RELATIONAL STORAGE AND EFFICIENT RETRIEVAL
OF RULES IN A DEDUCTIVE DBMS

Jean-Pierre Cheiney* and Christophe de Maindreville**

* Ecole Nationale Superieure des T&MC, 46 rue Barrault, 75013 Paris, France
** Institut National de la Recherche en Informatique et Automatique, Rocquencourt - BP 105 - 78153 Le Chesnay Cedex, France

electronic mail : cheiney@enst.ens.fr maindrev@inria.madonna.fr

In this paper, we address the problem of the storage and retrieval of rule programs in a deductive database system. In our proposal, when the rules are entered in the system, they are first analyzed and then compiled into a graphic execution model. The result of this compilation is finally stored in a relational database. When a query is submitted to the system, the first step is to retrieve from the database the rules able to produce results for the given query. Thus, efficient access to the rule base becomes an important issue for the performance of the system. The proposed rule base storage presents three main advantages: First, it allows integrated management of rules and facts in a deductive database system. Second, it allows rules to be stored in a pre-compiled form using the descriptive power of the graphic execution model. Third, a rule base is stored in a very compact form which is easily maintained. This last property permits efficient access to the relevant rules for solving a query. The relevant rules are accessed by traversing the graph model stored in the rule base. This traversal is performed by a join loop (transitive closure) over these relations. We present an algorithm using a specialized physical clustering to compute this retrieval. This algorithm can be easily extended to general transitive closure. The main feature of this algorithm is that input/output linearity requires only a relatively small main memory. A parallel version of the algorithm is also presented.

1. Introduction

One of the objectives for an integrated deductive database system is to offer relational storage and manipulation of the deductive rules. Thus, facts and rules can be stored in a unified way in the same system. When a query is submitted to the system, the first step is to retrieve the rules stored on disk in order to execute them. When the number of rules becomes large, this retrieval of rules becomes an important issue for the performances of the whole system. The aim of this retrieval consists in isolating the minimal set of rules which can produce all the results of a given query.

Several solutions have been proposed for the storage of deductive rules in a deductive database. In the ECRC prototype [BOCC86], the rules are stored as indexed files of texts. In contrast, our proposal adopts a relational storage for the rule program. The main advantage of such an approach is to provide a uniform way to manipulate rules and facts. The rules are manipulated through the same data manipulation language (DML) as that used to query the facts. For instance, the rule base can be queried and updated by a user through his usual DML. Thus, important database concepts such as physical and logical integrity, sharing and recovery are applied to the data representing the rules. Furthermore, the relational storage of rules will make it possible to benefit from the efficient techniques traditionally used in database management to access large amounts of data resident on disk.

In our proposal, the rules are compiled and stored into an original execution model called a PCN [MAIN88b]. This graphic model represents the interrelationships between the rules and the predicates used in them. The retrieval, for a given query, of a part of the stored rules consists in traversing the PCN to isolate a subset of the rules. This traversal is similar to that used in executing a transitive closure of a subset of a graph. In this paper, we propose an algorithm to reduce the amount of free main memory necessary for such a operation. The transitive closure operation uses an iterative algorithm where the clustering of the data ensures the linearity of each join, i.e., each page is read only once. This linearity is reached with weak conditions on the size of the main memory. The algorithm is based on a double hashing of the relation. The paper is organized as follows: Section 2 briefly presents the syntax of the RDL1 language which can be considered as an extension of logical languages such as Datalog®. Then, we present the PCN (Production Compilation Network) used to model and execute the RDL1 language. Section 3 presents the relational storage of the PCN in a particular rule base. In section 4, we present an algorithm based on a double hashing of the attributes participating in the join. This hashing is used to store on disk the relation storing the rules. A parallel version of the algorithm is also described. Section 5 presents some evaluations of the proposed algorithm and specifies the main memory size required for a single read execution for the join operation.

2. The rule language RDL1 and its execution model, the PCN

The purpose of this section is to give an overview of the rule language RDL1 and of its execution model called a PCN. We will concentrate on the static aspect of the language and of the PCN, since in this paper we only address the storage and the retrieval of rules in a Relational Deductive Database. Then, the dynamic aspect of the language and of the PCN, i.e. the semantics, will not be detailed. Let us recall here that the work presented in this paper can be integrated into any relational rule-based system.

