Efficient Relational Joins with Arithmetic Constraints on Multiple Attributes

Chuang liu
University of Chicago
chliu@cs.uchicago.edu

Lingyun Yang
University of Chicago
lyang@cs.uchicago.edu

Ian Foster
Argonne National Lab
forster@mcs.anl.gov

Abstract

We introduce and study a new class of queries that we refer to as ACMA (arithmetic constraints on multiple attributes) queries. Such combinatorial queries require the simultaneous satisfaction of arithmetic constraints on three or more attributes from different relations, and thus often involve expensive multi-join operations. Building on techniques from constraint programming, we develop preprocessing methods, algorithms, and a new constrained join operator that allow ACMA queries to be evaluated efficiently within a conventional relational database engine. We present the results of a careful performance evaluation of both our new approach and the conventional nested-loop join algorithm. Measurements of tuples read, intermediate tuples generated, and execution time shows that our approach achieves superior performance for ACMA joins.

1. Introduction

As database technologies are applied in ever more diverse fields, we encounter the need to perform new classes of queries: including, as we consider here, combinatorial queries that seek to identify tuples from multiple relations with particular characteristics. Frequently, the combinatorial query condition in such queries is expressed by arithmetic constraints on three or more attributes from these relations: what we refer to as ACMAs (arithmetic constraints on multiple attributes).

Queries involving ACMAs cannot be implemented efficiently using the conventional approach of composing pair-wise joins, as the evaluation process requires those multiple query conditions be satisfied simultaneously, and each condition refers to more than two attributes in different relations. Thus the question that we address in this paper: can we introduce support for such queries into relational database systems without requiring significant major modifications to the underlying database engine?

Example *Suppose a database with the following relations*

- supplier(sid, price), *denoted* S, *contains for every raw material supplier, the id of the supplier and the price for every unit of raw material.*
- preProcessFactory(pid, rate, price), *denoted* P, *contains for each raw material preprocessing factory, the factory id, the quantity of preprocessed material made from an unit of raw material, and the price for processing of an unit of raw material*.
- componentFactory(cid, rate, price), *denoted* C, *contains for each components manufacturing factory, the factory id, , the number of components made from an unit of preprocessed material, and the charge for the manufacture of every 1000 components.*
- delivery(did, price, rate), *denoted* D, *details the ids of the delivery companies, the unit price for the delivery of every 1000 components, and the percentage of components delivered without damage.*

Suppose the production process of a component has multi-stages: purchase of raw material, raw material preprocessing, components manufacturing, and components delivery. A component provider wants to assemble a raw material supplier, a material preprocessing factory, a component manufacturing factory, and a delivery company together to provide components. The component provider requires that the production process provides more than 900 components under a total cost 16 for every unit of raw material purchased. The request is described by a SQL query in Figure 1.

Note that the query condition is a conjunction of two arithmetic constraints on attributes from four relations. The first constraint is on the total price that is the sum of the cost in every stage. The second

constraint is on the number of components delivered by this production process. ■

S

tuple#	sid	price
1	1	1
2	2	3
3	3	5
4	4	7
5	5	9

P

tuple#	pid	rate	price
1	1	0.8	3
2	2	0.9	4
3	3	0.95	4
4	4	0.96	4
5	5	0.98	10

C

tuple#	cid	rate	price
1	1	900	5
2	2	950	6
3	3	920	5
4	4	980	7
5	5	980	10

D

tuple#	did	rate	price
1	1	0.9	6
2	2	0.95	7
3	3	0.97	8
4	4	0.98	9
5	5	0.99	10

```
SELECT *  FROM S, P, C, D WHERE
   S.price +  P.price + P.rate * C.rate * C.price
         + P.rate * C.rate * D.price <= 16          AND
   P.rate * C.rate * D.rate >=  900
```

Figure 1. An example of an ACMA join query, showing four relations and the query

Similar queries also arise in other areas including utility computing, decision support, scientific computing, and e-commerce [19]. For example, in e-commerce, a traveler might request airline tickets, a hotel within ten miles of the airport, and a rental car—all with a total cost under a specified budget.

Definition 1 (ACMA Query) *An* arithmetic constraint *is an equation or inequality on real or integer attributes and/or constants and involving arithmetic functions (such as +, -, *, /) of attributes and constants defined in database systems. An* arithmetic constraint on multiple attributes *(ACMA) is an arithmetic constraint that contains more than two attributes from more than one relation. An* ACMA join *is a join operation whose join condition is a conjunction of one or more ACMAs. We call each ACMA a* primitive constraint. ■

Current database systems implement ACMA joins by a set of binary join operators. Commonly used optimization techniques for pair-wise joins, such as sorting join and hashing join, cannot be applied to this kind of query because they work only for simple equality or inequality comparison join conditions. Nested-loop join is the only well-known technique for complex join conditions [23]. Furthermore, for an ACMA constraint such as $P.rate * C.rate * D.rate \geq 900$, current join operators cannot test the satisfiability of an intermediate tuple until all variables have been determined. A consequence of this inability to remove intermediate tuples that will not lead to any results is that the query evaluation process must ultimately evaluate the Cartesian product of all tuples of all join relations, which may lead to high memory and computation costs.

ACMA join can also be viewed as combinatorial search [2], a problem that has been thoroughly studied by researchers in constraint and mathematical programming [9,15,20]. Efficient combinatorial search algorithms have been developed that avoid evaluating all possible states by pruning unsatisfiable results from the search space. However, these algorithms assume that all data are stored in memory and that the required values have some basic data types, such as integers. No work has been done to apply these techniques to the problem of locating tuples from multiple tables.

In this paper, we introduce a new approach to ACMA joins that allows search space refinement technologies from constraint programming to be applied within the context of a database system. Specifically, we introduce a new constrained join operator, an extended version of the current nested-loop operator that is guided in its operations by bounds information summarizing the finite value domains of its target relations. This bounds information consumes little memory and allows the new operator to guide the search process rapidly to a specified number of results (or to failure) from the relations being joined. We also show how this new operator can be introduced into query plans along with additional selection operators to achieve significant reductions in both tuples read and intermediate tuples generated relative to conventional operators and plans.

