Algorithms for compressing and joining relations

Jeppe Nejsum Madsen

Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
Denmark

E-mail: nejsum@diku.dk

Copenhagen STL Report 2002-1, January 2002

Contents

Chapter 1

Introduction

A constraint satisfaction problem involves the assignment of values to variables subject to a set of constraints. A wide variety of problems can be viewed as constraint satisfaction problems:

- Many classic combinatorial problems, such as SATISFIABILITY from propositional logic and COLORABILITY from graph theory can be formulated as constraint satisfaction problems.

- Many planning and scheduling problems can be formulated as constraint satisfaction problems. In these problems the task is to create e.g., a production plan for a number of products on different machines. Constraints restrict which products can be produced on which machines and the order in which the products should be produced.

- In product configuration, a product with many options is modelled with all the available options and the constraints between them. The user can then be guided through the selection of options, with only legal combinations available. For e.g. a car, a constraint would be that a convertible model would not be available with a sun roof option.

- Combinatorial puzzles can often be formulated as constraint satisfaction problems. A classic example is the n-queen problem where n queens must be placed on a chess board of size $n \times n$ so that no queen can capture any other queen.

Constraints can be described using relational tables, listing valid values for combinations of variables. In this report, we describe algorithms for compressing and joining such tables of relational data. While the techniques are general, and thus applicable in e.g. relational database systems, we study the algorithms in the context of constraint satisfaction. Similar algorithms are used in the software product Array Database^{TM}, which is a

commercial solver for constraint satisfaction problems, created by the company Array Technology A/S. This enables us to compare the performance of our algorithms with existing algorithms used in a commercial product.

The model of computation used in this report is the *word RAM* (WRAM) model, which allows unit cost Boolean operations on w-bit *words* (see e.g. [1]). The structure of the report is as follows:

Chapter 2 Contains a formal definition of a constraint satisfaction problem with examples and identifies the links with relational databases.

Chapter 3 Contains a detailed description of the *compress* operation which compresses relational data by storing tuples as Cartesian product arguments.

Chapter 4 Contains a detailed description of the *join* operation, which joins two compressed relations.

Chapter 5 Contains an overview of an implementation of the algorithms described in Chapters 3 and 4.

Chapter 6 Contains a performance study of the implemented algorithms and compares the results with existing C++ and APL implementations.

Chapter 7 Contains a summary of the results obtained.

We conclude some of the chapters with a section entitled "Open Problems". This section lists some directions for future research that have been identified during the work with this report, but which we, due to time constraints, have not been able to pursue any further.

Chapter 2

Background

2.1 Constraint Satisfaction Problems

We will only consider constraint satisfaction problems in which there are a finite number of variables, each variable having a finite number of possible values.

Definition 2.1. Formally, the input of a *constraint satisfaction problem*, or *CSP* for short, is a triple $\mathcal{I} = (X, D, C)$ where

1. $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables.
2. D is a function that maps each variable x_i in X to a finite set of values, written $D(x_i)$, which it is allowed to take. The set $D(x_i)$, called the *domain* of x_i, is also denoted D_{x_i}.
3. $C = \{c_1, c_2, \ldots, c_m\}$ is a finite set of constraints. Each constraint c_j is a pair (S_j, R_j) where $S_j = \{x_{j_1}, x_{j_2}, \ldots, x_{j_{k_j}}\} \subseteq X$ is the *scope* of the constraint, $R_j \subseteq D_{x_{j_1}} \times D_{x_{j_2}} \times \cdots \times D_{x_{j_{k_j}}}$ is the *constraint relation* that specifies the combination of values which the constraint allows, and k_j denotes the *arity* of the constraint.

Definition 2.2. Let $\mathcal{I} = (X, D, C)$ be the input of a constraint satisfaction problem. An *assignment* to X is an n-tuple $t \in D_{x_1} \times D_{x_2} \times \cdots \times D_{x_n}$. A *solution* to the problem is an assignment t such that $t_{|S} \in R$ for all $(S, R) \in C$. Here $t_{|S}$ is a tuple obtained from t by removing entries corresponding to the variables in $X - S$.

We may want to find

- the set of all solutions, which will be denoted $Sol(\mathcal{I})$,
- a solution, with no preference as to which one,

- an optimal solution, where optimality is defined with an objective function defined in terms of some or all of the variables of the CSP.

Example 2.1. We will now construct a simple CSP to illustrate the definitions. Let $X = \{a, b, c, d\}$ and $D(x) = \{0, 1\}$ for $x \in X$ (i.e., all variables are in the Boolean domain). Let $C = \{c_1, c_2\}$ where

$$\begin{aligned} c_1 &= (\{a, b, c\}, \{<0,0,0>, <0,0,1>, <0,1,0>, <0,1,1>, \\ &\quad <1,0,1>, <1,1,0>, <1,1,1>\}) \\ c_2 &= (\{b, c, d\}, \{<0,0,0>, <0,0,1>, <0,1,1>, <1,0,1>, \\ &\quad <1,1,1>\}). \end{aligned}$$

To improve readability, we usually show the constraints as tables containing the valid tuples, with one column for each variable. The constraints c_1 and c_2 are shown below:

$c_1 =$

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	0	1
1	1	0
1	1	1

$c_2 =$

b	c	d
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

The constraints can sometimes be interpreted as statements in symbolic logic[1]. That is the case with the constraints in this example, which can be expressed in propositional logic. The corresponding statements are shown below.

$$\begin{aligned} a \rightarrow (b \vee c) \qquad & c_1 \\ (b \vee c) \rightarrow d \qquad & c_2 \end{aligned}$$

It is easy to see that one solution to the above problem is $(0, 0, 0, 0)$. The set of all solutions is:

[1]Indeed, symbolic logic is often used to define the constraints. In this case, the scope of a constraint is the set of variables in the symbolic expression and the constraint relation is the set of tuples which satisfies the expression

a	b	c	d
0	0	0	0
0	0	0	1
0	0	1	1
1	0	1	1
0	1	0	1
1	1	0	1
0	1	1	1
1	1	1	1

□

2.2 Links with relational database theory

As noted in [10] the definitions given so far have close connections to relational database theory, initially described by Codd [5]. Within this theory a database is a finite set of *relations*:

Definition 2.3 ([5])**.** A *relation* consists of a *scheme* and an *instance*:

1. A *scheme* is a finite set of attributes. Each attribute is associated with a set of values, called its *domain*.

2. A *tuple* over a scheme is a mapping that associates with each attribute of the scheme a value from its corresponding domain.

3. An *instance* over a scheme is a finite set of tuples over that scheme.

Thus, for any input of a constraint satisfaction problem $\mathcal{I} = (X, D, C)$, the following connections can be established:

- The variables in X can be interpreted as attributes.
- The domains D_x are the domains of these attributes.
- The valid combinations of values with scope $S \subseteq X$ is a tuple over the scheme with set of attributes S.

This gives rise to two alternative views of a constraint satisfaction problem with input $\mathcal{I} = (X, D, C)$:

1. The set of all solutions can be represented by the database consisting of a single relation with scheme X and instance $Sol(\mathcal{I})$.

2. The input can be represented by the database $\{R_{(s,r)} \mid (s, r) \in C\}$ where $R_{(s,r)}$ has scheme s and instance r.

Many operations have been defined on relations. These operations are part of the *relational algebra* [5]. Only one of those operations, *join*, will be used in this report. It is defined as follows [2].

Definition 2.4. Let R be a relation with scheme Y and instance r. Let S be a relation with scheme Z and instance s. The *join* of R and S, denoted $R \bowtie S$, is defined to be the relation with scheme $Y \cup Z$ and an instance containing the following set of tuples:

$$\{t \mid t \text{ is a tuple over } Y \cup Z \text{ , } t_{|Y} \in r, t_{|Z} \in s\}$$

Again, $t_{|Z}$ denotes the tuple t restricted to Z.

It follows from these definitions that for a CSP with input $\mathcal{I} = (X, D, C)$, $Sol(\mathcal{I})$ is equal to the relation $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$, where R_j is the relation corresponding to the constraint c_j in C.

The following table, reproduced from [19], summarises the terminology:

Constraint Terminology		Database Terminology
constraint satisfaction problem	$\equiv$	database
variable	$\equiv$	attribute
domain	$\equiv$	attribute domain
constraint	$\equiv$	relation
constraint scope	$\equiv$	scheme
constraint relation	$\equiv$	instance
set of solutions	$\equiv$	join of all tables

2.3 Complexity of constraint satisfaction

Given a CSP, we can derive a corresponding decision problem:

Definition 2.5. A decision version of a constraint satisfaction problem has a triple (X, D, C) as an input, where X, D and C are as in Definition 2.1, and yields "yes" or "no" as an output. The output is "yes" if the problem has a solution and "no" otherwise. This decision problem is denoted CSP_D.

Clearly, finding a solution to any form of CSP, can be used to solve the corresponding decision problem, i.e., finding a solution, all solutions or an optimal solution is as difficult as CSP_D. In the following we will prove that CSP_D is $\mathcal{NP}$-complete. First we need to introduce a well-known $\mathcal{NP}$-complete decision problem (see, e.g., [6] for more details):

[2] The operation defined here is sometimes referred to as *natural join* to distinguish from the more general *theta join*. Here we use the short form since it does not give rise to confusion.

Definition 2.6 (3-CNF-SAT)**.** An input of 3-CNF-SAT is a Boolean formula composed of

1. Boolean variables: $x_1, x_2, \ldots$;
2. Boolean connectives such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT); and
3. parentheses.

A *literal* in a Boolean formula is an occurrence of a variable or its negation. A Boolean formula is in *conjuntive normal form*, or *CNF*, if it is expressed as a logical AND of clauses, each of which is the logical OR or one or more literals. A Boolean formula is in *3-conjunctive normal form*, or *3-CNF*, if each clause has exactly three distinct literals.

In 3-CNF-SAT, we are asked whether a given Boolean formula in 3-CNF has a satisfying assignment, i.e., a set of values for the variables that causes the formula to evaluate to true.

Lemma 2.1. 3-CNF-SAT is polynomial-time reducible to CSP_D, i.e., 3-CNF-SAT $\leq_P \mathrm{CSP}_D$.

Proof. Let $\theta = C_1 \wedge C_2 \wedge \cdots C_k$ be a Boolean formula in 3-CNF with k clauses. From θ we need to create the input (X, D, C) to the decision problem CSP_D, where X is the variables, D the variable domains and C the constraints. It is created as follows.

1. For each distinct variable x_i in θ, we add x_i to X.
2. For each variable x_i in X, we let $D_{x_i} = \{0, 1\}$.
3. For each clause $C_r = (l_1 \vee l_2 \vee l_3)$ in θ, we create a constraint (S, R). The scope S contains the (not necessarily distinct) variables appearing in the literals l_1, l_2 and l_3, R contains a 3-tuple for each satisfying assignment to C_r.

This reduction is linear in the number of clauses in θ since there are at most $3k$ variables, k relations, each relation having at most $2^3 = 8$ tuples. It is clear that the 3-CNF form is satisfiable if and only if CSP_D has a solution. □

Theorem 2.2. CSP_D is $\mathcal{NP}$-complete.

Proof. Let $\mathcal{I} = (X, D, C)$ be the input of a CSP decision problem where X is the variables, D the variable domains and C the constraints. First we note that $\mathcal{CSP}_D$ is in $\mathcal{NP}$: Given an assignment t, verify $t_{|S} \in R$ for each $(S, R) \in C$. The verification can be done in time plynomial in the size of $\mathcal{I}$.

By $\mathcal{NP}$-completeness of 3-CNF-SAT and Lemma 2.1, $\mathcal{CSP}_D$ is $\mathcal{NP}$-complete. □

2.4 Solving constraint satisfaction problems

A CSP can be solved using the generate-and-test method, where each possible combination of the variables is systematically generated and then tested to see if it satisfies all the constraints. The first such combination found is a solution. The number of combinations generated in the worst case is the size of the Cartesian product of all the variable domains, i.e., exponential in the number of variables.

By using backtracking, a more efficient method can be constructed. Variables are instantiated sequentially and when all variables in a constraint have been instantiated, the constraint is checked. Whenever a partial instantiation violates a constraint, backtracking is performed to the most recently instantiated variable with valid values left in the domain, thereby eliminating a subspace from the Cartesian product of the variable domains.

While the backtracking method is never worse than the generate-and-test method,it is still exponential in the worst case.

2.4.1 Consistency techniques

A CSP where constraints are either unary or binary, is called a *Binary CSP*. For simplicity we only consider binary CSPs in this section. The principles are however general since it is possible to convert any general CSP to an equivalent binary CSP [20].

Binary constraint satisfaction problems can be depicted as an undirected graph, called a constraint graph or constraint network. In a constraint graph, each node represents a variable, and each arc represents a constraint. There is a node for each variable, and each node is augmented with the set of domain values which are valid for the corresponding variable. There is an arc for each constraint. An arc connects the nodes representing the variables in the scope of the constraint[3]. Each arc is augmented with the constraint relation for the corresponding constraint.

Example 2.2. Consider a binary CSP with the following input, where the constraints are specified using symbolic logic:

$$\begin{aligned} X &= \{a, b, c, d\} \\ D(x) &= \{1, \ldots, k\} \quad \forall x \in X \\ C &= \{a \neq b, a \neq c, a \neq d, b \neq c, b \neq d, c \neq d\} \end{aligned}$$

The corresponding constraint graph is shown in Figure 2.1 on the next page. For brevity, the valid domain values for the variables are not shown. □

[3] A general CSP can be depicted as a constraint hypergraph[3] where each hyperarc connects more than two nodes.

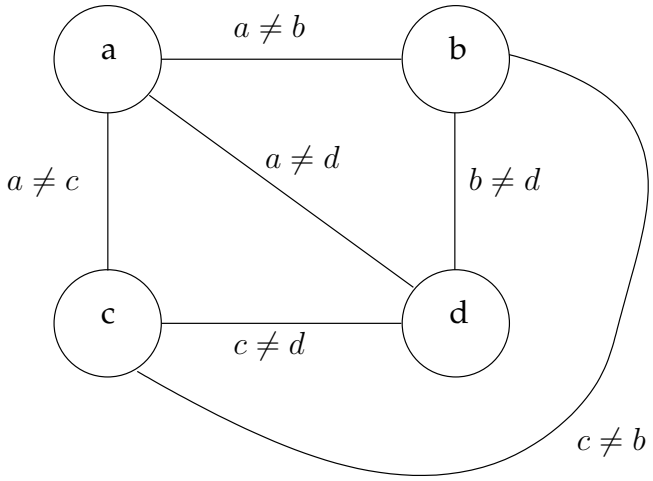


Figure 2.1: Constraint graph for a binary CSP

In the following, we assume that the CSP is represented by a graph $G = (V, E)$ where V is the set of nodes corresponding to the variables and $E \subseteq V \times V$ is the set of arcs corresponding to the constraints. With the above definitions, finding a solution to a CSP is equivalent to finding a global consistent labelling of the nodes in the corresponding constraint graph such that the constraints on all arcs are satisfied. To reduce the search space, the graph can be preprocessed to obtain various forms of local consistency. A number of different degrees of local consistency exists.

