DATA INTENSIVE PRODUCTION SYSTEMS: THE DIPS APPROACH

Timos Sellis, Chih–Chen Lin
Department of Computer Science

Louiqa Raschid
Department of Information Systems

and Institute for Advanced Computer Studies (UMIACS)
University of Maryland, College Park, MD 20742

Abstract

In this paper we highlight the basic approach taken in the design of the DIPS system, and briefly present the main contributions. These include the use of special data structures to store rule definitions; they are implemented using relations. A matching algorithm uses these structures to efficiently identify when the antecedents of productions are satisfied, making them applicable for execution. Partial match information stored in the data structures is used by the matching algorithm. We also describe a proposed concurrent execution strategy for applicable productions, which surpasses in performance, the traditional sequential OPS5 production execution algorithm. The requirements for a correct, serializable execution, based on locking, is described. An advantage of the matching technique in DIPS is that it is fully parallelizable, which makes it attractive for implementation in parallel computing environments.

1. Introduction

The integration of artificial intelligence (AI) and database management (DBMS) has been the focus of recent research [11,12]. An important aspect of this integration is identifying functional similarities in database processing and reasoning with rules. This allows techniques designed for use in either AI or DBMS technology to be tailored for use in a functionally integrated environment.

Existing relational database systems have some limited rule subsystems to provide integrity control and protection. Updates are "filtered" and committed only if several user–defined constraints are met. Updates to the database may trigger the firing of rules, which in turn, may perform several updates to the database. Triggers and their implementation in DBMS have been studied in various contexts [1,2,3,7,15] and they are similar to production rules in AI production systems [10]. The problem of supporting rule–based reasoning efficiently in a database environment is the focus of our research.

This research was sponsored partially by the National Science Foundation under Grant IRI-8719458 and by UMIACS.

In this paper, we describe the Data Intensive Production System (DIPS), designed to efficiently support production rules in a database environment. In DIPS, production definitions are stored using special data structures implemented using relations. A matching algorithm uses these structures to determine when the antecedents of productions are satisfied; these productions are then scheduled for execution. The algorithm stores information on partially satisfied productions, as well, using these relational data structures. Section 2 describes production systems and section 3 describes the architecture of DIPS. The relational data structures and the matching algorithm based on these structures are described in section 4.

In section 5, we examine a concurrent execution strategy for the execution of productions in DIPS. We demonstrate the equivalence of a serial execution strategy, as in OPS5, and the proposed concurrent execution strategy and we define the requirements for a correct, serializable execution, based on a locking technique.

The set–oriented matching mechanism in DIPS, together with the concurrent execution strategy, has the potential to be very efficient if the data base and rule base are very large. In addition, the DIPS matching algorithm is easily parallelizable. Using advanced database technology allows us to provide new methods and algorithms that are fully compatible with the relational model, yet appropriate for **very large** production systems.

2. Production Systems and OPS5

Production systems capture problem solving knowledge in the form of rules or productions and provide a good characterization of the process of reasoning using rules [10]. A production system is a collection of *Condition-Action* statements, called ***productions***. The Condition part is on the left-hand side of a production (LHS) and is satisfied by data stored in a global database, composed of *Working Memory* (WM) elements. The action part is on the right-hand side (RHS) and executes operations that can modify WM. A production system repeatedly performs the following operations, in sequence:

Match: For each production r, determine if LHS(r) is satisfied by the current WM contents. If so, add the qualifying production to the *Conflict Set.*

Select: Select one production out of the conflict set; if there is no such production, halt.

Act: Perform the actions in the RHS of the selected production. This will change the content of the WM and as a result, additional productions may have to be fired, or some productions may be deleted.

The OPS5 production system [8] has enjoyed much popularity in AI research and applications. An OPS5 production consists of (1) the symbol **p**, (2) the name of the production, (3) the LHS, (4) the symbol →, and (5) the RHS actions. The following are two productions:

```
(p R1
  (Emp ↑Name Mike ↑ Salary <S> ↑Dno <D>)
  (Dept ↑Dno <D> ↑D Toy ↑Floor 1 ↑Mgr <M>)
  → (remove 1))

(p R2
  (Emp ↑Name Mike ↑ Salary <S> ↑Dno <D>)
  (Dept ↑Dno <D> ↑D Shoe ↑Floor 1 ↑Mgr <M>)
  → (remove 1))
```

that remove Mike from the WM class Emp if he works on the first floor, in the Toy department (R1) or the Shoe department (R2). The ↑ symbol is used to indicate attribute names and <x> is a variable.

