Constraint Processing Techniques for Improving Join Computation: A Proof of Concept

Anagh Lal and **Berthe Y. Choueiry**

Constraint Systems Laboratory
Department of Computer Science and Engineering
University of Nebraska-Lincoln
{alal|choueiry}@cse.unl.edu

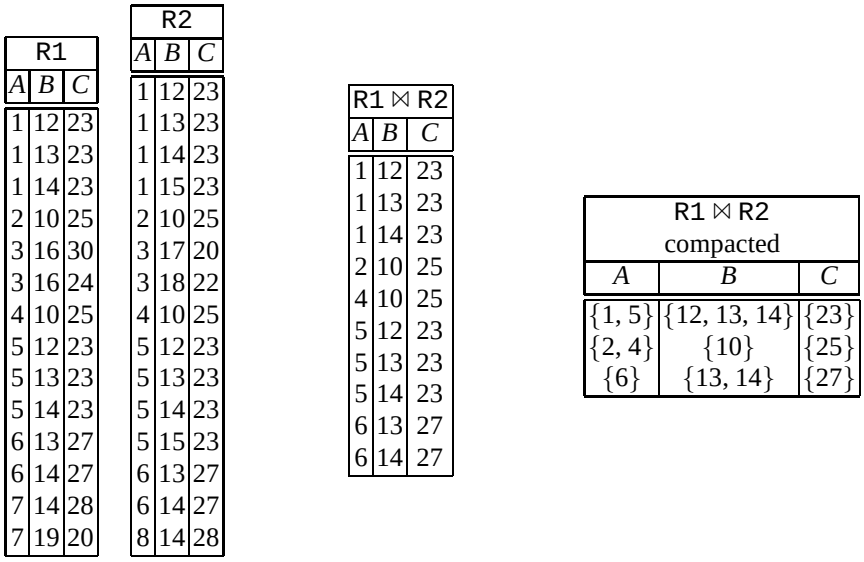
Abstract. Constraint Processing and Database techniques overlap significantly. We discuss here the application of a constraint satisfaction technique, called dynamic bundling, to databases. We model the join query computation as a Constraint Satisfaction Problem (CSP) and solve it by search using dynamic bundling. First, we introduce a sort-based technique for computing dynamic bundling. Then, we describe the join algorithm that produces nested tuples. The resulting process yields a compact solution space and savings of memory, disk-space, and/or network bandwidth. We realize further savings by using bundling to reduce the number of join-condition checks. We place our bundling technique in the framework of the Progressive Merge Join (PMJ) [1] and use the XXL library [2] for implementing and testing our algorithm. PMJ assists in effective query-result-size prediction by producing early results. Our algorithm reinforces this feature of PMJ by producing the tuples as multiple solutions and is thus useful for improving size estimation.

1 Introduction

Although not widely acknowledged, progress made in the area of Databases has historically greatly benefited the area of Constraint Processing, and vice versa. We present here one new such opportunity, in which a Constraint Satisfaction technique that we have developed and call bundling is used to improve the computation of a join. The join operation is extensively studied in the database literature and remains one of the most computationally expensive operations. To the best of our knowledge, no work has yet exploited the existence of symmetries within a relation to improve the performance of the task or reduce the space necessary for storing the result. Consider the two relations R1 and R2 shown in Figure 1 (left), and the following SQL query:

```
SELECT R1.A, R1.B, R1.C
FROM R1, R2
WHERE R1.A = R2.A AND R1.B = R2.B  AND R1.C = R2.C
```

The natural join R1 $\bowtie$ R2 has the tuples shown in Figure 1 (center). Consider $\sigma_{A=2}(R1)$ and $\sigma_{A=4}(R1)$. The tuples they yield in R1 $\bowtie$ R2 differ only for the value of A. We say that A=2 and A=4 are symmetric and use this symmetry to compact the resulting tuples



R1		
A	*B*	*C*
1	12	23
1	13	23
1	14	23
2	10	25
3	16	30
3	16	24
4	10	25
5	12	23
5	13	23
5	14	23
6	13	27
6	14	27
7	14	28
7	19	20

R2		
A	*B*	*C*
1	12	23
1	13	23
1	14	23
1	15	23
2	10	25
3	17	20
3	18	22
4	10	25
5	12	23
5	13	23
5	14	23
5	15	23
6	13	27
6	14	27
8	14	28

R1 ⋈ R2		
A	*B*	*C*
1	12	23
1	13	23
1	14	23
2	10	25
4	10	25
5	12	23
5	13	23
5	14	23
6	13	27
6	14	27

R1 ⋈ R2 compacted		
A	*B*	*C*
{1, 5}	{12, 13, 14}	{23}
{2, 4}	{10}	{25}
{6}	{13, 14}	{27}

Fig. 1. *Left:* R1, R2. *Center:* R1 ⋈ R2. *Right:* Compacted R1 ⋈ R2.

in R1 ⋈ R2. We propose here a technique for detecting such symmetries and exploiting them for compacting join results. Note that our technique is efficient but does not guarantee the maximal possible compaction. Our technique generates the compacted join relation shown in Figure 1 (right), which has only 3 tuples with nested values. When computing a sequence of join operations, the intermediate join results occupy less space. When these results are used in a subsequent join, more tuples are available per page, which reduces the I/O operations and thus saves time.

