DB_CSP: A Framework and Algorithms for Applying Constraint Solving within Relational Databases

Chuang Liu1, Ian Foster1,2
1University of Chicago, 1100 E. 58th street
Chicago, IL, 60637, USA
chliu@cs.uchicago.edu
2Mathematics and Computer Science Division, Argonne National Laboratory,
Argonne, IL 60439, USA

Abstract. We examine combinatorial search problems that arise when evaluating combinatorial queries in relational database systems. Such queries request tuples from multiple relations that satisfy a conjunction of constraints on tuple attribute values, and are not executed efficiently by current database systems. We propose an approach by which such problems can be addressed with constraint solving technology. First, we formalize combinatorial queries in databases as a new type of constraint satisfaction problem: what we call Database Constraint Satisfaction Problems (DB_CSPs). Then, we develop algorithms for solving DB_CSPs that integrate database data management technology with conventional CSP solving techniques. Our algorithms enable efficient domain refinement for tuple-valued data and minimize I/O operations when relations are stored on disk. Finally, we present experimental studies that both quantify the performance of our approach and permit comparison with a conventional database system. We show that our approach is superior for solving complex combinatorial queries.

1 Introduction

The relational data model and relational database technology have achieved great success as a means of managing, and implementing queries on, large amounts of data. However, new application domains and decision procedures lead to new classes of query that are not handled efficiently by today’s database systems. In the combinatorial queries that we consider in this paper, a request for tuples from multiple relations that satisfy a conjunction of constraints on tuple attribute values leads to a combinatorial search problem.

To illustrate how combinatorial search problems can arise, we consider an Internet information system that maintains a relation C storing descriptions of compute resources, a relation N storing descriptions of network resources, and a relation S storing descriptions of storage resources. Relation C has attributes cpuSpeed, price, and os; N has attributes bandwidth and price; and S has attributes ioSpeed and price. In the combinatorial query shown in Figure 1, a user requests a compute, network, and storage resource that collectively satisfy deadline (10 minutes) and budget ($5) constraints for an application that first calculates on a compute resource (time 10/cpuSpeed), then transmits results over a network resource (10/bandwidth), and finally stores results on a storage resource (4/ioSpeed). Note that the query condition is a conjunction of two arithmetic constraints on attributes from three relations.
SELECT * FROM C, N, D WHERE
C.price + N.price + D.price <= 5
AND
10/C.cpuSpeed + 10/N.bandwidth + 4/D.ioSpeed < 10

Figure 1. An example combinatorial query, showing the three relations and query.

Database systems implement such combinatorial queries by a set of pair-wise join operators. However, optimization techniques for pair-wise joins, such as sorted-join and hash-join, deal only with simple constraints such as equality or inequality comparisons between two attributes, and cannot, in general, process combinatorial queries efficiently. Thus, evaluation involves a complete traversal of the search space.

Combinatorial search has of course been thoroughly studied by researchers in constraint programming [8,13,15], who have developed efficient search algorithms that avoid evaluating all possible states by pruning unsatisfiable results from the search space. Thus, the question that we ask in this paper: can we exploit combinatorial search algorithms from constraint programming to optimize the execution of combinatorial queries in relational databases?

An investigation of this question is not straightforward. Algorithms developed for constraint programming apply only to simple data types and assume that all data is stored in memory. In contrast, combinatorial queries involve searches for tuples that may be stored on disk. Furthermore, it is important that new implementation techniques can be incorporated easily into existing database systems. Nevertheless, we show that it is indeed feasible to overcome these challenges and to integrate highly optimized data management mechanisms from databases with efficient combinatorial search algorithms from constraint programming.

Our approach to this problem proceeds in three stages. First, in Section 3, we show how combinatorial queries can be modeled as a new class of constraint satisfaction problem: what we call Database Constraint Satisfaction Problems (DB_CSPs). Constraint satisfaction problems (CSP) are often used to model combinatorial search problems, and numerous search algorithms are available. However, searching large databases introduces new challenges. We define DB_CSP to model these challenges.

