Clustering PROLOG Programs for Distributed Computations

Patrick O. Bobbie
Division of Computer Science, University of West Florida, Pensacola, Florida

Mike Papazoglou
Department of Computer Science, Australian National University, Canberra, Australia

A knowledge base (KB) is a collection of factual information pertaining to the objects of specialized domains or application areas. KB information may be acquired and represented using language paradigms which are based on formalisms of predicate calculus. Usually, the domains are not necessarily distinct due to the interrelatedness of the components of the problem or interdependency of the objects. Therefore, this interdependency could generate long search paths or references to the KB objects, particularly for large KB data. However, KB data can be reorganized into groups or clusters using some common relational information of the data objects. The reorganization process isolates the data into clusters and localizes the interdependency within the clusters. Therefore, the clusters offer opportunities for mapping the data into distributed or parallel processing environments to facilitate computational efficiency. This article focuses on methods for structuring, partitioning, and clustering logic-based KB data (rules and facts) for distributed computations.

1. INTRODUCTION

Knowledge-base (KB) systems have emerged as a technology to support systems (software and hardware) that rely on expert knowledge, imprecise or incomplete data, and deductive or inference mechanisms. The support is a welcome advantage, however, the trade-off is the added time required for accessing the KB layer for information to solve a given problem. Thus, the overall computational efficiency of the supported system depends largely on the speed of processing the underlying KB data.

The objectives of distributing and processing KB data are to reduce search time, increase data availability through redundancy, promote system modularity, and take advantage of a network of computers. To achieve these goals, a direction of artificial intelligence (AI) research has focused on exploiting parallelism in KB systems and the distributability of AI computations. In this article, we focus on methods for clustering PROLOG KB data objects to facilitate distributable computations [1]. We discuss methodologies for matching the attributes or relationships of the data objects from which to develop clusters. The clusters serve as units of distributable data objects suitable for efficient computations on a network of computers.

The next section presents an overview and the limitations of the clustering methods. Following the overview is a discussion of methods for data representation, analysis, structuring, and decomposition. Lastly, we discuss methods for clustering partitions of the data objects. The discussions are reinforced by an example that epitomizes a software system developed to transform PROLOG KB data into clusters.

1.1 Methods

One of the approaches for improving the efficiency and performance of KB computations is to exploit parallelism in domain knowledge. Other techniques for attaining efficiency include taking advantage of parallelism in hardware, employing the run-time support of underlying (distributed) operating systems, or reorganizing KB data by analyzing the language constructs

Address correspondence to Patrick Bobbie, Div. of Computer Science, University of West Florida, 11000 University Parkway, Pensacola, FL 32514.

0164-1212/91/$3.50

used for knowledge representation. We take the latter approach.

Languages such as concurrent PROLOG [2] and ACTOR-based systems [3] provide a linguistic format for representing and expressing parallelism in KB data. Although a language such as standard sequential PROLOG does not provide constructs for expressing parallelism, it has an easy-to-understand, expressive format for KB data representation. PROLOG is a logic programming language with three constructors: rules, facts, and queries. One approach to the detection of parallelism in PROLOG programs is to analyze the rules, facts, and queries for features that suggest the interdependency of the constructors. Several mechanisms currently in use involve insertion of additional language constructs in the program to convey semantics of parallel evaluation. Among these mechanisms are AND-parallelism, OR-parallelism, and argument-parallelism [4].

We focus on standard PROLOG representation of KB data. Specifically, we exploit a different kind of parallelism, namely parallelism due to clustering of rules, facts, and queries. By definition [1], a PROLOG rule is a procedure with a lefthand-side predicate name (optionally parametrized) and a righthand side (body) of one or more conjunctive and/or disjunctive goals (or subgoals). A subgoal is either a simple fact or the predicate name (optionally parametrized) of another or the same (for recursion) rule.

prolog-rule (definition):

predicate-name([parameters]):-
subgoal-name$_1$ ([parameters]), . . . ,
subgoal-name$_{k-1}$ ([parameters]),
[(**if** ⟨condition⟩ **then**
subgoal-name$_k$ ([parameters]), . . . ,
subgoal-name$_{k+m}$ ([parameters])
[**else** /* optional else-part */
subgoal$_{k+m+1}$ ([parameters]), . . . ,
subgoal$_n$ ([parameters])]
),] /* zero or more disjunctive subgoals */
subgoal-name$_{n+1}$ ([parameters]),
[*predicate-name* ([parameters])]./* for recursive *rules* */

The clustering process discussed here involves a syntactic analysis of predicate and subgoal names to determine the dependencies of rules on subgoals. We restrict the analysis to simple subgoals or facts in the body/definition of the rules. Thus, the predicate names of other rules that appear in a rule's definition are transformed into "remote activations" in distributed settings. Therefore, it suffices to analyze only rules and facts for dependencies.

The result from the analysis is a set of k-ary tuples: [*predicate-name*, subgoal-name$_1$, . . . , subgoal-name$_k$], depicting the relations between the KB rules and facts. The cardinality of different k-ary tuples varies and depends on the number of independent facts in a rule's body. The tuples are then transformed into a binary matrix where the row indices are represented by the predicate names of the rules and column indices are represented by the subgoal names of the facts. Therefore, the matrix is described by a rule-fact dependency relation. The cell-entries of the matrix are 1s and 0s, where 1 indicates the existence of a dependency relation and 0 is the absence thereof. The matrix is then decomposed into submatrices where each submatrix is equivalent to a partition of the KB rules and/or facts.

1.2 Limitations

Our methodology requires each matrix model to be characterized by a single binary relation. Thus, it is necessary to construct a matrix for each conceivable binary relation that might be defined over the objects being modeled. Also, when the objects under discussion are weakly related, the matrices are significantly sparse and, as such, alternative decomposition schemes which are computationally efficient and less costly may be used. However, for large systems with several interrelated objects, our modeling strategy is cost effective and brings structure to bear on the systems.

