An Algebraic Deductive Database Managing a Mass of Rule Clauses

Tadashi OHMORI and Hidehiko TANAKA
University Of Tokyo, Information Engineering Course

Abstract

This paper proposes a deductive database which manages a mass of rule-clauses in a disk as well as many fact-clauses in a disk. For this purpose, we propose a variant of relational algebra, *Relational Algebra extended with Unification* (RAU). Our original point consists in query-processing by RAU in this DBMS. This paper describes a compilation of a query to a RAU's expression, optimization strategies for it, and an algorithm for a heavy RAU-operator.

1 Introduction

Conventional deductive databases concentrate on fast retrieval of many fact-clauses in a disk through a few rule-clauses in a main memory [KHT86,Boc86,YSI86].

Practical applications, however, will need a DBMS which manages a mass of rule-clauses stored in a disk as well as many fact-clauses in a disk.

This paper proposes such a deductive database. This DBMS, which we call a *rule-DBMS*, must retrieve rule-clauses fast from a disk, execute them fast, and retrieve fact-clauses fast from a disk.

Our approach is simple; A mass of rule-clauses are managed as a set called a *meta-relation*. Each query to a rule-DBMS is described as a tree form of set-operators defined on meta-relations.

Those set-operators, which we call *RAU*, are a variant of relational algebra for dealing with unification. *RBU-operations* in [YI86] is the first that extends relational algebra for unification. Our RAU is, however, aimed at a faster query-processing in a large scale rule-DBMS.

The rest of the paper is organized as follows; Section 2 illustrates a large database of rule-clauses, and describes requirements for a rule-DBMS. Section 3 gives our basic ideas and defines RAU. Section 4 describes a compilation of queries to RAU-expressions and optimization strategies for them. Section 5 gives an algorithm for one heavy RAU-operator. Lastly, Section 6 discusses open problems.

2 Rule-DBMS

2.1 Preparations

Deductive databases in this paper allow functor symbols as in [Zan85]. A rule-clause is abbreviated as a *rule* and a fact-clause as a *fact*. We use usual notations in PROLOG. We restrict each rule to a non-recursive view definition. Hence it can be compiled

```
% store(Media,Type,Data):- Cond.
 10^4  { store( imagedb(T), image(sub1,A), D ):- q1(T,A,D).
       { store( textdb(sub2(T)),
 rules {   :                 text(f(b,F),A) , D ):- q2(T,F,A,D).
           :
% key(Type,Keyword,User):- Cond.
 10^4  { key( text(F,h(A,b)), K, japan(ic,X) ):- p1(F,A,K,X).
       { key( image(S,g(A)), story(K),
 rules {   :            japan(pie,tokyo(X)) ):- p2(S,A,K,X).
           :
% query
q(T,D):- ruledb(kb1, key(T, story(aaa), japan(P,X)), C1),
         ruledb(kb2, store(M, T, D), C2 ),
         demo( to, (C1,C2) ).
```

Figure 1: an example of rule database

into an expression of a modified relational algebra such as ERA in [Zan85]. Recursive views are discussed later in Section 6.

In general, deductive databases have two databases; a database of facts (factDB) and the other one of rules (ruleDB). We assume that those two databases are so large that both of them are stored in a disk. Let's call this DBMS a *rule-DBMS.* A factDB is managed by a relational database system; facts with a common predicate are regarded as tuples of a usual relation.

We use two meta-predicates `ruledb` and `demo` in [Bow81] for managing a ruleDB. `ruledb( KB, Head, Body)` says "a knowledgebase `KB` knows a rule `Head:-Body` ". `demo(T,Goal)` says " a theory `T` proves `Goal` ". In this paper, we say that a head-predicate of a given rule is the rule's *kind*, and the rule *belongs to* or *expresses* its kind. e.g. `p(a,X):-q(X).` and `p(f(X),b):-r(X).` belong to one common *kind* p.

2.2 Large rule database

In practical applications, two cases enlarge a ruleDB; either there are many *kinds* of rules, or many rules belong to one *kind.*

The latter case corresponds to "many different implementations for a common interface". In an object oriented paradigm, it is the case that a superclass C requires a common interface p and allows each subclass of C to implement p independently. With 10^4 subclasses, 10^4 rules belong to a kind p. We expect this large scalability will arise naturally when developing a database of "a mass of heterogeneous data with common interfaces". Possibly, each rule may be implemented by a simple relational algebra program such as a single selection or at most one large size join.

Figure 1 is an example of the latter case. `store(Media, Type, Data)` is a common interface of a class `Media`. It says "a `Data` with a `Type` is stored in a `Media`.". In Figure 1, subclasses and attributes' values in `Media, User...` are expressed by compound terms. e.g. `User` has a structure of *nation(group, city(id)).*

Suppose that each rule expressing `store` operates different relations differently, depending on a subclass of `Media` (e.g. image-database subtype1,...), and a `Type` of `Data` (image, text, format,..). Then, with 10^2 subclasses of `Media` and 10^2 `Types` of `Data`, 10^4 rules express the kind `store`.

In the same way, `key( Type, Keyword, User)` is a common interface of a class `Type` of data. It says "a `Type` of data is indexed with a `Keyword` by a `User`.". Depending on 10^2 `Types` of data and 10^2 properties of a `User`, 10^4 different rules belong to `key`.

Most of queries are issued via only those common interfaces regardless of different implementations. e.g. a query $Q1$ is given as follows; "`q(T,D):- key(T, story(aaa), japan(P,X)), store(M, T, D).`".

Assume that only 100 combinations of rules expressing "`key` and `store`" are necessary for this query $Q1$, though the ruleDB has 10^4 rules for either "`key`" or "`store`". Then, we should decide those combinations of rules at first before executing rules, in order to restrict rules and reduce useless facts-retrieval.

