1985

Using Expert Knowledge in Database-Oriented Problem Solving

Jiawei Han
University of Wisconsin

Larry Travis
University of Wisconsin

Using Expert Knowledge in Database-Oriented Problem Solving

Jiawei Han and Larry Travis
Computer Science Department
University of Wisconsin
Madison, Wisconsin 53706

Abstract

Database-oriented problem solving often involves the processing of deduction rules which may be recursive in relational database systems. In this kind of problem solving, expert knowledge plays an important role in the guidance of correct and efficient processing. This paper presents a modularized relational planner RELPLAN, which develops a knowledge-directed inference and planning mechanism for efficient processing of deduction rules in relational DB systems.

Introduction

Relational database technology provides us with a powerful tool for information processing. The conventional use of relational DB systems is aimed at the management and retrieval of *stored* data. With the emerging research on expert database systems, expert system technologies are being merged into relational database systems and the application domains of relational DB technology are being expanded to those that require knowledge-guided processing of *both stored and derived* data.

This paper studies the application of expert knowledge in DB-oriented problem solving. Problem solving is the process of developing a structure of (in the simplest case, a sequence of) actions to achieve a goal. Database-oriented problem solving is the problem solving involving large databases, in our discussion, large relational databases. As in many expert systems, DB-oriented problem solving is featured with deductive process. Our discussion is more concentrated on the deductive process which may involve recursive rules.

To efficiently implement such problem solving process in relational databases, two issues should be addressed. The first one is the transformation of recursion into iteration in relational databases. Two recent research papers &Hens 84é and &Ullm 85é deal with this problem from two different angles. &Ullm 85é develops query evaluation techniques based on "capture rules" on a graph representing clauses and predicates, while [Hens 84] presents us algorithms which compile queries involving recursive rules into iterative programs. The second issue is how to use expert knowledge and planning techniques to guide efficient processing of recursion or iteration in relational databases. This is the topic of this paper.

To effectively augment expert knowledge in relational DB system, a relational problem solving planner RELPLAN has been built. Similar to many expert systems, expert knowledge is encoded in RELPLAN in the form of rules and incorporated with queries in deductive compilation to answer queries and solve problems. In our design, the modularization of a rule system and the compilation technique are emphasized. A modularized rule system is built on top of a conventional relational DB system and RELPLAN uses these rules to transform user's deductive queries into non-deductive query programs, functioning as a deductive front-end of the relational DB system.

Complex DB-oriented problem solving requires *planning technique*, which develops a structure of query plans (programs) for a problem before actually solving it by DB operations. The planning technique implemented in our project is the means-ends analysis technique which develops hierarchical plans for complex queries based on the modification of the original deductive module. The planning process is divided into two phases: the selection of a planning strategy and the generation of the actual plan. The selection of planning strategy is based on the query provided by database user and the information stored in the database.

This paper first illustrates the architecture of the relational planner RELPLAN, then discusses the compilation of non-recursive and recursive queries using expert knowledge. A two-phase planning mechanism using plan modules is introduced and the efficiency of knowledge-directed inference and planning in database-oriented problem solving is also demonstrated in the paper.

The Architecture of the Relational Planner—RELPLAN

Here we present the architecture of RELPLAN, a problem solving planner for a relational DB system in Figure 1. The motivation of the development of such a relational planner is at introducing a modularized rule system and knowledge-guided problem solving to relational DB systems.

RELPLAN uses expert knowledge to transform user's deductive queries into non-deductive query programs. Expert knowledge is coded in the form of rules and entered into a rule base which consists of global rules and local rules. The global rules are available for all scopes of deductive queries, while local rules are confined in deductive and plan modules to incorporate the queries which reference these modules. The transformation of deductive query into non-deductive query program is based on the resolution principle and query modification techniques. The output query program of RELPLAN can be sent to query optimization routines to generate query access plans for database accessing.

The RELPLAN software is written in C language using YACC (a compiler-compiler) running under UNIX (VAX/11-750). The RELPLAN grammar is specified in appendix using the extended BNF grammar.

Like other relational database languages, RELPLAN contains data definition part and data manipulation (query) part. To ensure a high level query interface, RELPLAN query language is defined the same as conventional relational query language QUEL, except that the tuple variables that queries reference may also be deductive modules. The data definition part, where rules and modules are specified, is the major enhancement comparing with other relational languages. Rules are specified as virtual relations, search constraints and other stereotyped rules (such as *start*, *iteration*, *bound*, *etc*.) in deductive modules. The deductive module contains (i) the specification of local schemas (for temporary generic relations during problem solving process) and local rules, (ii) stereotyped rules, and (iii) an optional planning section which contains local specification and planning steps. Each planning step contains planning rules which append, delete or modify the corresponding stereotyped rules in the deductive module.

When processing a query which references a deductive module, RELPLAN uses the information provided in user's query and rules in the deductive module to decide which planning strategy should be adopted and what constraints should be augmented during problem solving process. The query is then resolved by using the knowledge provided in the rules base and/or deductive modules. A query program is thus generated with all virtual relations resolved and ready for further processing in relational database systems.

The Compilation of Non-Recursive Database Queries

Virtual relations are introduced in RELPLAN and the transformation of queries which involve virtual relations is based on the *deductive query compilation technique* developed in *logic and database* research [Reit 78] [Chan 81][Kell 81] and the *query modification techniques* developed in database research [Ston 75][Cham 75]. The compiled approach delays the database accessing in the deductive process until all intensional components (those reference to virtual relations) are resolved to the access of extensional database (which contains base relations only).

The following example shows the compilation of a non-recursive deductive query.