The matrix-based approach is also best suited for modeling and partitioning knowledge bases which are static over a reasonable period of time. Thus, when minor and short-term additions and deletions (or dynamics) of rules or facts do not warrant repartitioning and reclustering, the approach is certainly cost effective. With dynamically-changing knowledge bases (like all conventional data bases), the rules or facts must be reclustered to reflect the new relationships. With our methods, dynamic repartitioning and reclustering is feasible and, during such a process, the changing portion(s) of the KB must remain frozen, or locked, from user access. This problem is not unique, however, with our approach. Several systems, e.g., operating systems and conventional data bases, require periodic garbage collection, backups, recovery, maintenance of dangling pointers and processes, and dynamic updates during system operations.

1.3 Summary

The rationale for using matrix models as a knowledge analysis tool is twofold. The first reason is a conse-

quence of earlier research in using matrix-model partitioning schemes to group entities based on common relational attributes [5]. The second reason is that, in general, matrices are relational structures which serve as useful tools for information processing. Other kinds of matrix-based tools have been used for knowledge analysis [6, 7].

For very large knowledge bases, two levels of clustering are necessary. First, the knowledge base is analyzed at an abstract level and reorganized into topics, areas, species, classes, or domains of interest (depending on the problem domain). Second, the objects in each group are analyzed and modeled as discussed in the following sections. Consequently, each group produces object clusters, and the entire KB is also structured as a network of group clusters. The clustering methods and algorithms discussed in the following sections are fully developed and implemented. Currently, we are applying the techniques to cluster and search large PROLOG KB and other kinds of knowledge-base data sets for performance measurements and evaluation. The experiment is being conducted on an 8-node T800 Transputer system.

2. KNOWLEDGE REPRESENTATION

In addition to logic-based models, knowledge bases are represented using models based on frames, object-orientedness, semantic networks, or entity relationships. The focus here is on the logic-based model and an example is presented to motivate the discussions. Figure 1 is a listing of logic-based KB information pertaining to the family data, health plan, rank, and retirement provisions for employees of a company. The KB also includes information about the impact of violating the company's policy.

The data is represented in PROLOG format and is made up of rules (or conditions) for maintaining certain status or benefits in the company. (It is assumed that the reader has an introductory knowledge of PROLOG as a programming language.) For example, to be an old employee, one must earn over a 50,000-unit salary. To receive pension benefits, an employee must be earning over a 50,000-unit salary at the time of retirement, be eligible for retirement (i.e., over 60 years old), and have never violated the company's policy. In addition, the KB includes a set of concrete data (factual information) about three of the employees, viz. John, Peter, and Tom. The factual data are used to check employees' eligibility status for pension and health options. Of significant corporate concern are rules and facts pertaining to the selection of a management planning team and participants in a decision-making process.

2.1 An Example of KB Data Clustering

In this section, we present a simple scheme for clustering PROLOG KB information. In later sections, a more elaborate implementation approach will demonstrate the applicability of the methods. Unlike structure-oriented models, e.g., frames and the object-based paradigms, logic-based models lack structural features for component clustering. However, a structure can be brought to bear on logic-based data by clustering the rules, facts, and queries.

For example, in Figure 1, the facts and queries may be partitioned into four clusters based on the bound arguments: "john," "peter," "tom," and the remaining facts which are not associated with the three arguments. Thus, the bound arguments form a basis for isolating the facts. On the other hand, the scope of the unbound arguments, e.g., T, X, and Y, is locally confined to the definitions (righthand side) of the individual rules. The unbound arguments, therefore, do not isolate the rules to aid in the construction of rule clusters. Nonetheless, a less efficient way to achieve sets of distributable clusters is to make four copies of the rules, in addition to the four clusters of the facts and queries. Consequently, the following combinations become distributable units of the data in Figure 1: the first copy of the rules and the cluster related to "john," the second copy and the cluster related to "peter," the third copy and the cluster related to "tom," and the fourth copy and the "other" cluster containing the remaining facts.

Although a redundancy is introduced by these combinations, the trade-off is a potential improvement on the efficiency of processing the KB data. The improvement is a result of simultaneously processing the "john," "peter," "tom," and "other" clusters on separate, multiple computers. In this example, these bound arguments are uniquely defined in each of the four clusters. Hence, minimal or no interprocessor communication overhead would be incurred with respect to the facts. (Assuming that the primitives *not*, *fail*, and *greater* are considered as subgoals and replicated in each cluster. The notorious *cut* (!) operator translates to a "remote activation," if present in a rule; see later sections.) Additionally, since each processor's problem space is reduced to a fraction of the entire KB data, the likelihood of minimizing the overall computational time increases.

2.2 Analysis and Modeling

Unfortunately, clustering KB data by simply grouping the facts and queries according to the bound arguments is inefficient and could lead to less optimal perfor-

RULES

```
family(X) :-                spouse(X,Z),
                            father(Y,X),
                            mother(Y,Z),
                            age(Y,A),
                            not(greater(A,21)).
/* children over 21 are not dependents */

senior_executive(T) :-      manager(T),
                            not(trainees(T)),
                            planning_team(T).

old_employee(X) :-          salary(X,Y),
                            greater(Y, 50K).

junior_executive(X) :-      not(old_employee(X)),
                            family(X),
                            college(X),
                            conferences(X),
                            not(fired(X)).
planning_team(K) :-         old_employee(K),
                            family(K),
                            college(K).

trainees(Z) :-              age(Z,L),
                            not(greater(L,30)),
                            ppc_insured(Z),
                            not(family(Z)),
                            not(old_employee(Z)).
family_health_plan(F) :-    family(F),
                            ppc_insured(F),
                            not(retirement(F)).

single_health_plan(S) :-    not(family_health_plan(S)).

medicaid_plan(M) :-         pension_support(M),
                            not(ppc_insured(M)).
ppc_insured(P) :-           has_health_plan(P, Y),
                            provider(Y),
                            patient_preferred(Y).
fired(Z) :-                 violate_policy(Z).
retirement(Y) :-            age(Y,X),
                            old_employee(Y),
                            greater(X,60).
pension_support(X) :-       retirement(X),
                            old_employee(X),
                            not(fired(X)).
```