The above situation will often happen when we have a mass of heterogeneous data with a few common interfaces. Because more constraints will arise in this case. e.g. *this subclass of media stores only this type of data by this group of users.* The query in Figure 1 describes those above operations of $Q1$ by meta-predicates. In the query, "`to`" is a object-theory in [Bow81]. "`(C1,C2)`" refers to "`C1` and `C2`". The query retrieves applicable combinations of rules at first, and execute them.

2.3 Requirements

It needs two points for fast execution of the query in Figure 1.

(a) fast retrieval of necessary combinations of rules. It is important especially when many rules belong to one kind. We think this function is the most important for a large ruleDB. (b) to avoid random accesses to a ruleDB in a disk when executing rules; e.g. Executing a rule `p:-q.` calls another rule `q:-r`, which may be stored in another page of a disk.

A limited solution of (b) is a partial compilation; transformation of each rule to simpler ones which operate a factDB directly [Miy86]. e.g. a rule `p:-q.` is transformed into 100 rules `p:-` r_1 , ... `p:-` r_{100} if we have `q:-` r_i ($i = 1,...100$). (Possibly each r_i may be a relational algebra program for the factDB.) This transformation makes more rules belong to one kind as shown above. Thus the requirement (a) gets more fundamental for a fast query-processing in a large scale rule-DBMS. Note that (a) and (b) are caused only because a ruleDB is so large.

Other requirements arise from a fast retrieval of facts through executing rules. (c) to select an appropriate evaluation-strategy or capture rule [Ull85] for reducing irrelevant facts. (d) common subexpression sharing [Sel86]. These two are also inherent problems for conventional deductive databases.

2.4 Conventional studies

Conventional deductive databases are based on a topdown or bottom-up approach [Ull85]. Recently, both prepare a graph of given rules in advance. In orthodox studies based on the topdown approach, a given query is transformed into a relational algebra program so-called *plan* and executed in a relational database [Boc86,YSI86], [Rei78,GM78]. Studies based on bottom-up pick up a partition of graph-of-rules corresponding to a given query, optimize the graph, and execute it by a relational algebra engine [KHT86,Zan85] or a data-flow framework [KL86,Gel86].

Both conventional studies are naive in terms of the requirement (a) only because they don't assume a large ruleDB. They process (a) by *one rule at a time* and don't process this large scalability efficiently.

In topdown-based ones, plans are generated by *selecting one rule at a time*. In bottom-up based ones, the picked-up graph is *optimized by one rule at a time* before/during execution. Therefore in the example of Section 2.3, it takes 10^8 processing to decide only 100 applicable combinations of rules for the query.

On the other hand, conventional studies satisfy the requirements (c), (d) to some extent. Hence it is hopeful to develop dexterous mechanisms for (a) which is consistent with conventional techniques for (c), (d).

3 Algebraic approach

3.1 Basic ideas

We propose an *algebraic* approach for a large rule-DBMS as follows;

At first, we compile in advance each rule into simpler rules which operate the factDB directly. They can be expressed by a variant of relational algebra such as ERA. The variant must be able to deal with functor symbols.

Second, we make a set of rules expressing one common kind. In Figure 2, $R[A,B,C]$ or $T[A,D]$ is a set of rules "`r(`A`,`B`):-` C`.`" or "`t(`A`):-` D`.`" respectively. We call these sets of rules *meta-relations*. e.g. In the figure, the scheme $T[A,D]$ of a meta-relation T is depicted over a horizontal bar, and a tuple $(X, q(X))$ is under it. Corresponding to the interpretation of the scheme $T[A,D]$, this tuple refers to a rule "`t(X):- q(X)`". In a large ruleDB, each meta-relation will have many tuples; e.g. 10^4 tuples.

Third, we prepare four set-operators $\overset{u}{\bowtie}, \sigma, \mathrm{I}, \pi$ on those sets as follows; Their examples are in Figure 2. (R, T are meta-relations and $A, B, C, D, \dots$ are attribute-names.)

- $R \overset{u}{\bowtie} T$ with a scheme $[A,B,C,D]$ is a set of rules "`q(`A`,`B`):-` C`,`D`.`". These rules are defined by "`q(`A`,`B`):- r(`A`,`B`),t(`A`).`". It expresses "`r` and `t`".
- $\sigma_{A \overset{u}{=} \mathtt{f(X)}} T[A,D]$ with a scheme $[A,D]$ retrieves a set of rules "`q(`A`) :-` D`.`". This kind "q" is defined by "`q(`A`):-t(`A`),`A $\overset{u}{=}$`f(X).`". It expresses `t` whose value in the attribute T[A] is restricted to be unified with "`f(X)`".
- $\mathrm{I}_D\, T[A,D]$ is a set of facts satisfying a rule expressing `t`.
- $\pi_{[A,C]} R[A,B,C]$ with a scheme $[A,C]$ is a set of rules "`q(`A`):-` C`.`" defined by "`q(`A`):- r(`A`,` B`).`". Those rules express `r` which is restricted to the attribute $R[A]$ excluding $R[B]$.

These operators are called *Relational Algebra extended with Unification* (RAU).

Fourth, we express queries to a rule-DBMS as tree forms of those operators. e.g. Let a query Q be "retrieve a set of facts satisfying both a rule expressing `r` restricted by F1 and a rule expressing `t` restricted by F2." It is expressed as follows; $Q = \pi \mathrm{I}[(\sigma_{\mathrm{F1}} R) \overset{u}{\bowtie} (\sigma_{\mathrm{F2}} T)]$. This query-tree describes what rules to be retrieved and when to execute them.