Although most research in Constraint Satisfaction focuses on binary constraints, many real-life problems are more 'naturally' modeled as non-binary Constraint Satisfaction Problems (CPSs). The focus on binary CSPs has so far been tolerated because it is always possible, in principle, to reduce a finite non-binary CSP to a binary one [3, 4]. However, recent research has shown that it is sometimes more effective to operate on the non-binary encoding of the CSP than on its binary reduction [5]. At this point in time, research on non-binary constraints is still in its infancy. In [6, 7], we proposed a technique, *dynamic bundling*, for dynamically detecting and exploiting symmetries in binary CSPs. In [8], we extended this technique to non-binary constraints and showed a significant improvement in processing time and solution space. In this paper, we show how to use dynamic bundling for processing join queries and compacting the join results. We make the following contributions:

1. Provide a new way to map a join query into a CSP (Section 3).
2. Present an algorithm for dynamic bundling that improves the memory usage of our previous implementation [8] and is more suitable for databases (Section 4).

3. Present a join algorithm for producing bundled solutions that are more compact, thus saving memory (Section 5).
4. Identify new opportunities for exploiting the compact solution space in other database applications such as data analysis and materialized views.

This paper is structured as follows. Section 2 states our motivation and provides background information. Section 3 models the join operator as a CSP. Section 4 describes a technique for bundling the values of an attribute in a relation. Section 5 uses this bundling technique in a new join algorithm. Section 6 discusses our implementation and the results of our experiments. Section 7 summarizes related work. Finally, Section 8 concludes the paper and gives directions for future research.

2 Background

In this section we explain the motivations behind our research and summarize background information useful for reading the paper.

2.1 Motivation

Join algorithms can be classified into three categories: hash-based, sort-based, and nested-loop algorithms. All these algorithms attempt to optimize the join by minimizing the number of times relations are scanned. Hash-based algorithms use hash-tables to partition relations according to the values of an attribute, and then join the partitions corresponding to the same values. The sort-based approach partitions relations by sorting them on the attributes involved in the join condition. Thanks to sorting, each tuple in a relation is compared with tuples of the other relation lying within a fixed range of values, which are significantly fewer than all possible tuples. Sorting reduces the number of scans of both relations and speeds up join processing. Nested-loop algorithms are used when relations fit in memory or when no adequate hashing function or useful sorting order is available. None of these techniques attempts to compact query results, although this can be beneficial given the large size of join results. The reduction of the number of I/O operations during query evaluation is a key factor in determining the efficiency of a database. Extensive research is devoted to the development of query-evaluation plans and evaluation algorithms that minimize the number of I/O operations. Our technique of dynamic bundling produces results that are compact by automatically detecting symmetries within a relation. Our goal is to exploit the use of these compact solution spaces in order to reduce I/O operations and extract information from query results useful for data analysis and data mining. We achieve this goal by first reducing the space requirements of our bundling technique in order to adapt it to the bundling of solution tuples of a query in the context of databases (Section 4). Then, we design a join algorithm that uses bundling (Section 5).

We project two other important uses of our technique, namely: improving query-size estimation and supporting data analysis and mining. Indeed, the fact that the size of the compacted tuples produced by our technique is large is an indicator of high redundancy in the join relations. This information can be used to boost the estimate of

query-result size, which is important for query planning. Further, the compacted results of our new join algorithm represent similar solutions that are clustered together. Let us consider a scenario with the two relations

```
Customer_Choice(Custid, Favorite_Product, Cust_Category),
```

which stores customer choices from an online survey, and

```
Customer_List(Custid),
```

which stores the customers staying in Lincoln, and a query to find the result of the online survey for Lincoln:

```
SELECT Customer_List.Custid, Favorite_Product, Cust_Category
FROM Customer_Choice, Customer_List
WHERE  Customer_Choice.Custid = Cust_List.Custid
```

Our techniques will produce results where customers with same product and category choices are bundled up together. This is just one example of how bundling adds information to query results. This additional information can be used for data mining and in packages for data analysis.

2.2 Sort-based join algorithms

The join operator in relational algebra takes two relations as arguments and a condition (known as the join condition) that compares any two attributes, one from each of the two argument relations. The generic form of a join is $\mathsf{R}\bowtie_{x\theta y}\mathsf{S}$, where R and S are two relations, x and y are attributes from R and S respectively, and θ stands for a comparison operator (e.g., $=, \geq, \leq$, and $\neq$) called the *join condition*. Equality is the most commonly used join condition, and gives the *equi-join*. A *natural join* is a special case of an equi-join for which $x = y$, i.e. the attributes of the two relations are same. The join operation is among the most I/O-intensive operators of relational algebra because it may require multiple scans over the two input relations and also because the size of the result can be as large as the product of the sizes of these relations.

Our new join algorithm (Section 5) adopts the main idea of the Progressive Merge Join (PMJ) of [1]. PMJ is a join algorithm that produces query results early, and hence has the ability to provide valuable information to the query-size estimator. These are exactly the working conditions that we are targeting. PMJ is a special sort-merge join algorithm, which have two phases: the sorting phase and the merging phase. We first describe sort-merge algorithms in general, then discuss PMJ.

In the sorting phase of a sort-merge algorithm for computing the join of two relations, R1 and R2, the memory of size M pages is first filled with pages of R1. These loaded pages are then sorted on the join-condition attributes and stored back to disk as a sub-list or *run* of the relation. When R1 has N pages, $\frac{N}{M}$ runs are generated. This process is repeated for the second same-sized relation R2. At the end of the sorting phase, we have produced sorted runs of R1 and R2. Now, the merging phase can start. We first consider that $M \geq 2 \times \frac{N}{M}$. Now $\frac{M^2}{2\times N}$ pages from each of the $\frac{N}{M}$ runs of R1

are loaded into memory, and the same is done for R2. The smallest unprocessed tuples from the pages of R1 and R2, respectively, are tested for the join condition. Those that satisfy the condition are joined and the result written as output. A page is exhausted when all its tuples have been processed. When a page is exhausted a page from the same run is brought in. When $M < 2 \times \frac{N}{M}$, multiple merge phases are needed. Each intermediate merging phase produces longer but fewer sorted runs. This process of generating longer but fewer runs continues until the number of runs of the two relations is equal to the number of pages that can fit in memory. This sort-merge algorithm is a *blocking* algorithm in the sense that the first results come only after the sorting phase is completed.

