Toward Practical Constraint Databases

Alexander Brodsky Joxan Jaffar Michael J. Maher

I.B.M. Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

Linear constraint databases (LCDBs) extend relational databases to include linear arithmetic constraints in both relations and queries. A LCDB can also be viewed as a powerful extension of linear programming (LP) where the system of constraints is generalized to a database containing constraints and the objective function is generalized to a relational query containing constraints. Our major concern is query optimization in LCDBs. Traditional database approaches are not adequate for combination with LP technology. Instead, we propose a new query optimization approach, based on statistical estimations and iterated trials of potentially better evaluation plans. The resulting algorithms are not only effective on LCDBs, but also on existing query languages.

1 Introduction

Linear programming/linear constraints is a technology widely used in applications of economics and business, e.g. allocation of scarce resources, scheduling production and inventory, cutting stock and many others. This paper proposes a merger of linear programming (LP) and relational database technologies in the framework of *linear constraints databases* (LCDBs), that extend relational databases to include linear arithmetic constraints in both relations and queries. The motivation comes from the fact that classical LP applications do not stand alone, but rather operate over a large amount of stored data and usually require not just to optimize one objective function, but to answer more complex queries involving manipulation of both regular data and constraints. A second application realm is that of engineering design systems, which may operate over large catalogs of components, devices etc., and enable queries about design patterns that are described using constraints. LCDB technology, in particular, is important because it can enhance many existing software platforms such as relational DBMS, object oriented DBMS, operation research packages, constraint logic programming (CLP) systems etc.

Traditionally, there have been two major approaches to query optimization. One is based on compile time algebraic simplification of a query using heuristics as in [12, 30, 33, 38, 32, 43, 6, 44, 2]. The other is based on cost estimation of different strategies as in [1, 10, 7, 8, 29, 42]. The heuristics of the algebraic simplification approach, such as performing selections as early as possible, assume that the selection conditions are readily available. In fact, extracting such conditions from the constraints of a query involves linear programming techniques which are, in general, expensive because there may be, for example, thousands of variables. The cost estimation approach, on the other hand, has the similar problem of extracting explicit constraints on attributes which are needed for the estimation. Even if these constraints were readily available, there is a second problem: it is typically necessary to make assumptions about the distribution of data (like uniformity within, and independence of, attributes), and these appear unlikely to hold in LCDBs. In short, traditional optimization approaches are inadequate for LCDBs.

In this paper, we propose a new generic approach to query optimization, that is not only effective on LCDB, but also on existing query languages. The underlying philosophy is that expenditure of computational cost is necessary in order to obtain information

Proceedings of the 19th. VLDB Conference, Dublin, Ireland, 1993.

required to estimate which is the best evaluation plan. We use statistical sampling for the cost estimation of specific plans, which has the advantage of avoiding dependence on data distribution. Since it is impractical to consider all possible plans in the search for the best one (because cost estimation of each plan might be expensive), trials of evaluation plans are performed, one at a time, "gambling" some work required for the cost estimation of the plan in an attempt to discover a better plan. We bound the amount we can gamble, based on the best estimated cost so far. The gambling algorithm is then used for optimization of generalized select-project-join queries involving up to two generalized relations. This requires to develop algorithms for estimating costs of possible evaluation plans, based on statistical methods. The problem of how to perform reasonably accurate and computationally cheap cost estimations for a more general class of queries requires more study. As additional contribution, adapting the algorithm of [9, 37] for n-dimensional rectangle intersection, we show how to perform an analog of the sort-join.

There has been work on specific uses of constraints in databases, the earlier of which includes [17, 13, 5, 35, 4]. The work [18] proposed a framework for integrating abstract constraints into database query languages by providing a number of design principles. They proved important properties on specific instances of the framework, but did not focus on optimization. The work [13] considered optimiziting in the context arithmetic equations. However, constraint solving was limited to local propagation and hence not suitable for LP problems. More recent work on deductive databases [31, 41, 19, 20, 26] concentrate on optimizing by repositioning constraints and assume the implementation of selection, projection and join and optimization of expressions involving these operators.

The remainder of the paper is organized as follows. Motivating examples and discussion are next, in Section 2, and the definitions of our data model and query language appear in Section 3. An important aspect of our work, which pertains to practical use, is the use of the notion of constraint canonical forms. Section 4 covers relevant computational issues in constraint manipulation, which are fundamental to constraint query evaluation. Section 5 discusses why traditional optimization methods are inadequate, elaborating on the discussion above. Section 6 and beyond form the core technical presentation: we deal first with the selection/projection queries in Section 6, which motivates our generic "gambling" algorithm presented in Section 7. Section 8 gives an algorithm for sort join on constraint attributes and Section 9 presents the application of the gambling algorithm to optimizing select-project-join queries.

2 Introductory Examples

Suppose a company manufactures two products using two resources. Its database has the relations *orders1* and *orders2* for orders of its first and second products respectively. Each relation has the attributes *Order#*, *Customer* and *Product_quantity*. Another relation *product_resource*($P1, P2, R1, R2$) specifies a relationship between quantities of resources and products: $P1$ and $P2$ represent quantities of the first and second products respectively to be produced, while $R1$ and $R2$ represent amounts of the first and second resources available. A possible manufacturing process can be specified by (a conjunction of) the following constraints:

$$\begin{aligned} P1 + 100\,P2 &\leq R1 \\ 100\,P1 + P2 &\leq R2 \\ P1, P2, R1, R2 &\geq 0 \end{aligned}$$

This says that the amount of the first resource needed to produce $P1$ and $P2$ units of the first and the second products must not exceed the amount $R1$ of this resource available. Similarly about the second resource. Suppose that there is another manufacturing processes:

$$\begin{aligned} 1.7\,P1 + 13.1\,P2 &\leq R1 \\ 28.3\,P1 + 11.5\,P2 &\leq R2 \\ P1, P2, R1, R2 &\geq 0 \end{aligned}$$

Now, the relation *product_resource* is a disjunction of two conjunctions of (three) constraints, a finite description of the infinite number of tuples ($P1, P2, R1, R2$) of values satisfying the disjunction. Similarly to [18] we define a ***constraint tuple*** to be a (possibly existentially quantified*) conjunction of constraints and ***constraint relation*** to be disjunction of constraint tuples.

In addition to regular relational queries, one may have queries like: "given that profit for one unit of the first product is \$15 and of the second is \$4, and that there are 100 and 10000 units of the first and second resources respectively, and 10000 units of the second at stock, what is the maximum profit the company can make with each manufacturing pattern?" or "given certain quantities of resources, what are the ranges of and the connection between the quantities of the two

*In [18], existential quantifiers are not allowed.