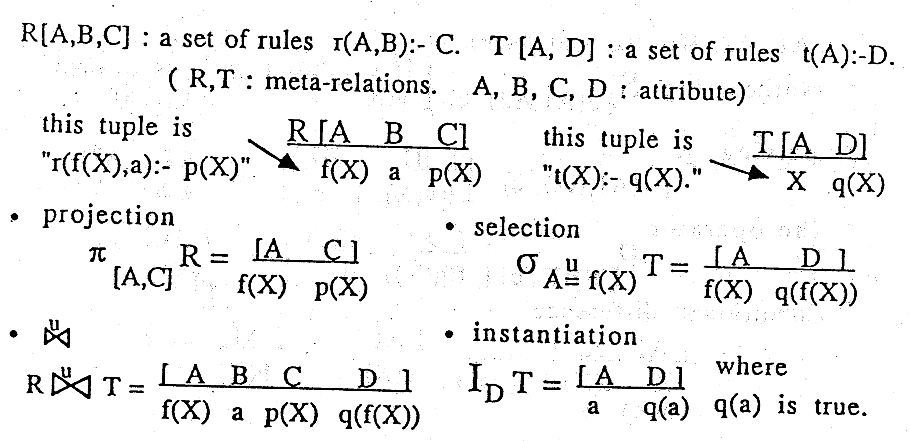

R[A,B,C] : a set of rules r(A,B):- C.
T [A, D] : a set of rules t(A):-D.
(R,T : meta-relations. A, B, C, D : attribute)
this tuple is "r(f(X),a):- p(X)"
R [A B C] / f(X) a p(X)
this tuple is "t(X):- q(X)."
T [A D] / X q(X)
• projection
$\pi_{[A,C]} R = \frac{[A \quad C]}{f(X) \quad p(X)}$
• selection
$\sigma_{A \overset{u}{=} f(X)} T = \frac{[A \quad D]}{f(X) \quad q(f(X))}$
• $\overset{u}{\bowtie}$
$R \overset{u}{\bowtie} T = \frac{[A \quad B \quad C \quad D]}{f(X) \quad a \quad p(X) \quad q(f(X))}$
• instantiation
$I_D T = \frac{[A \quad D]}{a \quad q(a)}$ where q(a) is true.

Figure 2: an example of meta-relations

Fifth, after optimizing the query-tree, we execute each operator, especially $\overset{u}{\bowtie}$ and I-operator by a fast set-operation algorithm. Note that I-operator execute a set of (modified) relational algebra programs on the factDB. We execute I-operator by a relational database system, after global query optimizations such as common subexpression sharings. □

Our approach satisfies the requirements (a), (b), and partly (c) and (d) in Section 2; (a) is satisfied by fast executions of RAU-operators, especially $\overset{u}{\bowtie}$ operator as described later in Section 5. So is (b) by exhaustive compilations. (c) is satisfied at a "macro level". i.e. a query including I-operator is transformed as follows; I($R \overset{u}{\bowtie} T$) = I((IR) $\overset{u}{\bowtie}$ T) = IR $\overset{u}{\bowtie}$ IT. They correspond to topdown, sideway, and bottom-up strategies at a "macro level" respectively.

(c) at a "micro level" refers to changes of strategies in executing each compiled form of rules. This function and (d) should be supported when implementing I-operator. Apparently I-operator is a function of conventional deductive databases.

Till Section 6, we concentrate on the requirement (a) in our approach.

3.2 Meta-relation

This section defines a *meta-relation* by using a meta-theory [Bow81].

Definition 1 Let p(arg1,.., arg$n$) be a predicate in a meta-theory T_m wrt an object-theory T_o. A *meta-relation rel(p)* is defined as follows.

- A scheme of *rel*(p) is [arg1,. . .,arg$n$], where each arg$i$ is an attribute. A domain of each attribute is a set of compound terms of T_m.
- A tuple satisfying the scheme is a function from each attribute to each domain in the scheme.
- A meta-relation *rel*(p) is a set of non-redundant tuples satisfying its scheme.

(A1, A2, B1, B2 : attribute)

- synthesizing $S_{[\ f(A1,A2),\ c]}\left[\begin{array}{cc}\text{A1} & \text{A2}\\ \hline g(X) & a\end{array}\right] = \begin{array}{cc}\text{B1} & \text{B2}\\ \hline f(g(X),a) & c\end{array}$
- parsing $P_{[\ f(A1,A2),\ c]}\left[\begin{array}{cc}\text{B1} & \text{B2}\\ \hline f(g(X),a) & c\end{array}\right] = \begin{array}{cc}\text{A1} & \text{A2}\\ \hline g(X) & a\end{array}$
- rho-operator $\rho_{[\ f(B1),\ c]}\left[\begin{array}{cc}\text{A1} & \text{A2}\\ \hline f(g(Y)) & c\end{array}\right] = \begin{array}{c}\text{B1}\\ \hline g(Y)\end{array}$
- Conditional difference

$$\begin{array}{cc}\text{A1} & \text{A2}\\ \hline X & f(X)\\ h(X) & a\end{array} \underset{A1}{\underline{}} \begin{array}{c}\text{A1}\\ \hline g(X)\end{array} = \begin{array}{cc}\text{A1} & \text{A2}\\ \hline h(X) & a\end{array}$$

Figure 3: an example of RAU-operators

Tuples t_1 and t_2 are redundant if $\exists\theta$: substitution; $t_1 = t_2\theta$. The scope of any variable symbol in each tuple is restricted within that tuple. Every tuple $[a_1, \ldots, a_n]$ in $rel(p)$ is mapped into $p(a_1, \ldots, a_n)$; an atomic formula of T_m. This mapping [tuple $\rightarrow$ wff] is defined by each meta-relation's scheme. □

Our meta-relation is almost the same as a *term-relation* in [YI86] except introducing a meta-theory. In Figure 2, a meta-relation $R[A, B, C]$ corresponds to a meta-predicate $R(A, B, C)$, which asserts that "there is an object-level wff $\mathtt{r}(A, B)$:- C." It is a set of object-level rules belonging to one common kind r.

In the rest of the paper, attributes are referred to by their names or positionID; e.g. an attribute B of $R[A, B, C]$ is referred to by $R[B]$, or $R[2]$, or only 2 if trivial.