In Section 4, we propose an algorithm for solving DB_CSPs. We address the difficulties introduced by the large value domains encountered in DB_CSPs by designing a value domain representation and a constraint-solving algorithm that together enable both compact storage and efficient execution. We use constraint-solving technology to control the search process, and database indexing technology to implement data manipulation operations such as domain refinement.

In Section 5, we report on experiments that allow us to quantify the performance characteristics of our approach, and to compare this performance with that of a conventional database system. We find that our approach is significantly more
efficient than conventional database join algorithms when handling queries with complex join conditions and small selectivity.

2 Related Work

Combinatorial search problems arise in many contexts, and we find a wide variety of approaches to their solution. Here we review the most relevant previous work.

In relational database management systems, combinatorial search is implemented by a multi-join operation, which combines information from multiple relations to determine if an aggregation of tuples from these relations satisfies search conditions. However, existing join algorithms, such as nested-loop, sorted-join, and hash-join [6], apply only to simple constraints. Heuristic combinatorial search algorithms [5,11] have been developed, but apply only to particular types of constraints.

Constraint programming [8,13,15] has been used to formulate combinatorial search problems in areas such as scheduling, routing, and timetabling that involve choosing from among a finite number of possibilities. Correspondingly, constraint-solving algorithms have been developed in several research communities, including node and arc consistency techniques [9,16] in artificial intelligence, bounds propagation techniques [13] in constraint programming, and integer programming techniques [17] in operations research. However, although these algorithms solve combinatorial search problems efficiently, implementations have been within modeling languages [1,3,4,10] and have not been used to manage queries on large data sets, as a database system does.

Constraint databases [12,14] extend the expressiveness of conventional relational databases by using constraints to model relations with an infinite number of data. In contrast, we use constraints to model the search process on conventional databases, with a view to improving the performance of combinatorial queries by incorporating constraint-solving algorithms into a database system.

3 Database Constraint Satisfaction Problems

The essence of our approach is to treat combinatorial search on databases as a constraint satisfaction problem and apply constraint-solving technologies to implement search and selection functions.

A combinatorial search on a database is triggered by a combinatorial query: a query that specifies selection conditions for multiple tuples from one or several relations. The selection condition is a conjunction of primitive constraints: comparisons (e.g., =, <, >) between arithmetic expressions or string values. Arithmetic expressions are arithmetic operators (such as +, -, *, /) on integer or real type variables or constants.

Existing constraint programming languages [2,3] usually formalize a combinatorial query as a constraint satisfaction problem (CSP) by mapping each database relation to a constraint, and associating a variable to each attribute appeared in the query condition. For example, we can express the search problem in Figure 1 as the following CSP, in which the constraint c, n, and d denote the relations.

\[
C\_price + N\_price + D\_price \leq 5 \land \\
10/C\_cpuSpeed + 10/N\_bandwidth + 4/D\_ioSpeed < 10 \land \\
C\_price \in \{1, 2, 3, 4, 5, 8\} \land C\_cpuSpeed \in \{1, 2, 3, 4, 6\}\ldots \land D\_ioSpeed \in [...] \land
\]
We argue that this method may have following limitations when dealing with queries on large size database.

- The constraint solver usually needs to read in tuples in database relations and transform them to a constraint in memory before solving a query. It may cause unnecessary I/O operations because a lot of tuples in a database relation won’t lead to any solution.
- The value domains of variables are often too large to maintain in memory, and thus domain refinement may cause expensive I/O operations. Database system may process multiples combinatorial queries concurrently that makes the memory efficiency of constraint solving more important.
- Using variables to represent attributes of a tuple may cause extra constraint checks in constraint propagation. For the constraint \( c(C\_tuple\#, C\_Price, C\_cpuSpeed) \) in previous example, if several values are removed from the domain of \( C\_Price \), we need to check the constraint \( c \) in order to remove values from the domain of \( C\_cpuSpeed \).

Based on these considerations, we propose to formalize a combinatorial query as a constraint satisfaction problem (CSP) by associating a variable with every required tuple. The domain of each variable comprises all tuples in the associated relation. Constraints on variables describe selection conditions. We call such a CSP a DB_CSP. For example, we can express the search problem in Figure 1 as the following CSP, in which the variables \( c \), \( n \), and \( d \) denote the required tuples.

\[
c(\_\_, C\_price, C\_cpuSpeed) \land n(\_\_, N\_bandwidth, N\_price) \land d(\_\_, D\_ioSpeed, D\_price) \land c(1, 6, 1) \land \ldots \land n(1,1,1) \land d(1, 6, 1) \land \ldots
\]

**Definition 1:** A database constraint satisfaction problem (DB_CSP) is a CSP in which variable domains are database relations.