FACTS

```
salary(peter, 35K).
age(peter, 39).
has_health_plan(tom, blue_cross)
spouse(peter, lurlyene).
father(kofi, peter).
spouse(john, sharon).
father(joshua, tom).
spouse(tom, lucy).
father(kwame, tom).
has_health_plan(john, hmo).
patient_preferred(blue_cross).
salary(john, 70K).
salary(tom, 30K).
age(tom, 55).
provider(blue_shield).
patient_preferred(blue_shield).
provider(hmo).
provider(blue_cross).
provider(blue_cross_blue_shield).
age(john, 65).
violate_policy(peter).
?- pension_support(tom).
?- pension_support(john).
?- planning_team(john).
?- junior_executive(peter).
?- family(john).
?- family_health_plan(tom).
mother(kofi,lurlyene).
mother(kwame,lucy).
mother(joshua,lucy).
mother(amma,sharon).
child(tom,joshua).
child(peter,frimpomaa).
```

Figure 1. Logic-based data representation.

mance. In addition, if the number of rules is significantly large, redundancy becomes a bottleneck in terms of modifications, additions, and deletions. A large degree of redundancy could also cause inconsistency problems in the KB data. Therefore, if the KB data is large, a formal, viable alternative must be employed to cluster the rules and subgoals. In our approach, the clusters are constructed by first analyzing the rules and

subgoals and then using the results of the analysis to structure the rules and subgoals as matrix models. The analysis involves determination of the dependency of the predicate names (rules) on the subgoals in the righthand-side definitions. We outline the procedure for analyzing the rules below.

2.2.1 Rules with only conjunctive subgoals. First of all, a set of unary tuples composed of the predicate names (head of the rules) is compiled. The predicate names of disjunctive rules are represented by aliases for the partitioning purposes. Each tuple is uniquely identified by one predicate name. The names of the facts or subgoals (subgoal names) to the right of the rules are appended to the tuples in the set to form k-ary tuples (k may vary). A matrix is then constructed such that the rows are labeled by only the predicate names in the tuples, because the rules depend on the subgoals. The columns are labeled by the subgoal names in the tuples.

Second, we employ the algorithm given in Figure 2 to structure the rules and subgoals as a matrix model. For example, in Figure 1, the predicate name *senior_executive* matches only the subgoal name *manager* in its definition. Since the predicate names *trainees* and *planning_team* represent other rules, they are not matched. The predicate name *family_health_plan* matches no subgoal names since *family*, *ppc_insured* and *retirement* are predicate names. The predicate names *family* matches all five of its subgoals: spouse, father, mother, age, and greater. Consequently, the following tuples (the predicate names and tuple elements—subgoal-names—represent the row and column indices of the matrix, respectively) are produced as the result of the analysis:

[*senior_executive*, {manager}]

[*family_health_plan*, { }]

[*family*, {spouse, father, mother, age, greater}].

2.2.2 Rules with both conjuctive and disjunctive subgoals. The algorithm in Figure 2 works for rules defined in terms of both conjunctive and disjunctive subgoals. The following discussion demonstrates that the clustering methods are invariant to rules with both conjunctive and disjunctive subgoals. To this end, we introduce three new rules (Figure 3) to the KB data in Figure 1. Figure 3 shows the two new rules, *family_health_plan* and *family*, which are alternatives (or disjunctive) to the rules in Figure 1. The new rules are aliased as *family_health_plan1* and *family1*, for the purposes of the analysis. The third rule, *exception*, is defined to satisfy the procedure call in the two new rules. An analysis of the new rules produces the following additional tuples:

[*family_health_plan1*,

{has_health_plan, violate_policy}]

[*exception*, { }]

[*family1*,

{spouse, age, greater, has_health_plan, child}].

Consequently, if these new rules were part of the KB data in Figure 1, the resultant matrix (Figure 4) will be augumented with a new column indexed by "child" and three new rows indexed by "family_health_plan1," "exception," and "family1." The new column and row indices are derived from the additional tuples.

In both cases, the resultant matrix is relational because of the dependency relation. The matrix is also binary because the entries are 1s and 0s. The matrix is then decomposed into submatrices, where each submatrix is equivalent to a partition of the KB rules and facts. Next, we discuss the specifics of the decomposition algorithm using the data in Figure 4 as a starting point.

3. DECOMPOSITION AND REORGANIZATION

In the following, we describe the algorithm for decomposing and reorganizing PROLOG KB data objects to 1) complement the analysis and modeling procedures already discussed, and 2) motivate its applicability. The procedure begins by computing the total number of nonzero entries in each row and column in the matrix. Consequently, a new matrix is constructed from the

```
match() {/* input: tuples/sets of predicate-names and subgoal-names (facts) and the KB data */
    for each predicate-nameI in a tuple {
        for each subgoal-nameJ {
            if the subgoal-nameJ (of some fact) is in the tuple or body of predicate-nameI
            then set Matrix[I,J]-entry to 1 else set Matrix[I,J]-entry to 0
        }
}
```

Figure 2. Algorithm for modeling KB rules and facts as a binary matrix.

```
family_health_plan(T) :-  exception(T,Y),            /* alternative to rule in Figure 1 */
                          (if Y = yes then           /* aliased as family_health_plan1 */
                             not(violate_policy(T)), /* disjunctive subgoal */
                             assert(has_health_plan(T, hmo))
                           else
                             medicaid_plan(T),
                             assert(has_health_plan(T, blue_cross))
                          ).

exception(T,X) :-         retirement(T), not(family(T)), X= yes. /* try both family rules */

family(M) :-              exception(M,E),            /* alternative to rule in Figure 1 */
                          (if E =yes then            /* aliased as family1  */
                             child(M,K), age(K,A),   /* disjunctive subgoal */
                             greater(A,21), not(spouse(M,V))
                           else
                             retract(has_health_plan(M,_)), !, fail
                          ). /* this rule allows single adults with older kids to be viewed as a
                             family */
```

Figure 3. Rules with conjunctive and disjunctive subgoals.

rearrangements. The size of the example used in motivating the specifics of the algorithm might not seem to warrant the trouble. The usefulness of the algorithm, however, is proven in situations where very large KB data has to be clustered for efficient, distributed processing.

