Query Evaluation as Constraint Search; An Overview of Early Results

Daniel P. Miranker
miranker@cs.utexas.edu
http://www.cs.utexas.edu/
users/bayardo/

Roberto J. Bayardo Jr.
bayardo@cs.utexas.edu
http://www.cs.utexas.edu/
users/bayardo/

Vasilis Samoladas
vsam@cs.utexas.edu

Dept. of Computer Sciences and Applied Research Laboratories
University of Texas at Austin
Austin, TX 78712

Abstract. We present early results on the development of database query evaluation algorithms that have been inspired by search methods from the domain of constraint satisfaction. We define a mapping between these two specialties and discuss how the differences in problem domains have instigated new results.

It appears that contemporary problems in databases which lead to queries requiring many-way joins (such as active and deductive databases) will be the primary beneficiaries of this approach. Object-oriented queries and queries which are not intended to return all solutions also benefit. Some obvious CSP interpretations of certain semantic database properties suggest open research opportunities.

1 Introduction

There is a direct computational correspondence between solving a constraint satisfaction problem (CSP) and calculating the results of a database query. Nevertheless, there has been little cross-fertilization among these fields. It is precisely this correspondence that has motivated a number of our recent papers [1,2,3,4,8]. Due to the specialization of target audiences for those papers the true motivation has not been expressed. Our intention in this paper is to make clear how these specialties relate and how the differences have motivated both synthetic and symbiotic results.

We define the correspondence between CSP and query evaluation as follows. Recall, a *constraint satisfaction problem* (CSP) is a set of *variables* and a set of *constraints*. Each variable is associated with a finite value domain, and each constraint consists of a subset of the problem variables called its *scheme* and a set of mappings of domain values to variables in the scheme. An assignment A *satisfies* a constraint C with scheme X if A restricted to the variables in X is a mapping in C. A *partial solution* to a CSP is an assignment that satisfies every constraint whose scheme consists entirely of variables mentioned in the assignment. A *solution* to a CSP is a partial solution mentioning every variable.

A CSP may be represented by a constraint graph G = (V, E) where vertex V_i corresponds to variable X_i, and edges connecting vertices represent binary constraints[1]. It is

1 Without loss of generality, we restrict attention to binary CSP.

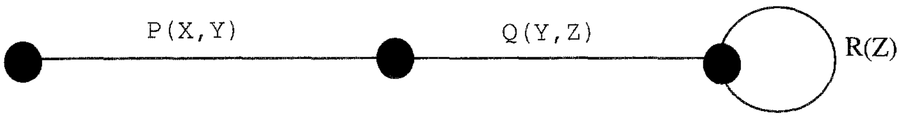
the illustration of this graph representation which makes the correspondence among the two problem domains most obvious. Given a relational query against the contents of a relational database, a query graph is defined as follows. Let each table mentioned in the query be represented by a vertex. For each join predicate expressed in the query, draw an edge connecting the pair of vertices representing the join arguments. In our interpretation, each tuple, in its entirety, is a single label and may be assigned to a single CSP variable. Thus, each database table represents a label set. The join predicate, most commonly equality of an attribute value from each table, defines a constraint.

```
select *
from emp X, customer Y, account Z
where X.name = Y.name
      and Z.credit = good
      and Y.cid = Z.cid
```

A Relational Query in SQL

Let:
P(X,Y) ⇔ X.name = Y.name
Q(Y,Z) ⇔ Y.cid = Z.cid
R(Z) ⇔ Z.credit = good

Constraint Predicates



emp (X)

name	eid
dan	10
bert	12
james	31

customer (Y)

name	cid
dan	1
bert	2
james	3

account (Z)

credit	cid
good	1
bad	2
bad	3

X		Y		Z	
name	eid	name	cid	credit	cid
dan	10	dan	1	good	cid

Query Result and Legal CSP Assignment to X,Y,Z

Fig. 1. A single problem as both a relational query and a CSP.

Figure 1 illustrates the elements of both problem domains by representing a single problem as both a relational query and a CSP. The query says that given a database concerning employees, customers and accounts, return those employees who are also customers with good credit. Viewed as a CSP, employees, customers and accounts form three label sets. The labels can be assigned to variables X,Y and Z respectively. P, Q, and R are three declaratively defined constraints over X,Y and Z that create the same intent as the SQL query.