PMJ delivers results early by joining relations already during the sorting phase [1]. Indeed, during the sorting phase, pages from both the relations are read into memory, sorted, and joined to produce early results. Because PMJ produces results early, it is a *non-blocking* or pipelined version of the sort-merge join algorithm. The number of runs generated for each relation is more than that by a general sort-merge algorithm and is given by $\frac{M^2}{4 \times N}$. The merging phase is similar to that of a sort-merge algorithm, except that PMJ ensures that pages of R1 and R2 from the same run are not joined again as they have already produced their results in the sorting phase. The memory requirements of PMJ are more than those of a sort-merge algorithm because the number of runs generated during the sorting phase is double that of a sort-merge algorithm. The number of runs generated doubles because the memory is now shared by both relations. Because of the increased number of runs, the chances of multiple merging phases taking place increases. The production of early results causes the results of PMJ to be unsorted. However, the unsorted results allow for more accurate estimation of the result size, which is an important feature.

2.3 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is defined by $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$= $\{V_i\}$ is a set of variables, $\mathcal{D}$= $\{D_{V_i}\}$ the set of their respective domains, and $\mathcal{C}$ a set of constraints that restrict the acceptable combination of values for variables. A constraint over the variables V_i, V_j, ..., V_k is specified as $C_{V_i,V_j,\ldots,V_k}$ = {($\langle V_i\ a_i\rangle$, $\langle V_j\ a_j\rangle$, ..., $\langle V_k\ a_k\rangle$)} with $a_i \in D_{V_i}$, $a_j \in D_{V_j}$, ..., $a_k \in D_{V_k}$. Solving a CSP requires assigning a value to each variable such that all constraints are simultaneously satisfied. The problem is **NP**-complete in general. The *scope* of a constraint is the set of variables to which the constraint applies, and its *arity* is the size of this set. Non-binary constraints are represented as hyper-edges in the constraint network. For sake of clarity, we represent a hyper-edge as another type of node connected to the variables in the scope of the constraint, see Figure 2.

Solving CSPs with search. CSPs are typically solved using depth-first search with backtracking. Depth-first search proceeds by choosing a current variable V_c and *instantiating* it, i.e. assigning to it a value a taken from its domain, $V_c \leftarrow a$. The variable and its assigned value define a variable-value pair (vvp) denoted by $\langle V_c\ a\rangle$. Uninstantiated variables are called future variables, and their set is denoted by $\mathcal{V}_f$. A look-ahead strategy called forward checking (FC) is then applied which removes from the domains of

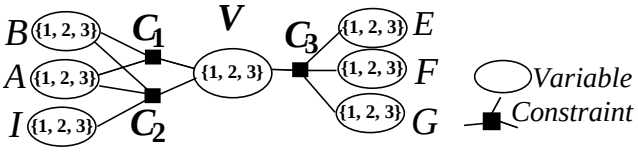


Fig. 2. *Example of a non-binary CSP.*

the future variables the values that are not consistent with the current assignment, thus *propagating* the effect of the instantiation $\langle V_c\ a \rangle$. V_c is then added to the set of instantiated variables, which we call past variables and denote as $\mathcal{V}_p$. If the instantiation does not wipe out the domain of any variable in $\mathcal{V}_f$, search considers the next variable in the ordering and moves one level down in the search tree. Otherwise, the instantiation is revoked, its effects are undone, and an alternative instantiation to the current variable is attempted. When all alternatives fail, search backtracks to the previous level in the tree. The process repeats until one or all solutions are found. Variables are considered in sequence according to a *variable ordering* heuristic. Common wisdom requires that the most-constrained variable be considered first in order to reduce the branching factor of the search tree and the number of backtracks.

2.4 Symmetry as value interchangeability

Interchangeability is a general concept that characterizes the types of symmetries that may arise in a CSP. The concept deals with redundancy in a CSP. In the broadest sense, when a CSP has more than one solution, one can define a mapping between the solutions such that one solution can be obtained from another without performing search. This is *functional* interchangeability [9]. We address here a restricted form of interchangeability: the interchangeability of values in the domain of a single variable. This type of interchangeability does not cover the permutation of values across variables, which is an isomorphic interchangeability. Consider the CSP shown in Figure 3 (A). In

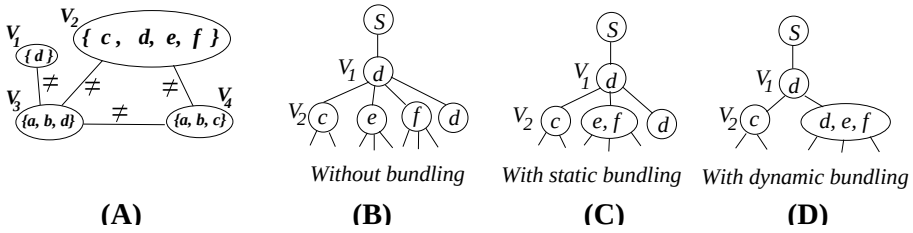


Fig. 3. Solving a CSP with search, with and without bundling.

the absence of any symmetry consideration, the depth-first search process described in Section 2.3 yields the tree shown in Figure 3 (B). A simple analysis of the values of the variable V_2 shows that the values e and f are consistent with exactly the same values in

the neighborhood of V_2, and consequently they are interchangeable in a solution of the CSP. Values e and f are said to be neighborhood interchangeable. Detecting neighborhood interchangeability can be efficiently done using the discrimination tree algorithm proposed in [9].