The major contributions of the paper are as follows.

- We define a new class of constraint satisfaction problems, DB_CSP, that models a set of challenging combinatorial search problems arising in database systems: ACMA joins. Constraint satisfaction problems (CSP), originally defined in the constraint programming area, are often used to model combinatorial search problems, and numerous search algorithms are available. However, searching on large databases introduces new challenges. We define DB_CSP to model these challenges.
- We design a combinatorial search algorithm that can handle DB_CSP problems, and show that this algorithm can be integrated into existing database join operators via the definition of query plans based on a new preprocessing operation and *constrained join operator*. We implement these techniques in a prototype system.
- We conduct experiments that allow us both to quantify the performance characteristics of the constrained join operator and to compare those characteristics with existing join operators in a relational database system. Our results show that our extensions can achieve order-of-magnitude performance improvements for ACMA joins.

The rest of this paper is structured as follows. Section 2 describes related work. In Section 3, we explain how ACMA join queries can be expressed as a class of constraint satisfaction problem with specific properties, examine the properties of these DB_CSP problems, and introduce our algorithm for solving such problems. In Section 4, we describe our implementation of this algorithm. In Section 5, we introduce a query that we use in Section 6 to compare the performance of our join algorithm with existing join algorithm. We conclude and outline our plans for future work in Section 7.

2. Related work

The problem of implementing ACMA join is closely related to the multi-join problem. Multi-joins are usually implemented by a set of pair-wise operations [3,5,18]. However, this pair-wise strategy does not work well for ACMA join queries.

Algorithms proposed by Fagin et al. [4] and Ilyas et al. [11] for top-k queries can be extended to implement queries with a constraint on the value of a monotone function, such as $A.attribute_1 + B.attribute_2 + C.attribute_3 > N$. They sort all join relations based on the value of attributes in the constraint, and then check the combination of tuples in the order of the value of the monotone function. However, this method can only use one constraint to guide the search process, which is not efficient when there are multiple constraints. In addition, this method is restricted to constraints on monotone functions.

Agarwal et al [1] address queries with linear constraints, and Guha et al. [7] address queries with aggregation constraints. However, their work is only valid for queries on one relation.

Semantic query optimization [8,12] exploits the knowledge about relations, tuples in relations, and various constraints associated them to translate a query condition into a semantically equivalent query condition that can be processed more efficiently. In this paper, we present a systematic scheme to rewrite ACMAs into query conditions that can be processed more efficiently. Besides, we introduce a mechanism to predict if an intermediate tuple will lead to a final result that helps reduce the number of intermediate tuples created during the join operation, thus improve the query performance.

Combinatorial search problems in areas such as scheduling, routing, and timetabling that involve choosing from among a finite number of possibilities have been formulated as constraint satisfaction problems [9,15,20] Constraint-solving algorithms have been developed in several research communities, including node and arc consistency techniques [10,22] in artificial intelligence, bounds propagation techniques [15] in constraint programming, and integer programming techniques [24] in operations research.

Vardi and Kolaitis [13,21] prove the intimate relationship between constraint satisfaction problems and database queries. So-called constraint database extended conventional relational databases to model relations with an infinite number of data [16]. However, no work has been done to study how to use constraint-solving algorithms to manage complex queries such as ACMA joins on conventional databases.

3. Supporting ACMA join operations

An ACMA join is triggered by a query that specifies arithmetic constraints on multiple tuples from several relations in the selection conditions. An example is shown in Figure 1. The essence of our approach is to treat ACMA join operation as a constraint satisfaction problem and apply constraint-solving technologies to implement search functions.

3.1 Background on constraint satisfaction

The following definition is from Marriott and Stuckey [15].

Definition 2 (Constraint Satisfaction Problem) *A* constraint satisfaction problem *(CSP) consists of a constraint C (the* selection conditions*) over variables* $x_1..x_n$ *and a domain D, that maps each variable* x_i *to a finite set of allowed values,* $D(x_i)$*. The problem is to find assignments to these variables such that the constraint C is satisfied. The complete CSP is understood to be a constraint* $C \wedge x_1 \in D(x_1) \wedge \ldots \wedge x_n \in D(x_n)$. ■

Constraint satisfaction problems have been used to model many real-life problems and many algorithms have been developed to optimize the search process for problems with arithmetic constraints. In the following sections, we describe how these algorithms can be adapted to improve the performance of ACMA joins in database systems.

3.2 ACMA join as a constraint satisfaction problem

We can formalize a database join operation as a CSP by associating a variable with every required tuple. The domain of each variable is all tuples in the associated relation. Constraints on variables describe selection conditions. For example, we can express the query of Figure 1 as the following CSP, with the variables T_i (i = 1 … 4) denoting the required tuples from S, P, C, and D, respectively. The first two lines are primitive constraints.

T_1.price + T_2.price + T_2.rate * T_3.rate * T_3.price + T_2.rate * T_3.rate * T_4.price $\leq 16 \quad \wedge$

T_2.rate * T_3.rate * T_4.rate $\geq 900 \quad \wedge$

$T_1 \in S \wedge T_2 \in P \wedge T_3 \in C \wedge T_4 \in D$

CSPs that express ACMA joins in this way share some common features: the values of variables are tuples with multiple attributes; constraints on variables are usually expressed by 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; and (if, as will often be the case, multiple queries are posed against the same database) different CSPs differ only in terms of their constraints, while sharing common variable domains corresponding to all tuples in all relations of the database. We call this kind of CSP a *DB_CSP*.

This formulation of ACMA join queries as a (particular class of) constraint satisfaction problem suggests that we may be able to apply constraint-solving algorithms to the solutions.

3.3 Search algorithm

In this section, we show how we can integrate search techniques developed for CSPs into database search engines to implement ACMA joins efficiently.