The transformation process can be divided into steps. (1) For each variable which references a virtual relation. e.g. "c" in Example 1., follow the query tree up to find *or node* or *where root*, as the rule augmentation point. (2) Substitute the variable by its rule definition, combine the query with the rule definition at the rule augmentation point to form a combined query tree, and rename the overlapped variable names if any, e.g. rename p1 to p hf in Example 1. (3) The merging, conflict removing and collapsing process can be performed on the combined query tree to simplify it. For example, in Example 1, the *medium* part in the *category* rule conflicts with *tall* uncle in the query and is thus collapsed. The same happens in the *female* part, which conflicts with the rule *brother* definition. Only the *male* subtree of the *tall* part in the *category* rule is augmented with the user's query. The collapsing technique is a kind of query optimization. (4) The above process is repeated for every virtual variable in the modified query until all virtual relation references are resolved. To concentrate our discussion on recursion, the details of non-recursive compilation algorithm are omitted here.

Knowledge Augmentation for Recursive Database Queries

Most database queries involving recursive rules can be transformed into iterative query programs using compilation techniques [Hens 84]. This paper discusses how to augment expert knowledge in compilation.

Example 1. Find Mary's tall uncle(s) who is (are) older than her father.

```
schema person ( name, age, sex, fa, mo, height)
range of p1, p2 ,p3 is person
define virtual relation b : brother(name = p1.name, bro = p2.name)
        where p1.fa = p2.fa and p1.mo = p2.mo and p2.sex = "male"
define virtual relation pa : parent(ch = p1.name, pr = p2.name)
        where p1.fa = p2.name or p1.mo = p2.name
define virtual relation u : uncle(name = p1.name, unc = p2.name)
        where p1.name = pa.ch and pa.pr = b.me and b.bro = p2.name
define virtual relation c : category(name , scale )[1]
        where c.name = p1.name and
        ((c.scale = "tall" and (p1.sex = "male" and p1.height > 6)
                or (p1.sex = "female" and p1.height > 5) )
        or (c.scale = "medium" and (p1.sex = "male" and p1.height <= 6 and p1.height > 5)
                or (p1.sex = "female" and p1.height <= 5 and p1.height > 4) ))
```

User's query:

```
range of x is uncle
retrieve (x.name, x.unc) where
        x.name = "mary" and x.unc = c.name and c.scale = "tall" and x.name = p1.name
        and x.unc = p3.name and p1.fa = p2.name and p2.age < p3.age
```

The resolved query by RELPLAN preprocessor:

```
range of p1 is person
range of p_hf is person
retrieve ( p1.name , p3.name )
        where p1.name = "mary" and p2.age < p3.age and p2.name = p1.fa
        and p3.height > 6 and ( p1.fa = p_hf.name or p1.mo = p_hf.name )
        and p_hf.fa = p3.fa and p_hf.mo = p3.mo and p3.sex = "male"
```

---

[1] In Prolog the "category" rule can be written as,

```
category ( Name, tall) :-
        person ( Name, _, Sex, _, _, Height),
        ((Sex = male, Height > 6); (Sex = female, Height > 5)).
category ( Name, medium) :-
        person ( Name, _, Sex, _, _, Height),
        ((Sex = male, Height > 5, Height =< 6);
        (Sex = female, Height > 4, Height < 5)).
```

```
                     Database Structure and Expert Knowledge
                     |------------------------------------------|
                     |       Schemas                            |
                     |       Rules                              |
User's               |       Modules                            |
Deductive    -->     |       ----------------------             |
Query                |       | Local Schemas      |             |
                     |       | Local Rules        |             |
                     |       | Stereotyped Rules  |             |
                     |       | Plans              |             |
                     |       ---------------------|             |
                     |------------------------------------------|
                                       |
                     |------------------------------------------|
                     |       RELPLAN Transformation             |
                     |------------------------------------------|
                                       |
                                       |
                          Non-Deductive Query Program
```

Figure 1

The architecture of the relational planner RELPLAN

SEARCH CONSTRAINTS AND EXPERT KNOWLEDGE

Combinatorial explosion is the major challenge in both AI and DB-oriented problem solving. Most AI problem solvers use various kinds of heuristics to reduce large search space. DB problem solvers search an even larger search space in general than AI problem solvers, due to the breadth-first search flavor of database operations in exploring all possible paths in the database. Obviously, the augmentation of search constraints is critical in DB-oriented problem solving. This can be seen from a practical example.

Example 2. The air-flight reservation problem: Suppose there are two relations *flight* and *airport* in the database. To schedule a flight from one airport to another *distant* airport, one step retrieval is generally inadequate and the problem solver must connect individual flights appropriately to form consecutive flights, which involves iteration or recursion on large data relations.

There are various kinds of constraints which could be augmented during the problem solving process. For example, the customer may require that the total flight time should be less than certain hours, the arrival time or the fare should be within some range, etc. The air-flight administrator may have some regulations such as that the interval of transfer should be within some range, etc. The travel agency may have some heuristic rules such as that each flight should be in the same direction as that from the initial departure to the final destination, etc.

In general, the more knowledge augmented, the more precise search. The key is how to incorporate with expert knowledge appropriately.

It is nontrivial in the augmentation of constraints for iterative processing of database queries. The first problem is termination problem for iterative process. In our example if there is no restrictive on connecting consecutive flights, new flight will be generated infinitely. (It could even fly around the globe many times!) A simple constraint such as the fare must be less than $1000 can terminate the iteration. Clearly, the upper bound of *fare* or *airtime* ensure the termination of the iterative process. This is the general case for recursive definitions which contain functions whose values increase/decrease mono-

Example 2

The new_flight (derived by connecting two consecutive flights) could be written in Prolog as,

```
new_flight (Flight_No, Departure, Arrival, Departure_Time, Arrival_Time, Fare) :-
        flight (Flight_No, Departure, Arrival, Departure_Time, Arrival_Time, Fare).

new_flight (Flight_No, Departure, Arrival, Departure_Time, Arrival_Time, Fare) :-
        new_flight (Flight_No1, Departure, Intermediate, Departure_Time, Arrival_Time, Fare1),
        flight (Flight_No2, Intermediate, Arrival, Departure_Time, Arrival_Time, Fare2),
        Fare is Fare1 + Fare2.
        Flight_No is Flight_No1 $ Flight_No2.
```
[2]

[2] $ is an operator which forms a virtual flightno Flight_No from Flight_No1 and Flight_No2.

iterative process, it is necessary to specify upper or lower bounds.