Search with static bundling. Haselböck proposed to 'bundle' up, in the search tree, neighborhood interchangeable values (e.g., e and f for V_2) since they necessarily yield equivalent sub-trees [10], see Figure 3 (C). We call this technique *static* bundling because the bundles are computed *prior* to search.

Search with dynamic bundling. When using a look-ahead strategy such as forward checking for searching the CSP, the effect of an instantiation of a current variable is propagated to the domains of the future variables. In the example of Figure 3 (A), $V_1 \leftarrow d$ results in the elimination of d from the domain of V_3. At this point, one notices that all three values d, e, and f become neighborhood interchangeable for V_2. Dynamic bundling is based on the idea of recomputing the bundles as search proceeds to take advantage of the new opportunities to bundle values enabled by decisions taken along a path of the tree. Figure 3 (D) shows the tree generated by dynamic bundling. In previous work we have established that dynamic bundling is always beneficial: it yields larger bundles and reduces the search effort [6, 7]. This unexpected result can be explained by the fact that, in addition to bundling solutions, dynamic bundling allows us to factor out larger no-goods (non solutions), thus eliminating more 'barren' portions of the search tree. Further, we showed that, in comparison to dynamic bundling, static bundling is prohibitively expensive, particularly ineffective, and should be avoided [7]. In [8], we extended this technique to non-binary constraints, and demonstrated significant improvements in processing time and solution space. In this paper, we show how to use dynamic bundling for processing join queries and compacting the join results.

3 Modeling a join query as a CSP

We show how to model a join query as a CSP using our running example:

```
SELECT R1.A, R1.B, R1.C
FROM R1, R2
WHERE R1.A = R2.A AND R1.B = R2.B AND R1.C = R2.C
```

We map the join query into the following CSP $\mathcal{P} = \{\mathcal{V}, \mathcal{D}, \mathcal{C}\}$, represented by the constraint network of Figure 4:

1. *The attributes as CSP variables.* $\mathcal{V}$ is the set of attributes in the join query. There are 6 variables in our example, which are the attributes `R1.A`, `R2.A`, `R1.B`, `R2.B`, `R1.C`, and `R2.C`.
   For an equi-join query, as it is the case here, the attributes joined using an equality constraint can be represented by a unique variable. In the example above, the CSP representing the query would consist of only 3 variables with `R1.A` = `R2.A`, `R1.B` =`R2.B`, and `R1.C` = `R2.C`. When the query lists the two equated attributes in its `SELECT` clause, the CSP variable is simply repeated in the output.

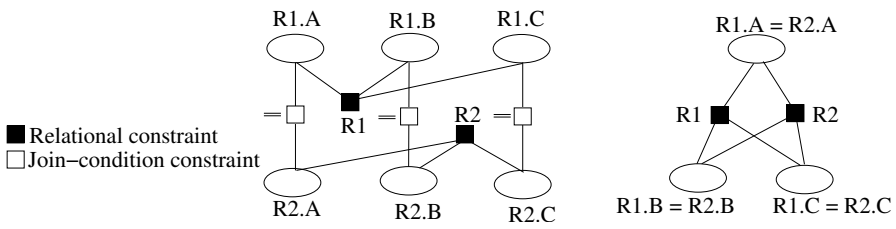


Fig. 4. *Two equivalent formulations of the join as a CSP.*

2. *The attribute values as variable domains.* $\mathcal{D}$ is the set of the domains of the variables. For each attribute, it is the set of values the attribute takes in a relation.
   For an equi-join query, the domain of the CSP variable representing the equated attribute is the union of the set of values that the equated attributes take in their respective relations.
3. *The relations and join conditions as CSP constraints.* $\mathcal{C}$ is the set of constraints of the CSP. These constraints originate from two sources, namely: the relations to be joined and the join conditions. The relations to be joined directly map to CSP constraints that are expressed extensionally. We call these constraints *relational constraint*. The join conditions map directly to CSP constraints expressed intensionally, which we call *join-condition constraints*. In our example, the relations to be joined are `R1` and `R2`, and there are 3 equality constraints due to the join conditions of the query.
   For an equi-join query where the equated attributes are represented by a unique variable, the join condition is implicit in the CSP representation and does not need to be expressed.

Table 1 maps the terminology of databases into that of Constraint Processing:

DB terminology	CSP terminology
Table, relation	Constraint (which we call relational constraint)
Join condition	Constraint (which we call join-condition constraint)
Relation arity	Constraint arity
Attribute	CSP variable
Value of an attribute	Value of a CSP variable
Domain of an attribute	Domain of a CSP variable
Tuple in a table	Tuple in a constraint Tuple allowed by a constraint Tuple consistent with a constraint
Constraint relation (in Constraint Databases)	Constraint of linear (in)quality
A sequence of natural joins	All the solutions of the corresponding CSP

Table 1. *Terminology mapping.*

Our algorithm for bundling non-binary CSPs required that constraints be enumerated [8]. However, for computing interchangeability in the database scenario, we do not have to enumerate the join-condition constraints and store them explicitly. Instead, we proceed as follows. When joining two relations specified in extension, the resulting tuple is checked for consistency with the join-conditions specified in intension as this tuple is being built up. When the values in the partially built tuple are not consistent with a join-condition constraint, the tuple is discarded, as we explain in Section 5.1. This is possible because we are guaranteed that all the CSP variables are present in at least one constraint defined in extension and thus all the join-condition constraints will be checked for consistency.

4 Bundling relations

This section describes the computation of interchangeable values (i.e., a bundle) of an attribute in a relation. Since our join algorithm is a sort-merge algorithm, the relations must first be sorted. Thus, we need to select the order of the attributes for sorting the relations. This order is necessarily static because we cannot afford to re-sort relations during processing. In terms of CSPs, this corresponds to a static ordering of the variables. We first describe our ordering heuristic then the technique for computing interchangeability.