The reader should already observe two primary differences among these scientific domains. By design there is only one record that fulfills the SQL query. Similarly, there is only one satisfying assignment to the CSP. However, though an SQL query is existentially quantified, the return value of a query is typically the set of all records that satisfy the constraints, not simply a single assignment developed by a constructive proof of satisfiability. Though the "all solutions" problem, in principle, falls within the scope of CSP, in practice virtually all papers in CSP explore the first solution problem. A second difference is, with respect to relational queries, that constraints are limited to relational predicates. Note that in the context of object-oriented databases this specialization does not apply.

By defining this correspondence among the two disciplines, we have that the database join algorithm known as pipelined nested loops is identical to the CSP chronological backtrack algorithm. Further, several results in semi-join reduction (from the domain of distributed databases) can be seen to be equivalent to arc-consistency results in the constraint literature. For instance, Bernstein and Chiu [5] define a "fully semi-join-reduced" database as one in which no semi-join eliminates tuples from the database. This definition can be seen to be equivalent to arc-consistency of the database when viewed as a CSP. They proceed to develop semi-join programs which can fully reduce one relation in an acyclic database (equivalent to Dechter's directional arc consistency [6]) and fully reduce all relations in an acyclic database (equivalent to Dechter's technique for fully reducing tree-structured CSP through two passes of directional arc consistency [6]).

Other work relating constraint satisfaction and equi-join query processing exploits the relational tables themselves to define the constraints such that the join column values serve as the domain values [7]. According to this transformation, chronological backtrack would iterate over column values and test constraints by probing the relations. This method has practical limitations since relations in databases are not necessarily indexed to allow efficient probing.

2 A Matter of Scale and Scaling

What prevents the direct application of several CSP results to query processing is the wide differences in features assumed of "typical" problem instances. In the CSP domain, the number of domain values is often assume to be small (e.g. SAT, scene labelling, graph coloring) and the constraint graph dense. In databases, the relations (number of domain values) are typically assumed to be large (and disk-resident), and the query graph tree-structured or nearly tree-structured. Another complication is the fact that query processing is typically concerned with producing all solutions, while most CSP results deal with only finding one.

With large disk-resident data, a single pass over a relation has a far higher cost than is typically assumed in most CSP analyses. For this reason, despite the appeal of arc-consistency in CSP, semi-join reduction has never been viewed as a practical query processing technique outside of the distributed database domain where communication costs dominate all others.

3 Algorithms and Results

Our start in this work began with an effort to develop optimal algorithms without incurring the cost of a preprocessing step and, otherwise, minimize the memory used by learning mechanisms. Both concerns anticipate the database requirement to reduce disk access. Toward these ends, we have developed algorithms for achieving the data-reduction benefits of arc consistency/ semi-join reduction on tree-structured CSP without a preprocessing step [1]. The algorithms extend chronological backtrack/pipelined joins with book-keeping enhancements in order to provably bound runtime. The book-keeping amounts to a marking per domain value. We demonstrate that known CSP backtrack enhancements combined with variable ordering provides runtime complexity within a logarithmic factor of optimality (TreeTracker I). We present an additional enhancement based on remembering successful branches of the search to achieve optimality (TreeTracker II). Both algorithms are shown to outperform directional arc consistency techniques for solving tree-structured CSP. The amount of improvement increases with domain size, indicating applicability to database-sized problems.

We have also extended the work to cyclic problems [3]. We find that low-space consuming backtrack enhancements can be used to effectively bound runtime on nearly-tree structured CSP across a wide range of variable ordering heuristics. Variable ordering is equivalent to the query optimization task of determining a join order, so freedom in variable ordering is important for obtaining good performance on average.

To date, our efforts within the database domain have been focussed on acyclic queries. Within databases this is not considered a severe restriction. It has been postulated, without rebuttal, that in practice virtually all database queries are acyclic. Even if this proves false, we have shown that straight forward integration of simple methods, such as cycle cutset or structural decomposition, is feasible and these basic results will still hold. More sophisticated opportunities exist and will be discussed below.