Order#	Customer	R_1 R_2
1	'Smith'	$R_1 \geq 262.31 \wedge R_2 \geq 4366.69$
1	'Smith'	$R_1 \geq 154.3 \wedge R_2 \geq 15430.0$
2	'Stone'	$R_1 \geq 49.708 \wedge R_2 \geq 486.76$
2	'Stone'	$R_1 \geq 17.2 \wedge R_2 \geq 1720.0$

Figure 1: Relation *orders1_resources*

products that can be produced with each manufacturing process?".

Typically the evaluation of queries involves both "regular" information and constraints, for example:

$$\begin{array}{ll} \text{CONSTRUCT} & orders1\_resources(O, C, R1, R2) \\ \text{FROM} & orders1(O, C, P1), \\ & products\_resources(P1, P2, R1, R2) \\ \text{WHERE} & P2 = 0 \end{array}$$

Note that in our notation the arguments O, C, $R1$, $R2$, $P1$ and $P2$ in the query are variables, not attribute names, but we sometimes use the same name for a variable and an attribute when the distinction is not important. Suppose the relation *orders1* consists of the two tuples $(1, 'Smith', 154.3)$ and $(2, 'Stone', 17.2)$, and two constraint tuples of the relation *products_resources* correspond to the manufacturing processes above. The answer to the query can be computed by considering all four pairs of tuples obtained from *orders1* and from *products_resources*. In each pair, set $P1$ to the value given by *orders*, set $P2$ to 0, and finally, simplify the constraints for $R1$ and $R2$. Figure 1 depicts the results. Note that we produce here a relation that is only partly constraint. *Order#* and *Customer* are regular and R_1 and R_2 are *constraint* attributes.

Clearly, regular relational database queries cannot produce this sort of relation as an answer. Although CLP can, in principle, implement this sort of query, it is not efficient. Consider another example query:

$$\begin{array}{ll} \text{CONSTRUCT} & both\_products(O1\#, O2\#) \\ \text{FROM} & orders1(O1, C, P1), \\ & orders2(O2, C, P2) \\ \text{WHERE} & P1 + 100\,P2 \leq R1, \\ & 100\,P1 + P2 \leq R2, \\ & P1, P2, R1, R2 \geq 0, \\ & R1 = 100, \\ & R2 = 10000 \end{array}$$

Note that the first three lines in the WHERE clause correspond to the first manufacturing pattern given above.

Note also that attributes (such as $R1$ and $R2$) do not have to appear in a relation.

	look -at	simple checks of constr.	proj. on single var.	satisf. tests
Naive	10^{12}	10^7	–	–
CLP($\mathcal{R}$)	10^9	10^4	–	10^6
SQL	10^7	10^7	–	–
Possible	10^2	10^2	2	–

Figure 2: Both_products: evaluation costs

In order to estimate the size of the answer to the query and its evaluation time, let *orders* denote either *orders1* or *orders2*, and make the following 3 assumptions. (1) The relation *orders* has 10^6 tuples. (2) The image size (that is, the number of different values) of *Customer* in *orders* is 10^5. (3) The range of P in *orders* is $[0, 10^5]$ and then, assuming values are uniformly distributed, there are approximately $size(orders) * (b - a)/10^5$ of tuples having a P value in the range $[a, b]$.

The table in Figure 2 depicts the costs of naive evaluation, CLP($\mathcal{R}$), SQL and the ideal possible evaluation. The naive evaluation simply considers all pairs of tuples. In CLP($\mathcal{R}$) the only tuples of *order1* that are consistet with the constraints are checked against *order2*. SQL takes advantage of using index on *Customer* in *orders2* for join operation.

The ideal evaluation does much better, as shown in Figure 2, as follows. First observe that we can deduce that $P1$ is in the range $[0.0, 100.0]$ and $P2$ is in the range $[0.0, 1.0]$. By the assumptions, there are about 10 relevant tuples in *orders2* and 1000 relevant tuples in *orders1*. If we can estimate this comparison, we will perform a selection on *orders2*. Assuming an index on $P2$ is maintained in *orders2*, the selection of 10 tuples will take about 10 ***look-at*** operations, in addition to some overhead of one indexed access. Instead of selecting about 1000 tuples from *orders1*, find a natural join on C of the tuples selected from *orders2* and the relation *orders1*. By assumption 2, the result has approximately 100 tuples and, assuming an index is maintained on *Customer* in *orders1*, this join will take about 100 ***look-at*** operations. Finally, check every tuple in the join to see that it satisfies the constraints.

We can see, in this example, the advantages of deducing ranges on attributes and of estimating costs before making a decision. Our approach attempts to balance these advantages with the costs of range deduction and estimation.

3 Data Model and Query Language

3.1 Data Model

A ***constraint tuple*** has the form $(t_1, \ldots, t_n)$ WHERE c in which the t_i's are either variables or constants and c is an existentially quantified conjunction of constraints, with free variables from $t_1, \ldots, t_n$. By using equality constraints we can write the tuple into the standard form $(x_1, \ldots, x_n)$ WHERE c', in which the x_i's are variables. When it is convenient, we will identify the constraint tuple with the constraint c'. A ***constraint relation*** (or simply relation) is a collection of constraint tuples. It can be understood as a finite representation for a possibly infinite regular relation; every assignment of values to variables which satisfies the constraint in a constraint tuple corresponds to a regular tuple.

A ***constraint relation scheme*** associates a type to each attribute of the relation, and specifies a canonical form for constraints. The type specifies the kind of values (integer, real, string, etc.) that the attribute may take and whether the value must appear explicitly in each constraint tuple, (as is usual for databases), or may be represented implicitly by constraints. In this paper, only one type allows constraints, ***constrained reals***. All other types (reals, integers, etc.) are ***regular*** database types. The constraints in each constraint tuple in the relation are required to be presented in the canonical form. We discuss canonical forms in the next section.

Thus our data model is almost an instance of the framework of [18]. The difference is that we consider explicitly the form in which constraints are presented and allow existentially quantified constraints to appear in constraint tuples.