An RDL1 production system (or an RDL1 program) is composed of a set of if-then rules called productions that make up the rule base, and a relational database called the fact database. The left-hand side of a production (corresponding to the if part of the rule) is a range-restricted expression of the tuple relational calculus [CODD71] [ULLM82], consisting of the conjunction of a range formula and a sub-formula. A range formula is a conjunction of positive (negative) range predicates
which indicate that a tuple variable ranges (does not range) over a relation. A sub-formula is a condition over the variables that appear in the range formula part. The right-hand side of a production (corresponding to the then part of the rule) is a set of actions. There are two elementary actions, denoted "+" and "-". The update action "+" takes a ground fact (i.e., a constant tuple) and maps a database state into another state which contains this fact. Thus, the action "+" inserts a tuple into a derived relation.

For instance, + Parent (sam, jeremie) inserts the tuple (sam, jeremie) into the Parent relation. On the contrary, the action "-" takes a fact and deletes it from a relation. In a parametrized action, variables that appear in the action part of a rule also appear in the condition part of the rule. Thus, the action "-" deletes a tuple from a derived relation.

The update action takes a ground fact (i.e., a constant tuple) and maps a database state into another state which contains this fact. Thus, the action "+" inserts a tuple into a derived relation.

For instance, + Parent (sam, jeremie) inserts the tuple (sam, jeremie) into the Parent relation. On the contrary, the action "-" takes a fact and deletes it from a relation. In a parametrized action, variables that appear in the action part of a rule also appear in the condition part of the rule. Thus, the action "-" deletes a tuple from a derived relation.

The update action takes a ground fact (i.e., a constant tuple) and maps a database state into another state which contains this fact. Thus, the action "+" inserts a tuple into a derived relation.

For instance, + Parent (sam, jeremie) inserts the tuple (sam, jeremie) into the Parent relation. On the contrary, the action "-" takes a fact and deletes it from a relation. In a parametrized action, variables that appear in the action part of a rule also appear in the condition part of the rule. Thus, the action "-" deletes a tuple from a derived relation.

In the following we will concentrate on the static (or structural) aspect of the PCN. The structural aspect of a PCN (which is the useful one in this paper) represents the relationships between rules and relational predicates as specified by a rule program.

The following associations can be made between rules and the PCN structure. We represent each rule by a transition and each relational predicate involved in the rule by a place. The relational predicates that occur in the condition part of a rule are input places to the transition representing the rule, and the relational predicates that occur in the action part of the rule are the output places of the transition. The condition itself of a rule is represented in the transition's inscription. More formally, we are able to set up an isomorphism between an RDLI program and a PCN structure which is summarized by the table below.

<table>
<thead>
<tr>
<th>RDLI</th>
<th>PCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>RULE</td>
<td>TRANSITION T</td>
</tr>
<tr>
<td>RELATIONAL PREDICATE P</td>
<td>PLACE P</td>
</tr>
<tr>
<td>Free tuple variables X1,...,Xn</td>
<td>ARC (P, T) labeled by: + X1 +...+Xn</td>
</tr>
<tr>
<td>Bound variables ranging over P</td>
<td>ARC (P,T) labeled by: ( )</td>
</tr>
<tr>
<td>Condition and negative range predicates</td>
<td>Transition's inscription</td>
</tr>
</tbody>
</table>

In this example, x and y denote tuple variables ranging respectively over the PARENT and ANCESTOR relations.
The second example is the PCN (Figure 2) associated with the RDL1 Rule Module WINGS. The base relations are PENGUINS, CROWS represented by the schema (Name, Color, Sex, Country, Predator).

```
MODULE : WINGS ;
TARGET : FLY (Name) ;
rules :
PENGUINS (x) → + BIRDS (Name = x.Name) ;
CROWS (x) → + BIRDS (Name = x.Name) ;
BIRDS (x) → + FLY (Name = x.Name) ;
PENGUINS (x) → - FLY (Name = x.Name) ;
en.
```

Figure 1: PCN for the Ancestor rules

The query expressed in the SQL language is:
```
SELECT Asc, Desc FROM ANCESTOR
where Desc = Georges;
```

Figure 2: PCN for the WINGS Module

One important feature of the PCN model is that it can represent in a uniform way rules and queries. A query is a production rule represented as a single transition, several input places (noted \([R_p]\) in the following) and a single output place representing the result of the query. Thus, a query can be represented as the combination of two PCNs. The first one represents all the rules needed to define the content of the input places of the query. The second net represents the query itself. The resulting PCN is called a query PCN. An example of a query PCN is given Figure 3.