We present two extensions of the CSP results for use as database query evaluation algorithms. Recall that the primary measure of interest in databases is disk I/O. Emphasis will be placed on the presentation of cost arguments with respect to database operations (e.g. disk access). They are both contrasted with previous techniques, and their value is assessed by theoretical as well as experimental evidence.

3.1 Finding the First-Few Answers

The computation of the first answer, or the first-few answers of a query, is becoming increasingly important with applications in data mining and distributed databases (such as the World Wide Web). Commercial systems are beginning to provide special support for such processing -- DB2 and Oracle being prominent ones. Cost arguments indicate join pipelining as the technique of choice, because in many cases a small number of answers can be computed without touching vast amounts of unnecessary data.

Ordinary pipelining of multiple joins often suffers from poor performance analogous to chronological backtracking. The problem is amplified in the database domain because queries typically have sparse query graphs. The common star query (whose query graph consists of a root node with branches of length 1) is extremely prone to poor performance when processed using naive join pipelining [4]. The problems can be remedied by augmenting the pipeline strategy with backjumping (also known as intelligent backtracking) techniques.

Given a tree-structured query graph, where nodes represent relations and edges represent joins, a task of query optimization is to assign an ordering to the joins of the query, usually so that cross products are avoided. For example, in the query of Figure 2 we have a pipeline of 3 joins on 4 relations. (In general we will have a pipeline of n joins on $n + 1$ relations.) The pipeline will thus have 4 stages, S_1 through S_4, filled in

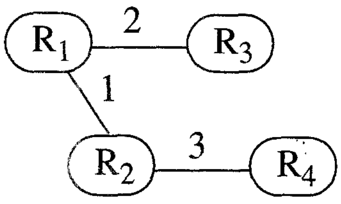
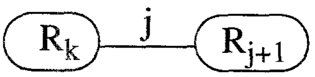


Fig. 2. An ordered query graph. Relation numbering indicates join order.

order by tuples from relations R_1 through R_4 of the figure. With ordinary pipelining, when the pipeline fails to fill S_4 (because the current contents of S_2 do not join with anything in R_4), control will return to S_3, whose contents of course are not responsible for the failure. We have our algorithm return control to S_2, which is the source of the problem. Our algorithm thereby eliminates futile lookups of joining tuples through backjumping. It also avoids redundant tuple lookups by carefully unfilling only the necessary pipeline stages with each backjump. For instance, after a backjump from S_4 to S_2, there is no need to unfill the contents of S_3. The algorithm can simply skip over stage S_3 should S_2 be successfully filled without further backjumps.

An important aspect of query processing is query optimization, which requires the ability to estimate the cost of various query plans in order to select the best one. This problem has been well-studied in the all-answer case in the database domain. Interestingly, we are not aware of any work in the CSP domain on estimating algorithm cost given a specific instance. Our work [4] describes a technique for predicting the cost of a first or first-few answer join query assuming a pipelined join execution model enhanced as described above. We summarize the technique here.

In order to have an accurate cost estimate for obtaining the first-few answers, it is necessary to use statistics not typically maintained in the data dictionary of centralized relational databases. Nevertheless, the statistics are easy to compute or maintain if desired, and can be derived from statistics maintained by distributed databases such as semi-join selectivity. Consider join j illustrated below between relations R_k and R_{j+1} (relations are numbered with their ordering in the ordered query graph).



We define the following two quantities:

$$p_j = 1 - \frac{|R_k \ltimes R_{j+1}|}{|R_k|}$$

$$b_j = \frac{|R_k \bowtie R_{j+1}|}{|R_k \ltimes R_{j+1}|}$$

From these quantities, through probabilistic arguments, we compute T_j, the probability that a lookup across predicate j has to be repeated during pipelined join execution in order to produce an answer to the query. If the average cost of looking up joining tuples along predicate j is w_j (dependent on access paths, clustering etc.), then the cost of the query plan can be predicted by the following:

$$Cost = \sum_j \frac{w_j}{1 - T_j}$$

In general, database statistics are centered around the estimation of join selectivity which provides estimates for join result size. Estimation of semi-join sizes is required by distributed database systems, as well as some centralized systems which utilize semi-join estimation techniques for predicting the effects of nested queries. These systems typically exploit the statistic s_j known as semi-join selectivity, which measures the fraction of tuples that remain after performing the semi-join. From this value, we have that $p_j = 1 - s_j$, and b_j can be derived from join selectivity and p_j.