3.2 Query Language

In general, a query will take the form

```
CONSTRUCT   a(X_1, ..., X_n)
                FROM  b_1(args), ..., b_k(args)
                WHERE cons_1(args)
OR
                FROM  c_1(args), ..., c_l(args)
                WHERE cons_2(args)
OR              ...
```

where each occurrence of $args$ denotes a sequence of variables from the set $\{X_1, \ldots, X_n, \ldots, X_m\}$. For convenience we assume that in each FROM clause no equality between two distinct variables is explicitly implied by $cons$ in the WHERE clause. (If this happen it is always possible to replace one variable by the other.) The query defines a relation a which contains the tuple $(v_1, \ldots, v_n)$ iff there are values $v_1, \ldots, v_n, \ldots, v_m$ which occur in the relations $b_1, \ldots, b_k$ and satisfy $cons_1$, or occur in the relations $c_1, \ldots, c_l$ and satisfy $cons_2$, or .... Since a is written with variable arguments, we sometimes abuse terminology and call an attribute a variable, or vice versa. This query incorporates selection, projection, join and union operations.

A ***linear arithmetic constraint*** has the form $r_1 X_1 + \cdots + r_m X_m$ $relop$ r, where $r, r_1, \ldots, r_m$ are real number constants and $relop$ is one of $=, <, \leq, >, \geq$. An arithmetic constraint is ***pseudo-linear*** with respect to a set of variables $\tilde{y}$ if, whenever the variables $\tilde{y}$ are replaced by real number constants, the resulting constraint is linear. We require that every constraint appearing in a WHERE clause be pseudo-linear with respect to those variables in the corresponding FROM clause which have regular types.

A straightforward extension of this language can incorporate views, cascades of views, complex types, and function symbols. These additional features do not significantly affect the issues we address in this paper. Other additional capabilities, such as recursion and the use of aggregation operators, introduce further complications, and we will not address them here.

Instead we direct our attention to a subset of this query language in which all constraints appearing in a query are linear. We consider selections and projections of relations, and the join of two relations, but we do not explicitly discuss the union operation.

4 Canonical Forms and Constraint Manipulation

In this paper, the constraint c associated with a constraint tuple is a (possibly existentially quantified) conjunction of linear equations and inequalities. In this section, we briefly discuss some computational issues on the manipulation of such constraints.

A ***canonical form*** for constraints is a useful standard form of the constraints, and is generally computed by simplification and the removal of redundancy. In addition to the advantages of a standard presentation of constraints, canonical forms can provide savings of space and time. In the class of linear arithmetic constraints there are many plausible canonical forms. However, they can be costly to compute.

Corresponding to a constraint relation is a disjunction of the constraints in each tuple. Some of these tuples might be redundant in the sense that omitting them does not alter the regular relation represented by the constraint relation. Clearly a canonical form that eliminates such tuples would be desirable. However, the problem of detecting such tuples is co-NP-complete [40], and so we will perform only two simplifications of disjunctions: the deletion of each tuple with an inconsistent constraint, and the deletion of duplicates when all values are regular.

Similarly, while it is theoretically possible to eliminate all existential quantifiers from our constraints (as required in the framework of [18]), the cost of this elimination and the size of the resulting constraint can grow exponentially in the size of the original constraint. Since we expect applications with large constraints, it is unrealistic to expect that all quantifiers can be eliminated. We suggest a method of only performing simplifying quantifier eliminations, similar to what is done in CLP($\mathcal{R}$) [16].

The conjunctive constraints offer the greatest scope in choosing a canonical form. One choice is to write all equations in the form $\{x_i = t_i \mid i = 1, \ldots, n\}$ where the x_i's are distinct and appear nowhere else in the constraint. A second choice is whether all equations which are implicit in the inequality constraints should be represented explicitly. (As a simple example of this, consider the constraints $x + y \leq 2, x + y \geq 2$.) A third is the extent to which redundancy within the inequalities should be removed. [23] presents a classification of redundancy that suggests simple forms of redundancy removal. A fourth choice is whether to keep the inequalities in a different form, such as simplex tableau form.

A fifth option is the addition of redundant information to the constraints. In particular, since range constraints will play an important role in our optimization and implementation methods, we consider a canonical form that requires explicit ranges for some variables. (A *range constraint* is a constraint on a single variable using inequalities or equations. A range constraint is *trivial* if it has the form $-\infty < X$ or $X < \infty$.) More specifically, we require the "tightest" such range, which can be obtained for each variable by projecting the conjunctive constraint onto the variable. Placing constraints in canonical form and, in particular, testing the satisfiability (or consistency) of constraints requires, in general, linear programming techniques.

For the purposes of this paper we consider just one class of canonical forms. We assume that there are no implicit equations, that equations are presented in the form suggested by the first choice, some simple redundancy in the inequalities is removed, and there are explicit range constraints for some variables.

In addition to choosing canonical forms for constraint relations, we must also consider the manipulations of constraints necessary in the evaluation of queries. The most important computation with query constraints is the extraction of a range on a variable. The extraction of a lower bound (for example) on x is exactly the linear programming problem of minimizing x subject to the constraints. The detection of implicit equalities in the query constraint is also a linear programming problem [22] as is, of course, testing for consistency.

5 Optimization: Differences in Approach

In this section we highlight differences between constraint databases and regular databases, which make the straightforward application of usual database techniques difficult or impossible. Consider, first, a simple problem of selection, that is, the query of the form

CONSTRUCT	$a(X_1, \ldots, X_n)$
FROM	$b(X_1, \ldots, X_i, \ldots, X_n)$
WHERE	$cons(X_i, \ldots, X_n, \ldots, X_m)$

Each constraint tuple of a can be constructed by taking a conjunction of a constraint tuple from b and *cons*, testing whether it is satisfiable, and if it is, finding a required canonical form for it. Note that depending on the canonical form for existential quantifiers, this may involve quantifier elimination (some) the variables $X_{n+1}, \ldots, X_m$. Thus, in general, processing a tuple in a constraint selection is significantly more expensive than in a regular selection.

To avoid unnecessary computation, we want to use the idea of *filtering*, similar to one used in spatial databases, that is, the discarding of irrelevant tuples of b by computationally cheap test. Suppose we have a range $c \leq X_k < d$ for X_k in *cons*, where c might be $-\infty$ and d might be ∞. If X_k is also a regular variable in b, we can discard all tuples in b whose X_k value does not lie in the range, since clearly those tuples are inconsistent with *cons*. Similarly, if X_k is a constrained variable and a range for X_k is stored for each tuple of b, then we can discard all tuples for which the ranges for X_k are disjoint.

(There is a larger class of constraints of use in filtering. A constraint is *simply checkable* wrt a relation r, if every variable in the constraint also occurs in the relation in an attribute that either is regular or has a range constraint in the canonical form for tuples of r. While testing such a constraint is a little more expensive, in general, than testing ranges, the cost still compares very favorably with the use of linear programming.)