Definition 2 given a tuple t and a meta-relation R,

- $t \in R \overset{\text{def}}{=} \exists t_1, \exists\theta_0 ; t = t_1\theta_0$, where
  θ_0 is a renaming substitution, and t_1 is a tuple and an element of R.
- $t \in_w R \overset{\text{def}}{=} \exists t_1$: tuple, $\exists\theta$: substitution ; $t_1 \in R$ and $t = t_1\theta$. □

e.g. In Figure 2, t_1 = [f(Y), a, p(Y)] $\in R$. t_2 = [f(c), a, p(c)] $\in_w R$, but $\notin R$. $t[i]$ is the i-th element of a tuple t; $t_1[3]$ = p(Y).

3.3 RAU operators

RAU-operators are defined below. (M, N, R refer to meta-relations. θ refers to a substitution. "$\overset{u}{=}$" refers to "be unified with". As logical connectives , {and, or, not, imply, equivalent} are expressed by { $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\equiv$}.) Figure 2 and 3 gives their examples.

Definition 3 The following operators are called *Relational Algebra extended with Unification* (RAU).

- ⟨ union ⟩ $M \cup N$ — same as that in relational algebra [Ull82], where the scheme of M is the same as that of N.
- ⟨ cartesian product ⟩ $M \times N$ — same as that in relational algebra. M and N share no common variable because of the scope restriction of their variables.
- ⟨ selection ⟩ $\sigma_{F(t)}\, M[A_1,\ldots,A_n] \overset{\text{def}}{=} \{\, t \mid t \in_w M \text{ and } F(t) \,\}$, where $F(t)$ is $\wedge_{i=1}^{n}(t[i] \overset{u}{=} C_i)$, abbreviated as $[A_1,\ldots,A_n] \overset{u}{=} [C_1,\ldots,C_n]$. C_i is a compound term allowing $t[j]$ $(i \neq j)$ as a variable symbol. The scope of variables in C_i is restricted within $F(t)$. e.g. $F(t)$ is $t[1] \overset{u}{=} f(t[2],X)$. (abbreviated as $A_1 \overset{u}{=} f(A_2,X)$).
- ⟨ synthesizing ⟩ $\mathrm{S}_{scheme}(M[A_1,\ldots,A_n]) \overset{\text{def}}{=} R[B_1,\ldots,B_m] =$ $\{t \mid \exists t_1; t_1 \in M, \text{ and } t = scheme(t_1)\}$, where $scheme = [B_1,\ldots,B_m]$, each B_i is a compound term. Its variable symbols must be $A_1,\ldots,A_n$. $scheme(t)$ is a compound term gained from $scheme$ by substituting $t[i]$ for A_i respectively.
- ⟨ parsing ⟩ $\mathrm{P}_{scheme}(R[B_1,\ldots,B_m]) \overset{\text{def}}{=} M[A_1,\ldots,A_n] =$ $\{\, t \mid \exists t_1; t_1 \in R, \text{ and } t_1 = scheme(t)\}$, where $scheme$ is the same as that in S-operator. $A_1,\ldots,A_n$ are those appeared in $scheme$.
- ⟨ instantiation ⟩ $\mathrm{I}_{cond} M[A_1,\ldots,A_n] \overset{\text{def}}{=} R[A_1,\ldots,A_n]$ $= \{\, t \mid \exists t_1, \exists\theta; t_1 \in M \text{ ,and } demo(T_o, cond(t_1)\theta) \text{ ,and } t = t_1\theta. \}$, such that $cond$ is a propositional wff whose propositional variables are $A_1,\ldots,A_n$. e.g. $A_1 \wedge A_2$. T_o is an object-theory in Definition 1. $cond(t_1)$ is a wff gained from $cond$ by substituting $t_1[i]$ for A_i respectively.
- ⟨ conditional difference ⟩
  $M[A_1,\ldots,A_n] -_{Alist} N[A_1,\ldots,A_m] \overset{\text{def}}{=} R[A_1,\ldots,A_n]$
  $= \{\, t \mid t \in M \text{ and } \forall s(s \in N \rightarrow \neg(t[Alist] \overset{u}{=} s[Alist]))\}$,
  where $Alist$ is a subset of $\{A_1,\ldots,A_n\}$.

Furthermore, abbreviations are defined.

- ⟨ projection ⟩ $\pi_{Alist} M[A_1,\ldots,A_n] \overset{\text{def}}{=} S_{Alist} M$, where $Alist$ is a subset of $\{A_1,\ldots,A_n\}$.
- ⟨ natural join extended with unification — $\overset{u}{\bowtie}$ ⟩
  $M[A,C] \overset{u}{\bowtie} N[B,C] \overset{\text{def}}{=} R[A,B,C] = \pi_{[A,B,C]}\, \sigma_{M[C] \overset{u}{=} N[C]} (M \times N)$.
  C is a list of common attributes of M and N.
- ⟨ *rho*-operator ⟩
  $\rho_{TERM}\, M[A_1,\ldots,A_n] \overset{\text{def}}{=} \pi_{Vlist}\, \mathrm{P}_{TERM}\, \sigma_{[A_1,\ldots,A_n] \overset{u}{=} TERM}\, M[A_1,\ldots,A_n]$,
  where $TERM$ be a list of compound terms. $Vlist$ is a list of distint variable symbols in $TERM$.

□

RAU-operators defined above can express those four operations in Section 3.1. Let's make a meta-relation $ruledb[K,H,B]$ by a meta predicate `ruledb(KB, Head, Body)` . Then the query in Figure 1 is described as a query tree $Q[T,D]$ as follows;

$$Q = \pi_{[T,D]}\ \mathrm{I}_{B1 \wedge B2}(\rho_{\ TERM1}\ ruledb[K,H,B1] \overset{u}{\bowtie} \rho_{\ TERM2}\ ruledb[K,H,B2]) \qquad (1)$$

such that $TERM1 = [kb1, key(T, story(aaa), japan(P,X)), B1]$ and
$TERM2 = [kb2, store(M,T,D), B2]$.

