$, then there is no index intersection required, and the gain is equal to just the first term in Equation 3. We see that the benefit function is just a linear function of each of the terms being merged, together with one component dependent on the number of RIDs corresponding to the merged predicate.

4.2 Choosing predicates to merge

The crux of the algorithm lies in its ability to compute the set of predicates to be merged in an incremental fashion, while also maintaining an associated benefit value G that measures the utility of the merging. Additionally, G can itself be computed incrementally, further improving the efficiency of the algorithm.

Algorithm 1 SelectBestMergedFactor(E, A)

```
1: BestCost=∞; FactoredForm=E;
2: for all indexed columns c in E do
3:    Let S= Set of all atomic ranges on Column c in E;
4:    while (true) do
5:       Select 2 overlapping predicates a1 and a2 from S,
         whose merge is a, such that G(a) > G(a1)+G(a2);
6:       If no such a1 and a2 exist, break;
7:       Remove a1 and a2 from S;
8:       Add a to S;
9:    end while
10:   a'=Predicate in S such that ∀p ∈ S G(a') ≥ G(p)
11:   Let E' be the factorized form of E using a' as a factor.
12:   CurrentPlanCost=AccessPath(E', A).cost
13:   if CurrentPlanCost < BestCost then
14:      BestCost=CurrentPlanCost;
15:      FactoredForm=E';
16:   end if
17: end for
18: return FactoredForm;
```

The algorithm starts out with the set of all atomic predicates on the same attributes. If there are two overlapping

predicates in the set a_1 and a_2, whose merge is another atomic predicate a, such that $G(a) > G(a_1) + G(a_2)$, then we delete a_1 and a_2 from the set and insert a instead. We continue this process while there are some two predicates that can be merged in this fashion. When there are no more predicates to merge, we simply select the best predicate a' in the set and return it. We also maintain additional information to identify the exact set of predicates that were merged to generate a'. We use "semi-naive" evaluation to perform merges efficiently. The complexity of identifying overlapping predicates is dependent on the form of overlap that we are looking for, but can be performed efficiently in many cases. We prove in the appendix that this algorithm provides the optimal solution to the problem of selecting the best merged predicate.

5. EXPERIMENTAL EVALUATION

We address the following three questions in our evaluation of the proposed factoring algorithms:

1. What is the performance improvement achieved by the algorithm in its most powerful form, for different query workloads?

2. What is the marginal utility obtained by progressively more complex variants of the algorithm? Specifically, what is the utility of doing approximate factoring instead of exact factoring, and the utility of general IIU plans as opposed to index-intersection plans?

3. What is the loss in quality due to our design decisions, specifically, the use of a greedy algorithm for factoring, and the use of in-place factoring instead of converting to DNF?

We address each of these questions using three different sets of workloads: data-mining queries, application-generated queries, and a synthetic workload on the TPC-H database. We first describe the experimental framework necessary for evaluating our algorithms, and then proceed to show results on each of the workload sets.

5.1 Experimental Framework

We use Microsoft SQL Server 2000 to perform our experiments. While we have proposed implementing a solution by modifying code of the optimizer, we use a simpler approach in order to evaluate our algorithms experimentally. We implement our algorithms as a stand-alone module that takes SQL queries as input. The module interacts with the optimizer in order to receive cost estimates and selectivity information. Using this information, our module identifies what it believes to be the optimal plan, and produces a new SQL expression for the query that is guaranteed to execute using the chosen plan. This new SQL expression is then fed to the database system through the normal channels for execution.

We identify the set of indexes that ought to be built on the tables involved using the index recommendation utility of the database system. We use the factored versions of the queries as input to determine the best indexes for the factored workload, and the original queries for the unfactored case. This structure is not always satisfactory, since the index recommender does not recommend the right set of indexes to enable efficient IIU plans. In such cases, we manually modify the generated recommendations to ensure efficient plans for the factored case. The original queries are also run against the database instance with the modified set of indexes, and we report the better of the two results for the original queries.