3.1 The Decomposition Algorithm

Before the steps of our algorithm are discussed, an explanation of three characteristics or states that a binary matrix could assume during its decomposition is in order.

1. A row or a column may have no nonzero entries.
2. An entry **(I, J)** may be the only nonzero element in a row **I**, but the corresponding column **J** may have one or more nonzero entries. The converse is also true.
3. All rows and columns may have two or more nonzero entries.

3.1.1 Step 1: Algorithm. The first step deals with the rearrangement of rows and columns of the matrix in states 1 and 2. To achieve this rearrangement, the number of nonzero entries in each row and column is totaled (see the rows and columns labeled SUM in Figure 4). The indices of rows and columns with SUMs equal to zero (or successively reduced to zero) are moved to the bottom rows and righthand columns of the new matrix, respectively. The indices of rows and columns with SUMs equal to one are moved to the main diagonal of the new matrix, along with the lone entries. Once the entries are moved, the corresponding row and column indices are deleted from the old matrix.

The deletion could cause the nonzero entries of other rows and columns to be deleted as well (due to state 2). Therefore, the SUMs of the rows and columns affected by the deletion are adjusted or reduced by one on each occurrence. The first step is repeated for both rows and columns until the old matrix assumes state 3. If at the end of Step 1 each row and column is already marked (i.e., entry is selected), the algorithm skips to Step 3 (discussed later). Appendix A shows the pseudocode of Step 1. The matrix in Figure 4 is the input to Step 1. Figure 5, the revised matrix, shows the results of Step 1 where the marked, or selected, row and column entries are marked "+". Also, the deleted rows and columns (" = ") under the "deleted" labels and the successive adjustments of SUMs are indicated under the labels "sum-1." The "sum-1" entry of the column labeled "great" illustrates successive reductions of SUMs from 4 to 3 to 2.

3.1.2 Step 1: Example. Two sets of rule-fact pairs are selected from Figure 4 to form candidate, main diagonal partitions in Figure 5. The first set of rule-facts pairs comprising (senior_executive,manager), (junior_executive,conference), (planning_team,college), and (fired,violate_policy) is formed when Step 1 is applied to the SUMs of rows in Figure 4 (see Appendix A). The second set, comprising (ppc_insured,has_health_plan), (old_employee,salary), and

facts / rules	spou	fath	mo	age	great	mang	coll	conf	sal	ha-h	prov	pa-pf	viola	SUM	sum - 1	deleted
family	1	1	1	1	1									5		
snr-ex						1								1		
jnr-ex							1	1						2		
old-emp					1				1					2		
pln-team							1							1		
trainees				1	1									2		
fam-hlp														0		
sing-hlp														0		
medica														0		
ppc-ins										1	1	1		3		
fired													1	1		
retirem				1	1									2		
pension														0		
SUM	1	1	1	3	4	1	2	1	1	1	1	1	1			
sum-1																
deleted																

Figure 4. Initial matrix model.

(family,spouse) is formed when Step 1 is applied to the SUMs of columns in Figure 4 (see Appendix A).

The respective row and column indices are deleted or not considered further in Step 2, and the corresponding entries are rearranged on the main diagonal of the new matrix as shown in Figure 6. The undeleted rows and columns constitute the inner square submatrix, which is further partitioned in Step 2. In practice, the result manifests the strength of the dependency relations and the effect of Step 1 in grouping the pairs of rules and facts into partitions. Note also that Step 1 is conservative because it avoids the elimination of several other entries by first selecting single, nonzero entries in the rows and columns (see Appendix Prologue).

3.1.3 Step 2: Algorithm. The input to this step is the remaining rows and columns of the inner, boxed submatrix in Figure 6. The new SUMs of the remaining nonzero entries are the values under the labels "sum-1" or SUMs of Figure 5. In general, the second step requires the rearrangement of the remaining column and row indices with more than a single nonzero entry. The aim is to move additional nonzero entries onto the main diagonal in order to obtain a maximum number of 1s on the main diagonal—the criterion for maximal decomposition discussed in the Appendix. Each of the remaining nonzero entries which have not been previously marked is selected such that the aggregate sum of the corresponding row and column SUMs is the minima. Ties are broken arbitrarily. (By analogy, the aggregate sums represent the total number of other edges which would be eliminated from the equivalent bipartite graph if the target one is selected; see Appendix.) Hence, the entry that incurs the smallest loss is preferable.

Once an entry is selected, it is moved onto the main diagonal of the new matrix along with its row and column indices. The process is repeated until the new main diagonal is completely filled with the maximum possible entries. The step completes the process of establishing the decomposition criterion. Appendix B shows the pseudocode of Step 2.

3.1.4 Step 2: Example. The rows and columns, or (drows,dcols) pairs, that are selected in Step 2 are marked "+" (Figure 6). The marked entries are moved onto the main diagonal. The main diagonal entries serve as a pivot for transitioning from rows with non-main diagonal entries to other rows containing pivotal entries. Hence, the length of the main diagonal determines the efficiency of grouping the rules and facts into independent partitions.