As illustrated in Figure 2(a), current database systems use multiple nested-loop operators to implement a single ACMA join. (We assume in that figure that the execution plan is pipelining.) However, this sort of execution plan must read in many tuples and create many intermediate tuples that cannot lead to any solutions. For example, the join operator used to join S and P in Figure 2(a) cannot remove any intermediate tuples based on the two ACMAs, and thus all intermediate tuples are passed to the upper level to join with C. Yet if we know that the value of T_2.rate * T_3.rate * T_3.price + T_2.rate * T_3.rate * T_4.price is always bigger than 10, then it is obvious that an intermediate tuple for which T_1.price + T_2.price > 6 cannot be in the final solution, given that the first ACMA T_1.price + T_2.price + T_2.rate * T_3.rate * T_3.price + T_2.rate * T_3.rate * T_4.price ≤ 16.

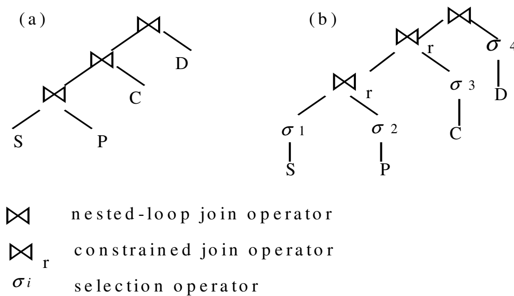


Figure 2. (a) Conventional and (b) improved execution plans for the join operation in Figure 1

These considerations lead us to propose the new execution plan shown in Figure 2. This new plan differs from the original in two respects. First, we add selection operators that filter tuples that cannot lead to a solution. Second, we introduce a new *constrained join operator*, which extends the nested-loop join operator to generate only intermediate tuples that possibly lead to a solution. Because every intermediate tuple will join tuples from other relations in the following join operators, this new plan saves computation time, I/O, and memory needed by the following join operations.

Although this execution plan is straightforward, it needs more intelligence in the database query processor to determine the selection conditions in the selection operators and to instruct join operators to create only intermediate tuples that possibly lead to a final solution.

3.3.1. Initializing selection operators. In order to initialize filter conditions in each selection operator, we need to translate the ACMAs in the join condition into a set of range constraints, such as P.rate > 0.9, that can be utilized by selection operators to filter tuples. This translation operation should be general such that it can be applied to any ACMA. Also, this operation should be powerful such that it can create range constraints with maximal selectivity. We implement this operation as function *CheckConsistency* (Figure 3) by utilizing and extending consistency checking algorithms from CSP area.

The *CheckConsistency* function applies consistency-checking algorithms to each ACMA in the query condition in turn. Traditionally, these algorithms use information about the domain of each variable in the constraint to eliminate, from the domains of both that variable and other variables, values that cannot lead to a solution. However, in the case of a DB_CSP problem, variable domains are tuples in relations that will typically be maintained on disk. Thus, if a consistency algorithm finds that a particular set of tuples cannot lead to a solution, it adds a range constraint in the related selection operator to filter those tuples. This filtering can reduce significantly the number of tuples passed up to the join operator. In addition, if an index is available for the attributes that appear in the selection condition, filtering can also reduce the number of tuples read from disk.

Consistency algorithms can be applied to any ACMAs. Also it keeps removing values from variable domains (in our case, adding new range constraints) until a consistent state is reached that means no further search space refinement (in our case, adding new range constraints) is possible.

A wide variety of consistency algorithms have been developed [15], with widely varying

characteristics in terms of the amount of information required to refine domains and their computational complexity. We choose to use the node consistency algorithm and bounds consistency algorithm because they need only bounds information for value domains. (As noted above, reducing accesses to value domains is particularly important in the case where values are tuples of a relation stored on disk.)

```
// C is the CSP corresponding to an ACMA query
1   CheckConsistency(C)
2      FOREACH primitive constraint c ∈ C
3            IF (c is not node consistent)
4               a = NodeConsistencyAlgorithm(c)
5            IF (c is not bounds consistent)
6               b = BoundsConsistencyAlgorithm(c)
7            ENDIF
8            If(!a || !b) return false
9      END_FOREACH
10     RETURN true
```

Figure 3. The consistency checking algorithm, used both to initialize selection operators and to filter unsatisfiable tuples within the constrained join operator

The node and bounds consistency algorithms work by removing all unsatisfied values from value domains until a node- or bounds-consistent state, respectively, is reached (Definitions 3 and 4)—or a domain becomes empty, in which case we can conclude that there is no solution.

Definition 3 (Node Consistent) *A primitive constraint (See definition 1) c is* node consistent [15] *if either the constraint involves multiple variable attributes or if all values in the domain of the single attribute are a solution of c.* ■

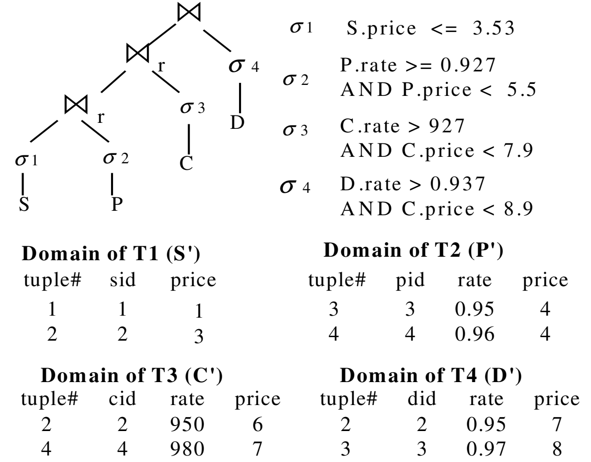
Definition 4 (Bound Consistent) *An arithmetic primitive constraint (see definition 1) c is* bounds consistent [15] *if for each variable attribute attr that appears in this constraint, there exist two solutions with attr assigned to maximal and minimal value of attr respectively.* ■

We consider the example of Figure 1 once again, this time to show how consistency checking proceeds.

Node consistency: All ACMAs are node consistent because they have more than one variable attribute.

Bounds consistency: The primitive constraint T_2.rate * T_3.rate * T_4.rate >= 900 is not bounds consistent because there is no assignment with T_2.rate= 0.8 that satisfies this constraint. Thus, we apply the bounds consistency algorithm on this constraint to update all variable value domains until a bounds-consistent state is reached.