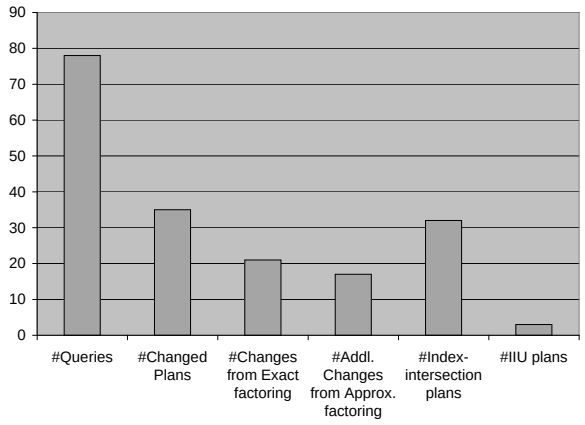

Figure 4: Data-Mining Queries: Number of Plans Modified

5.2 Data-Mining Queries

The embedding of mining models in relational databases often leads to syntactically complex queries involving disjunctions. For our experiments, we used workloads obtained from the authors of [6]. These workloads were generated by building various mining models on different databases in the commonly-used UCI Machine Learning [2] and the UCI KDD Repository [1]. The queries involved predicates on predicted attributes that were then translated into traditional relational queries on the original attributes of the database. We chose fifteen such workloads, all of which contained at least one query with disjunctions, and evaluated the utility of our factoring algorithm in its various avatars on each of these workloads. Queries in these workloads were mostly in DNF with the number of terms varying quite widely from zero (conjunctive queries) to more than a hundred, with most queries having less than 5 terms. Each table involved in the queries consisted of approximately one million rows, while the number and size of the attributes were different for each table. More details about how the queries were generated can be found in [6].

For each of these workloads, we compute the benefits of exact and approximate factoring in the following fashion. We first identify the improvement (if any) in running time obtained from performing exact factoring on the queries. We then enable both exact and approximate factoring and identify the incremental improvement, if any, from this configuration. The results of our experiments are summarized in Figures 4 and 5. Figure 4 shows the number of queries that benefit from factoring algorithms of different complexities. The greedy and dynamic-programming approaches to factorization produce exactly the same results, and we do not distinguish between them here. Nearly as many queries benefit from approximate factoring as they do from exact factoring. We also observe that only a small percentage of the queries benefit from IIU plans (both UoverI and arbitrary IIU plans). This observation can be explained by the nature of the workloads. All the queries in the workload