Table 1 shows the data structures and results of Step 2. Triplet tuples are constructed from the entries of each row in the inner submatrix (step [3], Appendix B). A triplet is a tuple of the following structure: (row-index, column-index, aggregate-of-SUMs). Table 1 shows the TRIPLET tuples for each row of the inner subma-

facts / rules	spou	fath	mo	age	great	mang	coll	conf	sal	ha-hl	prov	pa-pf	viola	SUM	sum - 1	deleted
family	1+	1	1	1	1									5		=
snr-ex						1+								1		=
jnr-ex							1	1+						2	1	=
old-emp					1				1+					2		=
pln-team							1+							1		=
trainees				1	1									2		
fam-hlp														0		=
sing-hlp														0		=
medica														0		=
ppc-ins										1+	1	1		3		=
fired													1+	1		=
retirem				1	1									2		
pension														0		=
SUM	1	1	1	3	4	1	2	1	1	1	1	1	1			
sum-1		0	0	2	3 2		1				0	0				
deleted	=	=	=			=	=	=	=	=	=	=	=			

Figure 5. Effect of Step 1 on the initial matrix model.

trix. For example, the triplet (trainees, age, 4) is selected arbitrarily to represent the row indexed by *trainees* because both triplets under the row have equal values of aggregate-of-SUMs. The triplet (retirem,age,4), is also selected arbitrarily to represent the row indexed by *retirem* for the same reason (see step [5], Appendix B).

The MINSET column of Table 1 indicates the selected triplets for each row. Because the triplets of the rows indexed by *trainees* and *retirem* are tied in both the column-index, age, and the aggregate-of-SUMs values, one is chosen arbitrarily and the other triplet (i.e., for the row *retirem*) is replaced (steps [8] and [9], Appendix B). Therefore, the next triplet for the row *retirem* with the minimum aggregate-of-SUMs value, (retirem,great,4)**, is selected to replace (retirem, age,4) (step [8], Appendix B). In Table 1, the MAINDIAG column shows the final selections or (drow,dcols) pairs for forming the main diagonal of the inner submatrix (rearranged according to steps [10] [11], Appendix B).

Table 1. Resulting Data of Step 2

ROW	TRIPLET	MINSET	MAINDIAG
trainees	(trainees,age,4)	(trainees,age,4)	(trainees,age,4)
	(trainees,great,4)		(retirem,great,4)
retirem	(retirem,age,4)	(retirem,age,4)-replaced	
	(retirem,great,4)	(retirem,great,4)**	

3.1.5 Step 3: Algorithm. The last step involves moving nonzero entries outside the main diagonal of the new matrix to cluster around the main diagonal. We construct a transitive closure of these entries such that there is a transition from the rows containing the entries, through the main diagonal entries (pivots), to the rows of the pivotal entries. As the entries are moved into the transitive closure sets, the corresponding row and column indices are rearranged in the new matrix to produce groups of row and column indices that constitute the partitions. An optimal partitioning is indicated by having all partitions on the main diagonal. The transitioning steps given in Appendix C are applied to the nonzero entries not lying on the main diagonal of Figure 6. The arrows in the figure illustrate the transitioning process for the inner submatrix. Table 2 illustrates the data structures and associated values of the transitioning steps of Step 3.

3.1.6 Step 3: Example. In Table 2, the column labeled TRANSET lists the row indices that were copied from one TRANSET set into others. The ROWSET set is composed of other row indices that establish the transitivity (or closure) principle (step [5], Appendix C). Thus, the ROWSET column contains row indices

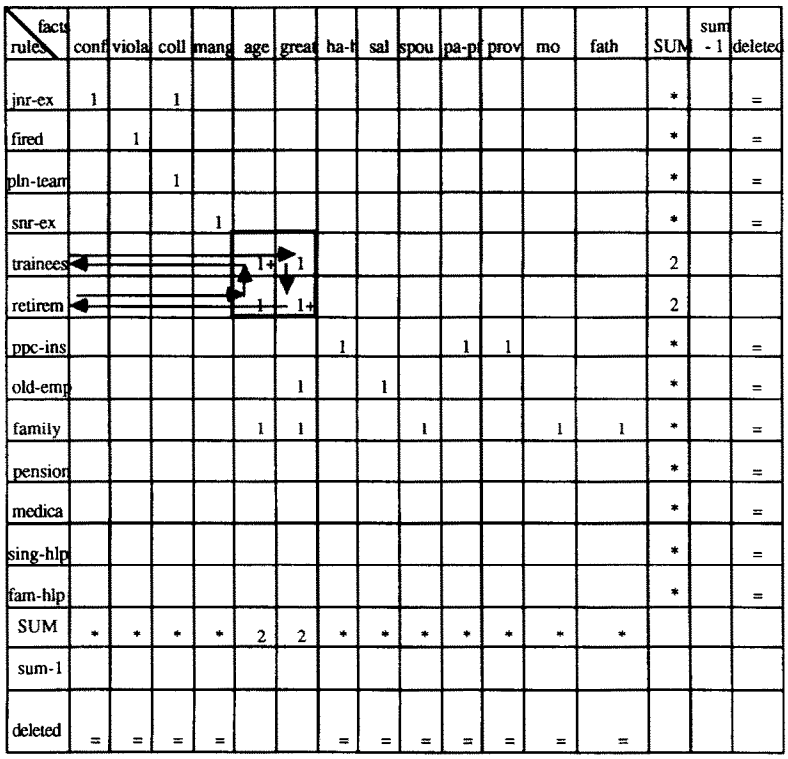

Figure 6. Result after Step 2 (with a boxed "inner" submatrix). *, SUMs not considered at Step 2.