The second problem is the augmentation of a query constraint at each iteration. Some query constraint should be augmented at each iteration, while some should not. For example, if the fare that a user likes to pay from Madison to Tokyo is between $800 to $1000, the $1000 maximum fare must be augmented to terminate those flights with accumulated fares exceeding $1000. But the $800 minimum should not be augmented until at the *final* stage, otherwise most of the possible answers would be cut-off at early iterations.

The augmentation of constraints in DB problem solving needs expert knowledge. To automate the problem solving in deductive database system, expert knowledge should be entered and used appropriately. One approach is to classify and modularize knowledge to make different constraints play different roles in constraint augmentation and isolate the interaction of different rules. With the help of a modularized rule system, the expert may append, delete or modify rules and constraints easily and the system may have an exact control on the deductive process. That leads to the design of deductive modules in RELPLAN.

THE DEVELOPMENT OF DEDUCTIVE MODULES

The deductive module is a module that consists of a group of rules, schemas and queries which form a problem solving package. It modularizes the rule system and makes the problem solving process focus on a small group of rules, which does not reduce the search effort in rule invocation but also minimizes the interaction among different rules and goals.

The deductive module itself can be viewed as a virtual relation by a database user. The name of the module is the same as either an extensional or intensional relation. If it is of the same name of an extensional (base) relation, the module represents the *"closure"* of the data relation which can be generated by the operations specified inside the module. If it is of the same name of a virtual relation, it defines the procedures in the module that specify how to obtain the named virtual relation.

The reference rule of RELPLAN follows a scope rule which is similar to the scope rules in conventional programming languages. Rules, schemas and queries defined inside the module can be referenced and executed only by rules and queries inside the same module. User's queries which reference the module relation treat the module as a virtual relation. No individual rule or relation inside the module can be referenced by user's query. The rules outside the module cannot reference the rules inside the module. The rules inside the module can reference the rules in the global rule space but not those in other modules. A module variable can be declared and used inside the module which reference the entire visual (*module*) relation.

To facilitate the iterative rule processing, which is a major motivation of the design of a deductive module, a sequence of stereotyped rules are defined inside the deductive module. It specifies start condition, iteration,

Example 3. The deductive module *flight* for air-flight reservation.

Constraint and heuristic rules posed by air-flight manager are considered as query independent knowledge which should be specified inside the module[3], while user's queries are considered as dynamic requirements which should be specified outside.

```
schema  flight( fno, dpt, arr, dpttime, arrtime, fare)
        airport(port, lat, long, size)        /* lat : latitude, long : longitude */

module flight

schema new_flight( fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)

range of mf is module flight
range of f is flight
range of n is new_flight
range of p0, p1, p2, p3 is airport
define constraint s : same_direction(dpt1,arr1,dpt2,arr2) is
        (p0.lat - p1.lat)* (p2.lat - p3.lat) > 0 and (p0.long - p1.long)* (p2.long - p3.long) > 0
        where
        s.dpt1 = p0.port and s.arr1 = p1.port and s.dpt2 = p2.port and s.arr2 = p3.port


start -> retrieve into new_flight (f.fno, 0, f.fno, f.dpt, f.arr, f.dpttime, f.arrtime, f.fare)
                where f.dpt = mf.dpt

iteration ->      retrieve into
        new_flight (n.fno $ f.fno, n.fno, f.fno, n.dpt, f.arr, n.dpttime, f.arrtime, n.fare + f.fare)

constraint ->     n.arrtime + 3 > f.dpttime and n.arrtime + 1 < f.dpttime

constraint for iteration ->      same_direction(f.dpt, f.arr, mf.dpt, mf.arr)

upper bound -> (1) mf.fare       (2) mf.arrtime

end module
```

[3] Rules inside the module can also be modified, added or deleted by experts using primitives similar to plan rules. This feature is not hard to be added but simply ignored in our prototyped implementation.

final condition, search constraints and upper and/or lower bounds.

Example 3. The deductive module *flight* for air-flight reservation.

Constraint and heuristic rules posed by air-flight manager are considered as query independent knowledge which should be specified inside the module[3], while user's queries are considered as dynamic requirements which should be specified outside.

The module *flight* contains several different components: (1) a local generic relation *new flight*, (2) the specification of local rules, e.g. the constraint rule *same direction*, and (3) a sequence of stereotyped rules to specify initialization, iteration, final states, constraints and bounds.

The generic relation *new flight* is used to iteratively generate the consecutive flights during the problem solving process. The local rules such as *same direction* is used for performing inference inside the local module. The stereotyped rules includes (1) general constraints, e.g., the transfer time between two flights should be between 1 to 3 hours (*n.arrtime* + 3 > *f.dpttime and n.arrtime* +*1* < *f.dpttime*), and constraints for iteratively connnecting the consecutive flights, e.g. flying in the same direction as the initial departure and the final arrival posed in query, i.e., *same direction)f.dpt, f.arr, mf.dpt, mf.arr)* (2) initial state: which is the portion of the base relation flight which has the same initial departure as the user's query. (3) iteration rule: iteratively connecting the flights to obtain *new flight* where the departure of the tuples in *flight* is the same of the arrival of the tuples in generic relation *new flight*, and (4) bound rules: which are used for terminating the iteration and implicit control of constraint augmentation. In our case, we specify that there must be upper bound rule for fare or arrival time.

THE TRANSFORMATION OF QUERIES USING A DEDUCTIVE MODULE

The algorithm for module transformation can be derived by studying the air-flight reservation example.

Example 4. The transformation of the deductive query using deductive modules for air-flight reservation.