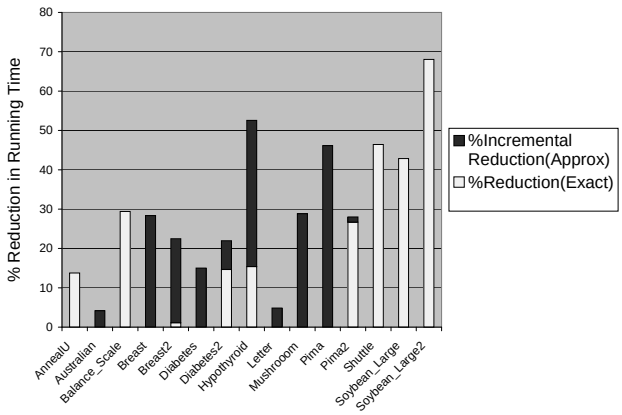

Figure 5: Data-Mining Queries: Reduction in Running Time

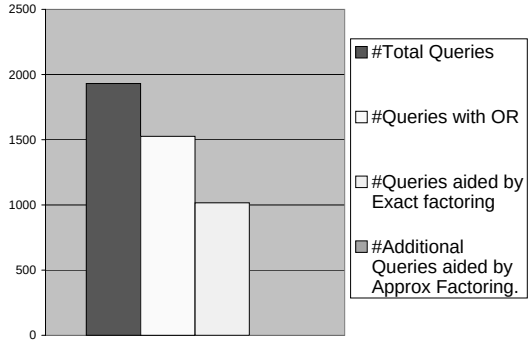

Figure 6: Application-Generated Queries: Number of Plans Modified

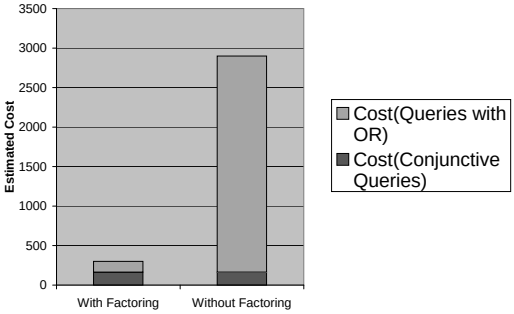

Figure 7: Percentage Reduction in Running Time from Exact and Approx. Factoring

are *SELECT ** queries which means that index-only plans are not possible unless we have indexes covering the whole table. Additionally, most of the queries select a considerable number of rows from the table, rendering the option of a data lookup after index-union somewhat expensive. So, only the queries with low values of selectivity end up being executed via IIU plans. Figure 5 shows the impact of exact and approximate factoring on the running time of all the queries in each of the fifteen workloads. We report the reduction in running time as a percentage of the running time of the workloads without factoring ; thus a reduction of 50% implies that the running time is halved. We see in Figure 5 that five workloads benefit solely from exact factoring and six, solely from approximate factoring. The remaining four workloads benefit from both, with the individual contributions of exact and approximate factoring varying across the workloads.

Overall, we notice that there is considerable benefit to be gained from factoring, with five of the workloads showing a reduction in running time of more than 40%. Note that some of the queries in the workloads are conjunctive and do not benefit from factoring. We also note that most of the queries in this workload are fairly unselective and do not permit index-only access. We would thus expect more selective variants of these queries, or variants involving aggregation, to show much more improvement in running times.

5.3 Application-Generated Queries

Queries that are automatically generated by applications tend to contain a lot of redundancy in them, in the sense that the queries can often be equivalently represented in a more compact form. One common source of such redundancy is the normalization to CNF or DNF that queries are often subjected to by the applications. We have already seen that neither CNF nor DNF are ideal representations for all queries. We used a short trace of queries on a real database in order to evaluate the utility of factoring on such queries. The database consisted of about 900 tables, with a total of 600MB of data. Our trace consisted of 1931 queries, which were predominantly pure selections, with a few aggregation and join queries. Most of the non-aggregation queries were highly selective, retrieving tens or hundreds of rows as output.

Figures 6 and 7 depict the results obtained on this workload. Both the greedy and dynamic-programming approaches to factoring produced exactly the same results on this workload too, and we therefore do not distinguish between the two here. We observe that nearly 80% of the queries contain a disjunction and two-thirds of these queries benefit from exact factoring. As hypothesized, nearly all the benefit obtained from factoring is due to the normalization of the query expressions, and there is no additional benefit obtained from approximate factoring. In fact, all the queries, once factorized, turn out to be in "near-conjunctive" form, that is, they can be represented as a conjunction of a lot of atomic predicates, together with one complex AND-OR expression. This abundance of conjunctive factors leads to approximate factoring not being useful for this workload. Another consequence of the near-conjunctiveness of the queries is that single-index and index-intersection plans are preferred to index-union plans. Thus, when trying to generate IIU plans, all the benefit of factorization comes from Algorithm UoverI_dnfand none from combining two terms in CNF using Algorithm UoverI_cnf.

Figure 7 shows the estimated costs of the factored and unfactored versions of the workload. The cost of the conjunctive queries in the workload remains the same in both cases. We see that the cost of the queries with disjunction is drastically lower when we use (exact) factoring. The overall cost for the factored workload is nearly 10 times lower than the cost for the unfactored workload. The improvement is largely due to the use of single-index and index-intersection plans when using factoring, as opposed to table scans for the original unfactored expression.