We can do filtering more efficiently using indices. Indexing on regular attributes is the same as usual, whereas indexing on a constraint attribute X of r works as follows. For each inserted constraint tuple t the range of X is extracted using linear programming techniques. This interval is inserted into an index structure maintaining intervals and has a reference to the corresponding tuple. Selection of all tuples t in r consistent with a given set of constraints c is done as follows. First the range I of X is extracted from c. Second, using the interval index all tuples whose corresponding ranges of X intersect I are retrieved. Third, the retrieved tuples are checked for consistency with c using linear programming methods. Of course, many different indices can be maintained and used for selection. Moreover, in order to improve filtering additional attributes can be defined as linear combinations of constraint attributes, as proposed in [3].

In general we need to have index structures supporting storage of values and intervals, and value and range queries. Two efficient access structures for intervals are the interval tree [9] and the priority search trees [28]. In one dimension, finding all intervals intersecting a given interval or containing a given point, takes at most $O(n \log n+k)$ time, where n is the size of the relation and k is the size of the output. Moreover, it requires only linear space in the size of a relation, and thus seems to be ideal as an indexing structure. The work in [21] proposes an efficient data structure for secondary storage, having the same space and time complexity and full clustering. There are different data structures to support access to multidimensional intervals, in particular based on combination of interval, segment and range trees [9, 37]. For 2-dimensional intervals (rectangles) R, R^+, R^*trees [11, 36, 39] are widely used in spatial databases.

In order to perform indexing and filtering, it is necessary to extract ranges of variables from *cons*. This extraction involves techniques of linear programming and can be very expensive, especially in applications coming from operational research in which *cons* might involve over a thousand constraints and variables. Thus, there is a trade-off to be made between an improvement gained by filtering and indexing and the cost paid for extracting ranges from *cons*.

Consider now projections, that is, the queries of the form

CONSTRUCT	$a(X_1,\ldots,X_n)$
FROM	$b(X_1,\ldots,X_n,\ldots,X_m)$

Computing a projection may involve, depending on the required canonical form, quantifier elimination of (some of) the variables $X_{n+1},\ldots,X_m$. In contrast to the usual database case, in which projection is a trivial operation, when constraints are involved it can be computationally expensive.

Consider now a "constraint" join, where the query is of the form

CONSTRUCT	$a(X_1,\ldots,X_i,\ldots,X_j,\ldots,X_n)$
FROM	$b(X_1,\ldots,X_j)$,
	$c(X_i,\ldots,X_n)$

In principle, each tuple in the answer to this query can be computed by taking a conjunction of a tuple from b and a tuple from c, testing its satisfiability and, if satisfiable, presenting it in the required canonical form. As with the constraint selection, we can use filtering to reduce the cost of satisfiability tests. The filtering step discards those pairs of tuples that have disjoint ranges on a common attribute. A refinement step then performs a full test for satisfiability for the remaining pairs. Note that the regular join does not involve constraints and hence does not require the refinement step. In this paper, we associate the notion of join only with the filtering step, and treat the full test for satisfiability as a separate operation.

We would like to use the ideas developed for regular joins for the filtering in the constraint join. The indexed join (i.e. for each tuple of one relation finding all corresponding tuples of the second using an index) for constraint relations differs from the indexed join for regular relations only in the different index structures that can be used. However, an analogy for the sort join (sorting both relations on common attributes and then finding all matching tuples in one merge) is not clear, since there is no appropriate total ordering on multidimensional intervals. In Section 8 we adapt work in computational geometry to give an analog to the sort join.

Finally, consider the two major approaches to query optimization for regular databases. One is based on algebraic simplification of a query and compile time heuristics. The other is based on cost estimation of different strategies. Neither of these is adequate for constraint database systems. The heuristics of the algebraic approach, such as performing selections as early as possible, are based on the assumption that selection conditions are readily available. In contrast, extracting such conditions from the constraints of a query involves linear programming techniques which are in general expensive. For the cost estimation approach, we have a similar problem of extracting explicit constraints which are needed for the estimation. Even if these constraints were readily available, there is a second problem: it is typically necessary to make assumptions about distribution of the data (like uniformity within, and independence of, columns) in the database, and these appear unlikely to hold in constraint databases.

6 Algorithm for Constraint Selection and Projection

Here the considered queries are of the form

CONSTRUCT	$a(\overline{Y})$
FROM	$b(\overline{Z})$
WHERE	$cons(\overline{W})$

We proceed by presenting an evaluation scheme which represents many evaluation plans. Evaluation schemes and plans are not intended to represent the decision-making process, but only to represent the decisions that need to be made, and the work that needs to be done. We then discuss the trial evaluation, which is necessary for estimating costs, and some heuristics which can be used to order evaluation plans. The generic gambling algorithm, described in the next section, uses this information to choose which plans receive a trial evaluation and, ultimately, to choose an evaluation plan.

We propose the following evaluation scheme for this query.

1. Choose a subset T of the common attributes $\overline{Z} \cap \overline{W}$. For each $X \in T$, extract from *cons* a range on X. Let S be the set of all attributes for which there are non-trivial range constraints (including those attributes from T).

2. Pick an index maintained on b whose selection condition can be explicitly checked by the range constraints on attributes in S. Using this index, select all tuples from b satisfying the constraints.

3. From these tuples, filter out those which do not satisfy simply checkable constraints from *cons* and the extracted range constraints.

4. Project out all regular attributes of b that do not appear elsewhere, eliminating duplicates.

5. For each remaining tuple t, check the satisfiability of the conjunction of t and *cons*. If it is satisfiable, put it in canonical form. If it is not, discard it.

This scheme leaves open the specific choice of a subset T of variables, and an index. Fixing a choice gives rise to a particular ***evaluation plan***.

The cost of an evaluation plan depends strongly on the ranges extracted in Step 1. Therefore, the ***estimation*** of such cost requires, in addition to statistical sampling, some amount of ***actual*** evaluation. Call this process of sampling/evaluation a ***trial evaluation*** of the plan. Now, even trial evaluation can be expensive and therefore it is unrealistic to estimate the cost of all evaluation plans. In fact the cost of estimation may exceed the cost of a naive evaluation.

In the next section, we provide our ***gambling algorithm*** that balances these two costs by considering evaluations plans one at a time and limiting the cost of the estimation of a plan to a portion of the cost of the best plan according to the estimation so far. For the remainder of this section, we detail the trial evaluation and provide heuristics on the order in which the plans are to be considered for estimation.

The trial evaluation for a given plan is described by steps (a) – (e) below. These steps comprise the subprocedure DO-TRIAL-EVAL-OF of the gambling algorithm for the case of selection-projection queries. (We provide different steps for other kinds of queries later.)

(a) Perform Step 1 above.

(b) Take a random sample of b. The number of tuples, say n, in the sample is a compile time parameter.