Among RAU-operators, I-operator is definitely unique to our RAU. S,P and ρ are almost the same as *combine, extended projection,* and *extended select/project* operators in ERA [Zan85] respectively. Hence compiled forms of rules are expressed by our RAU without unification. σ and $\overset{u}{\bowtie}$-operator are the same as *unification-restriction* and *unification-join* in RBU [MYNI86].

Our original points consist in the query-processing itself by RAU for a rule-DBMS; i.e. a RAU query tree is a PROLOG meta-interpreter with exhaustive compilation of rules to relational algebra programs. It expresses what combination of rules to be retrieved, when to execute them, and calls execution of relational algebra query trees to a factDB. We accelerate this processing by fast executions of RAU-operators after optimizing the query-tree.

4 Query processing by RAU

4.1 Compilation of query

A query "$q(D_1,\ldots,D_n)$:- $demo(kb_i,\ r_i(LIST_i))$." is given, where $LIST_i$ is a list of compound terms. If $LIST_i$ is $[f(X), a, X]$, $r1(LIST_i)$ refers to $r1(f(X), a, X)$. r_i is a name of either base relation or derived one in deductive databases. $D_1,\ldots,D_n$ are distinct variable symbols in $LIST_i$. kb_i is a subset of a object-theory.

Then this query is compiled to a RAU query-tree as follows;

$$R_i[D_1,\ldots,D_n] = \pi_{[D_1,\ldots,D_n]}\ \mathrm{I}_B\ \rho_{[kb_i,\, r_i(LISTi), B]} ruledb[K,H,B]$$

Note that R_i is ground because a base/derived relation r_i should be so. Apparently, the following theorem holds.

Theorem 1 A query "$q1(C_1,\ldots,C_m) : -w(D_1,\ldots,D_n)$" is given such that $C_1,\ldots,C_m$ is a subset of $D_1,\ldots,D_n$. Then, it is compiled into a RAU query tree if

1. $w(D_1,\ldots,D_n)$ is a wff composed by $\{\ \wedge, \neg, \exists\ \}$ of $demo(kb_i,\ r_i(LIST_i))$.

and

2. Let $R_i[V_1,\ldots,V_k]$ be a meta-relation expressing a query "$pi(V_1,\ldots,V_k) : -demo(kb_i,\ r_i(LIST_i))$." such that $V_1,\ldots V_k$ are variable symbols in $LISTi$. Then, $q1$ is a safe relational calculus expression by substituting a relational predicate $R_i(V_1,\ldots,V_k)$ for $demo(kb_i,\ r_i(LIST_i))$ in w respectively.

□

e.g. a query "$q(T,D)$: $-$ $demo(kb1, key(LIST1))$, $demo(kb2, store(LIST2))$." is given such that $LIST1 = [T, story(aaa), japan(P,X)]$ and $LIST2 = [M,T,D]$. This query is another form of that in Figure 1.

"$q(T,P,X) : -demo(kb1, key(LIST1))$" is compiled to R_1, and
"$q(M,T,D) : -\ demo(kb2, store(LIST2))$" is compiled to R_2 respectively as follows;

$$R_1[T,P,X] = \pi_{[T,P,X]}\ \mathrm{I}_B\ \rho_{[kb1,\,key(LIST1),B]}\ ruledb[K,H,B]. \qquad (2)$$

$$R_2[M,T,D] = \pi_{[M,T,D]}\ \mathrm{I}_B\ \rho_{[kb2,\,store(LIST2),B]}\ ruledb[K,H,B]. \qquad (3)$$

Apparently the given query $q(T,D)$ is a safe relational calculus expression "$q(T,D) :- R_1(T,P,X),\ R_2(M,T,D).$" . It is because R_1, R_2 are ground. The query is compiled into a relational algebra expression

$$Q[T,D] = \pi_{[T,D]}\ (R_1[T,P,X] \overset{u}{\bowtie} R_2[M,T,D]) \qquad (4)$$

($\overset{u}{\bowtie}$ is the same as usual join operation if it is restricted to a set of ground terms.) Q is just a RAU-expression if substituting (2), (3) for R_1, R_2 in (4).

This compilation generates a RAU-query tree having the following outline.

$$Q = IT_1 \overset{u}{\bowtie} \dots \overset{u}{\bowtie} IT_i -_{c1} IR_1 -_{c2} \dots -_{cn} IR_n$$

Their evaluation strategies are fixed but changed by commutative laws in the next section.

4.2 Commutative laws of RAU-operators

A given RAU-query tree is optimized by commutative laws for RAU-operators. Each law takes the form of *expression*1 $=_w$ *expression*2. "$=_w$" is defined as follows;

Definition 4 Given meta-relations M and N,

- $M \subset_w N \overset{\text{def}}{=} \forall t \in M; t \in_w N.$
- $M =_w N \overset{\text{def}}{=} M \subset_w N$ and $N \subset_w M.$

□

By Definition 1 , any meta-relation M expresses a wff $L(M)$; a conjunction of a universal closured wff mapped from each tuple in M. If meta-relations M and N have a common scheme and mapping [tuple $\rightarrow$ wff], $M =_w N$ implies $L(M) \equiv L(N)$.

Commutative laws hold as follows [Ohm87]; (M, N, R are meta-relations. 1,2,... and a, b, c are attributeID).