Table 2. Transitioning Steps of Step 3 and the Resultant Partition

INITIAL	TRANSET	ROWSET	DENSITY	GROUPSET
Part (A)				
trainees	{trainees}*	{retirem}	2	{ }
retirem	{retirem}	{trainees}	2	{ }
Part (B)				
retirem	{retirem, trainees}	{trainees}	4	{retirem, trainees}

*(TRANSET entry(ies) of "trainees" in Part (A) copied to TRANSET of "retirem" in Part (B).

for establishing transitions from one TRANSET set to other TRANSET sets. For example, part A of Table 2 shows the row index *trainees* copied into the TRANSET set of *retirem* following the transitivity principle. The DENSITY values are 2 in both cases (the same values under the column labeled SUM in Figure 6). The TRANSET set {retirem,trainees} or *retirem* in Part B of Table 2 is added to the GROUPSET set as a group with a total DENSITY of 4.

In practice, Step 3 groups the row indices (KB rules) using the main diagonal entries as the pivot to determine the transitivity. The resultant partitions of GROUPSET set are used to finally rearrange the row and column indices of the matrix. The DENSITY is used (step [11], Appendix C) in a heuristic sense to place the partitions on the main diagonal. The DENSITY also facilitates merging the partitions into clusters. Figure 7 shows the partitions, indexed by the rules and facts, and blocked out on and below the main diagonal. The nonzero entries marked "1"" are the ones lying outside the partitions. These outlying entries are merged with the diagonal partitions to form clusters. The resultant clusters are fact independent after the merging procedure. Next, the methods for forming fact-independent clusters are discussed.

4. FORMING DISTRIBUTABLE CLUSTERS

The predicate and subgoal names indexing the three diagonal partitions of Figure 7 form the respective clusters C1, C2, and C3 in Figure 8. The cluster C4 is formed by clustering the four rules [*pension_support*, *medicaid*, *single_health_plan*, *family_health_plan*] lying below the three diagonal partitions and having no subgoal dependencies. The cluster C1 is formed by clustering the rules [*senior_executive*, *planning_team*, *fired*, *junior_executive*] and the subgoals managers, college, violate_policy, and conference.

The subgoal names indexing the nonzero entries lying outside the main diagonal partitions are placed in the clusters C2 and C3, respectively, because the subgoal names are either adjacent to or below the C2 and C3 partitions. For example, cluster C3 is formed by

facts / rules	mang	coll	viola	conf	great	age	ha-h	sal	spou	pa-pf	prov	mo	fath	SUM	sum - 1	deleted
snr-ex	1													*		
pln-team		1												*		
fired			1											*		
jnr-ex		1		1										*		
retirem					1	1								*		
trainees					1	1								*		
ppc-ins							1			1"	1"			*		
old-emp					1"			1						*		
family					1"				1			1"	1"	*		
pensior														*		
medica														*		
sing-hlp														*		
fam-hlp														*		
SUM	*	*	*	*	*	*	*	*	*	*	*	*	*			
sum-1																
deleted																

Figure 7. Result after Step 3: three partitions on the main diagonal. *, Not considered further; 1″, outlying nonzero entries.

clustering the rules [*ppc_insured*, *old_employee*, *family*] and the subgoals has_health_plan, salary, and spouse, plus the outlying subgoals patient_preferred, provider, mother, and father. Similarly, the subgoal name [*greater*] is added to C3 because rules *old_employee* and *family* depend on *greater* (see outlying entries in Figure 7 and shown in Figure 8).

In Figure 8, the subgoals are labeled FACT partitions and placed below the rules (labeled RULE partitions). The crossed-out lines indicate remote activations among the rules. The arrows indicate the dependency on the subgoals before merging the partitions (clustering). The bottom half of Figure 8 shows the four fact-independent clusters after merging the facts.

4.1 Distributed Query Execution Map

Figure 9 shows a cluster-load map for executing the five distinct queries in Figure 1. Each row in Figure 9 represents one of the rules in the KB. The query execution map is constructed at the time of loading the KB clusters (fragments) into the available processors for distributed computations. The entries of Figure 9 indicate which cluster contains a given rule. It is not relevant to indicate the facts which are needed for completing the execution of a rule since the partitioning and clustering procedures established this requirement a priori. Thus, no facts would be migrated during the execution of rules.

Although there are rule-rule dependencies (as indicated in Figure 8 by the crossed-out lines), it suffices to indicate in Figure 9 which cluster contains an originating rule, as shown under the column labeled "orig." Thus, knowing the originating rule, parallel remote activations are performed using a message-passing scheme or accesses to a shared memory. The queries in Figure 1 (marked "*" in Figure 9) are distributed for execution by a controller (program on a host computer). The controller uses the map to load clusters into available processors and initiates the necessary remote activations. In this example, three slave processors are loaded with separate clusters for parallel query execution. In the context of the techniques discussed thus far, the degree of parallelism is simply measured by the total number of overlapped executions among the processors. Hence, the degree of parallelism or overlapped executions, obtained using four processors in this example, is 3.75 (including the computations of the host computer).

5. CONCLUSION

We have presented a methodology for analyzing, structuring, partitioning, and clustering KB data for distributed processing. We have demonstrated the applicability of these strategies and discussed a mechanism for constructing non or minimally-interfering clusters of rules and facts. Noninterfering clusters suggest an op-