(c) Select the tuples in the sample satisfying the selection condition of the index chosen in Step 2. Let n_1 denote the number of tuples selected in Step 2.

(d) Perform filtering (Step 3) and projection (Step 4) on these n_1 selected tuples. Measure the average cost per tuple, say a_1. Let n_2 denote the number of tuples selected in Step 3.

(e) Perform the satisfiability test (Step 4) on these n_2 selected tuples. Measure the average cost per tuple, say a_2.

Note the that Step (b) of the trial evaluation is done only once for all plans. The cost of the entire evaluation of the plan, except Step 1, can now be estimated as follows. (It is referred to as FIND `Estimated-cost` in the gambling algorithm.) First we estimate the number N_1 of the tuples selected from b using the chosen index in Step 2 by $(N/n) * n_1$ where N is the number of tuples in b. Then, we estimate the number N_2 of the tuples selected in Step 3 by $(N/n) * n_2$. The cost of Step 2 is estimated by $f(N, N_1)$, where f is a given cost function[†] for the index chosen. The costs of Step 3 and Step 4 are estimated by $N_1 * a_1$ and $N_2 * a_2$ respectively. Finally, in addition to the estimations of the costs above, we also compute the confidence intervals for the cost using standard statistical methods.

[†]Typically, $f(m,k) = O(logm + k)$.

```
set e_best to the first evaluation plan to be considered;
set Best-eval-plans to {e_best};
DO-TRIAL-EVAL-OF e_best and FIND
            Estimated--cost.e_best and the pair Bounds-of-estimated-cost.e_best;
set Total-spent-cost and Incremental-spent-cost to the work done so far;
set Best-total-cost to Estimated-cost.e_best + Total-spent-cost;
set the pair Bounds-of-best-total-cost to the sum of
            Total-spent-cost and the pair Bounds-of-estimated-cost.e_best;
COMPUTE Max-trials-cost as a function
            of Best-total-cost and Bounds-of-best-total-cost;

let e be the next plan to be considered;

      (      there is an evaluation plan e to consider          )
while (                         and                              ) do begin
      (   ESTIMATED-COST-OF-DO-TRIAL-EVAL-OF e ≤                 )
      (   Max-trials-cost − Incremental-spent-cost               )

    DO-TRIAL-EVAL-OF e and FIND
                Estimated-cost.e and Bounds-of-estimated-cost.e;
    update Incremental-spent-cost and Total-spent-cost to include the work above;
    if size of Best-eval-plans < MAX-EVAL-PLANS then
        add e to Best-eval-plans
    else if Estimated-cost.e < Estimated-cost.e_worst
                    for the worst plan e_worst in Best-eval-plans then
        discard e_worst from Best-eval-plans and add e to it;
    if e has been added to Best-eval-plans then begin
        if Estimated-cost.e < Estimated-cost.e_best then begin
            set e_best to e;
            set Old-best-total-cost to Best-total-cost;
            set Best-total-cost to Estimated-cost.e_best + Total-spent-cost;
            set the pair Bounds-of-best-total-cost to the sum of
                    Total-spent-cost and the pair Bounds-of-estimated-cost.e_best;
        end;
        if Best-total-cost < Old-best-total-cost then begin
            COMPUTE Max-trials-cost as a function
                    of Best-total-cost and Bounds-of-best-total-cost;
            set Incremental-spent-cost to 0;
        end
        let e be the next plan to consider (if there is one)
    end
end
return Best-eval-plans
```

Figure 3: Procedure CHOOSE-SMALL-SET-OF-BEST-PLANS

We conclude this section with a suggested heuristic on how to order the evaluation plans for the gambling algorithm, which can be used in conjunction with other heuristics. We propose to consider plans earlier when they:

1. require fewer additional range extractions in Step 1, and thus have potentially cheaper trial evaluation. In particular, we start with a plan that requires no extraction.

2. have a "stronger" index in Step 2. Indices on values are considered stronger than indices on intervals. Indices with an equality selection condition are stronger that those with range condition; the latter are stronger than those with one inequality condition. In particular, those plans having any index are stronger than the others.

7 The Gambling Optimization Algorithm

The input is $(e_1, \ldots, e_m)$, the list of evaluation plans to be considered in this order, induced by heuristics for a specific class of queries. The output is an evaluation plan e that is "recommended as the best". The basic idea of the algorithm is to perform trials of evaluation plans, one at a time, "gambling" some work required for estimation of the plan in an attempt to discover a better plan. The bound for the gambling cost depends on the best estimated cost so far. The algorithm consists of application of two major parts:

1. CHOOSE-SMALL-SET-OF-BEST-PLANS

2. CHOOSE-BEST-PLAN

The idea behind this split into two parts is as follows. When we are considering each of the plans in turn, we need to use statistical sampling in order to estimate the costs. In general, this estimation is expensive, especially for the more complex types of queries. If we take large samples for greater accuracy of the estimation, we might spend most of the gambling cost just on sampling, giving up consideration of many potential plans. On the other hand, taking small samples may lead, because the lack of accuracy, to recommend a plan that is significantly worse that the real best plan. Our two phase algorithm provides a balance. In the first phase, we use samples that are relatively small, so that we can spend the gambling time on considering many potential plans. However, instead of keeping just the best estimated plan we keep a small set of the best plans. Then, in the second phase we concentrate on a more accurate sampling, spending the remaining amount of the gambling time to try to find the best plan.

The CHOOSE-SMALL-SET-OF-BEST-PLANS procedure appears in Figure 3. It is in general self-explanatory[‡]; here we just clarify important points. The pair of lower and upper **Bounds-of-estimated-cost** of an evaluation plan is derived from the statistical confidence intervals. **Incremental-spent-cost** is the cost spent by the algorithm after the last improvement of the **Best-total-cost** is made. **Max-Trials-Cost** is used to bound the gambling time. To COMPUTE **Max-trials-cost**, which is redone after each improvement of **Best-total-cost**, we suggest the use of

$$\min\{\alpha * \mathbf{Best-total-cost}, \\ \beta * (\textit{lower bound of } \mathbf{Best-total-cost})\}$$

where α denotes some fraction of the entire evaluation cost we are ready to gamble. The parameter β should be a higher fraction than α and serves as a "watch dog", that is, if we overestimate the best **Best-total-cost**, then, in the worst case we are going to spend at most fraction β of the real cost. Finally, **MAX-EVAL-PLANS** is a compile-time parameter specifying the maximal number of best plans to be kept for the output.