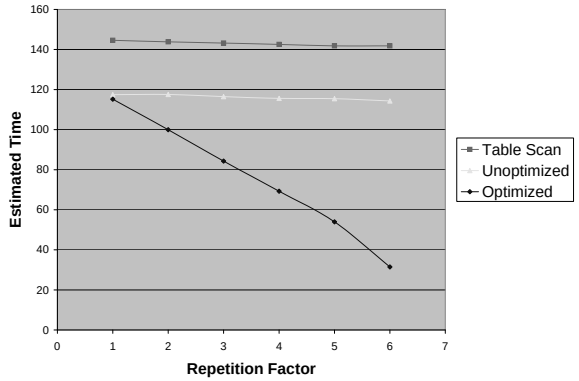

Figure 8: Sensitivity to number of repetitions

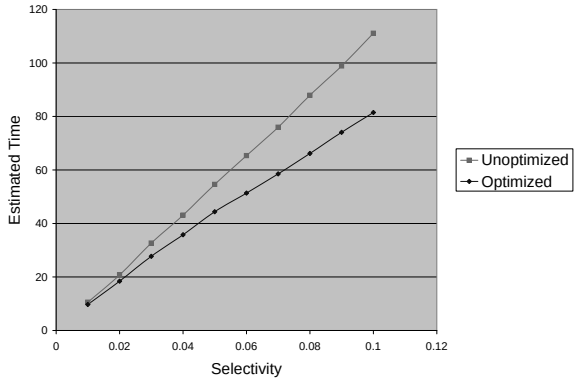

Figure 9: Scaling selectivities together, with an overlap factor

5.4 Synthetic Workloads

We used synthetically-generated workloads to understand the impact of various query characteristics on the utility of our factoring algorithms. Since we have already established the utility of factoring when it leads to index-intersection plans, our experiments in this section attempt to study queries that lend themselves to index-only plans involving both index-intersection and union. Our experiments were carried out on a 1GB TPCH database(with skew 1). Most of our queries were on the *lineitem* table, while some involved foreign-key joins of the lineitem table with other tables. Besides the primary index on the lineitem table, there were secondary indexes covering all the keys of the different tables, and covering all the columns involved in the generated queries. In order to generate overlapping predicates, we used range predicates on different fields of the table.

The following are the important query characteristics that were controlled:

1. Query Structure: We generated WHERE clauses in DNF. So, query structure was determined by the total number of terms generated, and the distribution of the number of literals per term.

2. Predicate Selectivities: The selectivities of the predicates in the query were chosen either from a Gaussian distribution, or from a uniform distribution with a certain peak value. The selectivities of predicates in the generated queries were controlled using the statistics maintained by the database system.

3. Repeating Predicate: In many of our experiments, only one predicate was made to repeat (either exactly or approximately) in order to isolate the effect of a single factorization. We had control over both the selectivity of the repeating predicate and the number of repetitions.

4. The Overlap Factor: This factor controlled the degree of overlap for the generation of overlapping predicates. If this factor is one, the overlapping predicates are identical. If the factor is zero, the overlap between the predicates is zero. Thus, the value of the overlap factor controls how much the overlapping factors intersect.

For each choice of these query-characteristic parameters, we generated 200 queries conforming to these parameters and present the average results over these 200 queries as output. We now demonstrate improvement in both estimated and actual execution times as a function of the various query characteristics.

5.4.1 Sensitivity to number of repetitions

In this experiment, we fix the query structure to be in disjunctive normal form consisting of 6 terms, with each term being covered by two indexes. Thus, the plan for the unfactored form would consist of six sets of index intersections, followed by an index union step. There is no data lookup. The selectivities of all the predicates are drawn from a Gaussian distribution with mean 0.1 and standard deviation 0.05. We vary the number of repetitions of predicates across terms from zero (no factoring opportunity) to six (a common factor across all the terms). Figure 8 shows the sensitivity to the number of repetitions. As we can see, there is an almost super-linear improvement in performance with an increase in the repetition factor.

5.4.2 Scaling Selectivities

For the experiment depicted in Figure 9, we use the same query structure as described earlier, but we freeze the number of repeating predicates at just one, while using an overlap factor of 0.5. The objective is to measure the utility of a single, approximately-recurring predicate as a function of the mean selectivity. Note that the individual predicate selectivities, along with the selectivity of the merged predicate, continue to be drawn from a Gaussian distribution, with the standard deviation being set to half the mean selectivity. We notice a consistent performance improvement through approximate factoring, reaching about 30% at selectivity 0.1.