Suppose a user wants to book a ticket from Madison to Shanghai. The price range is asked between $800 and $1000 and the total travel time is required less than 30 hours. Could we list the suitable flights (departure time, arrival time and fare) for him?

In the output query program, the constraint rule *same direction* is resolved, the user's query is properly augmented and the program will terminate when no tuple is obtained in the generic relation *new flight*.

The transformation process proceeds according to the following algorithm.

Algorithm 1. The transformation of a deductive query using a deductive module.

(1) The selection of the deductive module.

A deductive module, viewed by database users as a virtual relation, is selected when user's query references the module relation. (e.g. Module *flight* is selected by query: *range of x is flight retrieve (x.dpttime,. . . .)where. . .).*

(2) Initialization:

(1) Retrieve the data that stored in the database. (2) Initialize the iterative process by augmenting start rule with the constraints and user's query appropriately. (The unbounded part of the user's query is not augmented at this stage, e.g. *x.fare* > *800*).

(3) Iteration:

If the iteration part is missing in the deductive module (possibly by deletion), the module is non-iterative and there is nothing generated in iteration part.

The iteration part will be enclosed in a *loop . . end loop* statement for iterative processing.

i) Take the iteration rule specified in the module as the center rule, where the new generic relation is generated by using base relations and the old generic relation with constraints appropriately augmented. The bounded part of the user's query is also augmented with iteration rules.

ii) The tuples in the new generic relation which meet the user query requirements are retrieved and deleted from the generic relation if they are not to be re-used in generating new tuples in the generic relation.

Example 4. The transformation of the deductive query using deductive modules for air-flight reservation.

Suppose a user wants to book a ticket from Madison to Shanghai. The price range is asked between $ 800 and $ 1000 and the total travel time is required less than 30 hours. Could we list the suitable flights (departure time, arrival time and fare) for him?

A database user may write it in QUEL,

```
range of x is flight
retrieve (x.dpttime, x.arrtime, x.fare)
where   x.dpt = "Madison" and x.arr = "Shanghai"
        and x.fare > 800 and x.fare < 1000 and x.arrtime - x.dpttime < 30
```

The resolved query program by RELPLAN preprocessor is as follows,

```
range of x is flight
range of n is new_flight

retrieve ( x.dpttime , x.arrtime , x.fare )
        where x.dpt = "Madison" and x.arr = "Shanghai" and x.fare > 800
         and x.fare < 1000 and x.arrtime - x.dpttime < 30

retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)
        where n_.fno = x.fno and n_.f1no = 0 and n_.f2no = x.fno
         and n_.dpt = x.dpt and n_.arr = x.arr and n_.dpttime = x.dpttime
         and n_.arrtime = x.arrtime and n_.fare = x.fare
         and x.dpt = "Madison" and n_.fare < 1000
         and n_.arrtime - n_.dpttime < 30

loop

range of p0 is airport
range of p1 is airport
range of p2 is airport
range of p3 is airport
retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)
        where n_.fno = n.fno * 1000 + x.fno and n_.f1no = n.fno
        and n_.f2no = x.fno and n_.dpt = n.dpt and n_.arr = x.arr
        and n_.dpttime = n.dpttime and n_.arrtime = x.arrtime
        and n_.fare = n.fare + x.fare and n.arrtime + 3 > x.dpttime
        and n.arrtime + 1 < x.dpttime and p0.port = x.dpt and p1.port = x.arr
        and p2.port = x.dpt and p3.port = x.arr and n_.fare < 1000
        and n_.arrtime - n_.dpttime < 30
        and( p0.lat - p1.lat ) * ( p2.lat - p3.lat ) > 0
        and( p0.long - p1.long ) * ( p2.long - p3.long ) > 0

retrieve ( n_.dpttime , n_.arrtime , n_.fare ) and delete  new_flight
        where n_.dpt = "Madison" and n_.arr = "Shanghai" and n_.fare > 800
         and n_.fare < 1000 and n_.arrtime - n_.dpttime < 30

exit when new_flight is empty

end loop
```

iii) The iterative process terminates when there is no new tuple could be generated in an iteration.

(4) Constraint augmentation:

Constraints are augmented according to the module specification. The specification includes *constraint for* certain kind of stereotyped rules (e.g. *iteration, start*) and a general *constraint.*

(5) Deduction rule transformation:

The deductive components (rules referenced in user query and the augmented rules in the deductive module) are resolved by using rule definition defined inside the module or in global rule space if there is no corresponding local rule definition.

Planning Using Expert Knowledge

Planning is the mechanism that develops a representation of a course of actions before acting in problem solving process. Planning mechanism is widely used in AI problem solvers [Sace 77][Nils 80]. For complex problem solving in expert database systems, planning technique will also be a necessity [Han 84]. This can be seen in the air-flight reservation problem.

In the air-flight reservation, if a traveller wants to fly from a small port to another remote small port, the experience suggests us to schedule the flights like this: first fly from the departure port to a neighboring big port, then fly in the direction to the final destination *via big ports only*. The final flight would be the flight from the big port which is close to destination directly to the final destination. Because most of the small ports are ignored in our search, the search effort will be reduced considerably.

This scheduling technique is resulted from one useful planning strategy: means-ends analysis [Barr 81], which compares the current goal with a current task domain to extract a difference between them and select a relevant operator to reduce the difference. The small-big-big-small flight planning is essentially a hierarchical problem solving process which avoids passing through tiny ports in scheduling a long distance travel.

There are at least two approaches in scheduling such a search process, a top-down and a bottom-up approach. In the top-down approach, we first find the appropriate consecutive flights from a big-port closed to the initial departure to a big-port closed to the final destination, then find the local flight to connect these ports. In the bottom-up approach, we first find a flight from the initial departure to its neighboring big port, then find flights from this port to the big port near the final destination, etc. Here we demonstrate the bottom-up approach in our planning.