The input to CHOOSE-BEST-PLAN is **Best-eval-plans** which is provided by CHOOSE-SMALL-SET-OF-BEST-PLANS; the output is the recommended plan. In each iteration of the algorithm some computational cost is paid for additional sampling to estimate more accurately the costs of the current best plan e_{best} and the plan e which is more likely than other plans to replace the current best. Also, we discard all plans for which it can be statistically verified that they are either more expensive than e_{best} or close to it up to a certain small percentage ε. This ε denotes a marginal percentage of cost, and used to avoid useless sampling for comparing plans that have practically indistinguishable costs. The iterations end when either only one plan is left, or when we have exhausted **Max-sampling-cost**.

The procedure CHOOSE-BEST-PLAN appears in Figure 4. **Max-trials-cost** is computed as in the procedure CHOOSE-SMALL-SET-OF-BEST-PLANS, but with different coefficients, reflecting the fraction of the entire cost we are ready to gamble. **One-trial-cost** is the cost spend in one iteration; it depends on a compile time parameter **MAX-ITERATIONS**. It is important that **MAX-ITERATIONS** be sufficiently large, so that **Max-trials-cost** will be spent fairly and many

[‡]Subprocedures requiring additional explanation appear in the algorithm in SMALL CAPITALS.

let e_{best} ∈ `Best-eval-plans` *be the plan that has the least estimated cost;*
set `Spent-cost` *to* 0;
while *size of* `Best-eval-plans` > 1 **do begin**
    for each plan e', *except* e_{best}, COMPUTE *statistical confidence* $C_{e'}$ *with which the cost of* e'
        exceeds the cost of e_{best} *plus* ε *percent; suppose* C_e *is the lowest;*
    discard all plans e' *in* `Best-eval-plans` *with* $C_{e'} \geq$ `SIGNIFICANT_CONFIDENCE`;
    COMPUTE `Max-trials-cost` *as a function of*
        `Estimated-cost`.e_{best} *and* `Bounds-of-estimated-cost`.e_{best};
    set `One-trial-cost` *to* `Max-trials-cost` / `MAX-ITERATIONS`;
    if `One-trial-cost` > `Max-trials-cost` − `Spent-cost` **then**
        discard all plans but e_{best} *from* `Best-eval-plans`
    else begin
        increment `Spent-cost` *by* `One-trial-cost`;
        EVALUATE-OPTIMAL-PARTITION-OF `One-trial-cost` *giving costs* `Cost1` *and* `Cost2`
            of work to be spent on estimating costs of e_{best} *and* e *respectively;*
        TAKE *additional samples for estimating costs of* e_{best} *and* e *spending*
            `Cost1` *and* `Cost2` *respectively and re-estimate the costs of* e_{best} *and* e;
        if `Estimated-cost`.e_{best} > `Estimated-cost`.e **then**
            set e_{best} *to* e;
    end
end
return e_{best}

Figure 4: Procedure CHOOSE-BEST-PLAN

plans will have chance to compete for the first place. Note that, intuitively, there is a trend in the iterations to eventually discard e as the confidence intervals of costs for e_{best} and e get smaller, since it becomes more likely that the confidence C_e will exceed the confidence $C_{e'}$ of some other plan e'. On the other hand, `MAX-ITERATIONS` should not be too large, because of the overhead this can create. Finally, EVALUATE-OPTIMAL-PARTITION-OF `One-trial-cost` means, intuitively, maximizing the confidence of the decision which plan, e_{best} or e, is the best. This is done by minimizing the variance of the random variable `Estimated-cost`.e − `Estimated-cost`.e_{best}, which is a function of the sizes of the samples for e_{best} and e, subject to the constraint that the total cost on sampling is `One-trial-cost`. This problem usually translates to minimizing a quadratic function in one variable and can be easily done.

8 A Constraint Sort Join Algorithm

We adapt the algorithm of [9, 37] for n-dimensional rectangle intersection to perform an analog of the sorted equijoin. It is not possible to sort directly on a constrained attribute, since each tuple allows a range of values for that attribute and tuples may overlap. Instead we sort the endpoints of the ranges in the tuples, using not only the value of the endpoint, but also the type of the boundary: whether it is a point or a lower or upper boundary, and whether the boundary was caused by a strict or nonstrict inequality. (We must assume here that, for each common attribute X of type constrained real in the relations, there is a range for X in the canonical form of each relation.)

The value *value*(e) of an endpoint e may be any real number, $-\infty$ or ∞. For each endpoint e, there is a boundary type *bdry*(e), and these are ordered as follows: *upper-strict* < *lower-nonstrict* < *point* < *upper-nonstrict* < *lower-strict*. We write $e_1 \preceq e_2$ if *value*(e_1) ≤ *value*(e_2), or *value*(e_1) = *value*(e_2) and *bdry*(e_1) ≤ *bdry*(e_2).

To simplify the exposition, we assume initially that there is only one common attribute which is not regular in both relations. For each relation P, let p be the relation on the common attributes which is the projection of P except that there are, in general, two elements of p corresponding to each tuple, one for each endpoint[§]. (In practice it is not necessary to construct

[§]If a range is, in fact, a point then p contains only one element for that tuple.

```
construct p and q from input relations P and Q;
≼-sort p and q;
initialize Output to ∅;
initialize Active-set-for-p to ∅;
initialize Active-set-for-q to ∅;
initialize i and j to 1;
repeat
  if p_i ≼ q_j then begin
    if p_i is a point then
      add tuple(p_i) ⋈ Active-set-for-q to Output;
    if p_i is a lower boundary then begin
      add tuple(p_i) to Active-set-for-p;
      add tuple(p_i) ⋈ Active-set-for-q to Output
    end;
    if p_i is an upper boundary then begin
      remove tuple(p_i) from Active-set-for-p;
    increment i;
    end
  else
    We perform the same steps as in the then clause,
    with the roles of p and q, and i and j swapped;
until p or q has been exhausted;
return Output
```

Figure 5: A sort join algorithm

p explicitly.) We say that p is $\preceq$-sorted if it is sorted according to the lexicographic combination of the order on the regular attributes and $\preceq$. We write $tuple(p_i)$ to denote the tuple of P that produced the i'th element of p. We say $p_i \preceq q_j$ if p_i and q_j agree on values for the regular attributes and the value of p_i on the remaining attribute $\preceq$ the value of q_j on that attribute.

The algorithm (Figure 5) first $\preceq$-sorts p and q corresponding to the input relations P and Q. It then applies the plane-sweep technique [9], traversing the endpoints in order from least to greatest. At each stage of the sweep, `Active-set-for-p(Active-set-for-q)` holds the set of tuples of P (Q) which contain the current endpoint. If the current endpoint e comes from p then $tuple(e) \bowtie$ `Active-set-for-q` must be contained in $P \bowtie Q$, and similarly if e comes from q. We record this information at lower endpoints only, since upper endpoints only duplicate the information. The remainder of the algorithm updates `Active-set-for-p` and `Active-set-for-q`.