Node consistency: A node is *node consistent* (NC) if all unary constrains on the corresponding variable are satisfied by the valid values in the variable's domain. A CSP is node consistent if all nodes are node consistent.

Arc consistency: An arc (V_i, V_j) is *arc consistent* (AC) if, for every value x_i in the current domain of V_i, there exists a value x_j in the domain of V_j such that $V_i = x_i$ and $V_j = y_j$ is consistent with the constraint between V_i and V_j. A CSP is arc consistent if all arcs are arc consistent.

Path consistency: A natural extension of arc consistency is path consistency (PC). A path $(V_i, \ldots, V_j)$ is *path consistent* if, for all pairs (x_i, x_j) of values from the current domains of V_i and V_j, there exists values $x_{i+1}, \ldots, x_{j-1}$ for the current domains of $V_{i+1}, \ldots, V_{j-1}$ such that all constraints $(V_k, V_{k+1}), i \leq k < j$ are consistent . A CSP is path consistent if all paths are path consistent.

The above consistency notions are instances of a general consistency notion, called k-consistency. A CSP is *k-consistent*, if any consistent instantiation of $k-1$ variables can be extended to a consistent instantiation of k

variables by instantiating any of the remaining variables. A CSP as *strongly k-consistent* if the CSP is j-consistent for all $j \leq k$. We then have that node consistency is equivalent to strong 1-consistency, arc consistency is equivalent to strong 2-consistency, and path consistency is equivalent to strong 3-consistency. In [14], Kumar surveys algorithms to obtain different levels of consistency.

For a constraint graph with n nodes, it is clear that if the graph is strongly n-consistent, a solution can be found without search. Unfortunately, algorithms for achieving n-consistency in an n-node graph are exponential. So in general, backtracking cannot be avoided if the graph is k-consistent for $k < n$, thus consistency algorithms do not eliminate the need for search, but reduces the solution space which need to be searched.

General CSPs are usually solved by using some form of consistency algorithm in combination with a backtracking algorithm. The various backtracking algorithms differ in how they backtrack, what information is propagated etc. A theoretical evaluation of various backtracking algorithms can be found in [13].

Some CSPs can be solved without backtracking. This is usually due to the structure of the problem instance or to the nature of the constraints used. Freuder [9] describes conditions on the relationship between the constraints which allows solutions to be found without backtracking.

2.5 Array-based logic

Array-based logic (ABL) is an alternative approach to logical reasoning in the Boolean constraint domain which was first described by Franksen [8].

ABL is based on a geometrical representation of logic in terms of nested data arrays. The nested model of data provided by array theory was originally developed by More [18]. ABL simplifies the process of logical reasoning, since reasoning can be carried out in terms of only three, purely geometrical operations:

1. An outer product in the Cartesian sense, which establishes arrays of all possible solutions.

2. An operation, called *generalised transposition*, which takes out planes (or hyperplanes in the general case) by equating two or more axes.

3. A reduction by logical addition or multiplication which is used to derive information from the array.

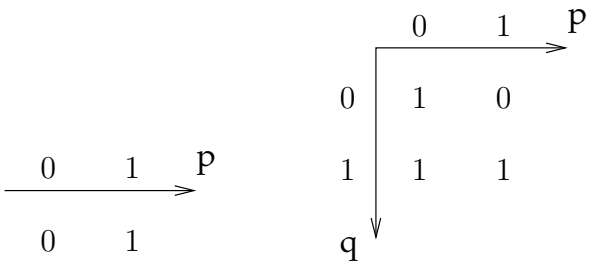
Example 2.3. In this example (which is based on [17, Ch. 1]) we demonstrate the basic operations used in array-based logic. Consider a binary CSP with the following input, where the constraints are specified using

symbolic logic:

$$\begin{aligned} X &= \{p, q\} \\ D(x) &= \{0, 1\} \quad \forall x \in X \\ C &= \{p, p \rightarrow q\} \end{aligned}$$

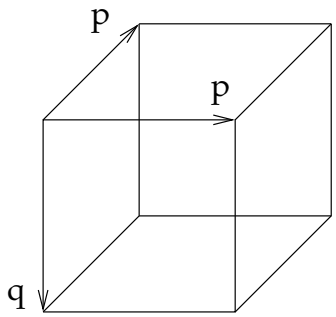
First, we create a binary array for each constraint. A j-ary constraint gives rise to a j-dimensional array. The array is essentially a truth table for the constraint, where each axis of the array corresponds to a variable and the values on the axis represents the domain values for the corresponding variable. Each j-dimensional coordinate in the array contains 1 or 0, which specifies whether the variable values, corresponding to the coordinate, satisfies the constraint or not.

Møller shows how to create the array for the common connectives in propositional logic using an *outer product*. For the two constraints in this example we get the following arrays of dimension 1 and 2 respectively:

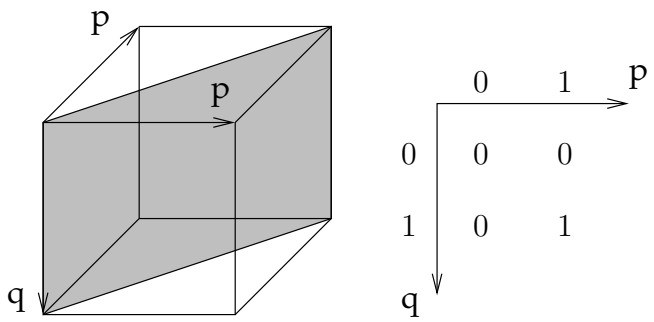


As can be seen, the array representing the constraint p only allows the value $p = 1$ and the array representing the constraint $p \rightarrow q$ allows all combinations except $p = 0$ and $q = 1$. This is consistent with the laws of propositional logic.

A solution to the CSP is an assignment, which satisfies the conjunction of the individual constraints, in this case $p \wedge (p \rightarrow q)$. An array corresponding to the conjunction of the individual constraints can be created using an outer product between the arrays representing the individual constraints. In this case, we get a 3-dimensional array with the following axes:



Note that the variable p is associated with more than one axis. This is not a suitable representation, since only the diagonal of the two p axes satisfies the implicit constraint that p can only be assigned one value at any given time. We can *fuse* such superfluous axes by a projection on the diagonal plane[4]. This projection is the array operation called generalised transposition. The diagonal plane and the resulting projection is shown below:



We have now calculated all solutions to the problem. As can be seen from the above array, only $p = 1$ and $q = 1$ satisfies all constraints.

When a binary array is established by means of the outer product and the generalised transposition, the next step is to derive information from the array by projections on different subspaces. This is the process of variable elimination, which is carried out by the array operation *reduction*. □

The outer product combined with colligation between two binary arrays is equivalent to a join between two relations.

2.5.1 Constraint satisfaction with array-based logic

The graph-based methods for solving CSPs, described briefly in Section 2.4, are all offline or static in the sense that all constraints must be known before we try to solve the problem. In many applications, we need the ability to dynamically add constraints to the problem (because of user interaction, changes in the system environment etc.). This type of CSP is called *dynamic CSP* (DCSP) and the concept was first introduced by Dechter and Decther [7].

The consistency methods described, are not able to reuse any information from the old problem, so we essentially need to solve the problem again. This may be prohibitive in an interactive or real time environment where the result may be needed fast. Specialised consistency and search algorithms have been devised for DCSPs.

Møller [17] described how ABL can be used to solve some dynamic CSPs. In Møller's DSCP model, the variables and their domains are static but the constraints are split in two sets:

[4] In the general case, this will be a projection on the diagonal hyperplane

1. A static set of constraints which must always be satisfied. There are no bounds on the type or scope of the static constraints.

2. A dynamic, initially empty, set of unary constraints. Only unary constraints are allowed but constraints can be added or removed.

At first it seems like a major restriction that only unary constraints are allowed to be dynamic, but for many applications this is sufficient (i.e., the user selects/discards values from a variable's domain).

The method described by Møller essentially creates the set of all solutions that satisfies the static constraints by repeatedly joining constraint relations until the remaining relations form a certain structure. Having computed the set of solutions that satisfies the static constraints, it is easy to add an unary constraint. All we need is to mark the tuples of the current set of solutions, which do not satisfy the new constraint, as invalid. We can remove the constraint by removing the marks.

The join operation is one of the two algorithms covered in this report. The order in which relations are joined is important for the applicability of ABL (probably more important than the join operation itself). This problem is equivalent to query optimisation in relational database management systems. In this report we focus on the low level operations, and we will therefore not study this problem further.

2.5.2 Compressed relations

While ABL is simple in theory, it quickly becomes impractical to implement on a computer. This is caused by the combinatorial explosion of the solution space (e.g., an unconstrained Boolean problem with n variables has 2^n possible solutions).

Møller [17] introduced a concept which is very important in a computer implementation: compression of relations. By storing the tuples of a relation in a compact way, and modifying the algorithms to work with this compressed format, the practical applicability of ABL is greatly increased.

The data representation introduced for relations, represents disjoint legal subspaces as operands to a Cartesian product operator, rather than the individual tuples.

Example 2.4. Consider the set of valid tuples for relation c_2 in Example 2.1. They are explicitly enumerated below:

$$\{< 0,0,0 >, < 0,0,1 >, < 0,1,1 >, < 1,0,1 >, < 1,1,1 >\}$$

An alternative representation is to use, where feasible, a Cartesian product to generate the set of valid tuples. Thus the same set can be represented as follows:

$$\{< 0, 0, 0 >\} \cup (\{0, 1\} \times \{0, 1\} \times \{1\})$$

The last representation is what we refer to as the *compressed relation form*. When using the tabular notation, we display the relation as shown below:

b	c	d
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{compress}$

b	c	d
0	0	0
$\{0, 1\}$	$\{0, 1\}$	1

When using the tabular notation, a Cartesian product is implied between sets on the same row. In addition, we omit the set delimiters $\{\}$ when the set is a singleton. □

As can probably be seen from the above example, a compressed relation should require less storage than explicitly listing the valid tuples. In the optimal case, the storage requirements for a relation in the Boolean domain, where all combinations are valid, is reduced from $\mathcal{O}(2^n)$ to $\mathcal{O}(n)$. It is worth pointing out, that it is not always possible to reduce the storage requirements of a relation, as the following example shows.

Example 2.5. Consider a relation with the following valid tuples:

a	b	c
1	0	0
0	1	0
0	0	1

It is not possible to further compress this relation (at least not when the Cartesian arguments are restricted to a single variable). □

Møller developed several heuristics for compressing a relation. The *compress* algorithm is the other algorithm covered in this report. It is covered in more detail in Chapter 3.

Chapter 3

Compressing relations

3.1 The minimal relation form

As have been discussed in Section 2.5.2 on page 13, several relation forms exists. We first need to define when two relation forms are equivalent.

Definition 3.1 (Equivalent relation forms)**.** Two relation forms are said to be *equivalent* if the underlying sets are equal.

We also need to establish a measure on the size of a relation form. We ignore any overhead needed to administrate the data structure, and therefore arrive at the following definition, which is valid for both uncompressed and compressed relation forms.

Definition 3.2. The *size* of a relation form is the number of scalar values contained in the relation.

Using this definition on the constraint relation c_2 from Example 2.4 on page 13, we see that the uncompressed form has size 15 while the compressed form has size 8. Note that the compressed form is not canonical, i.e. the same relation can be compressed in different ways. If we return to relation c_2 again, it is easy to see that

b	c	d
0	0	$\{0,1\}$
$\{0,1\}$	1	1
1	0	1

is an equivalent compressed form of size 11.

Ranking different compressed forms leads to a (not necessarily unique) optimal form, which is defined as follows:

Definition 3.3. A relation is said to be in *minimal relation form* if no equivalent relation form exists which has a smaller size.

3.2 Compression strategies

It is possible to compress a relation such that the Cartesian arguments span more than one variable. The relation shown in Example 2.5 on page 14 has size 9 and cannot be further compressed using Cartesian arguments spanning only one variable. By using arguments that span two variables (in this example, a and b), the following form (with size 8) can be obtained:

(a, b)	c
$\{(1, 0), (0, 1)\}$	0
$(0, 0)$	1

In general, the Cartesian arguments can be selected from subsets of the relation scope. We formalise this selection as follows:

Definition 3.4. A *compression strategy* for a relation with scope S is a (possibly empty) set of pairwise disjoint subsets of S. Each subset defines the scope of a Cartesian argument. An *optimal compression strategy* is a compression strategy, which yields a minimal relation form.

Using this definition, the compression strategy used above would be $\{\langle a, b\rangle, \langle c\rangle\}$ and the strategy used in Example 2.5 on page 14 is $\{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$. Note that the subsets need not cover S, i.e. variables, which are not member of the compression strategy will not have their values compressed. However, since we are interested in decreasing the size of a relation and compression never increases the size, we only consider compression strategies which actually cover S.

Proposition 3.1. The number of compression strategies that cover a n-ary relation with scope S is B_n, where B_n is defined by the following recurrence relation:

$$B_{n+1} = \sum_{i=0}^{n} \binom{n}{i} B_i \tag{3.1}$$

Proof. The number of compression strategies that cover S is equivalent to the number of ways the set S with n elements can be partitioned into nonempty subsets. This number is called a *Bell Number*, denoted B_n, and can be generated by the recurrence (3.1). □

3.3 A compression heuristic

The result in Proposition 3.1 clearly makes an exhaustive search for an optimal compression strategy intractable for all but the smallest scopes. When using the information in a relation (i.e., when joining two relations), we

usually need the values columnwise. Therefore we only consider the compression strategy containing singletons spanning the scope, i.e., the Cartesian arguments span a single variable.

Møller [17] introduced a heuristic to compress relations, in which the relation is compressed one column at a time. It works in two phases:

1. In the first phase, the relation is analysed to determine the order in which the columns are to be compressed.

   From the original relation, each variable is removed in turn, and duplicate tuples are eliminated from the resulting relation. The number of unique tuples in the new relation is the size of the *complement* associated with the variable removed. The columns are compressed in order of non-decreasing complement sizes.

2. In the second phase, the relation is compressed on each column according to the selected column order.

The heuristic as described above is called complement controlled compression. Møller also describes another way to determine the order in which columns are to be compressed, called domain controlled compression, where the order of compression is determined by the domain size of the variables. Columns are compressed in order of non-decreasing domain sizes.