TWO-PHASE PLANNING

There should be different planning strategies for different queries even in the air-flight reservation planner. If a user poses a query asking to book a local flight, say, from Madison to Chicago, the planner doesn't need to consider any hierarchical algorithms. If a user books a flight from New York to Tokyo, the planner should just consider the flight via big ports only. But if a user wants to book a cheap flight from Los Angeles to any cities in northern England, it is better to schedule both big and small ports in northern England. A travel agent can easily deal with such diverse queries because he has good knowledge on geography and airflights. For our poor planner, we even don't know which planning strategy should be considered before knowing the port information. Apparently, the planning process is hard to be completed in one phase and a two phase planning technique is suggested: First decide what kind of planning categories the query belongs to by retrieving more information from databases, then decide the scheduling process in detail.

In our two-phase planning process, the first phase is to retrieve port information into several small new relations according to user's query and determine the selection of planning strategies based on the results. Our new small relations are (1) *Local (dpt, arr)* which represents that the departure port is quite close to the arrival port and *only local schedule is needed*. All the others are non-local flight schedules. (2) *BigBig (dpt, arr)* which means that both the departure and arrival ports are big ports and scheduling flights *via big ports only* is the simple suggestion, (3) *BigSmall (dpt, arr)* which means that the departure port is a big one but the arrival is a small one. The planner will suggest to schedule *fly to a big port which is close to the destination via big ports only and then fly directly to the destination*. (4) *SmallBig (dpt, arr)* which flies from a small port to a distant big port, and (5) *Small-Small (dpt, arr)*, which flies from a small port to a distant small port.

Let's discuss the first phase of the two-phase planning. We first retrieve the airport information using user's query into five small relations: *Local*, *BigBig*, *BigSmall*, *SmallBig*, *SmallSmall*.
One example query is:

In general cases, the retrieval for port information will result in only a small number of tuples in one of the five

```
range of p1, p2 is airport
retrieve into s : SmallSmall(dpt = p1.port, arr = p2.port)
        where
                p1.port = mf.dpt and p2.port = mf.arr
                /* p1 and p2 are both small ports */
                and p1.size < 10 and p2.size < 10
                /* Two ports are located beyond local distance */
                and p1.lat - p2.lat > 5 and p1.lat - p2.lat > -5
```

relations. For example, a query asking for flights from Madison to Los Angeles will result in only one tuple in SmallBig relation.

The second phase of the planning will be the generation of a query program which processes the resulted tuple in the small relation. In RELPLAN syntax, we have "*for tuples in* variable: relname *do*. . . query program generation". For the empty relation, query program will not be generated and the corresponding planning strategy is ignored because it makes no sense to process on empty relations. This ensures the appropriate planning strategy selected based on diverse user query requirements and data in the database.

THE DEVELOPMENT OF PLAN MODULES

The plan module is a deductive module with a planning section augmented at the end of the module. The plan module is more complex than unplanned deductive modules. But with two phase planning, the generated query program will possibly be just slightly more complex than or the same as or even simpler then (e.g. in *Local* flight plan generation the iteration part is deleted) the unplanned process but result in efficient processing.

The two-phase planning splits the planning section of the module into two parts. The first part consists of a set of query statements which retrieve information for the selection of the planning strategy.

The second part of the planning section consists of one or several steps for each planning strategy. Each step in a planning strategy is a modification of some stereotyped rules of the original deductive module. For example, in SmallSmall planning strategy, the first step is to fly from the local port *directly* to the neighboring *big port*, which is written as,

```
append  constraint for start -> f.arr = p1.port and p1.size > 10    /* flying to a big port */
delete  iteration                                                   /* flying in one step */
```

Let's see the specification of a plan module.

Example 5. The plan module *flight*: only the case *SmallSmall of the planning section is demonstrated.*[4]

Ex 5. The plan module *flight*: only the case *SmallSmall* of the planning section is demonstrated.[4]

```
module flight
 ......
plan ->

schema SmallSmall(dpt,arr)
 ......

retrieve into SmallSmall(p1.port, p2.port)
        where . . .
 ......

for tuples in s : SmallSmall do
step 1:  /* First fly to a big port in one step. */
        append  constraint for start -> f.arr = p1.port and p1.size > 10
        delete  iteration

step 2:  /* Then fly via big ports only to a big port which is close to the destination. */
        append  constraint for iteration -> f.arr = p1.port and p1.size > 10
        replace final -> n.arr = p1.port and s.arr = p2.port and
                p1.lat - p2.lat < 5 and p1.lat - p2.lat > -5
                and p1.long - p2.long < 5 and p1.long - p2.long > -5

step 3:  /* Finally fly from that port directly to the destination. */
        delete  iteration

end for

end module
```

[4] To simplify our discussion, the other four cases *Local, BigBig, BigSmall, SmallBig* are not included here.

THE DEDUCTIVE QUERY TRANSFORMATION BASED ON PLAN MODULES

Because the specification of planning section is based on the modification of its deductive module, the transformation process is naturally based on the modification of the deductive module.

Algorithm 2. The transformation of deductive queries using planning techniques.

(1) Phase 1: The selection of planning strategies.

Data retrieval for the plan selection relations specified in planning section. Planning is prepared for each non-empty plan relation. For example, the query in Example 6 results in a non-empty plan relation *SmallSmall* because *Madison* and *Suzhou* are both small ports and located beyond local distance. The planning strategy for *SmallSmall* is selected.

(2) Phase 2: Plan generation:

Modify the module based on plan rules and generate the corresponding query program.

For each plan step do, append, delete or replace the original stereotyped rules by the rules specified in planning section. The modification forms a modified module and the generation of the queries based on the modified module follows Algorithm 1.

Because the initialization is needed only for the first step, the start rule in the deductive module is simply ignored in the rest steps. Because the final termination in the intermediate step (not the final step) does not generally follow user's query. A *final* part is usually added by rule *replace final* in the planning section and processed as final situation in the intermediate step transformation.

The algorithm is demonstrated by using the following example.