Any valid combinatorial database query can be expressed as a DB_CSP as just shown. DB_CSPs share some common features different from conventional CSPs, as follows.

- The values of variables are tuples with multiple attributes.
- Constraints on variables are expressed as constraints on the attributes of those variables.
- The number of variables is typically not large, but the domain of each variable may be extremely large.

These features lead us instead to pursue an approach to the solution of DB_CSP problems that focuses on their unique characteristics.

### 4 Solving DB_CSP Problems

Having modeling combinatorial queries as DB_CSPs, we now investigate how constraint-solving algorithms can be used to implement the required combinatorial search. In considering this question, we must recognize that database systems incorporate highly optimized mechanisms for managing large numbers of record-like data. These observations motivate us to develop an implementation of constraint solving that incorporates techniques from database systems to optimize performance.
In brief, we use a constraint-solving algorithm to control the search process and database index technology to implement data manipulation operations such as domain refinement, which removes values from variable domains.

Considering the large size of variable domains of a DB_CSP, we require a value domain representation and constraint-solving algorithm that together enables both compact storage and efficient execution for multiple concurrent or successive DB_CSPs, so that we can perform domain refinement efficiently without requiring numerous I/O operations and high memory costs.

4.1 Representation of Variable Domains

We represent a variable domain in terms of (a) the relation that stores tuples in the variable domain and (b) a set of simple constraints on variables that we term the *filter*. The variable domain then corresponds to all tuples from the relation that satisfy the constraints in the filter. All DB_CSPs on a database share the same relations, but each has its own filter. The constraint-solving algorithm refines the domains of variables by adding constraints to the filter. Thus, relations do not change during constraint solving, ensuring that different DB_CSPs do not interfere with each other.

![Figure 2. Representation of the filter for the example in Figure 1.](image)

We implement the filter as a set of symbol tables, one per variable appearing in the DB_CSP. These symbol tables maintain information about the attributes appearing in the DB_CSP. Every symbol table entry contains an attribute name and known basic constraints on this attribute. We maintain two types of *basic constraints* on attributes in the symbol table: *assignment* constraints such as \( \text{attr} = N \) and *range* constraints such as \( \text{attr} > N \). We use these two types of constraint because they can be easily used by database index techniques to locate values belonging to a value domain from a relation. Initially, attributes are described by range constraints representing attribute value bounds derived from relations. For example, the value domain for \( c \) as specified in Figure 1 provides the constraints \( c.\text{cpuSpeed} \geq 1 \) and \( c.\text{cpuSpeed} \leq 6 \) (Figure 2).

A filter requires little space and thus can be kept in memory. The basic constraints contained in the filter can be seen as a summary of the variable domains. As shown in Section 4.2, this information is both used to refine the variable domain and is updated by the constraint-solving algorithm as the variable domain is reduced.

Values in variable domains are tuples satisfying the constraints in the filter. A naive way to pick a value from variable domain is to read tuples repeatedly from the relation and evaluate the constraints in the filter until a satisfying tuple is found. A more efficient approach is to build indices on relations and use those indexes to locate satisfying tuples. A filter may contain multiple basic constraints on attributes and thus we use multi-dimensional indexing techniques from database systems. Different types of multi-dimensional index are available (e.g., grid file, bitmap) with efficiencies that depend on the data in the relations. In this paper, we use multi-key indexing to illustrate the use of indices for solving DB_CSPs.
For every variable, we build a multi-key index [6] on those of its attributes that appear in the DB_CSP. Figure 3 presents an example that illustrates the approach. The root of the tree is an index for attribute \( \text{cpuSpeed} \). The index could be any type of conventional index that support range queries, such as a B-tree. The index associated with each of its values is a pointer to another index on attribute \( \text{price} \). If \( V \) is a value of attribute \( \text{cpuSpeed} \), then by following value \( V \) and its pointer we reach an index into the set of tuples that have \( \text{cpuSpeed} \) equal to \( V \) and any value for attribute \( \text{price} \).