Example 3.1. We now illustrate the heuristic by compressing the relation c_2 from Example 2.4 on page 13 using complement controlled compression. First, we determine the size of each variable's complement:

b	c	d
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{removeCol(b)}$

c	d
0	0
0	1
1	1
0	1
1	1

$\xrightarrow{unique}$

c	d
0	0
0	1
1	1

$\xrightarrow{count}$ 3

b	c	d
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{removeCol(c)}$

b	d
0	0
0	1
0	1
1	1
1	1

$\xrightarrow{unique}$

b	d
0	0
0	1
1	1

$\xrightarrow{count}$ 3

b	c	d
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{removeCol(d)}$

b	c
0	0
0	0
0	1
1	0
1	1

$\xrightarrow{unique}$

b	c
0	0
0	1
1	0
1	1

$\xrightarrow{count}$ 4

From this it can be seen that (b, c, d) and (c, b, d) are valid column orders. If we choose compression order (b, c, d), the result is as follows.

b	c	d
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{compress(b)}$

b	c	d
0	0	0
$\{0,1\}$	0	1
$\{0,1\}$	1	1

$\xrightarrow{compress(c)}$

b	c	d
0	0	0
$\{0,1\}$	$\{0,1\}$	1

$\xrightarrow{compress(d)}$

b	c	d
0	0	0
$\{0,1\}$	$\{0,1\}$	1

□

Møller described the compression heuristic in terms of high-level array operations using APL notation. In Algorithm 1 on the facing page we present ORDER-COLUMNS-COMPLEMENT, an algorithm that orders the columns of the input relation R using the complement controlled heuristic, i.e., phase 1 described above. For each column in the input relation, ORDER-COLUMNS-COMPLEMENT builds a data structure T which holds the unique tuples of the current column's complement. An array, $complements$, holds the size of each column's complement and before returning, the columns of the input relation are ordered in non-decreasing complement sizes.

In Algorithm 2 on the next page we present COMPRESS, an algorithm which compresses the input relation R one column at a time, i.e., phase 2 described above. It assumes the input relation R has a field, $columns$, which is an array holding the columns. Each column is an array of sets (each set being a singleton when the relation is uncompressed). The columns are compressed in the order in which they are stored in the array $columns$.

For each column, COMPRESS constructs the data structure T, which holds an entry for each unique tuple in the complement. Each entry has two fields: $rowno$ holds the row number for the first occurrence of the unique tuple. The field $values$ contains the values from the current column, which occurs in the rows where the complement tuples are equal. When T has been constructed, a new relation is created with the current column replaced by the sets stored in $values$. The remaining columns are created from the existing columns, by picking out rows with row numbers stored in $rowno$.

Complexity

While ORDER-COLUMNS-COMPLEMENT and COMPRESS will work if the relation is already compressed, we will for the purpose of analysis, assume the input relation is uncompressed. Let c denote the number of variables

Algorithm 1 Order columns in relation R using the complement controlled heuristic

ORDER-COLUMNS-COMPLEMENT(R)

```
1: for columnno ← 1 to R.numColumns do
2:    INITIALISE(T)
3:    for rowno ← 1 to R.numRows do
4:       {Construct key from all columns except columnno}
5:       key ← CONSTRUCT-KEY(R, columnno, rowno)
6:       INSERT-OR-REPLACE(T, key)
7:    complements[columnno] ← T.size
8: {Sort R.columns according to complement sizes, stored in
   complements, in non-decreasing order}
```

Algorithm 2 Compress the relation R. Columns are compressed in the order stored in $R.columns$

COMPRESS(R)

```
1: for columnno ← 1 to R.numColumns do
2:    {Compress on column specified by columnno}
3:    INITIALISE(T)
4:    for rowno ← 1 to R.numRows do
5:       {Construct key from all columns except columnno. LOOKUP-OR-
         INSERT return a reference to an entry which equals key.}
6:       key ← CONSTRUCT-KEY(R, columnno, rowno)
7:       (inserted, entry) ← LOOKUP-OR-INSERT(T, key)
8:       if inserted then
9:          entry.rowno ← rowno
10:         entry.values ← R.columns[columnno][rowno]
11:      else
12:         entry.values ← APPEND(entry.values, R.columns[columnno][rowno])
13:   {Construct the new columns}
14:   for i ← 1 to R.numColumns do
15:      rowno ← 0
16:      for all entry in T do
17:         rowno ← rowno + 1
18:         if i = columnno then
19:            {Create new compressed column from values stored in T}
20:            {Remove duplicate values from entry.values}
21:            newcolumn[rowno] ← entry.values
22:         else
23:            {Copy values from existing columns}
24:            newcolumn[rowno] ← R.columns[i][entry.rowno]
25:      columns[i] ← newcolumn
26:   R.numRows ← rowno
```

and n the number of rows in the relation. The size of the input relation is then cn. We assume $n > c$ whenever needed.

The complexity of ORDER-COLUMNS-COMPLEMENT and COMPRESS is dependent on the data structure T used to implement the functions LOOKUP-OR-INSERT and INSERT-OR-REPLACE. If we use a balanced search tree, both functions can be implemented in time $\mathcal{O}(c \log_2 n)$ worst case since comparing keys takes $\mathcal{O}(c)$ time and there are n nodes in the tree if the relation cannot be compressed.

In ORDER-COLUMNS-COMPLEMENT we basically need to count the number of unique tuples in each variable's complement, i.e. remove duplicate tuples from the complement and count the remaining tuples. This gives a worst case running time of $\mathcal{O}(c^2 n \log_2 n)$ if INSERT-OR-REPLACE is implemented using a balanced search tree. By using a hash table, we can reduce this to $\mathcal{O}(c^2 n)$ expected. The worst-case running time is still $\mathcal{O}(c^2 n \log_2 n)$ if we implement each entry in the hash table as a binary tree.

If we use a balanced search tree, the running time of COMPRESS is: Lines $3 - 12$ take $\mathcal{O}(cn \log_2 n)$. At first, the running time of lines $14 - 25$ seem to depend on the size of T: If $T.size = n$ no compression has taken place and the removal of duplicates in line 19 takes $\mathcal{O}(1)$. If $T.size = 1$, all values are compressed into a single row and $entry.values$ is a set with n elements, so line 19 takes $\mathcal{O}(n)$ (using e.g. radix-sort to sort the integers before removing duplicates). But in general we always have $n = \sum_{entry \in T} |entry.values|$, so lines $13 - 24$ have running time $\mathcal{O}(cn)$ yielding a total running time for COMPRESS of $\mathcal{O}(c^2 n \log_2 n)$.

If instead we implement LOOKUP-OR-INSERT using a hash table, line $3 - 12$ take $\mathcal{O}(cn)$ expected time yielding a total expected running time $\mathcal{O}(c^2 n)$. The worst-case running time is still $\mathcal{O}(c^2 n \log_2 n)$ if we implement each entry in the hash table as a binary tree.

3.3.1 Strongly universal hashing

When hashing is used, we need to calculate the hash value for each column's complement. This takes $\mathcal{O}(cn)$ time. When we insert a key in the hash table and there is already a key with the same hash value, we need to compare the two tuples to determine if they are in fact equal. This comparison takes time $\mathcal{O}(c)$. Tuples can get the same hash value in two situations:

1. The tuples are different. The hash value is the same because the hash function, unless it is *perfect*, is not collision free. The hash function chosen should make the probability of collisions small.

2. The tuples are equal and therefore they get the same hash value. We cannot avoid this situation.

If we let n_i denote the number of tuples in column i's complement, which have the same hash value, the total time to compute the size of the complement for column i is $\mathcal{O}(cn + cn_i)$. Since n_i is proportional to n, this give us the running time of $\mathcal{O}(cn)$ expected, found in the previous section.

We note that ORDER-COLUMNS-COMPLEMENT merely performs a heuristic ordering of the columns, based on the sizes of the column's complement. If we use an approximation of the complement sizes, the size of the resulting compressed relation may not increase much. We can approximate the complement sizes by using the following method: We assume that tuples that get the same hash values are equal. Using this method, we save the comparison of tuples when they get the same hash value. We can thus reduce the running time of ORDER-COLUMNS-COMPLEMENT to $\mathcal{O}(cn)$ worst case by employing *strongly universal hashing*, defined by Carter and Wegman [24]. A class $\mathcal{H}$ of hash functions from $U = \{0, \ldots, m\}$ to $D = \{0, \ldots, n\}$ is said to be *strongly universal*$_2$, if each hash function $h \in \mathcal{H}$ maps keys pairwise independently, i.e. $\forall x \neq y \in U, \alpha, \beta \in D$: $\Pr_{h \in H}(h(x) = \alpha \wedge h(y) = \beta) = O(1/n^2)$. Strongly universal hash functions support the following type of vector hashing.

Proposition 3.2 ([22])**.** Suppose $\mathcal{H}$ is strongly universal$_2$. Let $\mathcal{H}^q$ define the class of functions from U^q to D such that $(h_1, \ldots, h_q)(x_1, \ldots, x_q) = h_1(x_1) \oplus \cdots \oplus h_q(x_q)$ where $\oplus$ is the binary XOR operation. Then $\mathcal{H}^q$ is strongly universal$_2$.

We can utilise vector hashing as follows. Let $h_v(i)$ denote the vector hash value for the tuple in the ith row, $h(r_{ij})$ the hash value for the ith row and the jth variable using a strongly universal hash function h. Then we have

$$h_v(i) = h(r_{i1}) \oplus \cdots \oplus h(r_{ic}).$$

We can compute the vector hash value h_v for each tuple in the relation in time $\mathcal{O}(cn)$ assuming that $h(r_{ij})$ takes $\mathcal{O}(1)$ time.

When we need the hash value $h_t(i, k)$ for the complement of variable k at row i, we can calculate this as

$$h_t(i,k) = h(r_{i1}) \oplus \cdots \oplus h(r_{ik-1}) \oplus h(r_{ik+1}) \oplus \cdots \oplus h(r_{ic}) = h_v(i) \oplus h(r_{ik})$$

This means we can calculate the hash values for each complement in time $\mathcal{O}(n)$ if we use $\mathcal{O}(cn)$ time to precompute the vector hash values. We thus arrive at a worst case running time for ORDER-COLUMNS-COMPLEMENT of $\mathcal{O}(cn)$. The worst case running time is thus linear in the size of the input.

Unfortunately, we cannot use the same trick in COMPRESS. While we can reduce the running time of lines $3 - 12$ from $\mathcal{O}(cn)$ to $\mathcal{O}(n)$ by using a strongly universal hash function, it does not seem possible to reduce the running time of lines $14 - 25$.

Open problems

- What is the complexity for finding the minimal relation form of a relation?
- Let S_{mrf} denote the size of the minimal relation form and S_{min} the size of the smallest relation form obtainable with a compression strategy consisting of singletons. If we can find a lower bound on the ratio S_{mrf}/S_{min}, we have an opportunity to judge whether it is worth investigating algorithms for finding S_{mrf}.
- Let S_{cc} denote the size of a relation compressed using the complement controlled heuristic and S_{max} the size of the largest relation form obtainable with a compression strategy consisting of singletons. If we can find a lower bound on the ratio S_{min}/S_{cc}, we have an opportunity to judge the quality of the complement controlled compression heuristic.

  If we can find an upper bound on the ratio S_{cc}/S_{max} and the bound is sufficiently close to 1, we may compress the relation using a random column ordering and obtain a relation form with size close to the size of the form found using the complement controlled heuristic without having to use ORDER-COLUMNS-COMPLEMENT.

Chapter 4

Joining compressed relations

4.1 Join techniques for relational database systems

The operation for joining two relations is one of the most frequently used and expensive query processing operation in relational database systems. Much research has therefore been focused on algorithms which can do this effectively. A survey of join techniques can be found in [16].

The most common join techniques can be grouped into three classes:

Nested-loops join: This is the simplest join algorithm and, as the name suggests, works by comparing each tuple of one relation (called the *outer relation*) against all tuples in the other relation (called the *inner relation*).

Sort-merge join: This is a two-phase algorithm. In the first sorting phase, each relation is sorted on the join attributes. In the second merging phase, a tuple from one relation is compared to a tuple from the other relation and joined if possible. Otherwise, the tuple with the smallest sort-order is discarded and the next tuple from this relation is used. In this way each tuple is only used once when the relations have been sorted.

Hash join: This is also a two-phase algorithm. In the first partitioning phase, each tuple is hashed on the join attributes and placed in a partition based on this hash value. In the second merging phase, one partition of the outer relation is joined with the corresponding partition of the inner relation.

Many variations on the above algorithms exists, mainly differing in how to minimise the number of disk accesses required.

Recently there has been some focus on how to optimise main-memory join algorithms to perform better on modern hardware [4, 21]. In [4] a par-

titioned hash-join algorithm is presented which is reported to be almost a factor of 7 faster than the simple hash-join for large relations.

4.2 Join algorithm for array-based logic

While there are many similarities between the join techniques used in relational database management systems and the join operation used in ABL, there are also some important differences:

- While not inherent to ABL, we restrict ourself to relations which fit in memory, i.e., we do not focus on disk I/O.
- The data structures used for storing relations need only be optimised to execute a join. In a relational database system, many other operations must also be implemented effectively (e.g., insert/modify/delete tuples, select subsets of tuples etc.).
- When two relations are joined, only the result is important. The two joined relations can be discarded. Consequently, a relation is only joined with another relation once. We cannot therefore afford to maintain expensive data structures for a relation, based on the assumption that the cost of this maintenance will be amortised when performing many join operations.
- When joining two relations, we not only need to compare values for equality but also, when joining compressed relations, determine the intersection between two sets in order to determine if the two tuples should be part of the result.
- We know the domain size of all the attributes in advance, i.e., by encoding all the domains D_x as integers in the range $\{0, 1, \cdots, |D_x| - 1\}$ we can, without loss of generality, restrict our attributes to be integers.

Ordering compressed relations

If a sort-merge join is to be used, we need a way to order the tuples based on the join attributes. This is not a problem when the attribute values are scalar and we only compare attributes for equality. When using compressed relations however, the attribute values are, in general, sets of integers and we need to calculate the intersection between these sets. While it is possible to order the tuples (i.e., using lexicographic ordering), we cannot merge the tuples based on this ordering since we need to perform an intersection between two tuples in order to determine whether the tuples should be part of the output. We conclude, that we cannot use sort-merge join to join compressed relations.

Hashing compressed relations

If a hash join is to be used, we need to define a hash function h which computes an integer value for each tuple, based on the values of the join attributes. For uncompressed relations, two tuples should be part of the result if the join attributes are equal, i.e., when two tuples are hashed, $h(x) \neq h(y)$ implies that the join attributes of the two tuples cannot be equal. But when a relation is compressed, the two tuples may still need to be part of the result, even if they are not equal, e.g. if $x = \{0, 1, 2\}$ and $y = \{0, 2, 4, 6\}$, the set $x \cap y = \{0, 2\}$ needs to be part of the result. So in order to effectively use hashing-like techniques we essentially need a signature function with the following property:

$$x \cap y = \emptyset \Rightarrow sig(x) \neq sig(y)$$

In [11] a signature-hash join algorithm is presented, which can be used to evaluate subset join predicates. A signature function with the following property is constructed:

$$x \subseteq y \Rightarrow sig(x) \subseteq sig(y)$$

Unfortunately, as noted in the paper, algorithms whose join predicate is based on non-empty intersection have yet to be developed. This makes it difficult to use any of the known hash-based join techniques for compressed relations.