Figure 3: Query PCN

The PCN provides a formal framework for describing general computation strategies and query optimization algorithms [MAIN88b] and for modeling in an unified way queries and updates in a deductive context.

When a query is submitted to the system, the first step consists in retrieving the pertinent PCN, i.e., the minimal PCN which is able to produce facts in the result place. Then the minimal Query PCN is built in main memory. The second and third states consist of an optimization and execution phase. These phases are detailed in [MAIN88a]. The purpose of the present paper is to detail the first phase.

3. Relational storage and retrieval of the relevant rules.

As seen in the previous section, a PCN can be used to model the interrelationship between predicates and rules. In the system we designed, the PCNs are first used as a way to compile and store rule programs in the database. Rules are stored in an internal form directly derived from the PCN representation. More precisely, when a rule module is defined, the form given by the user is saved in a file, and the rule module is compiled into a PCN. This net is added to the rule base by incrementally updating the PCN representing the rule base. Pre-compiled information, useful for query processing, is included in the rule base storage.

The proposed rule base storage presents three main advantages:

1. First, it allows integrated management of the rules and facts in a deductive database system. Thus, standard features of database systems such as physical and semantic integrity, recovery and sharing can be applied to the management of the rules. Moreover, rules can include predicates ranging over the relations that store the rules. Meta-knowledge can thus be easily integrated and handled in rule programs. For instance, the retrieval of the pertinent rules can be written in the rule language supported by the system, (in our case, the RDL1 language);
2. Second, it allows rules to be stored in a pre-compiled form using the descriptive power of the PCN model;
3. Third, a rule base is stored in a very compact form which is easily maintained. This last property permits efficient access to the relevant rules for solving a query.
The schema of the rule base consists of three relations. In the following, we describe the schema of the rule base. For the sake of simplicity, we only describe a restricted version of the schema of the relations. In particular, attributes describing pre-compiled information useful for query processing are intentionally dropped.

**Module** (#module, module_name)

**Place-Transition-Place** (#module, #preplace, #trans, input-label, output_label, #postplace)

**Transition** (#module, #trans, transition’s inscription)

The structure of the PCN is stored in the Place-Transition-Place (PTP) relation. The most costly part of the retrieval of the pertinent rules is the computation of the transitive closure of the PTP relation. This transitive closure represents the traversal of the PCN going from the queried places (Rp) towards the upper places of the PCN. To perform this operation more efficiently, we adopted a denormalized representation for the PTP relation [SCH81]. For the sake of simplicity we consider, in the rest of the paper (as the network representation), the following schema for the PTP relation:

PTP (#preplace, #trans, #postplace)

Denormalization leads to several classical drawbacks: update unconsistancies, supplementary storage cost, and non-transparency for the user. In our case, these drawbacks are not too important: the insertion of tuples in the PTP relation is only performed by a system procedure, the supplementary storage cost is reduced, and the end user does not access the PTP relation. On the other hand, this redundancy presents the following main advantage: the loop of joins is only performed on the PTP relation.

The retrieval of the relevant rules consists in computing a relational query over the rule base. Figure 4 presents the PCN corresponding to the query. The PTP relation, for Relevant PTP, is first built with the rule which produces tuples in the Query place. Its schema is similar to the PTP one, with the added attribute named “goal”, which keeps the number of the place for which there exists a path. Then, the loop of joins is performed over the PTP and the RPTP relations.

![Figure 4: PCN corresponding to the search for the relevant rules.](image)

Thus, the retrieval of the relevant rules consists of a unique query submitted to the system. It is performed by a transitive closure on a base relation which stores the rules. In order to have an efficient deductive system, this retrieval has to be optimized. It is the purpose of the rest of the paper to propose such an optimization. Note, that the algorithm we propose for computing the retrieval of the rules can be used to compute any transitive closure algorithm based on the loop of joins.

4. execution of the PCN corresponding to the search for the relevant rules

4.1 Principle

The PCN is executed through forward chaining by a sequence of relational algebraic operations. The most costly part of this computation consists of the join loop over the PTP relation. An important point to notice is that this loop of joins always takes a base relation as one of the arguments. The size of this relation can be very large and thus, no optimistic hypothesis can be made regarding the comparison between this size and the size of available main memory. The loop of joins will be performed by a semi-naive iterative algorithm [BANC86]. The major point we want to study is the limitation of Input/Output operations. In order to guarantee the linear aspect of the join operations, we want to reduce the size of the data which has to be in main memory at a given time. Our proposal consists in using a multi-attribute clustering method to read all the pages containing useful tuples at once in order to minimize the number of comparisons. To do this, we compute a double hashing of the PTP relation.