4.1 Heuristic for variable ordering

With $\mathcal{V}$ the set of variables in the CSP representing a query, we denote $\mathcal{V}_q$ a first-in first-out queue of the ordered variables. $\mathcal{V}_q$ is initialized to an arbitrary variable[1]. We also denote $\mathcal{V}_u$ the unordered variables (i.e., $\mathcal{V}_u = \mathcal{V} \setminus \mathcal{V}_q$). Let V_c be the last variable added to $\mathcal{V}_q$. The next variable in the order V_n is chosen from $\mathcal{V}_u$ as follows:

1. Consider the variables $\{V_i\} \subseteq \mathcal{V}_u$ such that V_i is linked with a join-condition constraint C_i to V_c. V_n is selected as the variable for which $\mid \mathcal{V}_u \cap scope(C_i) \mid$ is the smallest.
2. V_n is selected as a variable from the same relation as V_c.
3. V_n is selected arbitrarily from $\mathcal{V}_u$.

If no variables satisfy a rule in the sequence above, the next rule in sequence is applied to $\mathcal{V}_u$. When more than one variable satisfy a rule, the next rule in sequence is applied to discriminate among the qualifying variables. V_n is removed from $\mathcal{V}_u$ and added to $\mathcal{V}_q$. The process is repeated until $\mathcal{V}_u$ is empty. The goal of this ordering is to allow the checking of join-condition constraints as early as possible. For the example of Figure 4, one possible ordering is the sequence `R1.A`, `R2.A`, `R1.B`, `R2.B`, `R1.C`, and `R2.C`. Note that the ordering of the variables affects the size of the generated bundles and that different ordering heuristics need to be investigated.

[1] One can elaborate heuristics for choosing the first variable. One possibility is to exploit the meta-data maintained by the DBMS such as the number of unique values of an attribute. Other heuristics may choose first the attribute that participates in the largest number of constraints. The design and evaluation of such heuristics still needs to be investigated.

4.2 The principle

Given the queue $\mathcal{V}_q$ of ordered variables, we build the bundles dynamically while joining the tuples loaded in memory. Variables in the queue are considered in sequence. The variable under consideration is called the current variable V_c, the set of previous ones is called $\mathcal{V}_p$, and the set of remaining ones $\mathcal{V}_f$. $\mathcal{V}_f$ is initialized to $\mathcal{V}_q$, keeping the same order of variables, and $\mathcal{V}_p$ is set to nil. First, we find a bundle for V_c as described below. Then, we determine the subset of values in the bundle that is consistent with at least one bundle from each of the variables in $\mathcal{V}_f$ with a join-condition constraint with V_c (see Algorithm 2). If such a subset is not empty, we assign it to V_c. In terms of CSPs, this corresponds to instantiating V_c. We move V_c to $\mathcal{V}_p$, and a new V_c is chosen as the first variable in $\mathcal{V}_f$. Otherwise, if the consistent subset for V_c is empty, we compute the next bundle of V_c from the remaining tuples and repeat the above operation. We continue this process until all the variables are instantiated and then output these instantiations as the next nested tuple of the join. Consider the scenario where a next bundle for V_c, an attribute of relation R1, needs to be computed during a sequence of instantiations (see Figure 5).

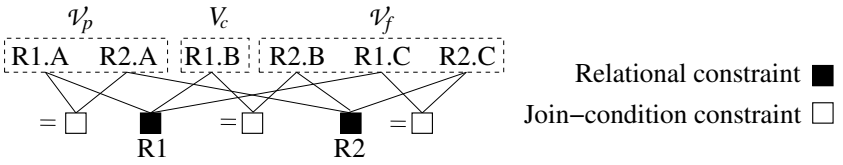


Fig. 5. *Instantiation sequence.*

The bundle depends on the instantiation of variables from R1 in $\mathcal{V}_p$ (i.e., previously instantiated variables). Although the computed bundle of V_c does not directly depend on the instantiations of past variables from R2, the bundle subset to be assigned to V_c must be consistent with those variables of $\mathcal{V}_p$ that share a join-condition constraint with V_c. When such a variable is from R2, then the instantiation of V_c is affected by the instantiations of variables from R2.

Below, we describe the method for computing a bundle of V_c, an attribute of relation R, given that some of the variables of R are in $\mathcal{V}_p$. The bundles are computed on the tuples of R present in the memory, called R′. First, R′ is sorted with the variable coming earliest in the static ordering (see Section 4.1) as the primary key, the one coming second as the secondary key, and so on. The sorting clusters tuples with the same values for variables as they appear in the static ordering.

4.3 Data structures

We first introduce the various data structures used for computing the bundles.

– *Current-Inst* is a record of size equal to the number of variables in the CSP. It is used to store the current instantiations of variables of R in $\mathcal{V}_p$. This corresponds

to a current path in a search tree. When a variable is assigned a bundle of size greater than one, only the smallest value in the bundle is stored in *Current-Inst*, as a representative of the bundle.
- *Processed-Values* is a similar record storing cumulatively all non-representative values of the assigned bundles. While computing bundles of V_c, tuples corresponding to values for V_c in *Processed-Values* are ignored.
- *Current-Constraint* is a selection of the relation R′ (of which V_c is an attribute) such that: (1) Past variables have the values stored in *Current-Inst*, and (2) the value of V_c is greater than the previous instantiation of V_c. Initially, the *Current-Constraint* is set to R′.