4.2.1 Nested-loop join

The above observations leave us with nested-loop join as the only useful method to join two compressed relations. To maximise locality of reference, we choose to store relations columnwise (see Section 5.2 for more detail). Therefore, we should preferably compare the tuples columnwise, which means we cannot construct the output relation until we have joined all common attributes.

When executing the join $R_1 \bowtie R_2$, we must keep track of which tuples are needed in the output relation. For this we use a data structure called a *join index* [23]. A join index is a set of pairs (i, j) of row numbers such that row i from input relation R_1, denoted $R_1[i]$, has a nonempty intersection (on the common attributes) with row j from input relation R_2, denoted $R_2[j]$. More formally.

$$J = \{(i, j) \mid R_1[i]_{|S_1 \cap S_2} \cap R_2[j]_{|S_1 \cap S_2} \neq \emptyset\}$$

Here, S_i denotes the scope of ith relation.

When joining on the first common attribute, we create the initial join index by intersecting all rows in relation R_1 with all rows in relation R_2.

For the subsequent common attributes (if any) we only check the intersection between the row number combinations which already are in the join index. All other row number combinations have already been discarded. If an intersection is empty, the corresponding row number combination is removed from the join index. We thus incrementally prune the join index, ending up with a list of row number combinations which should be combined to form the output relation.

Example 4.1. We now illustrate the join algorithm with a small example. The input relations R_1 and R_2 are shown below.

$$R_1 = \begin{array}{ccc} a & b & c \\ \hline 0 & \{0,1\} & 0 \\ \{1,2\} & 0 & 1 \\ 3 & \{2,3\} & 0 \\ 4 & 4 & 1 \end{array} \qquad R_2 = \begin{array}{ccc} b & c & d \\ \hline \{0,1\} & 1 & 1 \\ 2 & 1 & 3 \\ \{1,3\} & 0 & 2 \end{array}$$

The common attributes are b and c. The initial join index J_1 is created by joining on column b. We calculate the intersection between all sets from column b in R_1 with all sets in column b in R_2 and store the row number combinations that have a nonempty intersection. We then get:

$$J_1 = \begin{array}{cc} i & j \\ \hline 1 & 1 \\ 1 & 3 \\ 2 & 1 \\ 3 & 2 \end{array}$$

The size of J_1 is 4. When joining on column c, we thus only need to look at the 4 row number combinations present in the join index, not all 12 combinations. We then get the final join index J_2.

$$J_1 = \begin{array}{cc} i & j \\ \hline 1 & 3 \\ 2 & 1 \end{array}$$

From this we see that the output relation R has two rows, and it is straightforward to create it using the information in the join index.

$$R = \begin{array}{cccc} a & b & c & d \\ \hline 0 & 1 & 0 & 2 \\ \{1,2\} & 0 & 1 & 1 \end{array}$$

□

The CREATE-JOIN-INDEX algorithm, which creates a join index given two relations, is shown in Algorithm 3. The CONSTRUCT-JOIN-RESULT algorithm, which creates the output relation for a join result specified in a join index, is shown in Algorithm 4.

An improvement is possible in the following special case. If there exists pairs of common attributes that are not compressed, it is possible to utilise the well-known join algorithms (such as sort-merge or hash) to join on these non-compressed attributes before joining on the compressed attributes. This makes it possible to create the initial join index without looping through all rows in the first common column. We will not pursue this special case any further.

Algorithm 3 Create join index for relations R_1 and R_2. The join index is returned.

CREATE-JOIN-INDEX(R_1, R_2)

```
 1: J.size ← 0
 2: first ← true
 3: for all column in R1.columns ∩ R2.columns do
 4:   if first then
 5:     {Create join index}
 6:     first ← false
 7:     for i ← 1 to R1.size do
 8:       for j ← 1 to R2.size do
 9:         if R1.columns[column][i] ∩ R2.columns[column][j] ≠ ∅ then
10:           J.size ← J.size + 1
11:           J[J.size] ← (i, j)
12:   else
13:     {Prune join index}
14:     lastrow ← 1
15:     for x ← 1 to J.size do
16:       (i, j) ← J[x]
17:       if R1.columns[column][i] ∩ R2.columns[column][j] ≠ ∅ then
18:         J[lastrow] ← (i, j)
19:         lastrow ← lastrow + 1
20:     J.size ← lastrow − 1
21:
22: return J
```

Complexity

For ease of analysis, we assume that the scope of the two relations is the sane and that the size of R_1 is larger than the size of R_2. The size of the join index depends on the *join selectivity* factor, denoted JS, which is defined as follows.

$$JS = \frac{\|R_1 \bowtie R_2\|}{\|R_1\|\|R_2\|}$$

Algorithm 4 Construct output relation for join between relations R_1 and R_2 given join index J. The output relation is returned.

CONSTRUCT-JOIN-RESULT(R_1, R_2, J)

1: **for all** col **in** $R_1.columns \cup R_2.columns$ **do**
2: **if** $col \in R_1.columns \cap R_2.columns$ **then**
3: {Column is common, create new from intersection between values}
4: **for** $x \leftarrow 1$ **to** $J.size$ **do**
5: $(i, j) \leftarrow J[x]$
6: $new[x] \leftarrow R_1.columns[col][i] \cap R_2.columns[col][j]$
7: **else**
8: {Column is not common, create new by picking values from col in either R_1 or R_2}
9: **for** $x \leftarrow 1$ **to** $J.size$ **do**
10: $(i, j) \leftarrow J[x]$
11: **if** $col \in R_1.columns$ **then**
12: $new[x] \leftarrow R_1.columns[col][i]$
13: **else**
14: $new[x] \leftarrow R_2.columns[col][j]$
15: $result.columns[col] \leftarrow new$
16:
17: **return** $result$

where $\|R_i\|$ denotes the number of tuples in the ith relation. If $JS = 1$, the size of the join index equals the size of the Cartesian product of the two input relations.

If we let S_i denote the size of the ith relation, the average number of elements in the set of values for each tuple's attribute is $\frac{S_i}{c\|R_i\|}$. In general, the sets are represented as a sorted sequence of integers, which means that we, in the worst case, need to make $\frac{S_2}{c\|R_2\|}$ comparisons in order to determine whether the intersection is empty. We ignore the special case where the sets can be represented as bit vectors, which allows for intersection in constant time (see Section 5.2 on page 31 for more details on the data structures used to represent the columns).

It should be clear that the running time of CREATE-JOIN-INDEX depends on the join selectivity factor, the worst case being $JS = 1$. In this situation we make $c\|R_1\|\|R_2\|$ set intersections, yielding a running time of $\mathcal{O}(S_2\|R_1\|)$.

The running time of CONSTRUCT-JOIN-RESULT is $\mathcal{O}(c\|R_1 \bowtie R_2\|)$, i.e., proportional to the size of the join index.

Column order dependency

A weak point of CREATE-JOIN-INDEX is that whenever $JS < 1$, the running time depends on the order in which the common attributes are joined, as the following examples shows.

Example 4.2. Assume the input relations R_1 and R_2 have common attributes a and b. Assume further that the join selectivity for the join of column a is 1 and the join selectivity for the join of column b is ϵ.

If we join the columns in order a, b, the initial join index has $\|R_1\|\|R_2\|$ rows while the final join index has size $\epsilon\|R_1\|\|R_2\|$. If instead we join the columns in reverse order, the initial join index has $\epsilon\|R_1\|\|R_2\|$ rows, same as the final join index. □

The order in which columns are joined is thus important. In the following, we sketch how we can make the space complexity linear in the size of the output.

The idea is to always limit the number of non-finished row number pairs in the join index by a constant β. A non-finished row number pair is a pair of row numbers where we do not know yet, whether the rows should be part of the output or not. When we join a column and we reach β non-finished pairs, we need to skip the current column and start joining the next column, thus pruning the join index. We keep skipping columns, until we reach, say $\beta/2$ unfinished entries. Note, that it is always possible to reach 0 non-finished entries by joining on all common attributes. With this method, the join index will at any time have at most β entries more than the final join index.

Chapter 5

Implementation

In this chapter an implementation of the algorithms described in Chapters 3 and 4 is presented. The actual source code for the implementation can be found in Appendix A.

5.1 Objective

The objective was to implement the algorithms described, to facilitate performance experiments. In order to compare performance between this implementation and existing implementations of the algorithms, functionality which can read and write Array Database™ (ADB) files is implemented, too. An ADB file is a binary file which contains information about the variables, domains and relations of a CSP instance. This information is specified in a text file using symbolic logic and translated into the binary ADB file by a compiler. More information on the syntax of this text file can be found in [2]. Since these I/O routines are not interesting for this report, we will not discuss them further.

5.2 Data structures

There are basically two ways to layout a relation in memory:

Horizontally decomposed: The values of each tuple form a consecutive byte sequence.

Vertically decomposed: The values of each column form a consecutive byte sequence.

When joining two relations, we only need values for the columns common to both relations. This is usually not all the values in a tuple, so the *stride*, i.e., the offset between two subsequently accessed memory addresses, will

be larger than one. In [4], experiments show that the stride size is an important parameter for optimising memory access patterns. On a simple in-memory scan of a single byte from 200000 tuples, an increase of a factor of 8 was measured in execution times when the stride was increased from 1 to 150 bytes.

We therefore choose the vertical decomposition since values of a single column can then be accessed sequentially. This choice has another benefit: the tuple width need not be uniform. This means that values in a compressed column can vary in size.

A consequence of this choice is that all algorithms should preferably work columnwise instead of tuplewise in order to maximise the locality of memory access.

5.2.1 Representing columns

As noted earlier, each column in a relation contains values from a finite domain D_i. Without loss of generality, we therefore encode the domain values as integers $\{0, 1, \ldots, |D_i| - 1\}$.

When joining two relations, we compare (or in the general case, determine the intersection between) tuples of columns containing values from the same domain D_i. None, one or both of the columns can be compressed.

Uncompressed columns

Uncompressed columns are stored as a sequence of integers. The number of bits needed for each integer is $\lceil \log_2 |D_i| \rceil$. We represent an integer as the smallest data type that can hold $\lceil \log_2 |D_i| \rceil$ bits.

Compressed columns

There are several ways to represent the sets in a compressed column. We need to be able to determine efficiently

- whether the intersection between two sets is empty,
- the intersection between two sets, and
- whether a given element exists in a set.

For small domains a bit vector seems like a natural choice: the intersection operation can utilise the inherent word-parallelism of the bitwise AND operator and the single element test is simply a bit test. But for large domains, the memory usage of the bit vector becomes prohibitive. For variables with large domains, we therefore chose to represent the set as a vector of sorted integers.

A compressed column is thus stored as a sequence of either fixed-sized bit vectors or (pointers to) sorted integer vectors. In the current implementation, bit vectors are used when the domain has less than 33 values. That enables encoding the set in a single 32-bit word. It is likely that it is beneficial to encode even larger domains as bit vectors, since we avoid memory allocation for set elements and we exhibit better cache behaviour. The exact size where the encoding should switch to using a sorted vector, should be determined experimentally.

5.3 Important classes

The most important classes are as follows:

CSP: Contains the input of a given CSP instance.

Variable: Describes a variable. A variable has an id, a name, and a number of valid domain values.

Relation: Contains the valid tuples of a relation. The valid tuples are represented by a number of Columns, one for each variable in the relation.

Column: Contains the valid values for a single variable. Column is an abstract baseclass for all Columns (compressed or uncompressed).

UncompressedColumn: Baseclass for the uncompressed columns.

ConcreteUncompressedColumn¡T¿: Uncompressed column where the scalar values are of type T.

CompressedColumn: Baseclass for the compressed columns.

BitmapCompressedColumn¡BITS¿: Compressed column represented as a bit vector.

ConcreteCompressedColumn¡T¿: Compressed column represented as a sorted vector of type T.

The static structure of these classes are shown on Figure 5.1 on the next page.

5.4 Details

Strongly universal hashing

In order to compare this implementation with existing implementations, we have not implemented ORDER-COLUMNS-COMPLEMENT using the heuristic based on a strongly universal hash function. However, we do use a

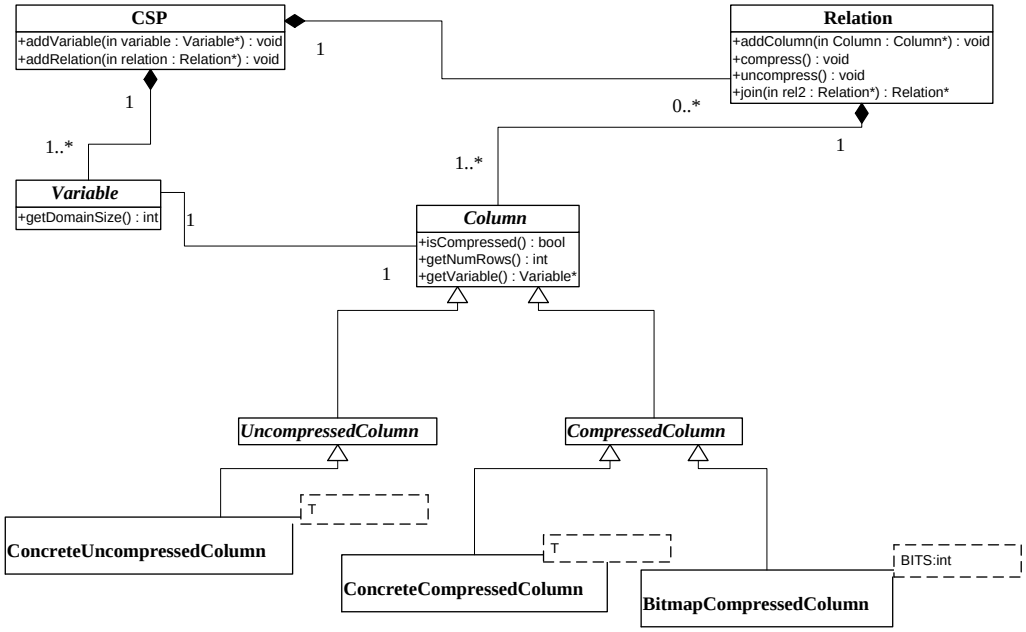


Figure 5.1: UML static structure diagram for the most important classes. Details are omitted for brevity.

strongly universal hash function in order to calculate the hash values faster using the vector hashing method described in Section 3.3.1 on page 20.

We need a strongly universal hash function that calculates a hash value for arbitrary n-bit tuples. A standard trick is to tabulate hash functions for 8-bit characters. A b bit key is then viewed as $\lceil b/8 \rceil$ characters, and the hash value for the ith character is found by a table lookup in the ith table using the 8-bit character as index. The hash value for the entire vector can then be calculated as described in Proposition 3.2 on page 21. This hash function is very fast on the platform considered in this report, the Intel Pentium (see [22] for a performance study of strongly universal hash functions), since only table lookups and XOR is involved. The drawback is, that we need independent tables for each coordinate in the vector we are trying to hash, making it unsuitable for long vectors.