Figure 4 shows both the selection conditions that result from this preprocessing and the relations S', P', C' and D', which represent the domains of T_1, T_2, T_3 and T_4, respectively, following preprocessing by the selection operators. We see that the size of the search space, which corresponds to the Cartesian product of all variable domains, is reduced from 5*5*5*5=625 to 2*2*2 *2=16.



Domain of T1 (S')

tuple#	sid	price
1	1	1
2	2	3

Domain of T2 (P')

tuple#	pid	rate	price
3	3	0.95	4
4	4	0.96	4

Domain of T3 (C')

tuple#	cid	rate	price
2	2	950	6
4	4	980	7

Domain of T4 (D')

tuple#	did	rate	price
2	2	0.95	7
3	3	0.97	8

Figure 4. The search plan of Figure 2(b) (upper left); the selection operators added by preprocessing (upper right); and the tuples left after selection operators (bottom).

3.3.2. Extending the join operator. We extend the current nested-loop join operator to incorporate logic for filtering intermediate tuples that cannot lead to any final solution. For clarification, we call the new operator a *constrained join operator*.

We assume the original nested-loop join operator is implemented in terms of three iterator functions, *Open*, *GetNext*, and *Close*, used to open relations, get the next tuple, and close relations, respectively. We extend the operator by rewriting the *GetNext* function. To distinguish the new *GetNext* function from the original, we rename the original *to OriginalGetNext*. Figure 5 shows the new GetNext function.

```
// CQ is the CSP corresponding to an ACMA query
1 GetNext()
2      WHILE (true)
3          // r is a join result
4             r = OriginalGetNext();
5             IF(r = NotFound)
6                    RETURN NotFound
7                      // join relations are exhausted
8             IF (CheckConsistency(CQ AND r))
9                      RETURN r
10     ENDWHILE
```

Figure 5. The new GetNext function used within the constrained nested-loop operator

The extended function first calls *OriginalGetNext* (line 4) to generate a tuple. It does not return this tuple, but instead calls the *CheckConsistency* function (line

8). The input to this function, *CQ AND r*, is a CSP representing the search for all solutions extended from the intermediate tuple *r*. If this function returns false, it means that this particular intermediate tuple will not lead to any solution. Thus, the new *GetNext* function uses *OriginalGetNext* repeatedly to generate intermediate tuples until *CheckConsistency* returns true, indicating a tuple that may lead to solutions.

For the example in Figure 4, the join of S' and P', the relations created by the selection operators on S and P, passes only two intermediate tuples, namely {tuple 1 in S, tuple 3 in P} and {tuple 1 in S, tuple 4 in P}, to the upper-level join operator. For a conventional nested-loop operator, the number of intermediate tuples would be the product of the size of the two join relations: in this case, four.

Also, when generating and passing the intermediate tuple {tuple1 in S, tuple 3 in P} to the upper level operator, the consistency algorithm further refines the search space by adding new selection conditions to the selection operator attached to C. By applying the bound consistency algorithm on the primitive constraints where T_1 and T_2 are determined in the intermediate tuple, we can conclude that only tuples in C that satisfy T_3.rate > 937 and T_3.price < 7.1 can join with this intermediate tuple. In other words, the bound consistency algorithm allows us to add two selection condition C.rate > 937 and C.price < 7.1 to the selection operator attached to relation C, thus removing more tuples from the domain of T_3.

This example shows how consistency-checking techniques can greatly reduce the size of the search space. The algorithms can also be implemented efficiently, as we discuss in the next section.

4. Implementation

We now present the implementation of the *CheckConsistency* function.

4.1 *CheckConsistency* function

Our implementation of the *CheckConsistency* function is structured in terms of three function modules: a constraint store, a set of propagators, and a guard.

The *constraint store* maintains two types of *basic constraints* on attributes: *assignment* constraints such as *attr* = *N* and *range* constraints such as *attr* > *N*. Initially, attributes are described by range constraints representing attribute value bounds derived from relations. For example, the value domain for T_1 as specified in Figure 1 provides the constraints T_1.price ≥ 1 and T_1.price ≤ 9. These constraints will be refined by consistency algorithms during the query processing. We give further details about this process in the following.

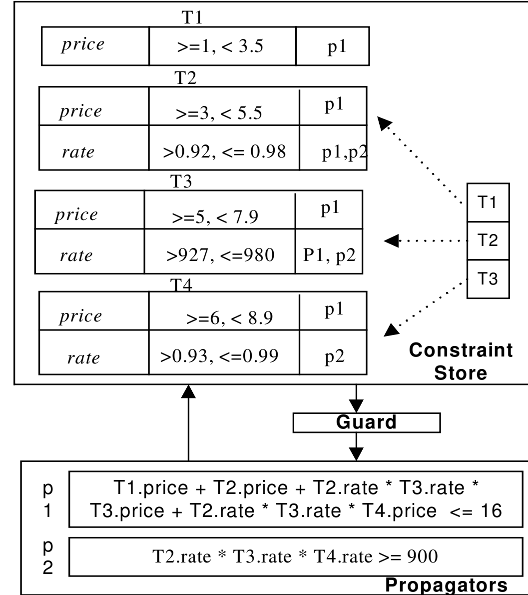


Figure 6. Functional modules for our implementation of the CheckConsistency function of Figure 3, showing their contents when used to implement the query of Figure 1.

We represent the constraint store as a set of symbol tables, one per relation appearing in the join query. This symbol table maintains information about the attributes appearing in ACMAs. Every symbol table entry contains an attribute name, known basic constraints, and pointers to related propagators (see Figure 6). Information in the constraint store is applied to selection operators as selection conditions.

Propagators impose additional complex constraints, such as $T_2.rate + T_3.rate + T_4.rate \leq 5$. A propagator for a constraint *c* is a computation module that tries to narrow the scope of variable attributes occurring in *c*. The node and bounds consistency algorithm is implemented in propagators. A propagator is in one of three states, *inactive*, *active*, and *redundant*, as follows.