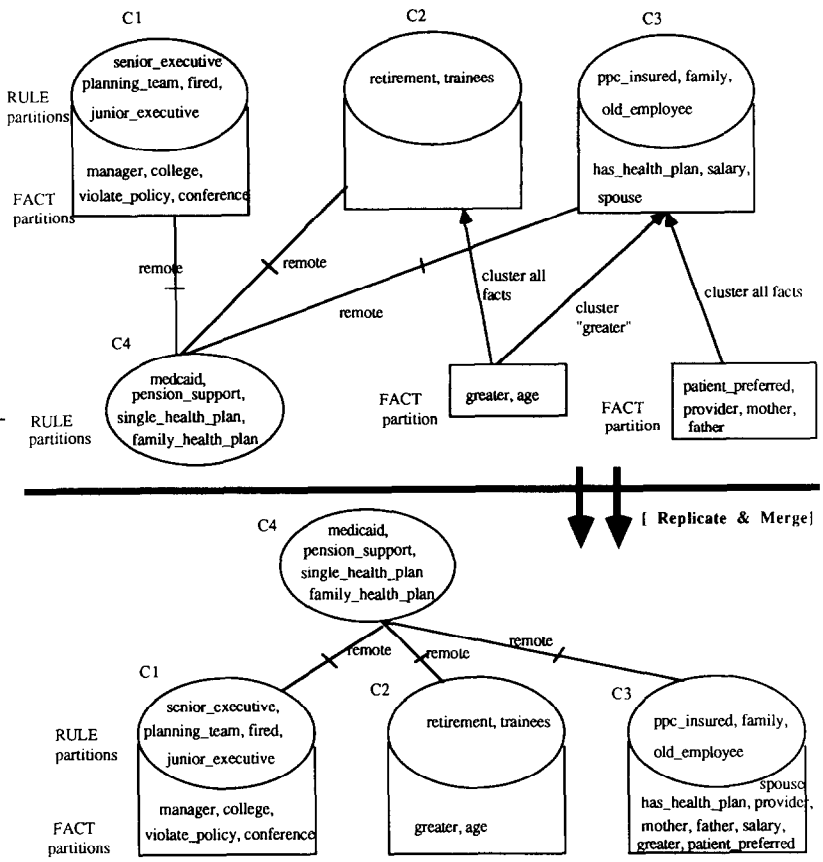

Figure 8. Clustering of rule and fact partitions.

process / rules	P1	P2	P3	host	orig	No. process
senior_executv					C1	
* planning_team	C1		C3	C4	C1	3
fired					C1	
* junior_executv	C1		C3	C4	C1	3
retirement					C2	
trainees					C2	
ppc_insured					C3	
old_employee					C3	
* family			C3	C4	C3	2
* pension_supp	C1	C2	C3	C4	C4	4
medicaid					C4	
single_health					C4	
* family_health		C2	C3	C4	C4	3
Degree of Par	-	-	-		-	3.75

Figure 9. Distributed query execution map (cluster-processor matrix). *, queries of Figure 1.

portunity for distributed processing with minimal or no interprocessor communication overhead. To this end, we have also discussed a method for constructing a query execution map for assigning clusters to available processors to facilitate distributed computations.

ACKNOWLEDGMENT

We thank Ross Huitt for his assistance in developing the PROLOG program-dependency analysis tool. We also thank the referees for their comments and suggestions on earlier drafts of this article. This work was partly supported by a grant from the National Science Foundation under the Research Experience for Undergraduates (REU) Program.

REFERENCES

1. L. Naish, Automatic Generation of Control for Logic Programs, Technical Report 83-6, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1983.
2. E. Y. Shapiro, A Subset of Concurrent PROLOG and its Interpreter, ICOT Technical Report TR-003, Institute for New Generation Computing, Tokyo, 1983.
3. G. Agha, *ACTORS: A Model for Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, MA, 1986.
4. J. S. Conery, *Parallel Execution of Logic Programs*, Kluwer Academic Pub., Norwell, MA, 1987.

5. P. O. Bobbie and J. E. Urban, A Model for Understanding Complexities of Developing Large-Scale Software Systems, *IEEE Computer Society International Workshop on Tools for AI*, Herndon, VA, 1989, pp. 16-23.
6. P. O. Bobbie, K. M. Ford, and E. G. Rodgers, Automated Elicitation of Software Requirements Specifications—An AI Approach, in *Advances in Artificial Intelligence Research* (M. B. Fishman, ed.), Vol. II, JAI Press, Greenwich, CT, 1990.
7. R. Braun, Expert System Tools FOR Knowledge Analysis, *AI EXPERT* 22-29 (1989).

APPENDICES: THE DECOMPOSITION ALGORITHM

Prologue

In theory, each nonzero entry of a binary matrix is equivalent to an edge in a bipartite graph [A1,A2]. A bipartite graph is a simple graph constructed from two sets of nodes in which nodes in one set are connected to nodes in the other set by edges (an onto-mapping). Several edges may emanate from or terminate at a single node. In practice, a bipartite graph is constructed from two sets of binary related elements, e.g., rules versus subgoals or rules versus rules. A nonzero entry in a binary matrix is an indication of a relation between two index elements, e.g., rule X depends on subgoal Y and will have the (X,Y)-entry set to 1. Isomorphically, the nonzero entry is represented by an edge between X and Y in a corresponding bipartite graph.

A bipartite graph may be matched [A1] using a systematic procedure to remove edges from the graph into a set (edge set) without having any pair of edges in the set coming from a common node. The ideal case is when the edge set contains the largest possible number of such edges, indicating a maximal matching. When an edge between two nodes is removed from the graph into the edge set, the two nodes are eliminated along with all other edges currently connected to the nodes. The additional disconnected edges are candidate edges that could be removed (if they were still connected) into the edge set to maximize the set's cardinality. Therefore, a prudent removal of an edge must be made to ensure maximal matching.

Maximum matching is a precondition for obtaining perfect matching. Isomorphically, the edges in the edge set represent the nonzero entries that are formed on the main diagonal of an equivalent binary matrix. Hence, the number of such nonzero entries must be as large as possible to guarantee an optimal decomposition or partitioning. Thus, the initial choice is to remove edges which do not eliminate several other potential edges or to start with lone nonzero entries in the rows and columns of the binary matrix (Appendix A).

The parallel between bipartite graphs and binary matrices is that the two are isomorphic. Thus, the two sets of nodes in the graph constitute the row and column indices of the matrix. The edges connecting the nodes in the graph represent the nonzero entries of the matrix.