We have chosen a pragmatic solution where we use only a fixed number of tables (in our case 8) which are then used cyclically [12].

Using optimal representations

The columns are represented using different classes, depending on whether the column is compressed and the domain size of the corresponding vari-

able. The creation of the optimal class for any given variable is implemented in a central factory function, which creates new columns. For compressed columns, it looks as follows.

```
CompressedColumn *CompressedColumn::createNew(const Variable *var,
    const urvector_t &ur)
  {
  size_t max_dom_index = var->getDomainSize()-1;
  if (max_dom_index < 8)
    return new BitmapCompressedColumn<8>(var, ur);
  if (max_dom_index < 16)
    return new BitmapCompressedColumn<16>(var, ur);
  if (max_dom_index < 32)
    return new BitmapCompressedColumn<32>(var, ur);
  if (max_dom_index ≤ std::numeric_limits<unsigned char>::max())
    return new ConcreteCompressedColumn<unsigned char>(var, ur);
  if (max_dom_index ≤ std::numeric_limits<unsigned short>::max())
    return new ConcreteCompressedColumn<unsigned short>(var, ur);
  return new ConcreteCompressedColumn<unsigned long>(var, ur);
  }
```

First we check if we can use a bit vector to represent the column cells. If this is not possible, we select a sequence of sorted integers using the smallest possible data type to represent the integers.

Using traits classes to simplify implementation

The performance critical part of the code is the functions that implement the compress and join functionality. Since different C++ types are used to represent the column data (3 types for the uncompressed columns and 6 for the compressed columns), the compress and join functions should potentially be implemented in several different versions. Compress needs an implementation for each type of column (9) and join needs an implementation for many combination of column types.[1] This gives maximum performance, since the type of the columns is known to the compiler, which makes full optimisation possible in the inner loops of the algorithms. An alternative solution is to create a single version of each algorithm and reference the column data using virtual functions in the column classes. This has the drawback that many virtual function calls are performed in the inner loops, and, more importantly, it is not possible for the compiler to inline the virtual methods.

By using the concept of *traits* classes, originally introduced by Meyers [15], it is possible to combine the two methods. Here we use the term traits

[1] Not all combinations need to be implemented. Since the relations are joined column-wise on common variables, each pair of columns joined represent the same variable, i.e. they have the same domain. A `ConcreteUncompressedColumn<short>` is thus never joined with a `ConcreteCompressedColumn<char>`

class to refer to a class that aggregates the basic operations (copy, intersection, hash value etc.) on a single cell value in a column. The traits class for a cell containing scalar values has the following signature:

```
template <typename T>struct ScalarCellTraits
  {
  typedef T value_type;
  static std::string toString(T cellValue)
  static T cloneCell(T cellValue)
  static bool intersectionEmpty(T v1, T v2)
  static void expandCell(T cellValue, idvector_t &values)
  static bool equal(T cellValue1, T cellValue2)
  static void hashCell(size_t *hashValue, size_t index, T cellValue,
      unsigned char domainBits)
  };
```

By supplying different traits classes to the same algorithm (e.g., join), this algorithm can be applied to different column types. The compiler will generate an appropriate implementation of the algorithm (using templates), specialised for the types passed. Since the compiler knows the types at the time of compilation, full optimisation and inlining can be applied in the critical inner loops. In this case, virtual functions are only used to call the correct (compiler generated) function.

Further improvements

- When all columns of a relation are of fixed width, i.e. represented by a `UncompressedColumn` or `BitmapCompressedColumn`, it should be possible to use sorting in order to remove duplicates (in ORDER-COLUMNS-COMPLEMENT). By using, e.g., merge-sort, it is expected that the cache behavior is improved over the current hashing method.
- The class `ConcreteCompressedColumn<T>` is implemented in terms of a `std::vector<std::vector<T *> >`. Since each element of the column is dynamically allocated, the locality of reference is probably not very good. Elements should be allocated from larger chunks to ensure better cache behaviour.
- The code has not been profiled, so it is probably possible to finetune it further.

Chapter 6

Performance results

In this chapter we present the results of a performance study in which we compared the performance of the implementation described in Chapter 5 to existing implementations in C++ and APL.

6.1 Test environment

Three different implementations were compared:

APL The original APL implementation described in [17].

Old The C++ implementation used in the commercial software product Array Database™ version 5.5. This implementation is mostly based on the APL implementation, using a general *Array* data structure to hold all data. Several optimisations, such as hashing and reference data types, not available in APL are used to speed up the algorithms.

New A C++ implementation based on the algorithms described in Chapters 3 and 4. The source code can be found in Appendix A.

The relations used in the experiments were collected from real-life CSP instances which have been used in various projects. To get the large relations, the compiler that generates the binary ADB file were modified to extract the relations during the compile process.

All the experiments were performed on a Dell Inspiron 5000e with a 750 MHz Pentium III processor and 384 MByte memory. The operating system used was Windows 2000, Service Pack 2. For the C++ tests, the compiler used was Microsoft Visual C++ Version 7 Beta 2. For the APL tests, we used Dyadic APL for Windows version 8.2 release 4. Each experiment was run multiple times, with no significant variance observed among the results.

6.2 Comparing compression implementations

The characteristics of the input data used for comparing the compression algorithms are shown in Table 6.1, where S denotes the scope of the relation, D_i the domain for variable x, n the number of tuples, s the size of the relation as defined in Definition 3.2 on page 15, n_c the number of tuples in the compressed relation, and s_c the size of the compressed relation. The two last columns show n_c and s_c as percentages of n and s, respectively. The values for the compressed relations are the same for all methods, since the same heuristic is used to compress the relations.

Name	$\|S\|$	$\sum_{x \in S} \|D_x\|$	n	s	n_c	s_c	n_c %	s_c %
heq	10	1643	151374	1513740	5020	158194	3.3%	10.5%
plan31	14	28	8192	114688	14	274	0.2%	0.2%
q10a	8	80	149552	1196416	13144	165486	8.8%	13.8%
q10b	8	80	55658	445264	6632	79668	11.9%	17.9%
ns11	11	65	333322	3666542	102	2654	0.03%	0.07%

Table 6.1: Characteristics of the input data used for performance comparison of compression.

The performance results are presented in Table 6.2. The speedup factor is calculated with the existing C++ implementation used as a base. As can

	APL		Old		New	
Name	Time	Speedup	Time	Speedup	Time	Speedup
heq	36.5	0.33	12.01	1	2.63	4.6
plan31	9.10	0.11	1.031	1	0.11	9.4
q10a	233	0.32	74.02	1	1.74	42.5
q10b	47.8	0.21	9.884	1	0.63	15.7
ns11	660	0.57	377.5	1	5.80	65.1

Table 6.2: Execution time of compression algorithms. All times are measured in CPU-seconds.

be seen, the new implementation is significantly faster than both existing implementations. The difference seems to increase as the size of the relation increases. It is also evident that the speedup is larger when the domains are small. This seems to indicate that encoding the compressed columns as bit vectors is beneficial.

6.3 Comparing join implementations

The characteristics of the input data used for comparing the join algorithms are shown in Table 6.3. For the ith relation, s_i, n_i, and S_i denote the size,

the number of tuples and the scope, respectively. Relations R_1 and R_2 are the input relations and R_0 the output relation. The last column lists the *join selectivity* for the join operation. It is an indication of the number of rows in the result relation compared to the total number of possible rows (which is $n_1 n_2$).

Name	s_1	n_1	$\|S_1\|$	s_2	n_2	$\|S_2\|$	$\|S_1 \cap S_2\|$	$\sum_{x \in S_1 \cap S_2} \|D_x\|$	s_0	$\frac{n_0}{n_1 n_2}$
1shelf	25286	710	3	77201	1283	4	2	1454	194099	0.0081
plan31b	113882	5132	15	59934	2964	15	14	28	294996	0.0008
q10c	33076	4200	7	43592	4898	7	6	60	66884	0.0004

Table 6.3: Characteristics of the input data used for performance comparison of join.

The performance results are presented in Table 6.4. The speedup factor is calculated with the existing C++ implementation used as base. Again, the

	APL		Old		New	
Name	Time	Speedup	Time	Speedup	Time	Speedup
1shelf	0.93	0.80	0.742	1	0.345	2.2
plan31b	17.3	0.88	15.3	1	2.40	6.4
q10ca	11.4	0.69	7.88	1	0.690	11.4

Table 6.4: Execution time of join algorithms. All times are measured in CPU-seconds.

new implementation is faster than both existing implementations, although the difference is not as significant as with compress. As with compress, the speedup seems to be larger when the common attributes can be encoded using bit vectors (as in case plan31b and q10c).

Chapter 7

Conclusion

The main contribution of this report is a detailed description and complexity analysis of a heuristic to compress relational tables. This heuristic was previously only informally described, with no complexity results.

Furthermore we have implemented the compress algorithm, and a join algorithm that works on compressed relations. These implementations have been benchmarked against existing implementations. The results obtained, show the algorithms presented here to be faster than any previous implementations for the data sets used.

Further work

The theoretical properties of the compression scheme presented in Chapter 3 should be further investigated.

For benchmarking CSP solvers it is important to (also) use real life problems as input, since they often have a certain structure which can be exploited during the solving process. This is why real life data was used as the basis for the benchmark in this report. In hindsight however, it seems like a better choice to mainly use synthetic data for benchmarking the low level algorithms. This will make it easier to control the different characteristics in order to measure which parameters are important for the actual running time.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts (1974).

[2] Array Technology A/S, *Array Database^{TM} Modelers Guide - Version 5.5* (2001). http://www.arraytechnology.com/documents/Modelers%20Guide.pdf

[3] C. Berge, *Graphs and Hypergraphs*, North-Holland (1973).

[4] P. A. Boncz, S. Manegold, and M. L. Kersten, Database architecture optimized for the new bottleneck: Memory access, *The VLDB Journal* **9** (2000), 231–246.

[5] E. F. Codd, A relational model of data for large shared data banks, *Communications of the ACM* **13**, 6 (1970), 377–387.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, *The MIT Electrical Engineering and Computer Science Series*, MIT Press/McGraw Hill (1990).

[7] R. Dechter and A. Dechter, Belief maintenance in dynamic constraint networks, *Proceedings of the Seventh Annual Conference of the American Association of Artificial Intelligence*, AAAI Press/MIT Press (1988), 37–42.

[8] O. I. Franksen, Group representations of finite polyvalent logic — a case study using APL notation, *A link between science and applications of automatic control*, Permangon Press (1979), 875–887.

[9] E. C. Freuder, A sufficient condition for backtrack-free search, *Journal of the ACM* **29**, 1 (1982), 24–32.

[10] M. Gyssens, P. Jeavons, and D. Cohen, Decomposing constraint satisfaction problems using database techniques, *Artificial Intelligence* **66**, 1 (1994), 57–89.

[11] S. Helmer and G. Moerkotte, Evaluation of main memory join algorithms for joins with set comparison join predicates, *Proceedings of 23rd International Conference on Very Large Data Bases*, Morgan Kaufmann (1997), 386–395.

[12] J. Katajainen and M. Lykke, Experiments with universal hashing, Technical Report **96/8**, Department of Computer Science, University of Copenhagen (1996).

[13] G. Kondrak and P. van Beek, A theoretical evaluation of selected backtracking algorithms, *Artificial Intelligence* **89** (1997), 365–387.

[14] V. Kumar, Algorithms for constraints satisfaction problems: A survey, *AI Magazine* **13**, 1 (1992), 32–44.

[15] N. C. Meyers, Traits: A new and useful template technique, *C++ Report* (1995). Available at `http://www.cantrip.org/traits.html`

[16] P. Mishra and M. H. Eich, Join processing in relational databases, *ACM Computing Surveys* **24**, 1 (1992), 63–113.

[17] G. L. Møller, On the technology of array-based logic, Ph. D. Thesis, Technical University of Denmark, Lyngby (1995). Available at `http://www.arraytechnology.com/documents/lic.pdf`

[18] T. More, Jr., Axioms and theorems for a theory of arrays, *IBM Journal of Research and Development* **17**, 2 (1973), 135–175.

[19] J. Pearson and P. Jeavons, A survey of tractable constraint satisfaction problems, Technical Report **CSD-TR-97-15**, Royal Holloway University of London (1997).

[20] F. Rossi, C. Petrie, and V. Dhar, On the equivalence of constraint satisfaction problems, *Proceedings of the 9th European Conference on Artificial Intelligence*, Pitman, Stockholm (1990), 550–556.

[21] A. Shatdal, C. Kant, and J. F. Naughton, Cache conscious algorithms for relational query processing, *Proceedings of 20th International Conference on Very Large Data Bases*, Morgan Kaufmann (1994), 510–521.

[22] M. Thorup, Even strongly universal hashing is pretty fast, *Proceedings of the 11th Annual Symposium on Discrete Algorithms*, ACM-SIAM (2000), 496–497.

[23] P. Valduriez, Join indices, *ACM Transactions on Database Systems* **12**, 2 (1987), 218–246.

[24] M. N. Wegman and J. L. Carter, New hash functions and their use in authentication and set equality, *Journal of Computer and System Sciences* **22**, 3 (1981), 265–279.

Appendix A

Source code

This appendix contains the source code, which implements the algorithms described in the main text. The files included are:

column.cpp: Contains implementation of the `Column` class.

compress.cpp: Contains implementation of the compression algorithm.

csp.cpp: Contains implementation of the `CSP` class.

hashfunction.cpp: Contains implementation of the hash function.

join.cpp: Contains implementation of the join algorithm.

relation.cpp: Contains implementation of `Relation` class.

uniquerows.cpp: Contains implementation of the classes used to find the complement of a relation.

variable.cpp: Contains implementation of the class `Variable`.

column.h: Contains declarations of the classes `Column`, `UncompressedColumn` and `CompressedColumn`.

columnimpl.h: Contains declaration and implementation of the classes `ConcreteUncompressedColu` `ConcreteCompressedColumn` and `BitmapCompressedColumn<N>`.

columntraits.h: Contains implementation of the column cell traits classes.

csp.h: Contains the declaration of the `CSP` class.

hashfunction.h: Contains the declaration and implementation of the `RandomTable<N>` class.

newbitset.h: Contains the declaration and implementation of the `NewBitset<N>` class.

relation.h: Contains the declaration of the `Relation` class.

uniquerows.h: Contains the declaration of classes used to find the complement of a relation.

variable.h: Contains the declaration of the `Variable` class.