5.4.3 Using a Uniform Distribution

Figure 10 shows the actual execution times when we draw selectivities from a uniform distribution with a peak value that is varied from 0.02 to 0.2. Again, there is exactly one repeating predicate whose selectivity is chosen to be half the peak value of the uniform distribution. Notice that this graph plots actual execution times rather than optimizer estimates, showing that benefits of factoring are visible both to the optimizer and in reality.

Figure 11 repeats the same experiment, but this time there is no exact repetition of a predicate. Instead, one predicate is duplicated with an overlap factor of 0.5. The

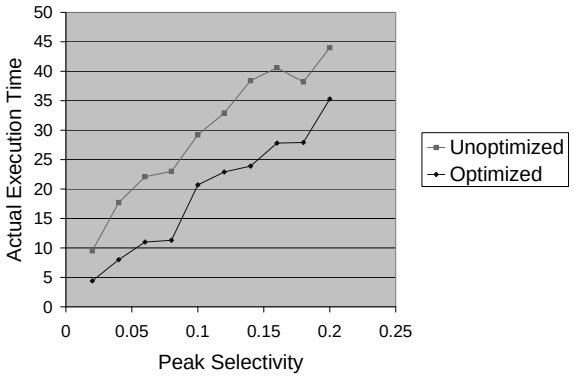

Figure 10: Uniform Distribution-Actual Times

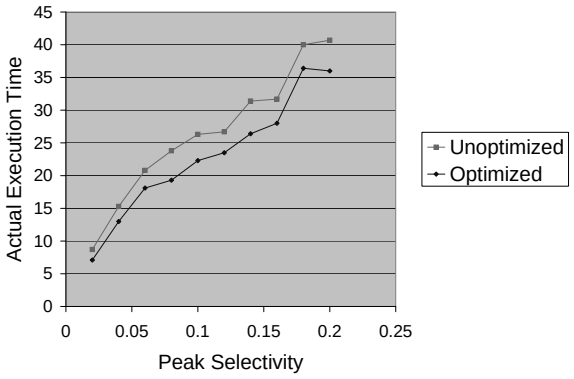

Figure 11: Uniform Distribution with an overlap factor-Actual Times

selectivity of the merged predicate is again chosen to be half the peak of the uniform distribution. We see that the performance improvement is not as considerable as in the earlier case but the optimized version is still clearly superior to the unoptimized version.

5.5 Summary of experimental results

We have evaluated the utility of factorization on two different sets of real workloads. We found that factorization was useful, both with expensive data-mining queries providing lots of scope for approximate factorization, and with highly-selective application-generated queries. Many of the application-generated queries were not in DNF, providing evidence of the viability of the algorithms dealing with arbitrary predicate expressions. We have also studied our algorithms with synthetic data and queries, and show the performance improvement one may expect for various query characteristics. Although we have not provided explicit measurements of compilation time, we believe that our algorithms are practical since the time taken to parse the queries completely dominated the actual running time of our algorithms.

6. RELATED WORK

To the best of our knowledge ours is the first paper to propose exact and approximate factorization for optimizing indexed access to a relation in a database system. However, the minimization of Boolean expressions through factorization has been used in several areas of computer science other than databases, including compiler optimization [20] and most notably in VLSI circuit design for reducing floor area of circuits. Factoring Boolean formulae to minimize the total number of literals is an important problem in VLSI design because the area taken up by a circuit for a Boolean formula is roughly proportional to the number of literals in the formula. This problem is NP-hard. However, the practical relevance of the logic minimization problem in VLSI has led to the design of several algorithms with various levels of complexity. Among factoring methods, *algebraic factoring* is the most popular since it provides very good results while being extremely fast [4, 3]. Our problem is somewhat different from the VLSI logic minimization problem, because every literal of our Boolean formula is associated with a different cost that depends on its selectivity and index. Also unlike in VLSI, the size of intermediate results is important for our problem since it affects index-intersection and index-union costs. Furthermore, we require more efficient algorithms because of the compile-time constraints, and because we cannot afford to convert expressions to DNF as most VLSI algorithms do.