When we have only one dimension (that is, only one constrained attribute) then `Active-set-for-p` and `Active-set-for-q` can be simple set data structures. For two dimensions, we want to filter out from `Active-set-for-q` those tuples which fail to intersect the current tuple due to the ranges on the second constrained attribute. The appropriate data structure is the interval tree [9] which allows us to do this filtering efficiently. In general, for d dimensions we use a combination of range and interval trees [9, 37]. This gives the algorithm for a d-dimensional sorted join a worst-case time of $O(N log N + M log M + log^{d-2} N + log^{d-2} M + K)$, and a worst-case space cost of $O(M log^{d-1} M + N log^{d-1} N)$, where P has M tuples, Q has N tuples and the output relation has K tuples.

We refer to the regular attributes and the first constrained attribute as *scanned* attributes and the remaining attributes, those for which filtering is done inside the active sets, are called *active* attributes.

9 Optimization for Constraint Select-Project-Join Queries

In this section we show how to use the gambling algorithm to evaluate constraint join-select-project queries. We consider queries having up to two relations, that is, of the form

CONSTRUCT	$a(\overline{X})$
FROM	$b(\overline{Y})$,
	$c(\overline{Z})$
WHERE	$cons(\overline{W})$

where b and c are constraint relations and $cons(\overline{X})$ a set of linear constraints. We propose the following evaluation scheme:

1. Decide on whether to use a regular join, or a constraint sort join or constraint indexed join algorithm.

2. For a constraint sort join choose *scanned* attributes that should include all regular (in both relations) common attributes, in addition to one selected constrained common attribute. Choose also a set of *active* common attributes. If the set of scanned attributes is already ordered, decide on whether selections are to be done on this relation (in the process probably destroying the ordering).

3. For an index join, decide which of the relations is to be scanned, and choose an index on common attribute(s) for the other relation. The selections before the join will be done only on the scanned relation.

4. Choose a subset T of attributes from $cons(\overline{W})$ and for each attribute $V \in T$ extract from *cons*

the range on V. Only useful attributes should be chosen in T, that is, those that appear in at least one of the relations on which selection is to be done. Let S denote the set of all attributes for which there are non-trivial range constraint.

5. For each relation r on which selection is to be done,
   (a) Pick an index whose selection condition can be explicitly checked by the range constraints on attributes in S.
   (b) Using this index, select all tuples from r which satisfy the range constraints.
   (c) From these tuples filter out those which do not satisfy simply checkable constraints w.r.t r in *cons* and the extracted range constraints.
   (d) Project out all regular attributes in r that do not appear elsewhere in the query, eliminating duplicates.¶
6. Perform the chosen join algorithm on the resulting relations.
7. Filter out all tuples in the new relation that do not satisfy constraints in *cons* that are simply checkable w.r.t. the new relation, or do not satisfy the extracted range constraints.
8. Project out all regular attributes in the new relation that do not appear in *cons* nor in the answer relation a, eliminating duplicates.
9. For each remaining tuple t, filter out those for which the conjunction of t and *cons* is unsatisfiable.
10. From the remaining tuples project out regular attributes that do not appear in a, eliminate duplicates and put the resulting tuples into the required for a canonical form;

Each series of choices in the evaluation scheme gives rise to a possible evaluation plan. We discuss only briefly the trial evaluation of a particular plan e, and estimating its cost, referred in the gambling algorithm as "DO-TRIAL-EVAL-OF e" and "FIND **Estimated-cost**.e." First we extract ranges from *cons* for variables in T in Step 4. Then, we take a sample of tuples from the relations on which selection is to be done. Exactly as in the case of select-project query in Section 6 we estimate the number of tuples satisfying the selecting condition of the index, and the number of tuples after the additional filtering and projection in Step 5(b,c) and using this information the cost of the index and the filtering.

Estimation of the cost of the join in Step 6 depends on the join method. For indexed join, we use the sample from the scanned relation. This sample is likely to have been taken already for estimation of selection cost. Then we actually join each tuple in the sample with the second relation using the chosen index. It is done in order to measure the average cost per tuple and to estimate the number of tuples in the result of the join in Step 6. For sort-join, we take sample of pairs of tuples from the relations b and c. Note that in order to get sufficient accuracy of the estimation the size of the sample should be significantly larger than that of indexed join. We use this sample to estimate the the average number of tuples in b that can be joined with one tuple in c in the sort join and vice versa and then to substitute these numbers in the formula for the sort-join cost. The cost estimation of the remaining steps is done by actually performing this steps on the result of the "simulated cost" and then normalizing the costs, analogically to what is done in the estimation for select-project queries. Here too we use statistical tests to compute **Bounds-of-best-total-cost** of e with significant statistical confidence.

The only non-trivial part of the estimation of trial evaluation cost, referred in the gambling algorithm as "ESTIMATED-COST-OF-DO-TRIAL-EVAL-OF e" is estimating range extraction costs. This is done exactly as in the case for select-project queries.

Finally we provide some heuristics on the order in which evaluation plans are to be considered in the gambling algorithm. We propose to consider plans earlier when they:

1. require fewer additional range extractions.
2. among the plans with indexed join, use a "stronger" index, where "stronger" is defined as for select-project queries. For the plans with sort join consider as follows. If there is at least one attribute to be active, consider first the plans with smaller number of active attributes. Among those, consider first those with active attribute that is regular in one relation‖.
3. have stronger indices for selection.
4. use an indexed join when picking a plan to be the first.

¶Two tuples are duplicate if they are identical including the canonical form of the constraints.

‖Recall that since we always put regular common attributes to be scanned, an active attribute cannot be regular in both relations.

5. among plans that are not distinguished by the previous criteria, pick any one, preserving fairness (for example pick one at random).

References

[1] M.M. Astrahan et. al., System R: A Relational Approach to Database Management, *ACM Trans. on Database Systems* 1, 2, 97–137, 1976.

[2] P.A. Bernstein & N. Goodman, The Power of Natural Semijoins, *SIAM Journal on Computing* 10, 4, 751–771, 1981.

[3] A. Brodsky, C. Lassez & J.-L. Lassez, Separability of Polyhedra and a New Approach to Spatial Storage, *Proc. Principles and Practice of Constraint Programming*, 6–13, 1993.