A.1 column.cpp

```
/*
 * Copyright (C) 2003 Jeppe Nejsum Madsen, nejsum@diku.dk
 *
 * This program is free software; you can redistribute it and/or
   modify
 * it under the terms of the GNU General Public License as published
   by
 * the Free Software Foundation; either version 2, or (at your option
   )
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */
#include "stdafx.h"
#include "variable.hpp"
#include "detail/column_impl.hpp"

// Create a new ConcreteUncompressedColumn<T> with the T being the
// smallest type large enough to contain all the domain indicies of
// the variable var. If rowCount > 0, it specifies the initial number
// of rows allocated in the new column
UncompressedColumn *UncompressedColumn::createNew(Variable &var,
    size_t rowCount)
  {
  size_t max_dom_index = var.getDomainSize()-1;

  if (max_dom_index < 8)
    return new ConcreteUncompressedColumn<unsigned char,
        BitsetCellTraits<NewBitset<8> > >(var, rowCount);

  if (max_dom_index < 16)
```

```
    return new ConcreteUncompressedColumn<unsigned char,
        BitsetCellTraits<NewBitset<16> > >(var, rowCount);

  if (max_dom_index < 32)
    return new ConcreteUncompressedColumn<unsigned char,
        BitsetCellTraits<NewBitset<32> > >(var, rowCount);
  if (max_dom_index < 64)
    return new ConcreteUncompressedColumn<unsigned char,
        BitsetCellTraits<NewBitset<64> > >(var, rowCount);

  if (max_dom_index < 128)
    return new ConcreteUncompressedColumn<unsigned char,
        BitsetCellTraits<NewBitset<128> > >(var, rowCount);

  if (max_dom_index < 256)
    return new ConcreteUncompressedColumn<unsigned char,
        BitsetCellTraits<NewBitset<256> > >(var, rowCount);

  if (max_dom_index <= std::numeric_limits<unsigned char>::max())
    return new ConcreteUncompressedColumn<unsigned char>(var,
        rowCount);

  if (max_dom_index <= std::numeric_limits<unsigned short>::max())
    return new ConcreteUncompressedColumn<unsigned short>(var,
        rowCount);

  if (max_dom_index <= std::numeric_limits<unsigned int>::max())
    return new ConcreteUncompressedColumn<unsigned int>(var, rowCount
        );
  return new ConcreteUncompressedColumn<unsigned long>(var, rowCount)
      ;
  }

// Create a new CompressedColumn from a urvector_t. If the domain of
// the variable is within certain limits, a BitmapCompressedColumn is
// created, otherwise a ConcreteCompressedColumn.
CompressedColumn *CompressedColumn::createNew(Variable &var, const
    urvector_t &ur)
  {
  size_t max_dom_index = var.getDomainSize()-1;
  if (max_dom_index < 8)
    return new BitmapCompressedColumn<8>(var, ur);

  if (max_dom_index < 16)
    return new BitmapCompressedColumn<16>(var, ur);

  if (max_dom_index < 32)
    return new BitmapCompressedColumn<32>(var, ur);
```

```
    if (max_dom_index < 64)
      return new BitmapCompressedColumn<64>(var, ur);

    if (max_dom_index < 128)
      return new BitmapCompressedColumn<128>(var, ur);

    if (max_dom_index < 256)
      return new BitmapCompressedColumn<256>(var, ur);
    if (max_dom_index <= std::numeric_limits<unsigned char>::max())
      return new ConcreteCompressedColumn<unsigned char>(var, ur);

    if (max_dom_index <= std::numeric_limits<unsigned short>::max())
      return new ConcreteCompressedColumn<unsigned short>(var, ur);

    if (max_dom_index <= std::numeric_limits<unsigned int>::max())
      return new ConcreteCompressedColumn<unsigned int>(var, ur);

    return new ConcreteCompressedColumn<unsigned long>(var, ur);
    }

  CompressedColumn *CompressedColumn::createNew(Variable &var, size_t
      rowCount)
    {
    size_t max_dom_index = var.getDomainSize()-1;

    if (max_dom_index < 8)
      return new BitmapCompressedColumn<8>(var, rowCount);

    if (max_dom_index < 16)
      return new BitmapCompressedColumn<16>(var, rowCount);

    if (max_dom_index < 32)
      return new BitmapCompressedColumn<32>(var, rowCount);

    if (max_dom_index < 64)
      return new BitmapCompressedColumn<64>(var, rowCount);

    if (max_dom_index < 128)
      return new BitmapCompressedColumn<128>(var, rowCount);

    if (max_dom_index < 256)
      return new BitmapCompressedColumn<256>(var, rowCount);

      if (max_dom_index <= std::numeric_limits<unsigned char>::max())
      return new ConcreteCompressedColumn<unsigned char>(var, rowCount)
          ;

    if (max_dom_index <= std::numeric_limits<unsigned short>::max())
```

```
    return new ConcreteCompressedColumn<unsigned short>(var, rowCount
        );

  if (max_dom_index <= std::numeric_limits<unsigned int>::max())
    return new ConcreteCompressedColumn<unsigned int>(var, rowCount);

  return new ConcreteCompressedColumn<unsigned long>(var, rowCount);
  }

std::ostream &operator<<(std::ostream &s, const Column *c)
  {
  return s << c->getVariable().getName();
  }

std::ostream &operator<<(std::ostream &s, const colpair_t &v)
  {
  return s << "(" << (v.first ? v.first->getVariable().getName() : "
      null")
           << "," << (v.second ? v.second->getVariable().getName() :
               "null") << ")";
  }

std::ostream &operator<<(std::ostream &s, const joinindex_t &ji)
  {
  joinindex_t::const_iterator it = ji.begin(), pend = ji.end();
  while (it != pend)
    {
    s << "(" << it->first << "," << it->second << ")\n";
    ++it;
    }
  return s;
  }
```

A.2 compress.cpp

```
/*
 * Copyright (C) 2003 Jeppe Nejsum Madsen, nejsum@diku.dk
 *
 * This program is free software; you can redistribute it and/or
    modify
 * it under the terms of the GNU General Public License as published
    by
 * the Free Software Foundation; either version 2, or (at your option
    )
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */
#include "stdafx.h"
#include "csplib/uniquerows.hpp"
#include "csplib/relation.hpp"
#include "csplib/utility/formatter.hpp"

namespace
{

// Utility types & structs to order columns
// based on unique tuples in complement
typedef std::pair<size_t, colvector_t::const_iterator> orderpair_t;
typedef std::vector<orderpair_t>                       comporder_t;
struct ColumnOrdering
  {
  bool operator()(const orderpair_t &p1, const orderpair_t &p2)
    {
    // If same size, use varid to order
    if (p1.first == p2.first)
      return (*(p1.second))->getVariable() < (*(p2.second))->
          getVariable();

    return p1.first < p2.first;
    }
  };

// Create the vector order, which contains columns in the order in
// which tey should be compressed
void determineCompressOrder(colvector_t &cols, comporder_t &order,
    const varset_t *vars_to_compress)
  {
  colvector_t::const_iterator it = cols.begin(), pend = cols.end();
  HashValues                  vals(cols, it);

  Formatter f("Compression order");
  int n = 1;
  while(it != pend)
    {
    Column *column = *it;

    if (vars_to_compress != 0 && vars_to_compress->find(&column->
        getVariable()) == vars_to_compress->end())
      {
      ++it;
```

```
58        continue;
          }

        f.writeProgress(n++, vars_to_compress ? vars_to_compress->size()
            : cols.size());
        UniqueRowCount  uc(cols, it, vals);
63
        if (uc.size() < column->getNumCells())
          order.push_back(std::make_pair(uc.size(), it));

        vals.xorColumn(*column);
68      if (++it ≠ pend)
          vals.xorColumn(*(*it));
        }
      std::sort(order.begin(), order.end(), ColumnOrdering());
      }
73
    // Compress relation on the specified column
    bool compressColumn(colvector_t &columns, colvector_t::const_iterator
        compress_col)
      {
      urvector_t urvector;
78
      UniqueComplement urs(columns, compress_col);

      (*compress_col)->fillInUniqueComplement(urs);

83    // Don't compress if there's no benefit
      if (urs.size() ≥ (*compress_col)->getNumCells())
        return false;
      // Extract info into sorted vector
      UniqueComplement::const_iterator it = urs.begin(), pend = urs.end()
          ;
88    while (it ≠ pend)
        {
        urvector.push_back(*it);
        it++;
        }
93
      std::sort(urvector.begin(), urvector.end());

      // Create the compressed column from the unique rows
      CompressedColumn *newCol = CompressedColumn::createNew((*
          compress_col)->getVariable(), urvector);
98
      // Remove the non-unique tuples of the remaining columns
      colvector_t::iterator colit = columns.begin(), colpend = columns.
          end();
      while(colit ≠ colpend)
```

```
    {
103   if (colit == compress_col)
        {
        // Replace the compressed column
        boost::checked_delete(*colit);
        *colit = newCol;
108     }
      else
        (*colit)->pruneCells(urvector.begin(), urvector.end());

      ++colit;
113   }
    return true;
    }

  // Uncompress a single row
118 // begin, end are begin()/end() from new uncompressed columns
  // new_col is the current new uncompressed column
  // compressed_col is the current column being uncompressed,
  // startRow is the row where values should be copied from
  // row is the row in comppressed_col to uncompress
123 // copyPrevious indicates whether
  void uncompressRow(bool uncompress_link, colvector_t::iterator begin,
      colvector_t::iterator end,
                     colvector_t::iterator new_col, colvector_t::
                         iterator compressed_col,
                     size_t startRow, size_t row )
    {
128 // Terminate recursion
    if (new_col == end)
      return;

    Variable &v = (*compressed_col)->getVariable();
133
    if (!(*compressed_col)->isCompressed() || (v.isLinkVar() && !
        uncompress_link))
      {
      if ((*compressed_col)->isCompressed())
        {
138     idvector_t cellValues;
        CompressedColumn *cc = dynamic_cast<CompressedColumn *>(*
            compressed_col);
        CompressedColumn *c = dynamic_cast<CompressedColumn *>(*new_col
            );
        cc->expandCell(row, cellValues);
        c->addCell(cellValues);
143     }
      else
        {
```

```
      UncompressedColumn *uc = dynamic_cast<UncompressedColumn *>(*
          compressed_col);
      UncompressedColumn *c = dynamic_cast<UncompressedColumn *>(*
          new_col);
148   c->addCell(uc->getCell(row));
      }

    uncompressRow(uncompress_link, begin, end, boost::next(new_col),
        boost::next(compressed_col),
                (*new_col)->getNumCells()-1, row);
153
    }
  else
    {
    idvector_t cellValues;
158 CompressedColumn *cc = dynamic_cast<CompressedColumn *>(*
        compressed_col);
    cc->expandCell(row, cellValues);

    idvector_t::iterator value;
    bool copyPrevious = false; // Don't copy previus columns for
        first value
163 for(value = cellValues.begin(); value != cellValues.end(); ++
        value)
      {
      // Copy values from previous columns
      colvector_t::iterator it = begin;

168   while(copyPrevious && it != new_col)
        {
        UncompressedColumn *c = dynamic_cast<UncompressedColumn *>(*
            it);
        assert(c);
        c->addCellCopy(startRow);
173     ++it;
        }

      // Copy value from this column
      UncompressedColumn *c = dynamic_cast<UncompressedColumn *>(*
          new_col);
178   assert(c);
      c->addCell(*value);

      // Insert values from following columns
      uncompressRow(uncompress_link, begin, end, boost::next(new_col)
          , boost::next(compressed_col),
183               c->getNumCells()-1, row);
      copyPrevious = true;
      }
```

```
      }
    }
188 }


  // Compress the relation
  bool Relation::compress(const varset_t *vars_to_compress)
193 {
    bool        compressed = false;
    comporder_t compressOrder;

    determineCompressOrder(m_Columns, compressOrder, vars_to_compress);
198
    comporder_t::const_iterator it = compressOrder.begin(), pend =
        compressOrder.end();

    if (it != pend && it->first < getNumRows())
      {
203   // Only compress if there's a benefit
      Formatter f("Compressing");
      int n = 1;
      while (it != pend)
        {
208     if (compressColumn(m_Columns, it->second))
          compressed = true;
        f.writeProgress(n++, getNumColumns(), getNumRows());
        ++it;
        }
213   }

    if (compressed)
      rowCountChanged();

218 return compressed;
    }

  // Uncompress the relation
  bool Relation::uncompress(bool uncompress_link)
223 {
    bool compressed = false;

    for(size_t i = 0; i < getNumColumns(); ++i)
      compressed |= m_Columns[i]->isCompressed();
228
    if(!compressed)
      return false;

    colvector_t newCols;
233
```

```
    for(size_t i = 0; i < getNumColumns(); ++i)
      {
      Variable &v = m_Columns[i]->getVariable();
      if (!uncompress_link && v.isLinkVar() && m_Columns[i]->
          isCompressed())
238     newCols.push_back(CompressedColumn::createNew(v));
      else
        newCols.push_back(UncompressedColumn::createNew(v));
      }

243 colvector_t::iterator it = m_Columns.begin();

    Formatter f("Uncompressing");

    for(size_t row = 0; row < getNumRows(); ++row, false)
248   {
      if ((row+1) % 10 == 0)
        f.writeProgress(row+1, getNumRows());

      uncompressRow(uncompress_link, newCols.begin(), newCols.end(),
          newCols.begin(), it, 0, row);
253   }

    for_all(m_Columns, delete_element<Column>());
    m_Columns = newCols;
    rowCountChanged();
258 return true;
    }
```