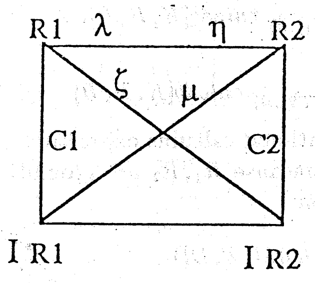
1. $\sigma_{p1\wedge p2}(M \times N) =_w \sigma_{p1\wedge p2}(\sigma_{p1}M \times \sigma_{p2}N)$,
   where $p1$ (or $p2$) is a selection-predicate about only attributes in M (or N).
   e.g.

$$\sigma_{1 \overset{u}{=} f(X) \wedge 2 \overset{u}{=} g(X)}(M[1] \times N[2]) =_w \sigma_{1 \overset{u}{=} f(X) \wedge 2 \overset{u}{=} g(X)}(\sigma_{1 \overset{u}{=} f(X)} M[1] \times \sigma_{2 \overset{u}{=} g(X)} N[2]).$$

2. $\mathrm{S}_{[p1,p2]}(M \times N) =_w \mathrm{S}_{p1}M \times \mathrm{S}_{p2}N$,
   where $p1$ (or $p2$) is a scheme consisting of only attributes in M (or N). e.g.

$$\mathrm{S}_{[f(1,2),g(3)]}(M[1,2] \times N[3,4]) =_w \mathrm{S}_{[f(1,2)]}M[1,2] \times \mathrm{S}_{[g(3)]}N[3,4].$$

3. $(M \overset{u}{\bowtie} N) \overset{u}{\bowtie} R =_w M \overset{u}{\bowtie} (N \overset{u}{\bowtie} R).$



$$\lambda = \frac{\text{number of R1's tuples generating tuples in R1 } \overset{u}{\bowtie} \text{ R2}}{\text{size of R1}}$$

ζ μ η are defined in the same way as λ

C1, C2 : execution cost for R1, R2

Figure 4: a query graph for $\mathrm{I}R_1 \overset{u}{\bowtie} \mathrm{I}R_2$

4. $\sigma_p \mathrm{I}_2 M[1,2] =_w \mathrm{I}_2 \sigma_p M[1,2]$,
   p is a selection predicate only about $M[1]$.

5. $\pi_1 \mathrm{I}_3 M[1,2,3] =_w \pi_1 \mathrm{I}_3 \pi_{1,3} M[1,2,3]$.

6. $\mathrm{I}_{a \wedge b}(M[a,c] \overset{u}{\bowtie} N[b,c]) =_w \mathrm{I}_b((\mathrm{I}_a M[a,c]) \overset{u}{\bowtie} N[b,c]) =_w \mathrm{I}_a M[a,c] \overset{u}{\bowtie} \mathrm{I}_b N[b,c]$.

7. $\mathrm{I}_b M[a,b] -_a \mathrm{I}_c N[a,c] =_w \mathrm{I}_b M[a,b] -_a \mathrm{I}_c((\pi_a \mathrm{I}_b M[a,b]) \overset{u}{\bowtie} N[a,c])$,
   where $\pi_a \mathrm{I}_b M$ must be ground.

4.3 Optimization strategies

The above laws 1 ~ 3 enable the same optimization strategies as those in *PSJ* query class in a relational database [Ull82]. The laws 4 and 5 restrict rules before execution; they reduce search-space when users give constraints interactively.

$$\text{e.g. } \pi_Y \mathrm{I}\,[p(X,Y) : -RAP] = \pi_Y \mathrm{I}\,[q(Y) : -\exists X(RAP)].$$

This $\exists X$ is π operator to relational algebra programs RAP.

The law 6 assures changing evaluation strategies for a query

$$q : - \wedge_{i=1}^{n}\, demo(kb_i,\, r_i(LIST_i)\,).$$

It is compiled into an expression $E = \mathrm{I}R_1 \overset{u}{\bowtie} \ldots \overset{u}{\bowtie} \mathrm{I}R_n$ (ρ, π are omitted.) . Each meta-relation R_i is a set of restricted rules expressing a kind "r_i". Each rule in R_i is a necessary one for evaluating $demo(kb_i,\, r_i(LIST_i)\,)$. E is optimized by changing I-operator's places and $\overset{u}{\bowtie}$'s sequence in it based on a "query graph".

Figure 4 illustrates a "query graph" for "$\mathrm{I}R_1 \overset{u}{\bowtie} \mathrm{I}R_2$". Each edge (n_1, n_2) except $(R_i, \mathrm{I}R_i)$ expresses a RAU-expression $n_1 \overset{u}{\bowtie} n_2$. Parameters on the edge is a success-ratio in unification. An edge $(R_i, \mathrm{I}R_i)$ is a RAU-expression $\mathrm{I}R_i$. A parameter on the edge is the cost of I-operators to R_i. Suppose that, when one edge in the query graph is marked, a RAU-expression corresponding to the edge is executed and the sum of execution-cost is incremented. Then , an optimization of I-operator and $\overset{u}{\bowtie}$ sequence is a problem to search the least-cost *terminated* sequence of edge-marking in the query graph.

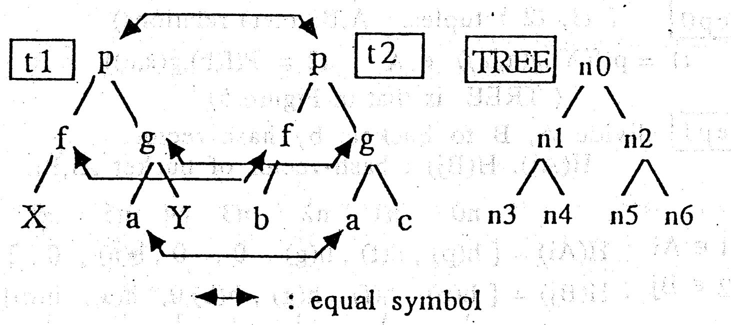


Figure 5: an example for unifiable atomic formulae

e.g. In Figure 4, a sequence $\langle(R_2,\mathrm{I}R_2),(R_1,\mathrm{I}R_2),(R_1,\mathrm{I}R_1)\rangle$ corresponds to a RAU-expression $\mathrm{I}(R_1 \overset{u}{\bowtie} (\mathrm{I}R_2))$. This sequence is called *terminated* because it executes all necessary operations for computing all tuples in $\mathrm{I}R_1 \overset{u}{\bowtie} \mathrm{I}R_2$.