![Diagram of multi-key indexes for attributes cpuSpeed and price in relation C of Figure 1.](image)

Figure 3. Multiple-key indexes for attributes cpuSpeed and price in relation C of Figure 1.

The multiple-key index works well for locating tuples with range constraints and assignment constraints, as long as the indexes themselves support range queries on their attributes. To locate tuples based on range constraints, we use the root index and the range of the first attribute to find all sub-indexes that might contain answer points. We then search each sub-index, using the range specified for the second attribute.

### 4.2 Clustering Value Domains

The search space of a CSP is the Cartesian product of the size of its variable domains. Variable domains may be large and thus even an efficient constraint-solving algorithm may still take a lot of time to find results.

We observe that many tuples have similar attribute values. For a DB_CSP with constraints on a subset of the attributes of a required tuple, tuples with the same value on those attributes can be viewed as identical even if they have different values for other attributes. For convenience, we call these tuples a *cluster*. If one tuple in a cluster will not lead to a solution, no tuple in the cluster will do so. On the other hand, if we find a solution containing one tuple in a cluster, we can get new solutions by selecting any other tuple in the cluster. Because we need to check only one tuple per cluster during the search process, the size of the search space is reduced to the Cartesian product of the cluster sizes.

This approach of clustering value domains may reduce the search space of a DB_CSP remarkably if attributes appearing in the constraint have a value chosen from a finite domain. For example, computers might have an attribute *operating system*, with permitted values of only *Linux* and *Windows*; thus, only two clusters are formed for this attribute.

For every DB_CSP, we preprocess variable domains such that tuples are organized into clusters. Indices on variable domains (relations) make it easy to cluster value domains. For the example in Figure 3, we check the five indexes on attribute \( \text{price} \). For every index, tuples with the same value on attribute \( \text{price} \) belong to the same cluster. Thus, tuples 1 and 6 in Figure 3 belong to a cluster.
4.3 Solving Algorithm

There are many CSP solving algorithms, with different characteristics [13]; our challenge is to build on this technology to develop a constraint-solving algorithm with I/O efficiency in addition to computational efficiency. To allow for variable domains stored on disk, we need a constraint-solving algorithm that does not need to access variable domains frequently and that can use database index techniques to achieve efficient implementations of data manipulation operations. We only consider a complete algorithm in this paper because, in many cases, database queries require that the search algorithm either return exact results or determine that the query is unsatisfiable. However, our approach can be easily applied to incomplete algorithms.

Based on these considerations, we develop a solving algorithm that combines complete backtracking search with the consistency techniques shown in Figure 4. This algorithm uses backtracking as its skeleton, and incorporates constraint consistency techniques to prune the search space during the backtracking process.

```plaintext
// C is a constraint of form c_1 \land \ldots \land c_n
// D is variable domains
// v is a variable
// D(v) is domain of variable v

Search_Algorithm(C, D)
1. If all variables in C are bound to a value
2. If constraint C holds
3. Output a solution
4. Else
5. Return // no solution for this constraint
6. If(!Consistency_Algorithm(C, D))
7. Return // no solution for this constraint
8. Choose variable v with the minimal value domain
9. For each d \in D(v)
10. Assign d to v
11. Search_Algorithm(C \land v=d)
12. Return

Consistency_Algorithm(C, D)
13. While (C is not node or bounds consistent)
14. For i = 1 to n do
15. a = NodeConsistencyAlgorithm(c_i, D)
16. b = BoundsConsistencyAlgorithm(c_i, D)
17. If(!a || !b) return false
18. Return true

Figure 4. The DB_CSP solving algorithm.
```

The function `Search_Algorithm` in Figure 4 uses backtracking to return all solutions for a constraint C. If all variables in C are instantiated, the function returns a solution if C holds (lines 3-4). Otherwise, the function assigns a value to an undetermined variable with minimum domain size and searches for solutions consistent with this assignment (lines 10-13).