A.3 csp.cpp

A.4 hashfunction.cpp

A.5 join.cpp

```
1 /*
  * Copyright (C) 2003 Jeppe Nejsum Madsen, nejsum@diku.dk
  *
  * This program is free software; you can redistribute it and/or
    modify
  * it under the terms of the GNU General Public License as published
    by
6 * the Free Software Foundation; either version 2, or (at your option
    )
  * any later version.
  *
  * This program is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */
#include "stdafx.h"
#include "csplib/relation.hpp"
#include "csplib/utility/formatter.hpp"
#include "csplib/uniquerows.hpp"

namespace
{
struct ColumnJoinOrder
  {
  bool operator()(const colpair_t &v1, const colpair_t &v2)
    {
    return v1.first->getVariable().getDomainSize() > v2.first->
        getVariable().getDomainSize();
    }
  };

// Constructs a joinindex for the common columns of two relations
void joinColumns(const colpairvector_t &columns, joinindex_t &ji,
    size_t numCommon, size_t maxrows)
  {
  ji.clear();
  bool done = false;
  size_t  start = 0, start_row = 0;

  while(!done)
    {
    colpairvector_t::const_iterator it = columns.begin(), pend =
        columns.end();
    Formatter f("Joining columns");
    int n = 1;

    // For the first common column, build initial joinindex
    while (it != pend)
      {
      Column *c1 = it->first, *c2 = it->second;
      ++it;
      // Only join on columns which are in both relations
      if (c1 && c2)
        {
        done = c1->createJoinIndex(ji, *c2, start_row, ji.size() +
            5000000);
        f.writeProgress(n++, numCommon, ji.size());
```

```
56          break;
            }
          }

        // For the remaining common columns (if any), incrementally prune
             join
61      // index
        //size_t new_start = ji.size();

        while(it ≠ pend)
          {
66        Column *c1 = it->first, *c2 = it->second;
          // Only join on columns which are in both relations
          if (c1 && c2)
            {
            c1->reviseJoinIndex(ji, *c2, start);
71          f.writeProgress(n++, numCommon, ji.size());
            }
          ++it;
          }
        start = ji.size();
76      if (start > maxrows)
          throw std::length_error("Join Columns");
        }
      }

81  // Create a vector containing pair of columns. first points to
        columns
    // in r1, second points to columns in r2. If a variable is in both
    // relations the pair will have two non-null fields. Returns the
    // number of common columns
    size_t unionColumns(const Relation *r1, const Relation *r2,
        colpairvector_t &columns)
86  {
      size_t numCommon = 0;
      colvector_t::const_iterator it1 = r1->getColumns().begin() , it2 =
          r2->getColumns().begin();
      colvector_t::const_iterator pend1 = r1->getColumns().end() , pend2
          = r2->getColumns().end();

91    while (it1 ≠ pend1 && it2 ≠ pend2)
        {
        size_t id1 = (*it1)->getVariable().getId();
        size_t id2 = (*it2)->getVariable().getId();

96      if (id1 < id2)
          columns.push_back(std::make_pair(*it1++, (Column *)0));
        else if (id1 > id2)
          columns.push_back(std::make_pair((Column *)0, *it2++));
```

```
    else
      {
      columns.push_back(std::make_pair(*it1++, *it2++));
      numCommon++;
      }
    }

  while (it1 != pend1)
    columns.push_back(std::make_pair(*it1++, (Column *)0));

  while (it2 != pend2)
    columns.push_back(std::make_pair((Column *)0, *it2++));

  return numCommon;
  }


// Construct output relation based on the join index
void constructRelation(Relation *newRel, const colpairvector_t &
    columns, const joinindex_t &ji, const varset_t &vars_to_remove)
  {
  colpairvector_t::const_iterator it;
  std::stringstream ss;
  ss << "Constructing relation " << newRel->getId();

  Formatter f(ss.str());
  int n = 1;
  for(it = columns.begin(); it != columns.end(); ++it)
    {
    colpair_t cp = *it;
    Column   *c;

    f.writeProgress(n++, columns.size());

    Variable &var = cp.first != 0 ? cp.first->getVariable() : cp.
        second->getVariable();
    if (vars_to_remove.find(&var) != vars_to_remove.end())
      continue;

    if (cp.first == 0 || cp.second == 0)
      {
      // Column is not common, clone the column, picking only the
      // values specified by the join index
      if (cp.first != 0)
        {
        c = cp.first->cloneFromJoinIndex(ji, true);
        }
      else
        {
```

```
        c = cp.second->cloneFromJoinIndex(ji, false);
        }
      }
    else
151   {
      // Column is common, create a new column based on the
      // intersection of values specified in the join index
      c = cp.first->createFromJoinIndex(ji, *cp.second);
      }
156   newRel->addColumn(std::auto_ptr<Column>(c));
    }
  }
}

161 // Construct cartesian product output relation
void constructCartesianProduct(Relation *newRel, const
    colpairvector_t &columns, size_t numNewRows, const varset_t &
    vars_to_remove)
  {
  colpairvector_t::const_iterator it;

166  Formatter f("Constructing cartesian product");
  int n = 1;
  for(it = columns.begin(); it != columns.end(); ++it)
    {
    f.writeProgress(n++, columns.size(), numNewRows);
171 colpair_t cp = *it;
    Column   *c;

    Variable &var = cp.first != 0 ?  cp.first->getVariable() : cp.
        second->getVariable();
    if (vars_to_remove.find(&var) != vars_to_remove.end())
176   continue;

    if (cp.first != 0)
      {
      c = cp.first->clone(numNewRows, false);
181   }
    else
      {
      c = cp.second->clone(numNewRows, true);
      }
186 newRel->addColumn(std::auto_ptr<Column>(c));
    }
}

std::auto_ptr<Relation> Relation::joinCompress(const Relation &r,
    size_t new_id, size_t maxrows, const varset_t &vars_to_remove)
    const
```

```
{
varset_t intersect = intersection(getVariables(), r.getVariables())
    ;

if (intersect.size() == 0)
  return join(r, new_id, maxrows, vars_to_remove);

varset_t unionvars = getVariables() + r.getVariables();

unionvars = unionvars - vars_to_remove;

std::auto_ptr<Relation> split1, split2, newrel;
std::auto_ptr<Relation> proj1, proj2;
std::auto_ptr<Variable> lv1, lv2;

const Relation *r1, *r2;

varset_t splitvars1 = getVariables() - intersect - vars_to_remove;
if (splitvars1.size() > 0)
  {
  proj1  = project(intersect, 42);
  split1 = project(splitvars1, 43);
  lv1    = proj1->link(*split1, 1000000);
  proj1->compress();
  split1->compress();
  r1 = proj1.get();
  }
else
  r1 = this;

varset_t splitvars2 = r.getVariables() - intersect - vars_to_remove
    ;
if (splitvars2.size() > 0)
  {
  proj2  = r.project(intersect, 44);
  split2 = r.project(splitvars2, 45);
  lv2    = proj2->link(*split2, 1000001);
  proj2->compress();
  split2->compress();
  r2 = proj2.get();
  }
else
  r2 = &r;

newrel = r1->join(*r2, new_id, maxrows, vars_to_remove);
newrel->compress();

if (split1.get() ≠ 0)
  {
```

```
      newrel = newrel->join(*split1, new_id, maxrows, split1->
          getVariables()->unionvars);
      newrel->compress();
      }
241   if (split2.get() != 0)
      {
      newrel = newrel->join(*split2, new_id, maxrows, split2->
          getVariables()->unionvars);
      newrel->compress();
      }
246
    return newrel;
    }

  // Join a relation with relation r, returning the result.
251 std::auto_ptr<Relation> Relation::join(const Relation &r, size_t
      new_id, size_t maxrows, const varset_t &vars_to_remove) const
    {
    std::auto_ptr<Relation> newRel(new Relation(new_id, "JOIN"));

    newRel->m_SourceIds = m_SourceIds;
256 newRel->m_SourceIds.insert(newRel->m_SourceIds.end(), r.m_SourceIds
        .begin(), r.m_SourceIds.end());

    // Find the smallest relation
    const Relation *smallRel, *largeRel;
    if (getNumRows() > r.getNumRows())
261   {
      smallRel = &r;
      largeRel = this;
      }
    else
266   {
      smallRel = this;
      largeRel = &r;
      }

271 colpairvector_t columns;
    size_t numCommon = unionColumns(smallRel, largeRel, columns);
    if (numCommon == 0)
      {
      // No common columns, create cartesian product
276   size_t numNewRows = smallRel->getNumRows() * largeRel->getNumRows
          ();
      if (numNewRows > maxrows)
        throw std::length_error("Cartesian product");

      constructCartesianProduct(newRel.get(), columns, numNewRows,
          vars_to_remove);
```

```
281       }
      else
        {
        joinindex_t ji;
        joinColumns(columns, ji, numCommon, maxrows);
286     constructRelation(newRel.get(), columns, ji, vars_to_remove);
        }

      return newRel;
      }
291
    size_t Relation::joinSize(const Relation &r, size_t maxrows)  const
      {
      size_t  size = 0;
      const Relation *small, *large;
296
      if (getNumRows() < r.getNumRows())
        {
        small = this;
        large = &r;
301     }
      else
        {
        small = &r;
        large = this;
306     }

      colpairvector_t colpairs;
      size_t numCommon = unionColumns(small, large, colpairs);

311   if (numCommon == 0)
        size = small->getNumRows() * large->getNumRows();
      else
        {
        for(size_t i = 0; i < small->getNumRows() && size < maxrows; ++i)
316       {
          boolvector_t  mask(large->getNumRows(), true);

          colpairvector_t::const_iterator it = colpairs.begin(), pend =
              colpairs.end();
          while(it ≠ pend)
321         {
            Column *c1 = it->first, *c2 = it->second;
            if (c1 && c2)
              {
              c1->updateTupleMask(i, *c2, mask);
326           }
            ++it;
            }
```

```
        size += std::count(mask.begin(), mask.end(), true);
331     }
      }
    if (size == 0)
      {
      std::cout << *small << *large;
336   }

    return size;
    }
```