- *Inactive*: the constraint attached to this propagator is node and bounds consistent.
- *Active*: the constraint attached to this propagator is inconsistent and consistency algorithms can be used to narrow down the scope of attributes in constraint store.
- *Redundant*: the propagator is subsumed by information in the constraint store, and thus cannot further narrow the scope of the attribute.

Active propagators refine the information in the constraint store. After refinement, the state of the propagator is changed to inactive or redundant, accordingly.

The guard watches the attribute information in the constraint store and changes the state of propagators accordingly. Information in the constraint store may

change during the search process due to, for example, a value assignment to a variable or domain refinement caused by propagators. These changes may cause inconsistency in related propagators. If inconsistency is found, the state of that propagator is changed to active. Thus, any changes in the constraint store are propagated to all related propagators, which lead to further refinement of the information in the constraint store.

Through these three modules, a complex constraint is translated into two types of basic constraints stored in the constraint store that can be used refine the search space of complex combinatorial search efficiently. For a complex constraint, the corresponding propagator realizes the translation based on attributes' bounds by using the bounds consistency algorithm (more details in Section 4.2). Because propagators simply translate query conditions into basic constraints, the consistency algorithm itself does not cause additional I/O operations.

Figure 6 shows the function modules as configured for the ACMA join of Figure 1. The query conditions include two primitive ACMA constraints $T_2.rate * T_3.rate * T_4.rate \geq 900$ and $T_1.price + T_2.price + T_2.rate * T_3.rate * T_3.price + T_2.rate\ T_3.rate * T_4.price \leq 16$. Attributes in the constraint store are described by bounds information collected from the database system. Two propagators are created for the two ACMA constraints. Based on the information in the constraint store, the guard puts these two propagators in the active state and they refine the bound of the value of attributes and the refinement results are shown in Figure 6.

4.2 Consistency algorithm

The node consistency algorithm is straightforward. For an ACMA constraint with only one variable attribute, it rewrites the constraint into an assignment constraint or a range constraint and put it into the constraint store.

The bounds consistency algorithm translates complex constraints into basic constraints that can be used to remove values from value domain. This algorithm is implemented in propagators. We employ the bounds consistency algorithm used in Oz [17]. A constraint has the following general form.

$$I_1 * D_{11} * \ldots * D_{1m_1} + \ldots + I_n * D_{n1} * \ldots * D_{nm_n} \sim D$$

Here, I_i and D are constants, D_{ij} are attributes that appear in this constraint, m_i is the number of attributes in the i_{th} sub-expression, and ~ represents a logic operator that can be '>', '<', '≥', '≤', or '='. By using a new attribute D_{n+1} with upper bound and low bound equal to D to replace constant D, this constraint can be represented as follows with $I_{n+1} = -1$ and $m_{n+1}=1$.

$$I_k * D_{k1} * \ldots * D_{km_k} \sim -\sum_{i=1, i \neq k}^{n+1} I_i * \prod_{j=1}^{m_i} D_{ij} \quad (1)$$

Assuming (as is common in constraint programming systems) that all attributes have values bigger than 0, we can calculate the bounds of the expression on the right side of this formula as follows.

$$L_k = -\sum_{i=1, i \neq k, I_i < 0}^{n+1} I_i * \prod_{j=1}^{m_i} \underline{D_{ij}} - \sum_{i=1, i \neq k, I_i > 0}^{n+1} I_i * \prod_{j=1}^{m_i} \overline{D_{ij}}$$

$$H_k = -\sum_{i=1, i \neq k, I_i > 0}^{n+1} I_i * \prod_{j=1}^{m_i} \underline{D_{ij}} - \sum_{i=1, i \neq k, I_i < 0}^{n+1} I_i * \prod_{j=1}^{m_i} \overline{D_{ij}}$$

Here, $\overline{D_{ij}}$ and $\underline{D_{ij}}$ denote the upper and lower bounds of the attribute D_{ij}, respectively, and H_k and L_k are the upper and lower bounds of the expression on the right side of formula (1). These values can be used to refine the domains of attributes on the left side of formula (1). For example, if ~ is the logic operator '<', then the range of D_{ki} is as follows.

$$D_{ki} < \frac{H_k}{I_k * \prod_{j=1, j \neq i}^{m_k} \underline{D_{kj}}} \quad \text{if } I_k > 0$$

$$D_{ki} > \frac{H_k}{I_k * \prod_{j=1, j \neq i}^{m_k} \overline{D_{kj}}} \quad \text{if } I_k < 0$$

Any value not in this range cannot possibly satisfy this constraint. Through this formula, we can obtain a range for all attributes in this constraint. We use this range to refine the description of the corresponding attribute in the constraint store. We process constraints involving other logic operators in a similar fashion.

As noted above, attributes in the constraint store are described by two types of constraints: assignment constraints and range constraints. Give a range R_1 on the attribute *attr* calculated by the previous step, we refine its descriptions in the constraint store as follows.

```
// S is the description of attr in the constraint store
If (S is attr = v)
   If v ∉ R1
      No consistent state for constraint
Else     // S is a range R2
   R3 = R1 ∩ R2  // R3 is intersection of two ranges
   If R3 is empty
      No consistent state for this constraint
   Else  S = R3
```

The bounds consistency algorithm is used both to check if there exists a consistent state for the current constraint and to reduce the domains of variables by refining the information of their attributes in the constraint store. If a constraint does not have a

consistent state, there is no solution for this constraint. Thus, the algorithm can determine rapidly that a query is unsatisfiable, without checking any combinations.

This bounds consistency algorithm can be used to refine the search space based on constraints described by a polynomial expression of attributes. Its simplicity allows it to be *implemented* efficiently (i.e., with few operations). In addition, experimental studies (Section 6) show that it can *refine the search space* efficiently (i.e., with few constraint checks).

5. Performance evaluation

We wish to evaluate the performance of our algorithm and to compare its performance with that of a conventional database system. To this end, we first define a benchmark query: a simple ACMA join query with parameters that can be used to vary its execution complexity and selectivity.