We also use the node consistency algorithm and the bounds consistency algorithms to refine variable domains (function `Consistency_Algorithm` in Figure 4). These algorithms need far less information from value domains when refining the search space than do, for example, the arc-consistency and hyper-arc consistency algorithms.
which must repeatedly read values. (As noted above, reducing accesses to value domains is important when values are stored on disk.) The node consistency algorithm can use database index techniques to refine value domains efficiently, while the bounds consistency algorithm needs only bounds information for value domains.

4.4 Consistency Algorithms

Consistency algorithms are usually developed for basic data types. Here we propose new consistency algorithms for structured data that refine variable domains without accessing tuples in the relation. Thus, if tuples are stored in disk, our algorithms are more I/O efficient than conventional node and bound consistency algorithms.

In a DB_CSP, constraints on variables are expressed by a set of primitive constraints on attributes of variables. We define attribute node consistent and attribute bound consistent as follows.

**Definition 2:** A primitive constraint $c$ is attribute node consistent [13] if either the constraint involves multiple variable attributes or if all values in the domain of the single attribute are a solution of $c$. The domain of an attribute is the collection of attribute values appeared in the variable domain.

$$\begin{align*}
\text{price} & \geq 1, \leq 6 \\
\text{bandwidth} & \geq 1, \leq 8 \\
\text{cpuspeed} & \geq 1, \leq 6 \\
\text{ioSpeed} & \geq 1, \leq 5 \\
\text{price} & \geq 1, \leq 8 \\
\text{cpuspeed} & \geq 1, \leq 6 \\
\text{price} & \geq 1, \leq 5
\end{align*}$$

**Figure 5.** The variable domains of Figure 1 after the execution of the node consistency algorithm on the constraint $c.price \leq 5$.

Given a primitive constraint $c$ and a variable domain, the attribute node consistency algorithm removes values from the variable domain until $c$ is attribute node consistent. As discussed in Section 4.1, we represent the variable domain by a filter and a relation. The attribute node algorithm reduces the variable domain by adding the unary constraint on one attribute to the filter. Thus, the refinement does not need to access the relation. To illustrate, consider the example in Figure 2 and a primitive constraint $c.price \leq 5$. The variable domain after the attribute node consistency algorithm is shown as Figure 5. In the symbol table for variable $c$, the range constraint on the attribute price has been changed to $1 \leq c.price \leq 5$, i.e., the combination of original constraint $1 \leq c.price \leq 8$ and $c.price \leq 5$.

The conventional bound consistency algorithm works on arithmetic constraints with integer or real type variables. However, in DB_CSPs the values of variables are tuples with multiple attributes. Thus we define attribute bound consistency as follows.

**Definition 3:** An arithmetic primitive constraint $c$ is attribute bounds consistent [13] if for each variable attribute $attr$ that appears in this constraint there is:

- An assignment of values $d_1$, $d_2$, …, $d_k$ to the remaining variable attributes in $c$, such that $\min(attr) \leq d_i \leq \max(attr)$ for each $d_i$, and $\{\text{attr}=\min(attr), \text{attr}_1=d_1, \text{attr}_2=d_2, ..., \text{attr}_k=d_k\}$ is a solution of $c$; and

- An assignment of values $d_1', d_2', ..., d_k'$ to the remaining variable attributes in $c$, such that $\min(attr) \leq d_i \leq \max(attr)$ for each $d_i$, and $\{\text{attr}=\min(attr), \text{attr}_1=d_1', \text{attr}_2=d_2', ..., \text{attr}_k=d_k'\}$ is a solution of $c$.

Where $\min(attr)$ (or $\max(attr)$ ) represents the minimal (maximal) value of attribute $attr$ in the domain of the corresponding variable.
Given a constraint \( c.price + n.price + d.price \leq 5 \) and variable domain in Figure 1, we can get the new range of attribute price of variable \( c \) by
\[
c.price \leq 5 - n.price - d.price \leq 5 - \text{Min}(n.price) - \text{Min}(d.price) = 3
\]
We can get the new range of attribute price of variables \( d \) and \( n \) in the same way. We refine the variable domain using this new range by adding it to the filter. We show the variable domains that are attribute bounds consistent for constraint \( c.price + n.price + d.price \leq 5 \) in Figure 6. Note that the range of the attribute price has been changed for each of the three variables.