The tuples with the same value for V_c in *Current-Constraint* form a partition p, and the value of V_c in this partition is denoted VALUE(p). Figure 6 shows these data-structures under various scenarios. A partition p is marked as *checked* when VALUE(p) is part

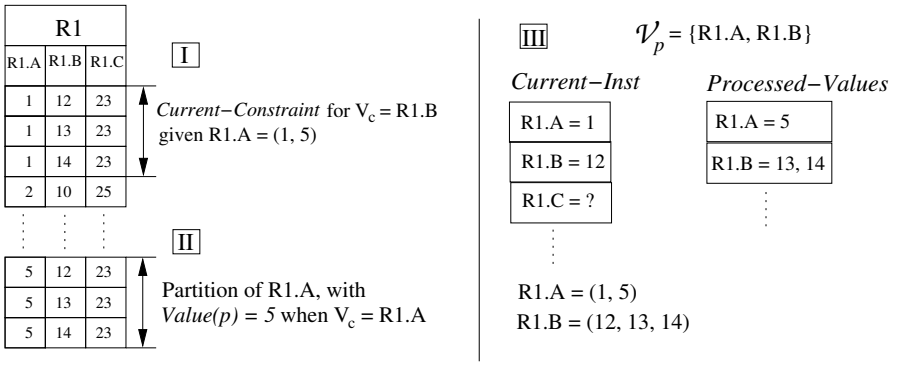


Fig. 6. *Data structures shown under 3 different scenarios.*

of an instantiation bundle or when p is selected to be compared with other partitions. Otherwise, the partition is considered *unchecked*. P_c refers to the unchecked partition with the lowest value of V_c in *Current-Constraint*. When no checked partition exists for V_c, P_c is set to a dummy such as -1.

4.4 Bundle computation

Algorithm 1 computes the next bundle of V_c given P_c. NEXT-PARTITION(p) returns the first *unchecked* partition in *Current-Constraint* following the partition p. For p= -1, NEXT-PARTITION(p) returns the first partition in *Current-Constraint*. P_c moves to the next unchecked partition at every call of Algorithm 1.

Algorithm 1 is called by Algorithm 2 of Section 5 for computing the bundle b_c of V_c and the bundles of the variables V_i connected to V_c with a join-condition constraint. Further, Algorithm 2 determines the subset *Inst* of the bundle b_c that is consistent with the variables V_i. This consistent set of values *Inst* is then used to instantiate

Input: V_c, *Current-Constraint*
$bundle \leftarrow$ `nil`, the bundle to return
$P_c \leftarrow$ NEXT-PARTITION(P_c)
Mark P_c as checked
Push VALUE(P_c) into $bundle$
$P'_c \leftarrow$ NEXT-PARTITION(P_c)
while P'_c **do**
  $t \leftarrow$ tuples of P_c
  $p \leftarrow$ tuples of P'_c
  if $\pi_{\mathcal{V}_f}(t) \equiv \pi_{\mathcal{V}_f}(p)$ **then** push VALUE(P'_c) in $bundle$
  $P'_c \leftarrow$ NEXT-PARTITION(P'_c)
end
Output: $bundle$

Algorithm 1: *Algorithm to generate the next bundle of V_c.*

V_c. This instantiation operation includes the update of the data structures *Current-Inst* and *Processed-Values*. In particular, the values in *Processed-Values* that are lesser than those associated with P_c are deleted.

We can compute all the bundles of V_c by repeatedly calling Algorithm 1, then assigning the returned bundle to V_c until Algorithm 1 returns `nil`. Thus, the algorithm described here implements a lazy approach for computing the bundles and avoids storing the entire partition of the domain of every variable.

In the method described above *Processed-Values* is the data structure that occupies the most space. Whereas all the other data structures have sizes proportional to the number of variables (and therefore cause insignificant memory overhead), the size of *Processed-Values* depends on the number of tuples and the amount of bundling performed. The worst-case scenario for *Processed-Values* occurs when all the values of a variable are in a single bundle. In this case, *Processed-Values* will hold all the unique values of that variable. Suppose that there are N tuples in the relation, the relation has k attributes, and the number of unique values of the variable is $\frac{N}{l}$, where l is the average length of each partition of V_c. Then, the size of *Processed-Values* is $\frac{N}{l \times k}$ tuples. However, if this bundle goes on to form a result tuple, it will save more space than required for bundling. Even when this bundle fails to yield a result tuple, it still saves on many comparisons thereby speeding up computation. Our current implementation is a proof of concept, and we are investigating how to improve its efficiency, possibly by the use of bit-maps.

5 Join algorithm using bundling

This section shows the use of bundling while computing a join as a depth-first search. The join algorithm discussed in this section is based on the Progressive Merge Join. The technique discussed here can be easily adapted to the simpler sort-merge join since PMJ is just an extension of sort-merge. We first describe the in-memory join algorithm, and then place it in the schema of the external join algorithm.

5.1 Join computation in memory

We present here the algorithm to join two sub-sets of relations that are currently in memory. For the sake of readability, Algorithm 2 is restricted to binary join conditions (where the join conditions are between two attributes from different relations). It can be easily extended to join conditions with more than two attributes. Algorithm 2 takes

```
Input: depth, Current-Solution
while (depth ≤| V |) and (depth ≥ 1) do
    V_c ← Variable[depth]
    b_c ← next bundle for V_c using Algorithm 1
    if b_c is empty then
        BACKTRACK, decrement depth, and GOTO L1
    end
    Inst ← b_c
    repeat
        foreach V_i ∈ V_f connected to V_c by a join-condition constraint do
            Consider R_i the relational constraint that applies to V_i
            Select r_i from R_i according to Current-Solution
            repeat
                Find a bundle b_i applying Algorithm 1 on V_i and r_i
                if b_i is empty then break
                I_i ← COMMON(b_i, b_c)
            until I_i is not empty;
            if no b_i then BACKTRACK, decrement depth and Goto L1
        end
        Inst ← COMMON(I_0, I_1, ..., I_n)
    until Inst is not empty;
    Instantiate V_c with Inst
    Current-Solution[V_c] ← Inst
    Increment depth
    L1:
end
Output: Current-Solution
```