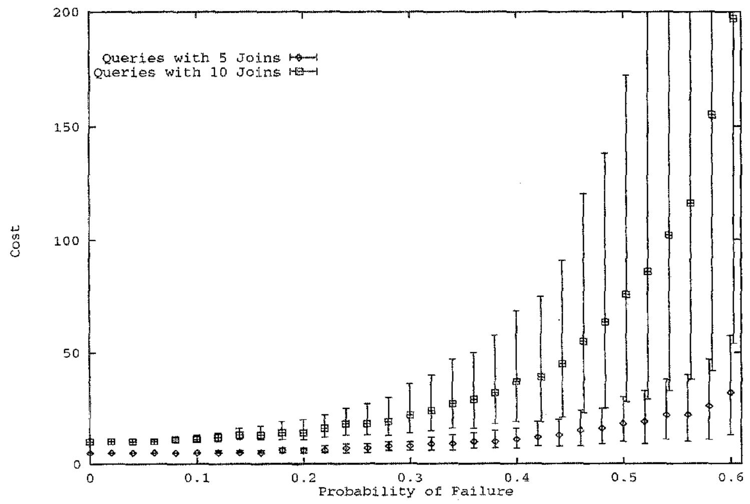


Fig. 3. Expected query plan cost (center ticks), worst and best query plan costs (top and bottom error marks respectively); from [4].

A proof of concept for this technique is provided through experimental results on randomly generated queries intended to capture several properties of real-world queries such as indexed join columns and primary-to-foreign key joins. We find that our cost models accurately predict performance of the different query plans for a given query. Moreover, they reveal that a wide performance gap exists between the expected, best, and worst plans (Fig. 3). Enhanced pipelining is found to reduce the I/O overhead usually by at least 30% and increasing with the number of joins and probability of tuple-lookup failure.

3.2 Finding All Answers in Object-Oriented Databases

Query processing in object-oriented databases (OODBs) is more challenging than in relational databases (RDBs) because of the richer schema capabilities and the presence of encapsulation and inheritance. More often than not, nested-loops evaluation is the only alternative. In this situation, CSP techniques can be valuable in query processing. However, because of the complex nature of the object-oriented data model, the application of CSP techniques becomes conceptually more complicated.

Relational queries explicitly define the data to be processed, as well as a number of propositional elements (predicates) to be applied to the data. OODB queries, apart from purely propositional elements, often declare navigational elements manifesting as path expressions. In other words, the data to be processed is not explicitly defined, but instead the data is defined through references to it (OIDs). A typical case is a query over complex objects (objects containing sets of object references). The following query in Object SQL demonstrates this duality, where variable y is defined navigationally from variable x, by the set-valued path expression x.a.

```
SELECT x, y, z
FROM x  IN C1,
     y  IN x.a
     z  IN C2
WHERE y.b  =  z.c
```

In this context, the analogy between CSP and query evaluation is not clear, until one admits that the navigational elements of an OODB query are indeed additional constraints, albeit of a special nature. The reduction of an OODB query to a CSP problem will associate a CSP variable with each top-level collection (C1 and C2 are such collections) and will also associate variables with set-valued path expressions. The intent is to capture the iterative parts of the query in the formulation of the CSP problem. Propositional elements of the query become constraints in the CSP domain. Navigational elements also become constraints, but their navigational nature has to be respected. This is illustrated in the following query graph, for the query above, where the navigational constraint has become a directed arc.

$$\begin{tikzcd} x \arrow[r] & y \arrow[r, dash] & z \end{tikzcd}$$

The application of the Treetracker algorithm [1] to the query evaluation problem is now formulated as follows. First, not only does the query graph need to be acyclic, but the navigational constraints have to be respected when a depth-first ordering is applied. The following CSP graph, although acyclic, does not admit such a depth-first ordering:

$$\begin{tikzcd} X \arrow[r] & Y \arrow[r, dash] & Z & W \arrow[l] \end{tikzcd}$$

Fortunately, it is often the case that the object schema will provide remedies for such queries, by declaring referential constraints which can be used for transforming the query graph, i.e. "reversing" the direction of particular edges.