Example 6. Find flights from Madison to Suzhou (a small port in China) using planning technique.

User's query:

```
range of x is flight
retrieve (x.dpttime, x.arrtime, x.fare)
where   x.dpt = "Madison" and x.arr = "Suzhou"
        and x.fare > 800 and x.fare < 1000 and x.arrtime - x.dpttime < 30
```

The resolved query program of RELPLAN preprocessor:
/* The first phase of two-phase planning, *retrieve into SmallSmall, etc.* has been discussed in 5.1. Only the second phase, *plan generation,* is demonstrated here. */

```
range of x is flight
retrieve (x.dpttime, x.arrtime, x.fare)
where   x.dpt = "Madison" and x.arr = "Suzhou"
        and x.fare > 800 and x.fare < 1000 and x.arrtime - x.dpttime < 30


retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)
        where
        n_.fno = x.fno and n_.f1no = 0 and n_.f2no = x.fno
        and n_.dpt = x.dpt and n_.arr = x.arr and n_.dpttime = x.dpttime
        and n_.arrtime = x.arrtime and n_.fare = x.fare and x.dpt = "Madison"
        and x.arr = p1.port and p1.size > 10 and n_.fare < 1000
        and n_.arrtime - n_.dpttime < 30

loop

range of p1 is airport
range of p2 is airport
/* Collect the new_flights whose arrival ports are close to the destination */
retrieve into tmp_relation and delete new_flight
        where
        n.arr = p1.port and s.arr = p2.port
        and p1.lat - p2.lat < 5 and p1.lat - p2.lat > -5
        and p1.long - p2.long < 5 and p1.long - p2.long > -5

range of n is new_flight
range of p0 is airport
range of p3 is airport
/* Obtain new_flight by connecting the old new_flight with the flight which
meets the constraints. */
retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)
```

```
        where
        n_.fno = n.fno $ x.fno and n_.f1no = n.fno and n_.f2no = x.fno
        and n_.dpt = n.dpt and n_.arr = x.arr and n_.dpttime = n.dpttime
        and n_.arrtime = x.arrtime and n_.fare = n.fare + x.fare
        and n.arrtime + 3 > x.dpttime and n.arrtime + 1 < x.dpttime
        and p0.port = x.dpt and p_mg.port = x.arr
        and p2.port = x.dpt and p3.port = x.arr and x.arr = p1.port
        and p1.size > 10 and n_.fare < 1000 and n_.arrtime - n_.dpttime < 30
        and( p0.lat - p_mg.lat ) * ( p2.lat - p3.lat ) > 0
        and( p0.long - p_mg.long ) * ( p2.long - p3.long ) > 0

exit when new_flight is empty

end loop

range of n is tmp_relation

/* Flying from the port which is close to the destination directly to the final destination. */
retrieve into n_ : new_flight (fno, f1no, f2no, dpt, arr, dpttime, arrtime, fare)
        where
        n_.fno = n.fno $ f.fno and n_.f1no = n.fno and n_.f2no = f.fno
        and n_.dpt = n.dpt and n_.arr = f.arr and n_.dpttime = n.dpttime
        and n_.arrtime = f.arrtime and n_.fare = n.fare + f.fare
        and n.arrtime + 3 > f.dpttime and n.arrtime + 1 < f.dpttime
        and n_.fare < 1000 and n_.arrtime - n_.dpttime < 30

retrieve ( n_.dpttime , n_.arrtime , n_.fare )
        where
        n_.dpt = "Madison" and n_.arr = "Suzhou" and n_.fare > 800
        and n_.fare < 1000 and n_.arrtime - n_.dpttime < 30
```

The Processing Efficiency Using Search Constraints and Planning

The planning process generates a longer query program than the unplanned process. In general, people will wonder whether they will generate more efficient processing. Let's analyze the processing efficiency based on primitive calculations for the query in Example 6.

Suppose there are 100 k tuples with each taking 100 bytes in relation *flight* and 5 k tuples with each taking 100 bytes in relation *port* in the database. The total database size will be 10.5 megabytes which cannot be processed by main memory algoritms and database processing is the necessity. Suppose that the average cost of each flight to a local small port is $50.00 and from one big port to another is $150.00. The average propagation cycle (number of flight connections) via small ports will be 20 and via big ports only will be 7.

We divide the discussion into several cases:

(1) Bare iterative search without any restriction and with user's query processed at the end:

The process will never terminate because without restriction the flight connection with new flight will iteratively generate infinite large number of tuples in generic relation *new flight*.

(2) Iterative search with user's bound information augmented during iteration:

With bound information (e.g. maximum fare $1000) augmented, the iteration will terminate. Suppose for each step, the averaged selectivity is 1/1000. The first time around 100 tuples selected, the second iteration will generate 10000 tuples. The total number of tuples processed in 20 times could be: $100 + 10000 + \ldots + 100^{20} =$

$100 * (100^{20} - 1)/(100 - 1) = 10^{40}$, a number too huge to be processed in a reasonable amount of computing time.

(3) Iterative search with (2) and same-direction constraint augmented:

With same direction constraint augmented, the selectivity will be increased by 4 times, the total number of tuples processed could be: $25 + 25^2 + \ldots + 25^{20} = 10^{28}$, which is a significant reduce but still too large to be processed.

(4) Iterative search with (3) and transfer time constraint augmented:

With transfer time augmented, the selectivity will be increased by around 10 times, the total number of tuples processed could be: $2.5\ 2.5^2 + \ldots + 2.5^{20} = 1.6 10^8$, another significant reduce and it requires reasonable processing power.

(5) Iterative search with (4) and planning technique augmented:

With planning technique augmented, the search on small ports are limited only at the end of the search (in SmallSmall example), the average number of iterative search will be significantly reduced, suppose to be 8 in the example. Then the total number of tuples processed could be: $2.5 + 2.5^2 + \ldots + 2.5^8 = 2500$. It is a quite efficient algorithm. The planning technique contributes significantly because it reduces the average number of iterations from 20 to 8.