[4] A. Brodsky & Y. Sagiv, Inference of monotonicity constraints in datalog programs. *Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems*, Philadelphia, 1989, pp. 190–199.

[5] J. Chomicki & T. Imielinski, Relational Specifications of Infinite Query Answers, *Proc. ACM SIGMOD*, 174–183, 1989.

[6] U.S. Chakravarthy & J. Minker, Multiple Query Processing in Deductive Databases using Query Graphs, *Proc. VLDB*, 384–391, 1986.

[7] D.D. Chamberlin et. al., Support for Repetitive Transactions and Adhoc Queries in System R, *ACM Trans. on Database Systems* 6, 1, 70–94, 1981.

[8] D. Daniels et. al., An Introduction to Distributed Query Compilation in R*, IBM Research Report RJ3497, 1982.

[9] H. Edelsbrunner, A new approach to rectangle intersections, Part II, *International Journal of Computer Mathematics*, 13, 221–229, 1983.

[10] P.P. Griffiths et. al., Access Path Selection in a Relational Database Management System, *Proc. SIGMOD*, 23–34, 1979.

[11] A. Guttman, R-trees: A dynamic index structure for spatial searching, *Proc. ACM SIGMOD*, 47–57, 1984.

[12] P.A.V. Hall, Optimization of a Single Relational Expression in a Relational Database, *IBM Journal of Research and Development* 20, 3, 244–257, 1976.

[13] M.R. Hansen, B.S. Hansen, P. Lucas & P. van Emde Boas, Integrating Relational Databases and Constraint Languages, *Computer Languages* 14, 2, 63–82, 1989.

[14] J. Jaffar, S. Michaylov, P. Stuckey & R. Yap. The CLP($\mathcal{R}$) Language and System, *ACM Transactions on Programming Languages*, 14(3), July 1992, 339–395.

[15] J. Jaffar & J-L. Lassez, Constraint Logic Programming, *Proc. Conf. on Principles of Programming Languages*, 1987, 111–119.

[16] J. Jaffar, M.J. Maher, P.J. Stuckey & R.H.C. Yap, Output in CLP($\mathcal{R}$), *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Tokyo, Japan, Vol. 2, 1992, 987–995.

[17] A. Klug, On Conjunctive Queries Containing Inequalities, *JACM* 35, 1, 146–160, 1988.

[18] P. Kanellakis, G. Kuper & P. Revesz, Constraint Query Languages, *Journal of Computer and System Sciences*, to appear. (A preliminary version appeared in *Proc. 9th PODS*, 299–313, 1990.)

[19] D. Kemp & P. Stuckey, Bottom Up Constraint Logic Programming Without Constraint Solving, Technical Report, Dept. of Computer Science, University of Melbourne, 1992.

[20] D. Kemp, K. Ramamohanarao, I. Balbin & K. Meenakshi, Propagating Constraints in Recursive Deductive Databases, *Proc. North American Conference on Logic Programming*, 981–998, 1989.

[21] P. Kanellakis, S. Ramaswamy, D.E. Vengroff & J.S. Vitter, Indexing for Data Models with Constraints and Classes, *Proc. PODS*, 1993.

[22] J-L. Lassez, Querying Constraints, *Proc. 9th PODS*, 288–298, 1990.

[23] J-L. Lassez, T. Huynh & K. McAloon. Simplification and Elimination of Redundant Linear Arithmetic Constraints. In *Proc. North American Conference on Logic Programming*, Cleveland, 1989. pp. 35–51.

[24] J-L. Lassez & K. McAloon, A Canonical Form for Generalized Linear Constraints, *Journal of Symbolic Computation*, to appear.

[25] R.J. Lipton & J.F. Naughton, Query Size Estimation by Adaptive Sampling, *Proc. 9th PODS*, 40–46, 1990.

[26] A. Levy, Y. Sagiv, Constraints and Redundancy in Datalog, *Proc. 11-th PODS*, 67-80, 1992.

[27] M.J. Maher, A Transformation System for Deductive Database Modules with Perfect Model Semantics, *Proc. 9th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, LNCS 405, 89–98, 1989.

[28] Priority Search Trees, *SIAM Journal of Computing 14(2)*, May 1985, 257-276.

[29] L.F. Mackert & G. M. Lohman, R* Optimizer Validation and Performance Evaluation for Local Queries, *Proc. SIGMOD*, 84-95, 1986.

[30] J. Minker, Search Strategy and Selection Function for an Inferential Relational System, *ACM Trans. on Database Systems* 3, 1, 1-31, 1978.

[31] I.S. Mumick, S.J. Finkelstein, H. Pirahesh & R. Ramakrishnan, Magic Conditions, *Proc. 9th PODS*, 314-330, 1990.

[32] F.P. Palermo, A Database Search Problem, in: *Information Systems COINS IV*, J.T. Tou (Ed), Plenum Press, 1974.

[33] R.M. Pecherer, Efficient Evaluation of Expressions in a Relational Algebra, *Proc. ACM Pacific Conf.*, 44-49, 1975.

[34] F.P. Preparata & M.I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.

[35] R. Ramakrishnan, Magic Templates: A Spellbinding Approach to Logic Programs, *Journal of Logic Programming*, 11, 189-216, 1991.

[36] N. Roussopoulos, D. Leifker, Direct spatial search on pictorial databases using packed R-trees, *Proc. ACM SIGMOD*, 17-31, 1985.

[37] H.W. Six & D. Wood, Counting and reporting intersections of d-ranges, *IEEE Trans. Computing C-31*, 181-187, 1982.

[38] J.M. Smith & P.Y. Chang, Optimizing the Performance of a Relational Algebra Database Interface, *Communications of the ACM* 18, 10, 568-579, 1975.

[39] T. Sellis, N. Roussopoulos, C. Faloutsus, The R^+-tree: A dynamic index for multidimensional objects, *Proc. 13th Int. Conf. Very Large Data Bases*, 507-518, 1987.

[40] D. Srivastava, Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints, *Annals of Mathematics and Artificial Intelligence*, to appear.

[41] D. Srivastava & R. Ramakrishnan, Pushing Constraint Selections, *Proc. 11th PODS*, 301-315, 1992.

[42] K.-Y. Whang & R. Krishnamurthy, Query Optimization in a Memory-Resident Domain Relational Calculus Database System, *ACM Trans. on Database Systems* 15, 1, 67-95, 1990.

[43] E. Wong & K. Youssefi, Decomposition - A Strategy for Query Processing, *ACM Trans. on Database Systems* 1, 3, 223-241, 1976.

[44] M. Yannakakis, Algorithms for Acyclic Database Schemes, *Proc. VLDB*, 82-94, 1981.