We next discuss related work in the database literature on optimizing queries with complex Boolean expressions. Early systems [21] rewrote the expression as a CNF and exploited only one index per expression. Others rewrote the expressions as a DNF and union-ed each disjunct evaluated independently [9, 8]. These simple approaches were augmented in later systems [17] to exploit multiple indexes by evaluated them as arbitrary IIU plans. [17] discusses how to choose the best subset of eligible indexes and sequence the RID mergings for best performance. However, they operate on the condition as directly expressed in the query and do not explore the space of alternative rewrites like we have. [15] shows how to efficiently access multi-dimensional B+-trees using partial key information, when some fields of the key are either not present in the query or have IN clauses associated with them. This is one of the few papers to exploit the interaction between the syntactic structure of the filter expression and its evaluation via an access method in the context of their proposed technique on OR optimization. However, unlike us, they rely on a transformation to a DNF-like normal form. Thus, their technique cannot leverage the gains of factorization that we achieve when combining multiple indexes. Some of the other techniques suggested in the paper such as detecting redundant predicates is complimentary to our approach.

Optimization of user-defined predicates with varying cost of evaluation and selectivity [12, 7, 22, 13, 5] has been studied extensively. Most of the above work has been on expressions with ANDs only with some exceptions[13, 5]. These techniques attempt to reduce the total cost of invocations of the user-defined predicates by determining an order for predicate evaluation that takes into account relative cost per tuple and selectivity of predicates. Mostly, the goal has been to minimize CPU cost instead of index access or I/O cost. Also, none of the above work exploit factorization as a technique. In contrast, [14] does consider factorization for optimizing Boolean expressions in object-oriented databases. However, it does so with the assumption that the input is a DNF and with the goal of minimizing CPU cost instead of index access cost for IIU plans. Another area of database query optimization where factorization has been exploited is for reducing the number of scans in multiple-relation multiple-disjunct join queries [18]. However, this

work assumes that no relation has an index and focuses on minimizing the number of joins that need to be performed.

7. CONCLUSIONS

In this paper we addressed the problem of optimizing queries with complex selection conditions using factorization. We proposed two novel ideas for transforming such expressions to simpler forms — exact and approximate factoring. Our algorithms were staged to produce plans at multiple complexity levels, enabling optimization for the common case by ensuring low overhead for producing simple plans that suffice for most queries. Our experiments on a commercial database system on two classes of real workloads has been encouraging. We have studied the utility of factorization as a function of different query characteristics using synthetic workloads. We outlined an approach for integrating the factorization routines with existing query optimizers' support for selecting indexes.

8. REFERENCES

[1] S. D. Bay. The UCI KDD archive [http://kdd.ics.uci.edu]. Irvine, CA: University of California, Department of Information and Computer Science, 1999.

[2] C. Blake and C. Merz. UCI repository of machine learning databases, 1998.

[3] R. K. Brayton. Factoring logic functions. *IBM Journal of Research and Development*, 31(2):187–198, 1987.

[4] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on CAD/ICAS, CAD-6*, 1987.

[5] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In H. V. Jagadish and I. S. Mumick, editors, *Proc. ACM SIGMOD 1996, Montreal*, pages 91–102, 1996.

[6] S. Chaudhuri, V. Narasayya, and S. Sarawagi. Efficient evaluation of queries with mining predicates. In *Proc. of the 18th Int'l Conference on Data Engineering (ICDE)*, San Jose, USA, April 2002.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates (a shorter version appears in vldb 1996). *TODS*, 24(2):177–228, 1999.

[8] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. SIGMOD 1992, San Diego, California*, pages 383–392, 1992.

[9] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of 13th International Conference on Very Large Data Bases, Brighton, England*, pages 197–208, 1987.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, Jun 1993.

[11] M. Z. Hanani. An optimal evaluation of Boolean expressions in an online query system. *CACM*, 20(5):344–347, 1977.

[12] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD Conference*, pages 267–276, 1993.

[13] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In R. T. Snodgrass and M. Winslett, editors, *Proc. SIGMOD 1994, Minneapolis*, pages 336–347, 1994.

[14] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing Boolean expressions in object-bases. In *Proc. of the VLDB Conference*, pages 79–90, Vancouver, Canada, August 1992.