5.1 The benchmark query Q

We define as a basis for evaluation of a simple ACMA join query on three relations A, B, and C. These relations are all defined according to either (a) the relation *TENKTUP1*defined in the Wisconsin Benchmark [6], which has various attributes named KN, with values distributed uniformly from 1 to N; or (b) the relation NORM, which has two real type attributes K1000 and K10000, whose values follow a larger normal distribution with μ =5000 and σ = 2500.

```
SELECT * FROM A, B, C
WHERE
    A.K1000 + B.K1000 + C.K1000 > N1   AND
   A.K1000 + B.K1000 + C.K1000 < N2   AND
    A.K10000 + B.K10000 + C.K10000  > N3
```

Figure 7. The benchmark query Q

This query specifies a request for three tuples from relation A, B, and C with three ACMA join conditions, each involving three relations. We choose to use linear ACMAs in this query only to simplify analysis of the results. As shown in Section 4, our algorithm can handle arbitrary arithmetic constraints.

Notice that varying the size of the relation A, B and C increases the size of the problem to be solved, while varying the parameters N_1, N_2, and/or N_3 varies the number of results obtained, i.e., the selectivity of Q. As we describe below, we vary these parameters in our experiments to obtain queries with various problem sizes and selectivity.

5.2 Evaluation plans for Q

We present in Figure 8 three different execution plans for Q. Plan I is the execution plan created by current database system. Plans II and III are variations of the algorithm suggested in this paper. Plan II uses the selection operator initialized by the consistency-checking function, but still uses the nested-loop operator. In contrast, plan III uses the new constrained operator. We compare the performance of Plans II and III to evaluate the efficiency of the constrained operator.

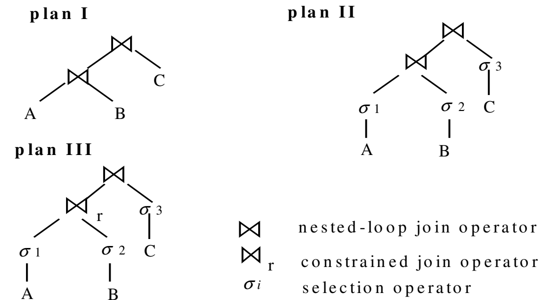


Figure 8. The conventional left-deep execution plan for Q (plan I) and two alternative plans

6. Experimental studies

We use three performance metrics to evaluate the search efficiency of the different execution plans: the number of tuple read operations, the number of intermediate tuples, and the elapsed time required to obtain query results.

- *Number of tuple read operations*. For large databases, query performance is determined by the number of I/O operations performed. Because the number of I/O operations in a particular system depends on many factors besides the search algorithm, we use the total number of tuple read operations performed during the query process as our I/O efficiency metric.
- *Number of intermediate tuples*. We measure this number by counting the number of intermediate tuples passed from an operator to its consumer in the execution pipeline.
- *Elapsed time*. This final metric is the time between query submission and the return of results.

In our experiments, we implement three execution plans in a prototype system, Redline [14] and measure the three metrics. We conducted all experiments on an IBM T20 with a single Pentium III 700MHz processor and 128 MB memory, running Suse Linux 7.2. In order to focus on the search efficiency of our algorithm, we do not build an index for any attributes. We note that the selection operators used in our algorithm could employ indexes on attributes, if they were provided, to reduce I/O operations. Thus, our experimental results provide only a lower bound on the improvements achievable by our algorithm.

6.1 Changing relation size

In this first set of experiments, we vary the number of tuples in the relation C from 10,000 (10K) to 1280K while fixing the number of tuples in the relations A and B at 10K and fixing the join selectivity by setting N_1=2, N_2=5, and N_3=27000 for TENKTUP1, and N_1=2, N_2=300, and N_3=33000 for NORM. We use different parameters for the two datasets to achieve similar selectivity.

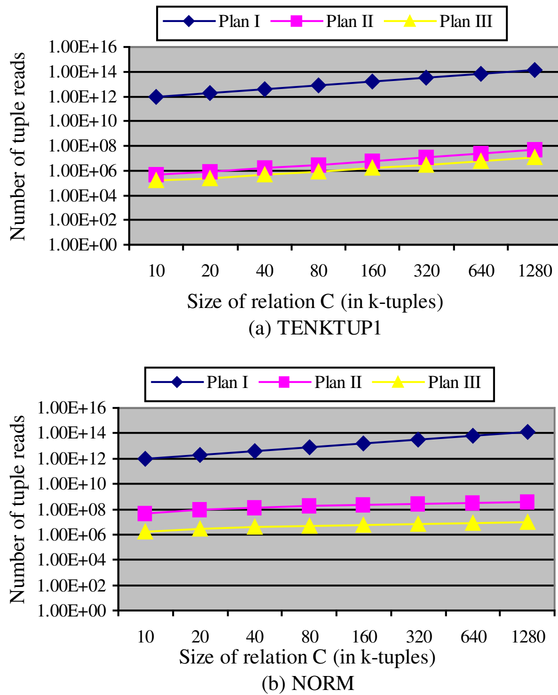


Figure 9. Number of tuple read operations performed when evaluating Q, as function of the size of relation C.

Figure 9 shows the number of tuple read operations performed for each of the three plans. Plan I reads from 10^4 to 10^6 times more tuples than do the other two plans. The reason for this large differential is that in the absence of ACMA filtering, plan I needs to calculate the Cartesian product of all join relations—and thus must perform a number of tuple read operations equal to the product of the size of all relations.

We see also that plan III performs a factor of 2.8 times fewer tuple reads than does plan II for TENKTUP1, and 36 times fewer for NORM. The reason for this difference is that the constrained join operator can filter intermediate tuples that cannot lead to any solution, thus reducing the number of tuple read operations performed by the next join operator in the pipeline. For example, in plan III, the operator that joins C with the intermediate results created by the join of A and B needs to read all tuples in C for every intermediate tuple created by the join of A and B. Thus, if the operator that joins A and B can filter an intermediate tuple, read operations on C are saved.