The PTP relation is first hashed into n buckets using a modulo function over the #preplace attribute. Then, each one of these buckets is hashed again by the same function applied on the #postplace attribute. To perform such a task, one can use a method based on predicate trees [GARD84] which guarantees a dynamic multi-attribute hashing. A semi-naive evaluation of this transitive closure is allowed since the transitive closure is linear. For each loop, new tuples are generated by the join between the PTP relation and the tuples produced by the previous iteration, i.e., in ∆PTP.

In contrast to other methods which use index techniques [STON86] [VALD87], our proposal uses directly physical clustering. The data are clustered by hashing techniques; this hashing is dynamic so that the rule base can grow and multi-attribute so as to have several partitioned views [CHEI86] of the relation which has to be joined.

4.2 Algorithm

For one iteration, two hashed relations have to be joined together. Both have been hashed by the same function. Thus, we use a hash-based join algorithm [KITS83], [DEWI84]. The join is performed between buckets having the same indices (buckets having different indices can not join together).

However, the hash-based algorithms cannot execute loops of joins. In fact, the result at one step has to be hashed on
a different attribute to constitute the buckets for the next step. Thus, if PTP is hashed according to the #postplace attribute and ΔTP according to the #preplace attribute, the join permits comparison of the only buckets for which (PTP.#postplace) modulo n = (ΔTP.#preplace) modulo n. But, in order to build the tuples of ΔTP useful for the next step, one has to hash the result according to the new #preplace attribute.

With our proposal, this rehashing is avoided. In fact, the tuples of the base relation PTP are hashed both on the #preplace and on the #postplace attributes. This relation can be seen as two different partitioned views [CHEI86]. The first one (hashed on #postplace) will be used with the join algorithm for reading the buckets having the same indices; The second one (hashed on #preplace) permits the preplace-hashed information for each tuple to be kept. This information is used to write new ΔTP buckets and to avoid a new hashing for the next step. Figure 6 portrays this aspect: it represents, for the iteration p, the join between the buckets of indices 3 of the PTP relation and the ΔTP relation which has been obtained at the previous step. For this step, the join between the buckets of indices 3 follows the join between the buckets of indices 0 to 2; it will be followed by the join between the buckets of indices 4. The tuples of the ΔTP relation which will be used at the next step are directly built without any rehashing.

4.2. Algorithm
We give in this section a more formal description of the algorithm. In order to simplify and generalize the algorithm, we denote the PTP relation by X and the relation which stores the tuples generated at one step by ΔX.

- X is the permanent relation having the schema (#preplace, #trans, #postplace)
- X is clustered on the two attributes #preplace and #postplace. The hashing function used for this clustering is the same for both attributes; it is a modulo n function. The X relation is recursively hashed into n^2 buckets. First, it is hashed in n buckets according to the #preplace attribute, and then, each one of these bucket is hashed into n buckets according to the #postplace attribute. The integer i represents the hash value for #preplace and j represents the hash value for #postplace.

\[ X = \bigcup_{i=0}^{n-1} X_{ij} \quad 0 \leq i \leq n-1, \quad 0 \leq j \leq n-1 \]

- ΔX is a temporary relation composed of n buckets named k. ΔX_k store the tuples from X having for k the hash value for the #preplace attribute.

\[ ΔX = \bigcup_{k=0}^{n-1} ΔX_k \quad 0 \leq k \leq n-1 \]

ΔX corresponds to one iteration of the construction of the RPTP (Relevant PTP) relation. The schema of this relation is (#preplace, #trans, #postplace).