[15] H. Leslie, R. Jain, D. Birdsall, and H. Yaghmai. Efficient search of multi-dimensional "b"-trees. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 710–719. Morgan Kaufmann, 1995.

[16] M.C. Golumbic and A. Mintz. Factoring logic functions using graph partitioning. In *Proc. IEEE/ACM Int'l. Conf. on Computer-Aided Design, (ICCAD-99), San Jose, CA*, pages 195–198, 1999.

[17] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single table access using multiple indexes: optimization, execution, and concurrency control techniques. In *Proc. International Conference on Extending Database Technology*, pages 29–43, 1990.

[18] M. Muralikrishna and D. J. DeWitt. Optimization of multiple-relation multiple-disjunct queries. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas*, pages 263–275, 1988.

[19] R. Brayton, G. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78:264–300, 1990.

[20] L. T. Reinwald and R. M. Soland. Conversion of limited-entry decision tables to optimal computer programs : Minimum average processing time. *JACM*, 13(3):339–358, 1966.

[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD International Conference on Management of Data, Boston*, pages 23–34, 1979.

[22] G. M. Wolfgang Scheufele. Efficient dynamic programming algorithms for ordering expensive joins and selections. In *Proc. of the 6th Int'l Conference on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.

APPENDIX

A. OPTIMALITY OF APPROXIMATE FACTORING ALGORITHM

We now prove the optimality of Algorithm 1. The algorithm is optimal so long as overlap of predicates is commutative and associative. For example, single-dimensional range predicates satisfy these properties. For simplicity, we will assume that the predicates to be merged are single-dimensional range predicates here on in, with the understanding that the same proof holds for other commutative and associative forms of overlap.

We first define an incrementally computed function F from which the benefit G can also be computed. For any

atomic predicate R occurring in the original expression as a conjunction Rx:

$$F(\{R\}) = -l_x + l_{Rx}$$

For any two overlapping predicates R_1 and R_2, with corresponding sets of predicates S_1 and S_2, whose merge is R:

$$F(S_1 \cup S_2) = F(S_1) + F(S_2) + (k_1 + 1)l_{R_1 \cap R_2}$$

The above formula also holds when R_1 and R_2 do not intersect. There is a special case where the cost functions change because there is no index-intersection required. This special case is easily handled though. When merging such a set of predicates a_i with the corresponding x_i all being $True$, $F(a_i)$ is simply defined to be zero.

We also define, for any set of predicates S, $G(S) = \max(F(S), 0)$.

At the end of Algorithm 1, we have a partition of all the range predicates on a particular column. Let us call the sets in this partition $S_1, S_2, S_3, ...$ Without loss of generality, let $\forall i G(S_1) >= G(S_i)$. The algorithm then produces S_1 as its output. In general, a "solution" is just a set of range predicates, all of which are to be merged together. We will find it useful to define benefit G for unions of disjoint solutions as the sum of the benefits of the individual solutions. We can unambiguously represent the union of multiple disjoint solutions by a single set, because there is only one way of partitioning a set into disjoint solutions.

LEMMA 1. *If S is the solution generated as the output of Algorithm 1, $G(S) > G(S')$ for any S' that is a subset of S.*

PROOF. We prove this lemma by induction on the number of merges performed to generate the final, merged range corresponding to S. Call this final range R. First, if R is obtained by merging two atomic predicates R_1 and R_2 from the original expression, we know, by the conditions of the algorithm, that:

$$G(R) > G(R_1) + G(R_2)$$

In this case, $S = \{R_1, R_2\}$ and, therefore, the gain corresponding to any subset of S is clearly less than $G(R)$.