There is one more issue to resolve for the extension of CSP techniques to an all-answer computation. This issue is closely related to the representation of the query's result. The solution to the problem exploits the acyclic structure of the query graph. Consider CSP variables X and Y, such that X is the parent of Y in the CSP graph. Let x be an object instantiating X, which appears in the query result. With x we associate a set $x.L(Y)$, called a goodlist, containing all objects y that appear in the result, with respect to object x. In relational terms, the definition of $x.L(Y)$ would be

$$x.L(Y) = \pi_Y(S) \ltimes x$$

where S is the query result and π is the relational project operator. Clearly, computation of goodlists is equivalent to the computation of the query result. TreeTracker can now be used to compute the goodlists, by employing both intelligent backtracking and marking. The desirable asymptotic property of polynomial work is retained, since the overall goodlist space is polynomially bounded (although the actual query result can be exponentially large). More importantly, the size of goodlists is also bound by the size of the result, which implies that even for queries with small answers space is used efficiently.

Integration of the technique to OODBs is quite straightforward, since it employs OID semantics in the construction of goodlists. The addition of markings during backtracking does not in any way preclude the use of special join algorithms (such as hash-join) or acceleration mechanisms (such as indexes) in the computation of the result. The overall computation can be viewed as pipelined query evaluation, where pipeline operation is dictated by intelligent backtracking and previously recorded information.

Experiments comparing our query engine to a commercially available engine revealed that the Tree-Tracker based query evaluation algorithm outperforms intelligent backtracking alone, sometime modestly, though more often by a wide margin (Table 1).

Table 1. Three queries over, 30,000 students, 1000 professors, 5000 courses, from [8].

	CPU Time (sec.)		
	Query 1	**Query 2**	**Query 3**
Us	3975	12.5	4965
A commercially available system	9876	14.5	18796

The improvement over naive nested-loops evaluation will be even more dramatic. The savings come for a number of reasons:

- Redundant predicate applications are virtually eliminated. Although predicate application does not directly involve I/O, in practice predicate parameters are path expressions, whose evaluation would incur I/O. By examining the marking of the root object of a path expression, we can avoid its evaluation.
- In contrast with simple intelligent backtracking, as employed in many systems (e.g. ObjectStore, ODE), our technique involves no more work than algebraic, set-oriented processing -and it typically involves less work. The impact on the I/O, but also on CPU times, can be quite dramatic.
- Because of the depth-first nature of the computation, good locality is achieved. Locality is important in client-server OODBs, where data is typically accessed through an object cache on the client side.

Many of the challenges of Object-Oriented query processing are due to the tight integration of query languages and general-purpose programming languages, such as C++. The majority of systems address these challenges by adapting relational database technology, augmented with a variety of new techniques. This approach usually increases the complexity of the OODB systems significantly. We have discovered that our technique combines many of the advantages of previous techniques in the scope of acyclic queries, while maintaining low system complexity.[1]

The above technique can be applied to relational systems as well, although parts of it would have to be adjusted to the relational database system architecture. Virtually all relational systems do support the necessary machinery, namely tuple IDs. Current optimizers would have to be extended, which is not trivial, but the savings can be very substantial. Unfortunately, we do not yet have experimental results on relational systems.

4 Discussion

Among all the differences between CSP search and database query evaluation the single most critical feature is the effective size of the label sets and the concomitant implication that the data is stored on disk. The direct consequence has been our exploration of the true advantages of particular CSP search techniques as a function of their use of memory and the development of techniques whose impact on performance is large relative to memory usage. This has led first to the tree-tracker algorithms. Per above, using $O(n^2)$ space and without a preprocessing step, we have shown that it is possible to attain optimal performance. In TT-1 we showed that the elimination of any significant storage can be done with only a O(log n) penalty in time.

With these results on search in hand we have defined improvements in the handling of relational joins when time to the first solution is the priority. We have extended the TT-2 to the all solutions case and shown how object-oriented queries may be repre-

1 Low system complexity has practical impact to loosely coupled data access systems, e.g. the World Wide Web.

sented so as to exploit both the physical structuring of the objects as well as CSP search techniques.