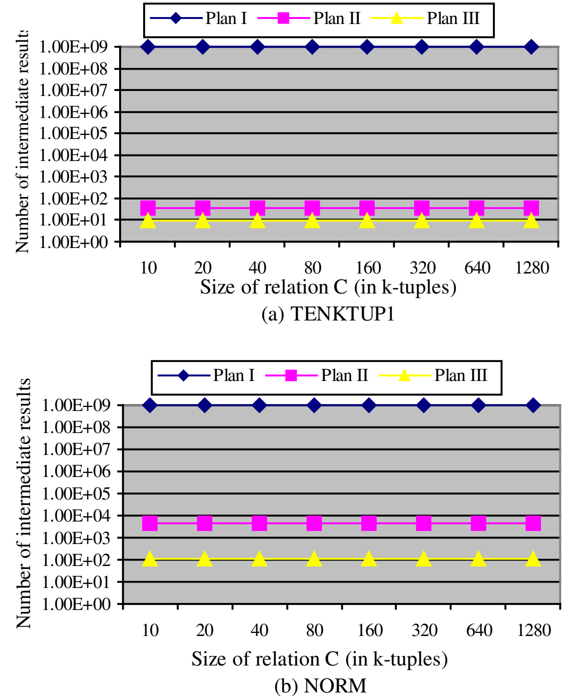


Figure 10. Number of intermediate tuples generated when evaluating Q, as a function of the size of the relation C.

Figure 10 shows the number of intermediate tuples generated by the different plans as a function of the size of the relation C. This number does not change with the size of C because intermediate tuples are created by the join of A and B whose sizes are fixed. However, we do see that plan III with the constrained join operator generated fewer intermediate tuples than plans I or II, and thus performs fewer tuple read operations (Figure 9) and has a smaller query elapsed time (Figure 11).

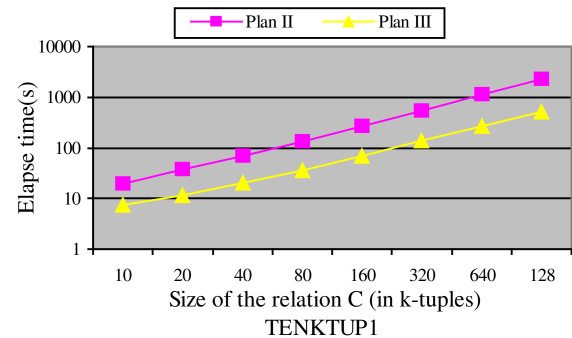
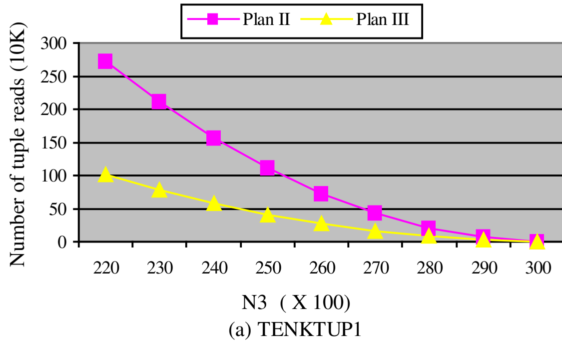


Figure 11. Time taken to evaluate Q, as a function of the size of the relation C.

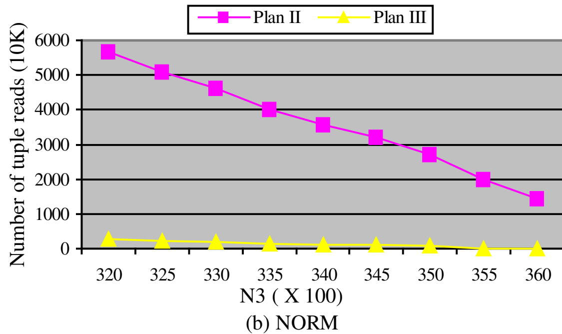
Figure 11 shows the execution times measured for plans II and III when A, B, and C are *TENKTUP1* relations. As expected, plan III is faster than plan II. We do not show the elapsed time for plan I because, as suggested by the analysis of the first two metrics, it cannot finish in a reasonable time. We also do not show the execution times on *NORM* relations, because plan II cannot finish in a reasonable time. The query Q takes longer on NORM because its attribute values have a larger range, which makes the bound consistency check less effective than on *TENKTUP1*.

6.2 Changing Selectivity

In this second set of experiments, we fix the number of tuples in the relations A, B, and C to 10000, and vary the join selectivity by varying the constant N_3 in the selection condition of Q. The constraint $A.K10000 + B.K10000 + C.K10000 > N_3$, causes the selectivity of query Q to decreases as N3 increases.



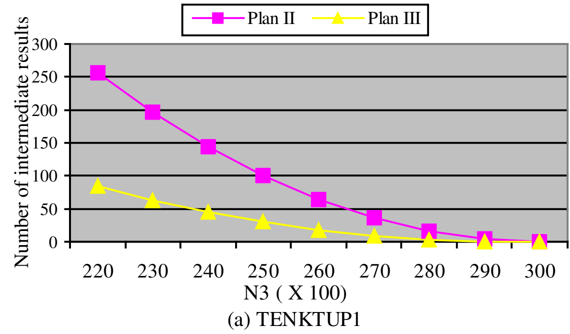
(a) TENKTUP1



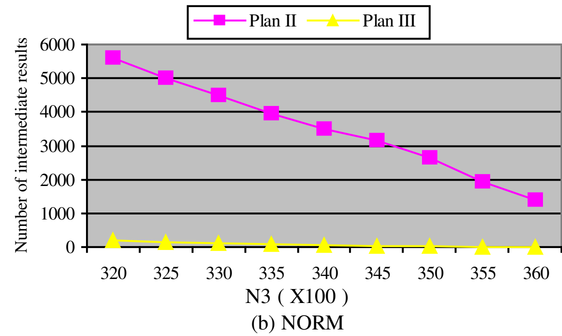
(b) NORM

Figure 12. Number of tuple read operations performed when evaluating Q, as a function of N3

We see from Figures 12-14 that plan III consistently outperforms plan II. We do not show results for plan I because (as shown in Figure 9) it is much more expensive than the other two plans, and its performance does not change with the change of selectivity. Note that when N_3>=28000 in Figure 12(a) and N_3>=33500 in Figure 12(b), the selectivity is 0. In this situation, plans II and III return results quickly. The reason is that our algorithm determines that this query is unsatisfiable because it is not bounds consistent. Thus, plans II and III can evaluate this query without really probing the search space. In contrast, plan I must perform an exhaustive search. Thus, we see that our ACMA join algorithm achieves even greater improvements relative to the conventional database join algorithm when query selectivity is small.