Algorithm 2: *Algorithm to compute the in-memory join using bundling.*

as input the level of V_c in the search tree (i.e., *depth*) and the current path represented by the data structure *Current-Solution*. *Current-Solution* is a record that stores the assigned bundles to the variables in $\mathcal{V}_p$ (note that *Current-Solution* cannot be obtained from *Current-Inst* and *Processed-Values*). *Variable*[] is the array of variables in the same order as the static ordering of Section 4.1. When BACKTRACK is called the value for *Variable*[*depth*] in *Current-Inst* is reset, the *Processed-Values* for the variable is emptied, the value for the variable in *Current-Solution* is reset, and *Current-Constraint* is re-computed, thus undoing the effects of the previous instantiation. The function COMMON() computes the set of values in the input bundles that are consistent with each other

according to the applicable join-condition constraints. Because this algorithm combines sorting and constraint propagation with bundling, it produces solutions quickly, which compensates for the effort spent on bundling.

5.2 The structure of the overall join

We have discussed join computation of tuples that are in memory and now describe the steps for computing the join of complete relations using our in-memory join algorithm, Algorithm 2. The join of the two input relations is computed using an approach similar to the PMJ, in the two phases shown below.

Sorting phase. The sorting phase is similar to the PMJ, except that for joining the pages of relations in memory we use the bundling-based technique of Algorithm 2. The sorting phase produces the early results and also a sorted sub-list or runs of the relations. These runs are stored back on disk and used in the merging phase of the join. Since the memory is filled with pages from both the relations, the number of runs generated for each relation is $\frac{2N}{M}$.

Merging phase. In the merging phase, as for the PMJ, $\frac{M^2}{4\times N}$ pages from every run created from the sorting phase are kept in memory. Let P_i^{rel} represent the pages in memory of relation rel and i^{th} run, where $rel \in \{0,1\}$ and $i \in \{1,2,\ldots,\frac{2N}{M}\}$. We store one solution each from the join of pages in an array, called *solution*, defined by Equation (1).

$$solution[i][j] = P_i^0 \bowtie P_j^1\ ,\ i \neq j \tag{1}$$

The minimum solution from *solution*[][] is the next result of the join. The next solution from the pages that gave the minimum solution is then computed and used to fill the corresponding place in *solution*[][]. A page P_i^{rel} is removed from memory and replaced with another page from the same run only if it satisfies the following two conditions for every page P_j^{1-rel}. P_i^{rel} is being joined with: (1) No more join tuples result from $P_i^{rel} \bowtie P_j^{1-rel}$, and (2) the last tuple in P_i^{rel} is less than that of P_j^{rel}. The tuples are compared using the same comparison criteria as the ones used for sorting. These conditions ensure the tuples are produced in sorted order (during the merging phase) and that the algorithm is complete.

6 Implementation and experiments

One of the goals of the XXL library [2] is to provide an infrastructure for evaluating new algorithms in databases. For example, PMJ was evaluated experimentally using this library. In our experience, XXL provides a good infrastructure for building new database algorithms through its rich cursor algebra built on top of Java's iterator interface. We implemented our join algorithm by extending the BUFFEREDCURSOR class of the XXL library.

The current implementation is a proof of concept and offers much room for improvement. To show the feasibility of our technique, we tested our join algorithm on randomly generated relations and on data from a real-world resource allocation problem in development in our group. For the real-world application, we computed the sequence of the natural join of three relations, with respectively 3, 4, and 3 attributes. The corresponding CSP has 4 variables, with domain size 3, 3, 300, and 250 respectively. The resulting join of size 69 was compressed down to 32 nested tuples. For the random problems, we used relations of n = 10,000 tuples. We set the page size to 200 tuples and the available memory size to $M = \frac{2N}{5}$, where $N = 10000/200$. We executed the query of our running example over five such pairs of relations. The result of the query had an average of 8,500 tuples, signifying that the query was selective. The number of tuples produced by bundling was reduced to 5,760 bundled tuples, an average of 1.48 tuples per bundle. The number of pages saved was more than $\frac{N}{4}$. Even if the worst-case scenario for the join occurred for every in-memory join (which is a highly unlikely event), the additional cost due to bundling is given by $\frac{N}{l \times k}$, where $\frac{N}{l}$ is the number of unique values of an attribute and k is the number of attributes in one relation (which is 3 here). For the worst case when $l = 1$, there are still savings in terms of pages. Again, the worst-case described here is of the current implementation, which offers much room for improvement.

7 Related work

The idea of data compression is not new and is used in compressed database systems [11]. In these systems, data is stored in a compressed format on disk. It is decompressed either while loading it into memory or while processing a query. The compression algorithms are applied at the attribute level and are typically dictionary-based techniques, which are less CPU-intensive than other classical compression techniques [12]. Although most of the work in compressed databases applied to relations with numerical attributes [11], some work on string attributes has also been done [13]. Another feature of compressed databases that differs from our approach is that the query results passed to the next operator are uncompressed and likely to be large. Our work differs from the above in that we reduce some of the redundancy present between tuples of a given relation. Our techniques are independent of the data type of an attribute. Further, the results of our queries are compacted, thereby assisting the next operator and reducing the storage of materialized views on disk. When these compacted results are loaded into memory for query processing, the de-compaction is effectively cost-free. The only costs associated with our techniques are those for performing the compaction. Finally, the compaction is carried out while the query is being evaluated, and is not a distinct function performed in separation.