(6) More efficient execution strategies could be explored. For example, with the *big port* constraint (*pl.size > 10*) augmented in the start and iteration part of the plan module, the relation flight can be first restricted to the portion which contains big ports only, which will reduce the size of the relation to be iteratively executed. Some heuristics such as cost or time preference may also be augmented to cut-off the growing of the intermediate relations.

A more strict simulation model can be built for comparison of the performance of database execution. Our coarse estimation shows the order of magnitude difference on number of tuples processed, it is reasonable to expect that the more detailed simulation and performance testing will still be in favor of knowledge augmentation and planning techniques.

Conclusion

The database oriented problem solving often involves recursive or iterative processing of large data relations in relational database systems. To achieve efficient processing, besides the query processing strategy and optimization schemes in database technology, the most important factor is knowledge-directed inference and planning techniques.

This paper presents a prototyped relational planner RELPLAN, which develops an inference and planning mechanism for the augmentation of knowledge in processing recursive rules in relational DB systems. Using the example of air-flight reservation a knowledge-directed deduction and planning mechanism is presented for database oriented problem solving. It shows: (1) The modularization of a rule system benefits the appropriate augmentation of expert knowledge in deductive process; (2) Planning and constraining will significantly reduce search space and result in more efficient problem solving process; (3) The selection of planning strategy is based on user's query and the knowledge stored in expert database systems, which is the first-phase of the two-phase planning approach; the second phase, plan generation, is based on the planning rules defined in the planning section of the deductive modules; and (4) The deductive and plan modules will result in high level query interface with transparent underlying deductive and planning process.

This is just a preliminary step for the development of knowledge-directed deduction and planning in expert database systems. A variety of different planning mechanisms should be explored. Their effectiveness, limitations, relationship and differences comparing with planning mechanisms in AI research need to be explored in depth.

Acknowledgements: The authors wish to thank David DeWitt, Michael Carey and Charles Dyer for their helpful discussions and comments.

REFERENCES

[Barr 81] Barr, A. and Feigenbaum, E., *The Handbook of Artificial Intelligence (Vol. I-III)*, Heuristic Press William Kaufmann Inc., 1981

[Brod 84] M. Brodie, J. Mylopoulos, and J. Schmidt, *"On Conceptual Modeling"*, Spring-Verlag, 1984.

[Cham 75] D. Chamberlin, J. Gray and I. Traiger "Views, Authorization and Locking in a Relational Data Base System", *Proceedings of the National Computing Conference*, May 1975.

[Chan 81] C. Chang, "On the Evaluation of Queries Containing Derived Relations in a Relational Data Bases", In [Gall 81].

[Gall 78] H. Gallaire, and J. Minker, *Logic and Databases*, Plenum Press, New York, 1978.

[Gall 81] H. Gallaire, J. Minker, and J. Nicolas, *Advances in Data Base Theory, Volume 1*, Plenum Press, New York, 1981.

[Han 84] J. Han, "Planning in Expert Database System by Using Rules", *Proceedings of the First International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, October 1984.

[Hens 84] L. Henschen, and S. Naqvi, "On Compiling Queries in Recursive First-Order Databases", *Journal of ACM 31(1)*, 1984.

[Kell 81] C. Kellog, and L. Travis, "Reasoning with Data in a Deductively Augmented Data Management System", in [Gall 81].

[Kung 84] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro, and M. Stonebraker, "Heuristic Search in Data Base Systems", *Proceedings of the First International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, October 1984.

[Naqv 83] S. Naqvi and L. Henschen, "Synthesizing Least Fixed Point Queries into Non-Recursive Iterative Programs", *Proceedings International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, August 1983.

[Nils 80] Nilsson, N.J., *Principles of Artificial Intelligence*, Palo Alto, California, 1980.

[Reit 78] R. Reiter, "Deductive Question-Answering on Relational Databases", In [Gall 78].

[Reit 84] R. Reiter, "Towards a Logical Reconstruction of Relational Database Theory", In [Brod 84].

[Sace 77] E. Sacerdoti, *A Structure for Plans and Behavior*, American Elsevier, New York, 1977.

[Ston 75] M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification", *Proceedings 1975 ACM-SIGMOD Conference on Management of Data*, 1975.

[Ston 76] M. Stonebraker, E. Wong and P. Kreps, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems 1(3)*, September 1976.

[Ullm 82] J. D. Ullman, *Principles of Database Systems*, 2nd ed. Computer Science Press, Potomac, Maryland, 1982.

[Ullm 85] J. D. Ullman, "Implementation of Logical Query Languages for Databases", *Proceedings 1985 ACM-SIGMOD Conference on Management of Data*, Austin, Texas, 1985.

Acknowledgements: The authors wish to thank David DeWitt, Michael Carey and Charles Dyer for their helpful discussions and comments.

APPENDIX : THE SYNTACTIC SPECIFICATION OF RELPLAN[5]

<RELPLAN > : <Data_Definition> <Data_Manipulation>

<Data_Definition> : { <Data_Defl> } { <Module_Definition> }

<Data_Defl> : <Schema_Definition> | <Variable_Declaration> | <Rule_Definition>

<Schema_Definition> : *schema* { Rel_Name '(' Attr_Name {',' Attr_Name } ')' }

<Variable_Declaration> : *range of* Var_Name {',' Var_Name } *is* [*module*] Rel_Name

<Rule_Definition> : <Virtual_Relation_Defn> | <Search_Constraint_Defn>

<Virtual_Relation_Defn> : *define virtual relation* [Var_Name ':'] Rel_Name '(' <Attr_Reference>
{',' <Attr_Reference> } ')' [*where* <Qualification>]

<Search_Constraint_Defn> : *define constraint* [Var_Name ':'] Constraint_Name '('
<Attr_Reference> {',' <Attr_Reference> } ')' [*where* <Qualification>]