A.6 relation.cpp

```
1  /*
   * Copyright (C) 2003 Jeppe Nejsum Madsen, nejsum@diku.dk
   *
   * This program is free software; you can redistribute it and/or
     modify
   * it under the terms of the GNU General Public License as published
     by
6  * the Free Software Foundation; either version 2, or (at your option
     )
   * any later version.
   *
   * This program is distributed in the hope that it will be useful,
   * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   * GNU General Public License for more details.
   *
   * You should have received a copy of the GNU General Public License
   * along with this program; if not, write to the Free Software
16 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
   */
  #include "stdafx.h"
  #include <algorithm>
  #include <functional>
21 #include <numeric>
  #include <iomanip>
  #include "relation.hpp"
  #include "uniquerows.hpp"

26 Relation::Relation(size_t id, const std::string &expr)
      : m_Id(id),
        m_Expr(expr)
    {
    }
31
  Relation::~Relation()
```

```
    {
    for_all(m_Columns, delete_element<Column>());
    }

std::auto_ptr<Variable> Relation::link(Relation &r, size_t new_id)
    {
    std::auto_ptr<Variable> lv;

    if (getNumRows() != r.getNumRows())
      throw std::logic_error("link: Relations does not have same row
          count");

    UniqueRows ur1(getColumns());
    (*getColumns().begin())->fillInUniqueRows(ur1);

    UniqueRows ur2(r.getColumns());
    (*r.getColumns().begin())->fillInUniqueRows(ur2);

    size_t numArgs = std::min(ur1.size(), ur2.size());

    // Only link relations if there is more than 1 unique argument in
        either
    // relation
    if (numArgs > 1)
      {
      lv.reset(new LinkVariable(new_id, "Link "+boost::lexical_cast<std
          ::string>(new_id), numArgs));

      UncompressedColumn *uc1 = UncompressedColumn::createNew(*lv,
          getNumRows());
      UncompressedColumn *uc2 = UncompressedColumn::createNew(*lv, r.
          getNumRows());

      std::cout << lv->getName() << " size=" << numArgs << ", other="
          << std::max(ur1.size(), ur2.size()) << "\n";

      UniqueRows::const_iterator it, pend;
      if (ur1.size() < ur2.size())
        {
        it = ur1.begin();
        pend = ur1.end();
        }
      else
        {
        it = ur2.begin();
        pend = ur2.end();
        }

      for(size_t i = 0; i < numArgs; i++)
```

```
76        {
          size_t cell = it->first;
          uc1->setCell(cell, i);
          uc2->setCell(cell, i);
          idvector_t *idv = it->second;
81        idvector_t::const_iterator it1 = idv->begin(), pend1 = idv->end
              ();
          while(it1 ≠ pend1)
            {
            cell = *it1++;
            uc1->setCell(cell, i);
86          uc2->setCell(cell, i);
            }
          ++it;
          }
        addColumn(std::auto_ptr<Column>(uc1));
91      r.addColumn(std::auto_ptr<Column>(uc2));
        }

      return lv;
      }
96
    void Relation::assign(const Relation &rel)
      {
      for_all(m_Columns, delete_element<Column>());

101   m_Columns.clear();
      m_Variables.clear();

      colvector_t::const_iterator it = rel.m_Columns.begin(), pend = rel.
          m_Columns.end();
      while(it ≠ pend)
        {
106     Column   *c = *it;
        addColumn(std::auto_ptr<Column>(c->clone()));
        ++it;
        }
111   m_ValidRows = rel.m_ValidRows;
      }

    void Relation::addColumn(std::auto_ptr<Column> col)
      {
116   // Ensure all columns have the same number of rows
      if (!m_Columns.empty() && col->getNumCells() ≠ getNumRows())
        throw std::logic_error("Columns don't have same rowcount");

      if (m_Columns.empty())
121     {
        m_ValidRows = boolvector_t(col->getNumCells(), true);
```

```
    }

  col->setIndex(getNumColumns());
  m_Variables.insert(&col->getVariable());
  m_Columns.push_back(col.release());

  // Ensure all columns are inserted in variable id order
  assert(std::is_sorted(m_Columns.begin(), m_Columns.end(),
      ColumnPtrVariableOrder()));
  }

void Relation::addTautologyColumn(Variable &var)
  {
  size_t n = m_Columns.empty() ? 1 : getNumRows();

  CompressedColumn *col = CompressedColumn::createNew(var, n);

  idvector_t  domain_values;
  for(size_t i = 0; i < var.getDomainSize(); ++i)
    {
    if (var.isValueValid(i))
      domain_values.push_back(i);
    }

  for(size_t i = 0; i < n; ++i)
    {
    col->addCell(domain_values);
    }
  col->setIndex(getNumColumns());
  if (m_Columns.empty())
    {
    m_ValidRows = boolvector_t(col->getNumCells(), true);
    }

  m_Variables.insert(&col->getVariable());
  m_Columns.push_back(col);

  // Ensure all columns are inserted in variable id order
  std::sort(m_Columns.begin(), m_Columns.end(),
      ColumnPtrVariableOrder());
  }

Column &Relation::getColumn(const Variable &var)
  {
  colvector_t::iterator it = m_Columns.begin(), pend = m_Columns.end
      ();
  while (it ≠ pend)
    {
    if ((*it)->getVariable() == var)
```

```
      return *(*it);
    ++it;
    }
  throw std::logic_error("getColumn: searched column not found!");
  }

const Column &Relation::getColumn(const Variable &var) const
  {
  colvector_t::const_iterator it = m_Columns.begin(), pend =
      m_Columns.end();
  while (it ≠ pend)
    {
    if ((*it)->getVariable() == var)
      return *(*it);
    ++it;
    }
  throw std::logic_error("getColumn: searched column not found!");
  }

double Relation::getNumExpandedTuples() const
  {
  std::vector<double> row_counts(getNumRows(), 1.0);

  colvector_t::const_iterator it = m_Columns.begin(), pend =
      m_Columns.end();

  while(it ≠ pend)
    {
    Column *c = *it;
    if (c->isCompressed())
      c->updateRowCounts(row_counts);
    ++it;
    }
  return std::accumulate(row_counts.begin(), row_counts.end(), 0.0);;
  }

double Relation::getSize() const
  {
  double size = 0;
  colvector_t::const_iterator it = m_Columns.begin(), pend =
      m_Columns.end();
  while(it ≠ pend)
    {
    size += (*it)->getSize();
    ++it;
    }
  return size;
  }
```

```
bool Relation::identifyUnaryConstraints(varset_t *changed_variables)
{
colvector_t::const_iterator it = m_Columns.begin(), pend =
    m_Columns.end();
bool boundsChanged = false;

while(it ≠ pend)
  {
  Column  *c = *it;
  Variable &var = c->getVariable();
  size_t domSize = var.getDomainSize();
  boolvector_t mask(domSize, false);
  c->createUniqueMask(getValidRows(), mask);

  bool first = true;
  for(size_t i = 0; i < domSize; ++i)
    {
    if (var.isValueValid(i) && !mask[i])
      {
      boundsChanged = true;
      var.setValueInvalid(i);
      if (changed_variables)
        changed_variables->insert(&var);
      }
    }
  ++it;
  }
return boundsChanged;
}


bool Relation::revise()
{
bool     changed = false;
colvector_t::iterator it = getColumns().begin(), pend = getColumns
    ().end();

while(it ≠ pend)
  {
  bool colchange = (*it)->revise(getValidRows());
  if (colchange)
    {
    changed = true;
    }
  it++;
  }

return changed;
}
```

```
bool Relation::updateValidRowMask()
  {
  bool     changed = false;
  colvector_t::iterator it = getColumns().begin(), pend = getColumns
      ().end();

  while(it != pend)
    {
    changed |= (*it)->updateValidRowMask(getValidRows());
    it++;
    }

  return changed;
  }

size_t Relation::getNumCommonColumns(const Relation &r2) const
  {
  size_t numCommon = 0;
  colvector_t::const_iterator it1 = getColumns().begin() , it2 = r2.
      getColumns().begin();
  colvector_t::const_iterator pend1 = getColumns().end() , pend2 = r2
      .getColumns().end();

  while (it1 != pend1 && it2 != pend2)
    {
    size_t id1 = (*it1)->getVariable().getId();
    size_t id2 = (*it2)->getVariable().getId();

    if (id1 < id2)
      ++it1;
    else if (id1 > id2)
      ++it2;
    else
      {
      ++it1;
      ++it2;
      numCommon++;
      }
    }
  return numCommon;
  }

std::auto_ptr<Relation> Relation::split(colvector_t &remove_columns,
    size_t new_id)
  {
  Relation *newRel = new Relation(new_id, "SPLIT");

  newRel->m_SourceIds   = m_SourceIds;
```

```
    colvector_t::iterator it = remove_columns.begin(), pend =
        remove_columns.end();
    while(it ≠ pend)
      {
311   newRel->addColumn(std::auto_ptr<Column>(*it));
      m_Variables.erase(&(*it)->getVariable());
      m_Columns.erase(std::find(m_Columns.begin(), m_Columns.end(), *it
          ));
      ++it;
      }
316 return std::auto_ptr<Relation>(newRel);
    }

  std::auto_ptr<Relation> Relation::split(varset_t &remove_vars, size_t
      new_id)
    {
321 Relation *newRel = new Relation(new_id, "SPLIT");

    newRel->m_SourceIds   = m_SourceIds;

    varset_t::iterator it = remove_vars.begin(), pend = remove_vars.end
        ();
326 while(it ≠ pend)
      {
      Column *c = &getColumn(*(*it));

      newRel->addColumn(std::auto_ptr<Column>(c));
331   m_Variables.erase(*it);
      m_Columns.erase(std::find(m_Columns.begin(), m_Columns.end(), c))
          ;
      ++it;
      }
    return std::auto_ptr<Relation>(newRel);
336 }

  std::auto_ptr<Relation> Relation::project(const varset_t &vars,
      size_t new_id)  const
    {
    Relation *newRel = new Relation(new_id, "PROJ");
341
    colvector_t::const_iterator it = m_Columns.begin(), pend =
        m_Columns.end();
    while(it ≠ pend)
      {
      Column   *c = *it;
346   Variable &v = c->getVariable();

      if (vars.find(&v) ≠ vars.end())
```

```
        {
        newRel->addColumn(std::auto_ptr<Column>(c->clone()));
        }
      ++it;
      }
    newRel->m_ValidRows = this->m_ValidRows;

    return std::auto_ptr<Relation>(newRel);
    }

  bool Relation::isCompressed() const
    {
    bool compressed = false;
    colvector_t::const_iterator it = m_Columns.begin(), pend =
        m_Columns.end();

    for(;it != pend; ++it)
      compressed |= (*it)->isCompressed();

    return compressed;
    }

  bool Relation::pruneInvalidTuples()
    {
    size_t      old_row_count = getNumRows();
    urvector_t  ur;
    size_t      numValid = 0;
    bool        pruned = false;

    for(size_t i = 0; i < getNumRows(); ++i)
      {
      if (m_ValidRows[i])
        {
        ur.push_back(std::make_pair(i, (idvector_t*)0));
        numValid++;
        }
      }

    if (numValid < getNumRows())
      {
      pruned = true;
      colvector_t::iterator it = m_Columns.begin(), pend = m_Columns.
          end();
      while(it != pend)
        {
        (*it)->pruneCells(ur.begin(), ur.end());
        it++;
        }
      rowCountChanged();
```

```
396     std::cout << "Pruned relation " << m_Id << ": old="<<
            old_row_count << ", new=" << numValid << "\n";
        }
    return pruned;
    }

401 bool Relation::removeTautologyColumns(varset_t &tautology_vars)
    {
    colvector_t::iterator it = m_Columns.begin();

    bool first = true;
406
    while(it != m_Columns.end())
      {
      colvector_t column;
      column.push_back(*it);
411   UniqueRows  ur(column);
      (*it)->fillInUniqueRows(ur);

      if(ur.size() == 1)
        {
416     if (first)
          {
          identifyUnaryConstraints();
          first = false;
          }
421
        Variable &var = (*it)->getVariable();
        tautology_vars.insert(&var);
        m_Variables.erase(&var);
        delete *it;
426     it = m_Columns.erase(it);
        }
      else
        it++;
      }
431
    if (tautology_vars.size() > 0)
      {
      std::cout << "Removed " << tautology_vars.size() << " tautology
          column(s): " << (*tautology_vars.begin())->getName() << "\n";
      }
436 return tautology_vars.size() > 0;
    }

  void Relation::checkInvariants() const
    {
441 bool ok = true;
```

```
    if (!std::is_sorted(m_Columns.begin(), m_Columns.end(),
        ColumnPtrVariableOrder()))
      {
      ok = false;
      std::cerr << "Columns are not sorted\n";
446   colvector_t::const_iterator it,it_end;
      for(it = m_Columns.begin(), it_end = m_Columns.end(); it ≠
          it_end; ++it)
        {
        std::cerr << (*it)->getVariable().getId() << "\n";
        }
451   }

    if (m_ValidRows.size() ≠ getNumRows())
      {
      ok = false;
456   std::cerr << "m_ValidRows.size(" << m_ValidRows.size() << ") ≠
          getNumRows(" << getNumRows() << ")\n";
      }

    if (!ok)
      throw std::logic_error("Invariants do not hold");
461 }

  void Relation::rowCountChanged()
    {
    m_ValidRows = boolvector_t(getNumRows(), true);
466 }


  bool Relation::removeRedundancy()
    {
471 colvector_t::iterator it, pend = m_Columns.end();
    bool                   changed = false;

    for(size_t row = 0; row < getNumRows(); ++row)
      {
476   boolvector_t  contained(getValidRows());

      for(it = m_Columns.begin(); it ≠ pend; ++it)
        {
        Column      *c = *it;
481     c->updateContainmentMask(row, *c, contained);
        }

      for (size_t row2 = row+1; row2 < getNumRows(); ++row2)
        {
486     if (contained[row2])
          {
```

```
          changed = true;
          setRowValid(row2, false);
          }
        }
      }

    if (changed)
      pruneInvalidTuples();

    return changed;
    }

  std::string Relation::getRowAsString(size_t row) const
    {
    std::stringstream s;

    s << "[" << row << "]=";
    for(size_t col = 0; col < getNumColumns(); col++)
      {
      s << getColumns()[col]->cellToString(row) << "\t";
      }
    return s.str();
    }

  std::ostream &operator<<(std::ostream &s, const Relation &r)
    {
    size_t  numRows = r.getColumns()[0]->getNumCells();

    s << "Rel #" << r.getId() << "(" << r.getNumRows() << "/" << r.
        getNumExpandedTuples() << ")" << " ";

    for(size_t col = 0; col < r.getNumColumns(); col++)
      {
      s << r.getColumns()[col]->getVariable().getName();
      if (r.getColumns()[col]->isCompressed())
        s << "+ ";
      else
        s << " ";
      }

    return s;
    }

  std::ostream &rel_tuples(std::ostream &s, const Relation &r)
    {
    size_t  numRows = r.getColumns()[0]->getNumCells();

    std::vector<size_t> colwidths(r.getNumColumns(), 0);
    // Find column widths
```

```
536   for(size_t col = 0; col < r.getNumColumns(); col++)
        {
        size_t len = r.getColumns()[col]->getVariable().getName().length
            ();
        colwidths[col] = len;
        for(size_t i = 0; i < numRows; ++i)
541       {
          len = r.getColumns()[col]->cellToString(i).length();
          if (len > colwidths[col])
            colwidths[col] = len;
          }
546     }
      std::cout << "      ";
      for(size_t col = 0; col < r.getNumColumns(); col++)
        {
        s << std::left << std::setw(colwidths[col]) << r.getColumns()[
            col]->getVariable().getName();
551     if (r.getColumns()[col]->isCompressed())
          s << "+";
        else
          s << " ";
        }
556
      s << "\n";
      for(size_t i = 0; i < numRows; ++i)
        {
        std::cout << (r.isRowValid(i) ? " " : "-");
561     std::cout << std::setw(4) << i << " ";
        for(size_t col = 0; col < r.getNumColumns(); col++)
          {
          s << std::setw(colwidths[col]) << r.getColumns()[col]->
              cellToString(i) << " ";
          }
566     s << "\n";
        }
      return s;
      }

571 std::ostream &operator<<(std::ostream &s, const Relation *r)
      {
      return s << *r;
      }
```

A.7 uniquerows.cpp

```
1 /*
  * Copyright (C) 2003 Jeppe Nejsum Madsen, nejsum@diku.dk
  *
```

```
 * This program is free software; you can redistribute it and/or
     modify
 * it under the terms of the GNU General Public License as published
     by
 * the Free Software Foundation; either version 2, or (at your option
     )
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */
#include "stdafx.h"
#include <fstream>
#include "column.hpp"
#include "uniquerows.hpp"

const size_t primes[] = {2, 5, 11, 23, 47, 97, 197, 397, 797, 1597,
                         3203, 6421, 12853, 25717, 51437, 102877,
                             205759,
                         411527, 823117, 1646237, 3292489, 6584983,
                         13169977, 26339969, 52679969, 105359939,
                         210719881, 421439783, 842879579,
                             1685759167};

const size_t NUM_PRIMES = sizeof(primes)/sizeof(primes[0]);

// Return the number of buckets needed to hold the specified
// number of entries
size_t calculateBucketCount(size_t numRows)
  {
  size_t first = 0, last = NUM_PRIMES - 1;
  while (first < last)
    {
    size_t mid = (first + last) / 2;
    if(numRows < primes[mid])
      last = mid;
    else
      first = mid + 1;
    }
  return primes[last];
  }

UniqueComplement::UniqueComplement(const colvector_t &columns,
```

```
                        colvector_t::const_iterator colmask)
    : base_t(calculateBucketCount((*colmask)->getNumCells()),
             Hasher(values), detail::ColumnComplementEq(columns,
                 colmask)),
51    values(columns, colmask)
  {
  }

UniqueRows::UniqueRows(const colvector_t &columns)
56    : base_t(calculateBucketCount((*columns.begin())->getNumCells()),
             Hasher(values), detail::ColumnRowEq(columns)),
    values(columns)
  {
  }
61
UniqueRowCount::UniqueRowCount(const colvector_t &columns,
                               colvector_t::const_iterator colmask,
                               const HashValues &vals)
    : base_t(calculateBucketCount((*colmask)->getNumCells()),
66           Hasher(vals), CEQ(columns, colmask))
  {
  size_t row, numRows = (*colmask)->getNumCells();
  for(row = 0; row < numRows; row++)
    insert(row);
71 }

std::ostream &operator<<(std::ostream &s, const UniqueComplement &uc)
  {
  UniqueComplement::const_iterator it;
76
  for(it = uc.begin(); it != uc.end(); ++it)
    {
    s << it->first << " : ";
    write_set<idvector_t>(s, *(it->second));
81  s << "\n";
    }
  return s;
  }
```

A.8 variable.cpp

```
1 /*
  * Copyright (C) 2003 Jeppe Nejsum Madsen, nejsum@diku.dk
  *
  * This program is free software; you can redistribute it and/or
    modify
  * it under the terms of the GNU General Public License as published
    by
```

```
 * the Free Software Foundation; either version 2, or (at your option
   )
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */
#include "stdafx.h"
#include <math.h>
#include "variable.hpp"

Variable::Variable(size_t id, const std::string &name, size_t
    domainsize, bool link_var)
: m_Name(name), m_ValueValid(domainsize, true), m_Id(id),
  m_DomainSize(domainsize), m_DomainBits(0), m_IsLinkVar(link_var)
  {

  }

unsigned char Variable::getDomainBits() const
  {
  // Cache domain bits
  if (m_DomainBits == 0)
    {
    m_DomainBits = (unsigned char)(ceil(log(m_DomainSize)/log(2.0)));
    }
  return m_DomainBits;
  }

size_t Variable::getValidDomainSize() const
  {
  return std::count(m_ValueValid.begin(), m_ValueValid.end(), true);
  }

void Variable::removeInvalidValues()
  {
  m_ValueValid = std::vector<bool>(getValidDomainSize(), true);
  m_DomainSize = m_ValueValid.size();
  m_DomainBits = 0;
  }

varset_t intersection(const varset_t &u, const varset_t &v)
  {
```

```
  varset_t res;
  std::set_intersection(u.begin(), u.end(), v.begin(), v.end(),
    std::insert_iterator<varset_t>(res, res.begin()),
        VariablePtrOrder());
  return res;
  }

varset_t operator-(const varset_t &u, const varset_t &v)
  {
  varset_t res;
  std::set_difference(u.begin(), u.end(), v.begin(), v.end(),
    std::insert_iterator<varset_t>(res, res.begin()),
        VariablePtrOrder());
  return res;
  }

varset_t operator+(const varset_t &u, const varset_t &v)
  {
  varset_t res;
  std::set_union(u.begin(), u.end(), v.begin(), v.end(),
    std::insert_iterator<varset_t>(res, res.begin()),
        VariablePtrOrder());
  return res;
  }

std::ostream &operator<<(std::ostream &s, const Variable &v)
  {
  s << "Var " << v.getName() << " = {";
  for (size_t i = 0; i < v.getDomainSize(); i++)
    {
    if (i > 0)
      s << ",";
    s << (int)i << "=" << v.getDomainValueAsString(i);
    }
  s << "}";
  return s;
  }

std::ostream &operator<<(std::ostream &s, const Variable *v)
  {
  return s << v->getName();
  }
```

A.9 column.h

A.10 columnimpl.h

A.11 columntraits.h

A.12 csp.h

A.13 hashfunction.h

A.14 newbitset.h

A.15 relation.h

A.16 uniquerows.h

A.17 variable.h