In [14], Mamoulis and Papadias present a spatial-join algorithm using mechanisms of search with forward checking, which are fundamental in Constraint Processing. They store the relations representing spatial data in R-tree structures and use the structures to avoid unnecessary operations when computing a join. The constraints under consideration are binary. The key idea is to reduce the computational cost by propagating the effects of search, thereby detecting failure early. Our technique is not restricted to bi-

nary constraints, and is applicable to constraints of any arity. Further, it differs from the approach of Mamoulis and Papadias in that it reduces I/O operation and compacts join results in addition to reducing computational operations.

Bernstein and Chiu [15], Wallace et al. [16], Bayardo [17], Miranker et al. [18] exploit the standard consistency checking techniques of Constraint Processing to reduce the number of the intermediate tuples of a sequence of joins. While Wallace et al. consider Datalog queries, Bayardo and Miranker et al. study relational and object-oriented databases. Our CSP model of join query differs from their work in that the constraints in our model include both relational and join-condition constraints, whereas the latter models the relational constraints as CSP variables and only the join-condition constraints as CSP constraints. Thus, our model is finer in that it allows a more flexible ordering of the variables of the CSP, which increases the performance of bundling. Finally, Rich et al. [19] propose to group the tuples with the same value of the join attribute (redundant value). Their approach does not bundle up the values of the join attribute or exploit that redundancies that may be present in the grouped sub-relations.

8 Conclusions and Future work

We described a new method for computing interchangeability and use it in a new join algorithm, thus establishing the usefulness of dynamic bundling techniques for join computation. In the future, we plan to address the following the issues:

- Refine our implementation by the use of lighter data structures.
- Test the usefulness of these techniques in the context of constraint databases where the value of an attribute is a continuous interval such as spatial databases [20].
- Conduct thorough evaluations of overall performance and overhead (memory and cpu) on different data distributions. And,
- Investigate the benefit of using bundling for query size estimation and materialized views.

Acknowledgments

This work is supported by the Maude Hammond Fling Faculty Research Fellowship and CAREER Award #0133568 from the National Science Foundation. We are grateful to the reviewers for their comments.

References

1. Dittrich, J.P., Seeger, B., Taylor, D.S., Widmayer, P.: On Producing Join Results Early. In: 22^{nd} ACM Symposium on Principles of Database Systems. (2003) 134–142
2. den Bercken, J.V., Blohsfeld, B., Dittrich, J.P., Krämer, J., Schäfer, T., Schneider, M., Seeger, B.: XXL–A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In: 27^{th} International Conference on Very Large Data Bases. (2001) 39–48

3. Rossi, F., Petrie, C., Dhar, V.: On the Equivalence of Constraint Satisfaction Problems. In: Proc. of the 9 th ECAI, Stockholm, Sweden (1990) 550–556
4. Bacchus, F., van Beek, P.: On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems Using the Hidden Variable Method. In: Proc. of AAAI-98, Madison, Wisconsin (1998) 311–318
5. Bessière, C., Meseguer, P., Freuder, E.C., Larrosa, J.: On Forward Checking for Non-binary Constraint Satisfaction. Artificial Intelligence **141 (1-2)** (2002) 205–224
6. Beckwith, A.M., Choueiry, B.Y., Zou, H.: How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. In: AI 2001: Advances in Artificial Intelligence, 14th Australian Joint Conference on Artificial Intelligence. LNAI Vol. 2256, Adelaide, Australia, Springer Verlag (2001) 50–61
7. Choueiry, B.Y., Davis, A.M.: Dynamic Bundling: Less Effort for More Solutions. In Koenig, S., Holte, R., eds.: 5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002). Volume 2371 of Lecture Notes in Artificial Intelligence., Springer Verlag (2002) 64–82
8. Lal, A., Choueiry, B.Y.: Dynamic Detection and Exploitation of Value Symmetries for Non-Binary Finite CSPs. In: Third International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'03), Kinsale, County Cork, Ireland (2003) 112–126
9. Freuder, E.C.: Eliminating Interchangeable Values in Constraint Satisfaction Problems. In: Proc. of AAAI-91, Anaheim, CA (1991) 227–233
10. Haselböck, A.: Exploiting Interchangeabilities in Constraint Satisfaction Problems. In: Proc. of the 13 th IJCAI, Chambéry, France (1993) 282–287
11. Roth, M.A., Horn, S.J.V.: Database compression. SIGMOD Record **22** (1993) 31–39
12. Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. SIGMOD Record **29** (2000) 55–67
13. Chen, Z., Gehrke, J., Korn, F.: Query optimization in compressed database systems. In: 2001 ACM International Conference on Management of Data (SIGMOD). (2001) 271–282
14. Mamoulis, N., Papadias, D.: Constraint-based Algorithms for Computing Clique Intersection Joins. In: Sixth ACM International Symposium on Advances in Geographic Information Systems. (1998) 118–123
15. Bernstein, P.A., Chiu, D.M.W.: Using semi-joins to solve relational queries. J. ACM **28** (1981) 25–40
16. Wallace, M., Bressan, S., Provost, T.L.: Magic checking: Constraint checking for database query optimization. In: CDB 1995. (1995) 148–166
17. Bayardo, R.J.: Processing Multi-Join Queries. PhD thesis, University of Texas, Austin (1996)
18. Miranker, D.P., Bayardo, R.J., Samoladas, V.: Query evaluation as constraint search; an overview of early results. In Gaede, V., Brodsky, A., Günther, O., Srivastava, D., Vianu, V., Wallace, M., eds.: Second International Workshop on Constraint Database Systems (CDB '97). LNCS 1191, Springer (1997) 53–63
19. Rich, C., Rosenthal, A., Scholl, M.H.: Reducing duplicate work in relational join(s): A unified approach. In: International Conference on Information Systems and Management of Data. (1993) 87–102
20. Revesz, P.: Introduction to Constraint Databases. Springer-Verlag, New York (2001)