The first phase of the algorithm consists in firing the initialization transition. It initializes RPTP with the tuples of PTP which satisfy the selection condition on the goal attribute. This corresponds to pushing up the constant given in the query. Since the PTP relation is clustered on the postplace attribute, only useful pages are read to perform this operation. The second phase of the algorithm consists of the loop of joins which perform the recursive transition of the PCN.

\[ \text{procedure } \text{relevant}(\text{PTP}); \]
\[ \text{begin} \]
\[ \quad \text{computation of } [1] = \text{values of } h(x.#postplace= \text{set of the deduced relations appearing in the query}); \]
\[ \quad \text{for each value of } l \text{ do} \]
\[ \quad \quad \text{from } k=0 \text{ to } n-1 \text{ do} \]
\[ \quad \quad \quad \text{begin} \]
\[ \quad \quad \quad \quad \text{read } X_{kl}; \]
\[ \quad \quad \quad \quad \text{write } ΔX_k:= ΔX_k + X_{kl}; \]
\[ \quad \quad \quad \text{end}; \]
\[ \quad \text{end}; \]
\[ \quad \text{[ loop of joins ]} \]
\[ \quad \text{RPTP} := ΔX; \]
\[ \quad \text{while } \text{AX contains new tuples do} \]
\[ \quad \quad \text{begin} \]
\[ \quad \quad \quad \text{for } k=0 \text{ to } n-1 \text{ do} \]
\[ \quad \quad \quad \quad \text{for } i=0 \text{ to } n-1 \text{ do} \]
\[ \quad \quad \quad \quad \text{begin} \]
\[ \quad \quad \quad \quad \quad \text{RT}_{ik} := ΔX_k \times X_{kl}; \]
\[ \quad \quad \quad \quad \quad Z_i := Z_i \cup RT_{ik}; \]
\[ \quad \quad \quad \quad \text{end}; \]
\[ \quad \quad \quad ΔX := \bigcup Z_i; \]
\[ \quad \quad \quad \text{RPTP} := \text{RPTP} \cup ΔX; \]
\[ \quad \quad \text{end}; \]
\[ \text{end}; \]
\[ \text{end } \text{relevant}(\text{PTP}); \]

Fig 6 Principle of loop of joins with double hashing technique.
The Z (\(\cup Z_i\)) relation is necessary to stack all the tuples built in one iteration: \(\Delta X\) has to be assigned to \(\cup Z_i\) only when all the tuples of the previous generation are generated. In fact, the tuples of \(Z_i\) are obtained from the \(\Delta X_i\), with \(k\) between 0 and \(n-1\). Note that the computation cost in performing the union between \(\text{RTPP}\) and \(\Delta X\) is reduced if we make use of the partition of \(\Delta X\) into several \(\Delta X_i\). One has to partition the \(\text{RTPP}\) relation into \(k\) \(\text{RTPP}_i\) buckets such that the elimination of the doubles is performed bucket by bucket: \(\text{RTPP} = \Sigma_k (\text{RTPP}_k \cup \Delta X_k)\).

### 4.3 Parallel algorithm

Due to its hashing techniques, this algorithm can be easily parallelized. With an architecture such as multiple-backend as presented in [CHEI86], the computation cost for the loop of joins can be divided by a factor \(p\), where \(p\) represents the number of processors.

Let \(p\) be the indices of a processor; the bucket of indices \(i\) of the \(\text{PTP}\) relation is written on the disk processed by the processor \(p\) (we assume here one disk per processor). The buckets of indices \(p\) of the \(\text{RTPP}\) relation are also clustered. Thus, the join between the buckets \(X_p\) and \(Y_p\) is local for each processor. The computation time of this phase is then divided by a factor \(p\). The results of each join \(X_p \times Y_p\) have then to be written on the disk \(i\). A serialization is possible according to the order in which the results are written.

In fact, if each processor \(p\) computes the buckets \(X_p\) with the ordering \(i=0\) to \(n\), the serialization by the bus might slow down the entire computation: the processors compute all the buckets \(X_p\) at the same time and try to send their results towards the same processor \(i\). This problem can be solved by a cyclic organization of the data transfers. Each processor first computes the bucket \(X_p\) and thus builds its own result. Then it computes the following bucket \(X_{(i+1)p}\) and transfers the result to the processor \(p+1\). Thus, the data transfer is done in a cyclic way, each processor receiving one subrelation at a time.

The algorithm is:

1. phase 1: (parallel selection)
2. phase 2: (parallel loop of joins)

for each processor \(p\) do

\[ \text{repeat} \]
\[ i := p; \]
\[ Z_{ip} := X_p \times Y_p; \]
\[ \text{transfer } Z_{ip} \text{ towards the disk used by the processor } i; \]
\[ i := (i+1) \text{ modulo } p; \]
\[ \text{until } i = p; \]

### 5. Evaluation of the main memory size required for a single read execution

One difficult problem with large joins is making sure that their execution stays "linear": each relation is read from disk only once. Therefore, this problem is very important with an iterative transitive closure algorithm, where a join is made in each iteration. The proposed algorithm reduces the memory requirement for a single read execution. As a first approximation, we may consider that the number of times each tuple is read depends on three factors:

(i) the feasibility of performing a delta algorithm which uses, at each step, only the prior tuples and the stored relation;

(ii) the iteration number of the join loop;

(iii) the main memory requirement of the different algorithms (for reading a relation once).