(a) TENKTUP1



(b) NORM

Figure 13. Number of intermediate tuples generated when evaluating Q, as a function of N3.

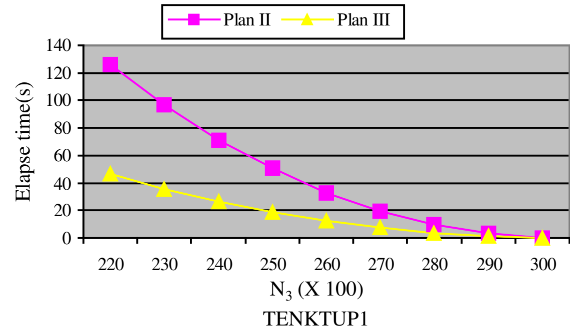


Figure 14. Time taken to evaluate Q, as a function of N3.

In summary, our results show that both of our proposed ACMA join algorithm features (new selection operators and the constrained join operator) can provide order-of-magnitude performance improvements for ACMA queries when compared to the conventional database join algorithm, particularly when query selectivity is small.

7. Summary

We have presented a new approach to supporting multi-join queries with arithmetic constraints in relational database systems. We introduce a new join algorithm that integrates constraint-programming techniques into relational database search. We show how this new algorithm can be implemented in terms of a new *constrained join operator* and a new execution plan that applies this operator. We experimentally evaluate the proposed join algorithm, comparing its performance with current join algorithms used in relational databases. The results of these experiments show a significant performance enhancement relative to conventional algorithms, particularly when query selectivity is small.

Our results suggest that we can expand the range of applicability for relational databases to encompass a significant new class of queries, and that we can do so without major changes to underlying relational database engines.

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, Anne Rogers, Svetlozar Nestorov, and Matei Ripeanu for comments on a draft of this paper.

References

[1] Agarwal, P.K., Arge, L., Erickson, J., Franciosa, P.G. and Vitter, J.S., Efficient searching with linear constraints. *PODS*, ACM press, Seatle, Washington, United States, 1998.

[2] Aigner, M., *Combinatorial Search*, John Wiley & Sons, 1988, 372 pp.

[3] Avnur, R. and Hellerstein, J.M., Eddies: continuously adaptive query processing. *SIGMOD*, Dallas, TX, USA, 2000, pp. 261-272.

[4] Fagin, R., Lotem, A. and Naor, M., Optimal aggregation algorithms for middleware, *Journal of Computer and System Sciences*, 66 (2003) 614-656.

[5] Garcia-Molina, H., Ullman, J.D. and Widom, J., *Database Systems: the Complete Book*, Prentice Hall, Upper Saddle River, NJ, 2002, xxvii, 1119 pp.

[6] Gray, J., *The Benchmark Handbook for Database and Transaction Systems*, 2nd edn., Morgan Kaufmann, 1993.

[7] Guha, S., Gunopoulos, D., Koudas, N., Srivastava, D. and Vlachos, M., Efficient approximation of optimization queries under parametric aggregation constraints. *Proc. of 29th International Conference on Very Large DataBases*, Berlin, Germany, 2003.

[8] Hammer, M. and Zdonik, S.B., Knowledge based query processing. *VLDB*, 1980, pp. 137 -147.

[9] Hentenryck, P.V., *Constraint Satisfaction in Logic Programming*, The MIT Press, 1989.

[10] Hentenryck, P.V., Deville, Y. and Teng, C.-M., Generic arc-consistency algorithm and its specializations, *Artificial Intelligence*, 57 (1992) 291-321.

[11] Ilyas, I.F., Aref, W.G. and Elmagarmid, A.K., Supporting Top-K Join Queries in Relational Databases. *VLDB 2003*, Morgan Kaufmann, Berlin, Germany, 2003.

[12] King, J.J., "QUIST A system for semantic query optimization in relational databases. *7th Int. VLDB Conference*, Cannes, France, 1981, pp. 510-517.

[13] Kolaitis, P.G. and Vardi, M.Y., Conjunctive-query containment and constraint satisfaction. *PODS*, ACM press, Seattle, Washington, United States, 1998, pp. 205-213.

[14] Liu, C. and Foster, I., A Constraint Language Approach to Grid Resource Selection. University of Chicago, Chicago, 2003.

[15] Marriott, K. and Stuckey, P.J., *Programming with Constraints: An Introduction*, The MIT Press, Cambridge, Massachusetts, 1998.

[16] Revesz, P., *Introduction to Constraint Databases*, Springer, New York, 2002.

[17] Schulte, C. and Smolka, G., Finite domain constraint programming in Oz. http://www.mozart-oz.org/documentation/fdt/index.html, 2002.

[18] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A. and Price, T.G., Access path selection in a relational database management system. *ACM SIGMOD Intl. Conf. on Management of Data*, Boston, USA, 1979, pp. 23-34.

[19] Stonebraker, M. and Hellerstein, J.M., Content integration for e-business. *PODS*, ACM press, Santa Barbara, California, United States, 2001, pp. 552-560.

[20] Tsang, E., *Foundations of Constraint Satisfaction*, Academic Press, London, 1993.

[21] Vardi, M.Y., Constraint satisfaction and database theory: a tutorial. *PODS*, ACM Press, Dallas, Texas, United States, 2000, pp. 75-85.

[22] Waltz, D.L., Understanding line drawings of scenes with shadows. In P.H. Winston (Ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975, pp. 19-21.

[23] Wolf, J.L., Iyer, B.R., Pattipati, K.R. and Turek, J., Optimal buffer partitioning for the nested block join algorithm. *ICDE*, Kobe, Japan, 1991.

[24] Wolsey, L.A. and Nemhauser, G.L., *Integer and Combinatorial Optimization*, first edn., Wiley-Interscience, 1999.