This optimization-problem is NP-complete. We use a little modified version of Nearest Neighbourhood strategy for the optimization [Ohm87]. e.g. The query-tree (4) in Section 4.1 is transformed to the query-tree (1) in Section 3.3 by the laws 5, 6 and other trivial laws about π and $\overset{u}{\bowtie}$.

Lastly, the law 7 corresponds to *Negation as Failure* [Llo84].

5 An algorithm for a RAU-operator

Among RAU-operators, σ, S can be executed by a filtering processor. The heaviest operators are $\overset{u}{\bowtie}$ and I-operator. This section proposes an algorithm for $A \overset{u}{\bowtie} B$ such that A, B : metarelations. I-operator is discussed in the next section.

For simplicity, we assume that both A and B consist of one attribute. Moreover, they are sets of atomic formulae with one common $\langle predicate/arity\rangle$.

e.g. $A = \{p(f(X),g(a,Y)),\ p(a,h(X))\}$, $B = \{p(X,h(a)),\ p(f(b),g(a,c))\}$. Then, $A \overset{u}{\bowtie} B = \{p(f(b),g(a,c)),\ p(a,h(a))\}$. □

Our algorithm's strategy is explained at first.

(STRATEGY) Tuple t_1, t_2 are given. They are atomic formulae having a common predicate. Depending on the predicate's arity, we make a tree-structure TREE which has nodes $\{n_i\}_{(i=1,\ldots)}$. Figure 5 illustrates a TREE when the arity is 2. Next, we parse t_1 and t_2 based on the structure of TREE as in Figure 5. Let S_1, S_2 be respectively the symbols of the parsed tuples t_1 , t_2 on the node n_i of TREE. Then, the following assertion holds; (*) *if t_1 and t_2 are unifiable and neither S_1 nor S_2 is a variable symbol, then S_1 must be equal to S_2.*

Our algorithm filters out pairs of non-unifiable tuples by this assertion (*). Figure 6 is an example of the algorithm.

[$\overset{u}{\bowtie}$ algorithm]

step0 Set up a tree-structure TREE having nodes $\{n_i\}(i = 1, \ldots)$.

step1 For each tuple $t \in A$ or B, parse t on TREE into a sequence of symbols $\{S_i\}(i = 1, \ldots)$. Each S_i corresponds to a node n_i of TREE.

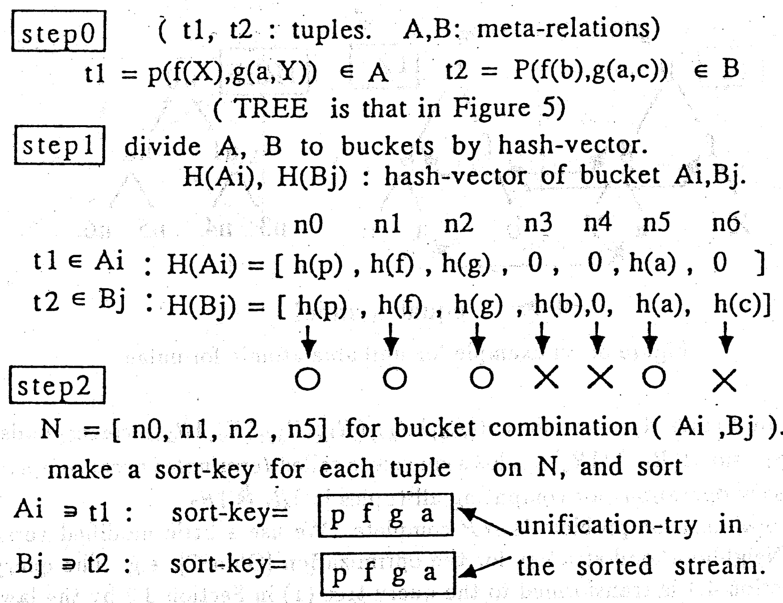

step0 (t1, t2 : tuples. A,B: meta-relations)
t1 = p(f(X),g(a,Y)) ∈ A t2 = P(f(b),g(a,c)) ∈ B
(TREE is that in Figure 5)
step1 divide A, B to buckets by hash-vector.
H(Ai), H(Bj) : hash-vector of bucket Ai,Bj.
n0 n1 n2 n3 n4 n5 n6
t1 ∈ Ai : H(Ai) = [h(p) , h(f) , h(g) , 0 , 0 , h(a) , 0]
t2 ∈ Bj : H(Bj) = [h(p) , h(f) , h(g) , h(b),0, h(a), h(c)]
○ ○ ○ × × ○ ×
step2
N = [n0, n1, n2 , n5] for bucket combination (Ai ,Bj).
make a sort-key for each tuple on N, and sort
Ai ∋ t1 : sort-key= p f g a
Bj ∋ t2 : sort-key= p f g a
unification-try in the sorted stream.

Figure 6: an example of a $\overset{u}{\bowtie}$-algorithm

Compute a vector-formed hash value $H(t) = [h(S_0), h(S_1), \ldots]$ for each tuple t where $h(\langle\text{variable}\rangle) = 0$, $h(\langle\text{non}-\text{variable}\rangle) \neq 0$, $h(S_i) = 0$ if S_i does not exist This H is called "hash-vector" of a tuple t.

Cluster A, B into buckets $\{A_i\}_{(i=1,\ldots)}$, $\{B_i\}_{(i=1,\ldots)}$ by each tuple's hash-vector H.

step2 For all bucket-combination (A_i, B_j), let $H(A_i) = [p_1, \ldots, p_n]$ and $H(B_j) = [q_1, \ldots, q_n]$ be the hash-vectors of the bucket A_i, B_j respectively.

If there is some k such that $p_k \neq 0$, $q_k \neq 0$, and $p_k \neq q_k$, then there is no pair of unifiable tuples in (A_i, B_j) by the assertion (*).