It is clear that these efforts must push beyond acyclic problems. We have shown that the tree-tracker algorithms combine well with cycle cut set methods. Nevertheless, there are many other opportunities to be explored. Toward these ends we have explored the structure and advantages of higher order CSP learning methods with respect to their use of memory [2, 3]. The extension and evaluation of these formal results in the context of database queries is still open.

5 Other Open Problems

A number of other open research issues present themselves, either on their own or necessary for full fruition of this direction.

5.1 Semantic Query Optimization

In the section of querying object-oriented databases we defined a form of constraint graph which at the onset contained some directed arcs. In an OODB, these directed arcs represented navigational elements of the database. In the context of search, these directed arcs represented the fact that without any preprocessing we could assume that the constraint was directed arc consistent. We suggest that within modern relational database systems there are ample occasions when directed arc consistency can be identified. Most readily, the SQL create table primitive now includes an option to declare an attribute, or set of attributes to be a foreign key[1].The integration of other consistency constraints within databases, also a topic of this workshop, will create further opportunities for predetermining the search characteristics of a query. Given the integral relationship of directed-arc-consistency with optimal search behavior, we can, with confidence, suggest that such semantic information can have dramatic impact on the execution time of a query.

5.2 An Algebra for the Element of CSP Algorithms

Database query optimizers are strongly rooted in the relational paradigm. Such optimizers represent a query as a sequence of operators, both logical and physical, such that the sequence of operators may be permuted consistent with algebraic identity rules and/or replaced with other operators. The algebra forms a basis for correctness proofs as well as an underlying basis for software engineering of the optimizers. The integration of CSP methods with main stream databases would be facilitated greatly if we were able to define the elements of CSP methods as individual operators in an algebra. This is not just philosophical. Where we see no fundamental reason preventing CSP techniques from coexisting with existing relational techniques, even within a single query evaluation, an algebra might clearly demonstrate this to be true.

1 The foreign key defines a many to one relation among the tuples in a pair of tables such that the value of an attribute in one table, (or set of attributes which when conjoined) must appear exactly once in a column (or set of columns) in another table.

5.3 Implications for Parallel and Distributed Processing

Interoperator concurrency is a most effective way to organize both distributed and/ or parallel evaluation of queries. In our context we can think of this in terms of pipelined nested loops for a three-way join. The join of the first two relations is computed on one processor and the results streamed to a second processor where they are joined with the third relation. Given the substantial overhead required to initiate a transfer between pairs of processors the stream of data between the two processors is more often organized as a sequence of block transfers, each block containing many tuples. The techniques we described in this paper are defined at the granularity of single tuples. At this level of granularity, these ideas will have little advantage in either distributed or parallel evaluation environments.

References

1. Bayardo, R. J. and Miranker, D. P. 1994. An Optimal Backtrack Algorithm for Tree-Structured Constraint Satisfaction Problems. *Artificial Intelligence* 71(1):159-181.
2. Bayardo, R. J. and Miranker, D. P. 1995. On the Space-Time Trade-Off in Solving Constraint Satisfaction Problems. In *Proc. 14th Intl. Joint Conf. on Artificial Intelligence*, 558-562.
3. Bayardo, R. J. and Miranker, D. P. 1996. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proc. of the 13th National Conf. on Artificial Intelligence*, 298-304.
4. Bayardo, R. J. and Miranker, D. P. 1996. Processing queries for first few answers. In *Proc. of the Fifth International Conference on Information and Knowledge Management*, 45-52.
5. Bernstein, P. A. and Chiu, D.-M. W. 1981. Using semijoins to solve relational queries, *J. ACM* 28(1), 25-30.
6. Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41(3):273-312.
7. Gyssens, M., Jeavons, P. G. and Cohen, D. A. 1994. Decomposing constraint satisfaction problems using database techniques, *Artificial Intelligence* 66, 57-89.
8. Samoladas, V. and Miranker, D. P. 1996. Loop optimizations for acyclic object-oriented queries. Technical Report TR96-10, Dept. of Computer Sciences, University of Texas at Austin Available at http://www.cs.utexas.edu/users/miranker/ooq.ps.