Now, let R be obtained by merging an atomic range R_1 with a merged range R_2. If S' contains the atomic range R_1, we know that $S' - \{R_1\} \subset S - \{R_1\}$ and, by induction, $G(S - \{R_1\}) > G(S' - \{R_1\})$ and, consequently, the same inequality holds when we replace G by F. We know that

$$\begin{aligned} G(S) &= F(S - \{R_1\}) + F(R_1) + (k_1 + 1)l_{R_1 \cap R_2} \\ G(S) &> F(S' - \{R_1\}) + F(R_1) + (k_1 + 1)l_{R_1 \cap R_2} \\ G(S) &> G(S') \end{aligned}$$

If S' does not contain R_1, then $S' \subseteq S - \{R_1\}$ and, therefore, $G(S - \{R_1\}) \geq G(S')$. Also, since Algorithm 1 chose to merge R_1 and R_2, we know that

$$\begin{aligned} G(S) &> G(S - \{R_1\}) + G(R_1) \\ G(S) &> G(S') + G(R_1) \\ G(S) &> G(S') \end{aligned}$$

We have now proved that if R is obtained by merging an atomic predicate R_1 from the original expression, with a merged predicate R_2, $G(S)$ is greater than G for any subset of S.

Now, let R be obtained by merging two merged predicates R_1 and R_2, with corresponding sets of predicates S_1 and S_2. Let $S_1' = S_1 \cap S'$ and $S_2' = S_2 \cap S'$. Without loss of generality, assume that $S_1' \subset S_1$. Then, $S_2' \subseteq S_2$. We know that

$$F(S) = F(S_1) + F(S_2) + (k_1 + 1)l_{R_1 \cap R_2}$$
$$F(S) > F(S_1') + F(S_2') + (k_1 + 1)l_{R_1 \cap R_2}$$
$$\text{(by induction assumption)}$$
$$F(S) > F(S')$$

This concludes the proof. □

LEMMA 2. *If S is the output generated by Algorithm 1, and S' is the optimal solution, every pair of predicates in S' are merged together into the same partition at some point in the execution of Algorithm 1 and are consequently in S.*

PROOF. We prove this statement by induction on the number of predicates in S'. As the base case, let the number of predicates in S' be 2, and call the predicates R_1 and R_2. We prove by contradiction. If R_1 and R_2 are not in the same partition, by the algorithm's decision not to merge R_1 and R_2, we know that $G(S') <= G(R_1) + G(R_2) = 0$. Therefore, solution S' is sub-optimal compared to S.

Now, let Algorithm 1 produce partitions $S_1, S_2, \ldots$ with $S = S_1$. If, for all intersecting predicates R_1 and R_2 in S', R_1 and R_2 both belong to the same partition, we can see that S' is, itself, equal to S_i for some i. (Since, according to our cost function, it is never profitable to merge two predicates that don't have an intersection, the "intersection graph" of the predicates in S' needs to be connected.) If S' is equal to S_i, then, by the algorithm, $G(S_1) \geq G(S_i)$ and, therefore, $G(S) \geq G(S')$.

Therefore, let there be some two intersecting predicates R_1 and R_2 in S' which belong to different partitions generated by the algorithm, say S_i and S_j, with corresponding merged predicates R_i and R_j. If $|S_i| > 1$ and $|S_j| > 1$, then $F(S_i \cup S_j) = F(S_i) + F(S_j) + (k_1 + 1)l_{R_i \cap R_j}$. We know that $F(S_i) = G(S_i)$ and $F(S_j) = G(S_j)$, and $l_{R_i \cap R_j}$ is non-zero, since $R_1 \cap R_2$ is non-empty. Therefore, $G(S_i \cup \S_j) > G(S_i) + G(S_j)$, which implies that the algorithm would have merged S_i and S_j.

Therefore, at least one of S_i and S_j has cardinality 1. Without loss of generality let S_i be a singleton with the atomic range R_i. By the induction assumption, S_j is exactly equal to $S' - \{R_i\}$. Since the algorithm did not merge S_i and S_j, we know that

$$\begin{aligned} G(S') &\leq G(S_i) + G(S_j) \\ &= G(S_j) \\ G(S') &\leq G(S) \end{aligned}$$

This completes the induction step. □

THEOREM 3. *Algorithm 1 generates the optimal set of predicates to be merged.*

PROOF. We observe that $G(S)$ accurately captures the benefit of approximating all the factors in the set S. Together with this observation, Lemmas 1 and 2 directly imply that Algorithm 1 produces the optimal solution. □