In searching for the relevant rules, we may apply a delta algorithm because the search rules are linear. However, the depth depends on the semantics of the stored rules. Logarithmic algorithms [VALD86], which reduce the number of iterations, are not reviewed in this paper. In fact, they increase the number of tuples that need to be in main memory at one time. The third factor (iii) is evaluated in this section.

We used the following parameters to evaluate the main memory requirement:

- \(|X_i|\): size of \(X\) in pages;
- \(|M|\): main memory size in pages;
- \(F\): uniformity ratio of hashing function.

Thus, when one relation \(R\) (size \(|R|\)) is hashed in \(n\) buckets, the size of the largest bucket (or sub-relation) is \(F|R|/n\).

We choose a cylinder structure [BANC86] for modelling tuple distribution. This modelling is well adapted to our problem (compared to random or tree distributions). A cylinder is a structure in which, on the average, each level has \(B\) nodes and each node has \(S\) input arcs and \(D\) output arcs. \(h\) is the height of the cylinder.

![Cylinder h=3, B=3, S=D=2](image)

Figure 7: Cylinder \(h=3, B=3, S=D=2\)

Thus, we consider B base relations and \(B*(h-1)\) derived relations. Let \(n\) be the number of different hash values on \#preplace and \#postplace. We want to determine the size of main memory required to guarantee one read of base relation, for each iteration. The first step performs the initialization transition. This step is a simple selection in which the bucket size is reduced. We keep only the tuples whose \#postplace belong to the values list of derived relations used by the given query. The resulting bucket size is therefore:

\[
F(|X|) \frac{s}{n},
\]

where

- \(s\) is the query selectivity factor
- \(p\) is the number of relevant relations, \(s=p/B\).
During the join step, the buckets of identical hash values X and AX must join in main memory. The smallest sub-relation (AXj) stays in F. IXj / n pages (the Xij pages for one i value, are read one after another). In order to have full pages, we must keep n pages for 1 \times 1. They correspond to the stacking of the produced tuples during an iteration (according to the postplace hash values). These pages are written on disk as soon as they are full. X is hashed in n^2 buckets.

![Figure 8: "One read join"

The condition is:  IMI \geq \frac{F}{n} \times X + 1 + n

With a cylindrical distribution of tuples, we consider that the size of \angle X angle (new tuples generated during one step) is equal to the size of the base relation. Thus, a sufficient condition is

IMI \geq \frac{F}{n} \times X + 1 + n \quad \text{and}

n^2 + (1 - IMI) n + F \times X \leq 0

This condition is true when (1 - IMI) \geq 4 F \times X . This constraint is easily met, it is:

IMI \geq \sqrt{4 X} \quad \text{condition 1}

The in-memory execution of each join loop step is guaranteed. However, the size that is considered here is the size of the RPTP relation reduced by the selection RPTP.P. If the memory is larger, the performance is improved because the Z result is kept in memory. The substitution of the old AXj buckets by the new AXj buckets is entirely performed in main memory. With this distribution, the relation has the same size X:

IMI \geq \frac{F}{n} \times X + 1 + X

let IMI \geq \left(\frac{F}{n} + 1\right) X + 1 \quad \text{condition 2}

Condition 2 is harder to meet but guarantees an execution of the transitive closure in one read.

6 Conclusion

In this paper we proposed a method to store and efficiently retrieve rules in a deductive database. In this proposal, the rules are compiled into an execution model called a PCN. The PCN is stored as a particular relational database which consists of three relations. For a given end-user query the first step of the inference process consists in searching for the relevant rules, by traversing of PCN structure. This traversal is computed by a transitive closure of a part of the PCN. The transitive closure is performed by a loop of joins over the relations storing the PCN. In this context, we proposed a new transitive closure algorithm based on a physical clustering of the relations. This clustering consists of a double hashing technique. It ensures a linear cost for the join operation with weak conditions on the size of the main memory. The present work reduces the number of I/O operations for seminaive transitive closure operation. In our system, this algorithm will be applied both to the search for relevant rules and to the transitive computation of linear recursive relations.

7 References