<table>
<thead>
<tr>
<th>( c )</th>
<th>( d )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>( \geq 1, \leq 3 )</td>
<td>price</td>
</tr>
<tr>
<td>( \geq 1, \leq 6 )</td>
<td>bandwidth</td>
<td>( \geq 1, \leq 8 )</td>
</tr>
</tbody>
</table>

Figure 6. The variable domains of Figure 1 after the execution of the bounds consistency algorithm on the constraint \( c.price + n.price + d.price \leq 5 \).

The bounds consistency algorithm needs only range information for attributes that are maintained in the filter. Because we can keep the filter, the algorithm requires no additional I/O operations to read tuples in the relations.

### 4.5 Assignment Operation

During backtracking, the solver picks a value from the variable domain and assigns this value to the variable (line 11). The variable domain is represented as a filter and relations (Section 4.1). The values of a variable are tuples in relations and thus we use the basic constraints in the filter to pick a value that belongs to the variable domain. Recall that basic constraints are range constraints and assignment constraints on one attribute; thus selection operations can be implemented efficiently by using database index techniques. For example, given the filter shown in Figure 6, when picking a value for variable \( c \) we use constraints in the corresponding symbol table (\( 1 \leq price \leq 3 \) and \( 1 \leq \text{cpuSpeed} \leq 6 \)) to query the relation \( C \). The multi-key index shown in Figure 3 allows the database to locate tuples that satisfy this selection condition efficiently.

Assigning a tuple to a variable involves replacing attributes with values appearing in the constraint. We also update the symbol table in the filter, which may cause bounds inconsistency and thus trigger a new domain refinement operation.

### 5 Performance Evaluation

We now present an experimental evaluation of the combinatorial query performance of both our DB_CSP solving algorithm and a conventional database system.

#### 5.1 The Benchmark Query \( Q \)

We first define a simple combinatorial query \( Q \) with parameters that can be used to vary its execution complexity and selectivity. This query operates on three relations \( A, B, \) and \( C \), defined according to relations \( TENKTUP1 \) from the Wisconsin Benchmark [7]. In order to simulate complex combinatorial queries, \( Q \) comprises two constraints, each involving three relations.

\[
Q: \text{SELECT * FROM } A, B, C \\
\text{WHERE} \\
A.K50 + B.K50 + C.K50 < N_1 \text{ AND}
\]
This query specifies a request for three tuples from relations A, B, and C such that the sum of their attribute K50 is less than a constant \( N_1 \) and the sum of their attribute K200 is more than a constant \( N_2 \). The value of attribute K50 is an integer uniformly distributed between 1 and 50, and the value of attribute K200 is an integer uniformly distributed between 1 and 200.

Notice that varying the sizes of A, B, and C changes the size of the problem to be solved, while varying the parameter(s) \( N_1 \) and \( N_2 \) changes the number of results obtained, i.e., the selectivity of Q. We vary both the relation sizes and \( N_2 \) in our experiments to obtain queries with various problem sizes and selectivities.

5.2 Experimental Studies

We use three performance metrics to evaluate the search efficiency of both our algorithm and conventional database search algorithms: the total number of I/O operation performed, the number of constraint check operations per result, and the elapsed time required to obtain query results.

- **I/O operations.** It is widely used to measure the performance of a database search. Because the algorithm of Section 4.3 reads a tuple only when it extends an intermediate result and we perform a constraint check for every intermediate result, the number of read operations is equal to the number of constraint check operations. Thus, we use the number of constraint check to measure I/O operations.

- **Constraint check operations per result.** The number of query results returned by a search represents a lower bound on the number of constraint check operations that a search algorithm must perform. We use the ratio of the number of constraint checks performed to the number of results obtained as a normalized measure of how close an algorithm is to this bound.

- **Elapsed time.** The final metric is the time between query submission and the return of results.

In our experiments, we compare the elapsed times of the implementation of our DB_CSP algorithm and the algorithm implemented in the Postgres relational database system (Version 7.3), which we choose for its accessibility. Postgres implement the search process by pair-wise join operations. We recognize that Postgres may perform less well than other database systems, but argue that because different database systems use similar execution algorithms, our results extend broadly.