If not, make N $= \{n_k, \ldots\}$ of nodes in TREE such that $p_k = q_k \neq 0$ as in Figure 6. Mergesort all tuples in $A_i \cup B_j$. Each tuple's sort-key is a sub-sequence $[S_{k1}, S_{k2}, \ldots]$ of the tuple's parsed symbols such that each S_{ki} corresponds to n_{ki} in N. Try a unification of each pair of tuples (t_1, t_2) in the sorted stream where t_1 and t_2 have a equal sort-key.

□

TREE in step0 should consist of those nodes $\{n_i\}$ such that the symbol S_i of parsed tuples on n_i become non-variable symbols frequently over operand relations. Probably, those nodes generate many tuples in each relation because they take many different constant values. In Step2, the sort-key consists of those "tuple-generating" nodes, and filters out many pairs of non-unifiable tuples in $A_i \cup B_j$.

If TREE includes a node which tends to become variable symbols, sorting in step2 will not reduce unification-load. Therefore, it is important to set up an appropriate TREE in step0. For this purpose, we use statistic informations about distributions of tuples' values in meta-relations [Ohm87].

6 Discussion

This paper has proposed a deductive database by *Relational Algebra extended with Unification* (RAU) for managing both a mass of rules and many facts in a disk. We have given a compilation of query to a RAU query-tree, optimization for it, and an algorithm for a RAU-operator $\overset{u}{\bowtie}$.

In a RAU query-tree, retrieval of rules is accelerated by fast set-operation algorithms for RAU-operators, especially $\overset{u}{\bowtie}$. Retrieval of facts is a role of I-operator, including common subexpression sharing. In our experimental system, each rule is a simple relational algebra program such as a single selection or at most one large size join. I-operator is executed by usual relational algebra algorithms with naive common subexpression sharings.

Our approach has several open problems as follows;

1. implementation of I-operator by conventional deductive databases. In conventional fields, our RAU accelerates optimizing a "graph-of-rules" before/ during facts-retrieval. We think our system and conventional ones are supplements to each other.

2. recursive predicate. A recursive system is a strongly connected component in a "graph-of-rules" [CGL86]. It should be treated as a non-decomposable unit. Because our approach only needs a unit of compiled relational algebra program, current trends in this field are consistent with ours.

3. random access to a ruleDB. In general, each rule is transformed into simpler ones, but they don't always operate a factDB directly. In this case, a solution is a virtual memory for PROLOG machines though it may be difficult. A trivial one is a main-memory database.

Lastly, we must tell the difference between our RAU and *RBU* in [MYNI86]; *RBU* describes SLD-resolution in [Llo84] on a database machine. On the other hand, our DBMS processes both a large ruleDB and a large factDB through a common paradigm. i.e. RAU query-trees for the ruleDB and relational algebra query-trees for the factDB.

References

[Boc86] J. Bocca. On the Evaluation Strategy of EDUCE. In *Proc. of ACM-SIGMOD International Conference on Management of Data '86*, pages 368–378, 1986.

[Bow81] K. Bowen. *AMALGAMATING LANGUAGE AND METALANGUAGE IN LOGIC PROGRAMMING.* Technical Report, Syracuse University, June 1981.

[CGL86] S. Ceri, G. Gottlob, and L. Lavazza. Translation and optimization of logic queries: the algebraic approach. In *Proc. of the 12th Conference on Very Large Data Base*, pages 395–402, 1986.

[Gel86] A.V. Gelder. A Message Passing Framework for Logical Query Evaluation. In *Proc. of ACM-SIGMOD International Conference on Management of Data '86*, pages 155–165, 1986.

[GM78] H. Gallaire and J. Minker, editors. *LOGIC AND DATABASES*. Plenum press, 1978.

[KHT86] C. Kellog, A.O'. Hare, and L. Travis. Optimizing the Rule-Data Interface in a KBMS. In *Proc. of the 12th Conference on Very Large Data Base*, pages 42–51, 1986.

[KL86] M. Kifer and E.L. Lozinskii. A Framework for an Efficient Implementation of Deductive Databases. In *Proc. of the 6th Advanced Database Symposium*, pages 109–116, Information Processing Society of Japan, 1986.

[Llo84] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[Miy86] N. Miyazaki. *Compiling Horn Clause Queries in Deductive Databases: A Horn Clause Transformation Approach*. Technical Report 183, ICOT, 1986.

[MYNI86] Y. Morita, H. Yokota, K. Nishida, and I. Itoh. Retrieval-By-Unification Operation in a Relational Knowledge Base. In *Proc. of the 12th Conference on Very Large Data Base*, pages 52–59, 1986.

[Ohm87] T. Ohmori. *An Algebraic Approach to Deductive Database system for managing a large amount of procedural knowledge*. Master's thesis, The University of Tokyo, 1987.

[Rei78] R. Reiter. Deductive Question-Answering on Relational Data Bases. In [GM78], pages 149–177, 1978.

[Sel86] T.K. Sellis. GROBAL QUERY OPTIMIZATION. In *Proc. of ACM-SIGMOD International Conference on Management of Data '86*, pages 191–205, 1986.

[Ull82] J. Ullman. *Principles of Data Base Systems*. Computer Science Press, 1982.

[Ull85] J. Ullman. Implementation of Logical Query Languages for Databases. *ACM Transaction on Database Systems*, vol.10(No.3):pp.289–321, 1985.

[YI86] H. Yokota and H. Itoh. A Model and an Architecture for a Relational Knowledge Base. In *Proc. of the 13th International Symposium on Computer Architecture*, pages 2–9, 1986.

[YSI86] H. Yokota, K. Sakai, and H. Itoh. Deductive Database System based on Unit Resolution. In *Proc. of the 2nd International Conference on Data Engineering*, 1986.

[Zan85] C. Zaniolo. The Representation and Deductive Retrieval of Complex Objects. In *Proc. of the 11th Conference on Very Large Data Base*, pages 458–469, 1985.