<Attr_Reference> : Attr_Name | Attr_Name '=' <Expression>

<Module_Definition> : *module* Module_Name <Module Body> *end module*

<Module Body> : { <Data_Defl> } { <Stereo_Typed_Rule_Defn>} [<Plan_Definition>]

<Stereo_Typed_Rule_Defn> : <Step> '=>' <Num_Query> { <Num_Query> }
| *constraint* [*for* <Step>] '=>' <Num_Clause> { <Num_Clause> }
| (*upper* | *lower*) *bound* '=>' <Num_Attribute>

<Step> : *start* | *iteration* | *final*

<Num_Attribute> : ['(' Integer ')'] <Attribute>

<Num_Query> : ['(' Integer ')'] <Query>

<Num_Clause> : ['(' Integer ')'] <Clause>

<Attribute> : Var_Name '.' Attr_Name

<Plan_Definition> : *plan* '=>' <Plan Prelude> <Plan_Steps>

<Plan Prelude> : <Data_Defl> <Data_Manipulation>

<Plan_Steps> : *for tuples in* Var_Name ':' Rel_Name *do* <Step> { <Step> } *end for*

<Step> : *step* Integer ':' <Modification> { <Modification> }

<Modification> : (*append* | *replace*) <Stereo_Typed_Rule_Defn>

| *delete* (<Step> | *constraint* [*for* <Step>] | (*upper* | *lower*) *bound*)

<Data_Manipulation> : { { <Variable_Declaration> } <Query> }

<Query> : *retrieve* <Target List1> *where* [<Qualification>]

| *retrieve into* Rel_Name <Target List2> *where* [<Qualification>]

<Target List1> : '(' <Attribute> {',' <Attribute> } ')'

<Target List2> : '(' <Expression> {',' <Expression> } ')'

<Qualification> : '(' <Qualification> ')'

| *not* <Qualification>

| <Qualification> (*and* | *or*) <Qualification>

| <Clause>

<Clause> : <Expression> <Relop> <Expression>

| Constraint_Name '(' <Attribute> {',' <Attribute> } ')'

<Relop> : = | != | < | <= | > | >=

<Expression> : <Term> (+ | -) <Term>

<Term> : <Factor> (* | /) <Factor>

<Factor> : <Attribute> | Constant | String

---

[5] { ... } denotes a set of zero or more occurrences, [...] denotes one or zero occurrences, and (.. | ..) denotes one of several occurrences.

APPENDIX : THE SYNTACTIC SPECIFICATION OF RELPLAN[5]

```
<RELPLAN >              : <Data_Definition> <Data_Manipulation>
<Data_Definition>       : { <Data_Defl > } { <Module_Definition> }
<Data_Defl >            : <Schema_Definition> | <Variable_Declaration> | <Rule_Definition>
<Schema_Definition>     : schema { Rel_Name '(' Attr_Name {',' Attr_Name } ')' }
<Variable_Declaration>  : range of Var_Name {',' Var_Name } is [module] Rel_Name
<Rule_Definition>       : <Virtual_Relation_Defn> | <Search_Constraint_Defn>
<Virtual_Relation_Defn> : define virtual relation [ Var_Name ':' ] Rel_Name '(' <Attr_Reference> {','
                          <Attr_Reference> } ')' [ where <Qualification> ]
<Search_Constraint_Defn> : define constraint [ Var_Name ':' ] Constraint_Name '(' <Attr_Reference>
                          {',' <Attr_Reference> } ')' [ where <Qualification>]
<Attr_Reference>        : Attr_Name | Attr_Name '=' <Expression>
<Module_Definition>     : module Module_Name <Module Body> end module
<Module_Body>           : { <Data_Defl > } { <Stereo_Typed_Rule_Defn>} [<Plan_Definition> ]
<Stereo_Typed_Rule_Defn> : <Step> '=>' <Num_Query> { <Num_Query> }
                        | constraint [for <Step> ] '=>' <Num_Clause> { <Num_Clause> }
                        | (upper | lower) bound '=>' <Num_Attribute>
<Step>                  : start | iteration | final
<Num_Attribute>         : [ '(' Integer ')' ] <Attribute>
<Num_Query>             : [ '(' Integer ')' ] <Query>
<Num_Clause>            : [ '(' Integer ')' ] <Clause>
<Attribute>             : Var_Name '.' Attr_Name
<Plan_Definition>       : plan '=>' <Plan_Prelude> <Plan_Steps>
<Plan_Prelude>          : <Data_Defl > <Data_Manipulation>
<Plan_Steps>            : for tuples in Var_Name ':' Rel_Name do <Step> { <Step> } end for
<Step>                  : step Integer ':' <Modification> { <Modification> }
<Modification>          : (append | replace) <Stereo_Typed_Rule_Defn>
                        | delete ( <Step> | constraint [for <Step> ] | (upper | lower) bound )
<Data_Manipulation>     : { { <Variable_Declaration> } <Query> }
<Query>                 : retrieve <Target List1 > where [<Qualification>]
                        | retrieve into Rel_Name <Target List2> where [<Qualification>]
<Target List1 >         : '(' <Attribute> {',' <Attribute> } ')'
<Target List2>          : '(' <Expression> {',' <Expression> } ')'
<Qualification>         : '(' <Qualification> ')'
                        | not <Qualification>
                        | <Qualification> (and | or) <Qualification>
                        | <Clause>
<Clause>                : <Expression> <Relop> <Expression>
                        | Constraint_Name '(' <Attribute> {',' <Attribute> } ')'
<Relop>                 : = | != | < | <= | > | >=
<Expression>            : <Term> ( + | - ) <Term>
<Term>                  : <Factor> ( * | / ) <Factor>
<Factor>                : <Attribute> | Constant | String
```

[5] { ... } denotes a set of zero or more occurrences, [...] denotes one or zero occurrences, and (.. | ..) denotes one of several occurrences.