The first two steps of the three-step decomposition algorithm are based on the matching principle of bipartite graphs [A1,A2]. The third step is an extension to the matching principle. A descriptive, pseudocode of the steps are presented in Appendices A to C with focus on the applications detailed in Sections 1 to 4. The matching principle establishes a maximum decomposition criterion (i.e., to obtain a maximum number of nonzero entries on the main diagonal).

There is a significant number of bipartite graph-matching algorithms with varying degrees of complexity. Specifically, our methods are based on maximum matching algorithms. Among the maximum matching algorithms are the Hungarian Trees algorithm which takes $O(|V||E|)$ steps overall, for V vertices and E edges (represented as 1s in our binary matrices); the Tutte Matrix Condition procedure which has a complexity of $O(|V|^3)$, where V is the cardinality of the vertex set; and the integer programming model, which suffers from NP-completeness [A2,A3]. The algorithm discussed in this paper has an overall worst-case running time of $O(|V_1||V_2|)$, where V_1 and V_2 are the cardinalities of the row and column index sets, respectively.

REFERENCES

A1. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
A2. J. McHugh, *Algorithmic Graph Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
A3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 1990, p. 1028.

Appendix A: Step 1

[1] compute SUMs as the sum of the non-zero entries of each row and column of the matrix **M**.
[2] **for** each row_I (or col_I) **do**
[3] **if** SUM of row_I (or col_I) = 0

```
[3.a]          then move row_I (or col_I) to bottommostrow (or rightmostcol) of M' (new)
[3.b]          else
[4]                 if SUM of row_I (or col_I) = 1
                    then
                        /*move M[row_I,col_J] (or M[row_J,col_I]) as follows:
                                J is the corresponding column or row */
                        move row_I (or col_I) to toprow (or rightcol) of M' (new)
                        move the corr. col_J (row_J) to leftcol (or bottomrow) of M'
                        delete row_I and col_J (or col_I and row_J) from M
                        subtract 1 from SUMs of all row_K (or col_K) where
                          M[row_K,col_J] = 1 (or M[row_J,col_K] = 1
   end /* the "or" parts of the steps are applied to SUMs of columns */
```

Appendix B: Step 2

```
[1] recompute the SUMs of rows and columns of the inner submatrix (see Figure 3c).
[2] for each row_I do
[3]       for each non-zero entry in col_J do
[3.1]           compute SUM-VALUE = SUM of row_I + SUM of col_J
[3.2]           form a TRIPLET set (row_I,col_J,SUM-VALUE) /* for all J */
          end
    end
[4] for each row_I do /* the number of triplets for each row_I = the number of its non-zeros */
[5]       for each TRIPLET set do
[6]           if a TRIPLET set is empty
[6.a]             then add row_I to bottommostrow and col_J to
                      rightmostcol of M' (new matrix)
[6.b]             else
                  select a triplet from TRIPLET set with the minimum SUM-VALUE
                      and add it to MINSET set -- ties are broken arbitrarily
                  delete the selected triplet from the TRIPLET
              endif
          end
    end
/* Form a MAINDIAG set of (row,column)-pairs from the candidate triplets of MINSET set
       in increasing value of the corresponding SUM-VALUEs */
[7] while MINSET is not empty do
[8]       if two or more triplets in MINSET have the same col_J
              then replace the triplets whose aggregate sum of the
                   SUM-VALUEs (in row_I's TRIPLET sets) is the
                   maximum, by re-executing steps [4]-[6.a] for the affected row_I's
          endif
[9]       if there are no ties in the SUM-VALUEs
[9.a]         then select a triplet with the minimum SUM-VALUE
[9.b]         else
                  select one arbitrarily
          endif
     end
[10] move the (row,column)-pairs from MAINDIAG set to the inner submatrix of M'
[11] move all non-zero entries from M to M' (the non-zero main-diagonal is formed).
```

Appendix C: Step 3

[1] consider only the *inner* submatrix (e.g., the boxed out submatrix in boldface in Figure 3c)
[2] form an INITIAL set of the row indices, a TRANSET set (initially of row indices) for each

```
    row_I, a ROWSET set (initially empty) for each row_I, and a GROUPSET set (initially empty).
[3] set a pointer1 to INITIAL, pointer2 to ROWSET set, and pointer 3 to the GROUPSET set
     /* Form ROWSET sets (composed of row indices) for each row such that */
[4] for each row_I in the inner submatrix do
[5]         for each non-zero entry in col_J (not on the main diagonal) do
                scan (up or down) the col_J to the main diagonal
                add the index of row_K containing the main diagonal entry to
                    ROWSET of row_I
            end
    end /* arrows in Figure 3c illustrate steps [4] and [5] */
[6] for each INITIAL set of row_I and the corresponding TRANSET and ROWSET sets do
[7]         for each row_I do /* determine a transitive closure */
[8]             if there is a match (intersection) between the ROWSET of row_I
                        and any TRANSET set of other row_K's
[8.a]                   then
[8.a.1]                     if there is a match (intersection) between the ROWSET set of
                                the matched row_K's and TRANSET set of row_I
[8.a.1.1]                   then
                                copy TRANSET set of row_I into TRANSET
                                    sets of the row_K's
                            endif
[8.b]                   else
                            copy TRANSET set of row_I into GROUPSET set
                            set pointer3 to next slot in GROUPSET set
                endif
[9]             delete TRANSET set of row_I
                set pointer1 to the next row_I (if not the last but one)
                set pointer2 to the next row_K (if any)
            end
    end
[10] for each group in GROUPSET set do
            compute DENSITY as the aggregate sum of the SUMs of the rows in the group
    end
[11] alternating between the groups with smallest and largest DENSITY values, relabel the
         row indices of the inner submatrix of M'
[12] rearrange the column indices of the inner submatrix to maintain the non-zero main
         diagonal entries of M' and move all other non-zero entries from M to M'
[13] block out the partitions on, below, or adjacent to the main diagonal of M'.
```