We conducted all experiments on an IBM T20 with a single Pentium III 700MHz processor and 128 MB memory, running Suse Linux 7.2.

5.3 Changing Relation Size

In this first set of experiments, we vary the number of tuples in each of the relations A, B, and C from 125 to 64,000 while fixing the join selectivity by setting \( N_1 = 6 \) and \( N_2 = 520 \). Figure 7 shows the elapsed time measured for both our DB_CSP algorithm and Postgres. Our algorithm is faster than Postgres by from two to four orders of magnitude, with the difference becoming larger as the relation size increases.
The reason for this difference in execution times is made clear by Figure 8, which shows the number of I/O operations performed. We see that Postgres performs six orders of magnitude more constraint checks than our algorithm. This difference arises because Postgres uses an execution plan that creates all combinations of tuples and checks them one by one. For a query with low selectivity, this strategy may lead to many unnecessary computations. In contrast, our DB_CSP algorithm can use constraints in Q to filter intermediate results and thus performs far fewer constraint checks. This improvement in efficiency is achieved by using consistency techniques to prune parts of the search space containing no solutions.

We also see that, for our algorithm, the execution time and the number of constraint checks both increase less rapidly with relation size than in the case of Postgres. For example, the execution time of Q does not change much when the relation size increases from 32,000 to 64,000. The reason is that our algorithm uses clustering (Section 4.2) to organize tuples into clusters and considers only one tuple per cluster during the search process. Query Q specifies constraints on attribute $K_{50}$ and $K_{200}$ that are used to cluster value tuples. $K_{50}$ picks values from 1 to 50 and $K_{200}$ picks values from 1 to 200; thus, the number of possible combinations of $K_{50}$ and $K_{200}$ is 10,000. After clustering, no matter what the relation sizes, the variable domains of the DB_CSP are smaller than 10,000, which puts an upper bound on the execution time of query Q.

Figure 9 shows the ratio of the number of constraints checked to the number of query results. Postgres performs around 2,100,000 constraint checks per result. In contrast, our DB_CSP algorithm performs only a few constraint checks per result. Indeed, for large relations this number declines to little more than one, which is the lower bound for any search algorithm.

In summary, our algorithm is more efficient at evaluating queries with arithmetic constraints such as Q because it performs many fewer constraint checks. We are confident that similar results would be obtained for other similar queries.
5.4 Changing Selectivity

In this second set of experiments, we fix the number of tuples in relations A, B, and C to 500 and vary the join selectivity by varying the constant $N_2$ in the selection condition of Q.

Figure 10 compares the elapsed times to evaluate the query. When query selectivity is small, our DB_CSP algorithm has a large performance advantage relative to Postgres. As selectivity increases, the difference between the two systems decreases. The reason is that as the number of results increases, the search space becomes filled with more results, leaving fewer opportunities for a more efficient search algorithm to improve search performance. In an extreme case in which all combinations of tuples in the joined relations satisfy the selection conditions, our algorithm will have the same complexity as conventional database join operations, because both strategies must check all combinations. In this case, the simplest algorithm has the best performance because it avoids the cost of backtracking and the consistency algorithm.

In summary, our results show that our algorithm performs particularly well when combinatorial queries are complex and query selectivity is small.

6 Summary and Future Work

Combinatorial queries in database systems seek tuples from multiple relations with a particular relationship. Existing relational database systems can express combinatorial queries but cannot execute them efficiently. To improve on this situation, we show how combinatorial queries can be modeled as a new type of CSP problem, DB_CSP, and design and implement a combinatorial search algorithm that integrates constraint-solving techniques with database search techniques. Experimental results show that our algorithm has significant performance advantages relative to traditional database technology when solving combinatorial queries with complex selection conditions, particularly when query selectivity is low. Our algorithm can also be incorporated easily into existing relational database systems.
Acknowledgements
This work was supported by the Grid Application Development Software project of the NSF Next Generation Software program, under Grant No. 9975020. We are grateful to Alvaro Fernandes, Mike Franklin, Svetlozar Nestorov, Anne Rogers, Matei Ripeanu, Alain Roy, and Lingyun Yang for comments on a draft of this paper.

References