The efficiency of OPS5 [8], has been attributed to the efficiency of the Rete algorithm and its implementation [8,9]. Here, the database resides entirely in virtual memory, and does not persist after the execution of a program. The Rete algorithm exploits temporal redundancy, i.e., the execution of a single production results in very small changes to WM. The Rete algorithm compiles the LHS condition elements, in lexical ordering, into a binary discrimination network. Figure 1 illustrates the result of compiling the above two productions.

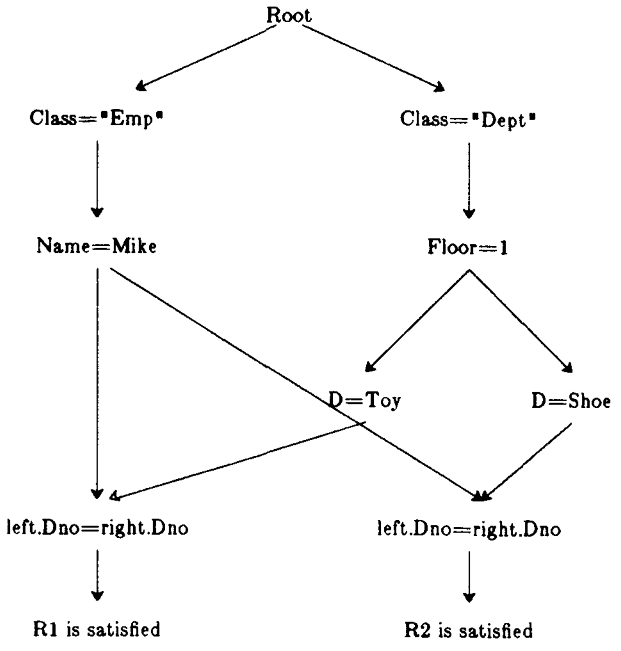


Figure 1 The Rete Network for R1 and R2

There is a *root* node which receives all input tuples. *One-input* nodes are used to check single attribute conditions of the form attr *op* constant, where $op \in \{<,>,\leq,\geq,=,\neq\}$. *Two-input* nodes are used to check joins of the form

left.attribute *op* right attribute

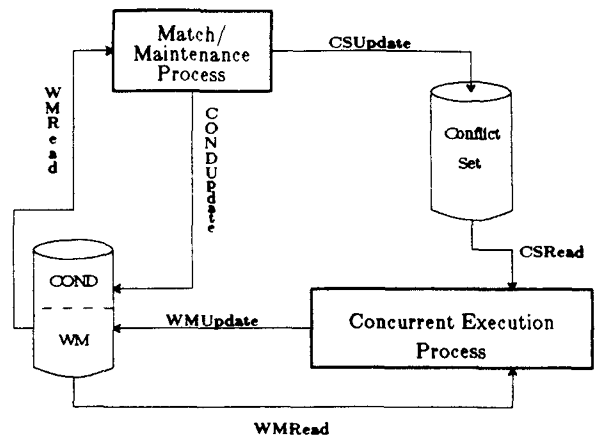
A tuple t is first checked at one-input nodes to determine if it is an Emp or a Dept tuple. If so, it is then propagated to the successors of the qualifying node of the network. In case a check is performed at a two-input node, and a matching value is not found at the corresponding join branch, the tuple is queued up at the network waiting for a future arrival of a matching tuple. When such a tuple comes through the network, the result of the join is a token that is propagated to the successors of the two-input node. Finally, if a token makes it all the way till the "bottom" of the Rete Network, a production or set of productions have qualified and the system adds these productions to the conflict set, together with the token that caused the production to become active.

In OPS5, productions placed in the conflict set are executed in a serial order. In each cycle, a single production is selected and its RHS actions are then executed; this may result in changes to the WM. This triggers the next match phase. Updates to the WM are propagated through the Rete network. Consequently, productions in the conflict set may be deleted, or new productions may be added. When several combinations of the WM elements satisfy a single production, OPS5 stores each combination as a separate instantiation of the production in the conflict set, and each is executed independently. Similarly, a WM element could simultaneously satisfy several productions. Again in OPS5, only a serial execution strategy is followed.

3. The DIPS System

The DIPS system views the task of satisfying the LHS of a production from a DBMS perspective. In general, we can consider the LHS as a well formed formula describing a condition that must be satisfied by the data stored in a knowledge base. The LHS is equivalent to a retrieval operation in a non-procedural SQL-like language or a tree of relational algebraic operations. Satisfying the LHS of a production is equivalent to executing a retrieval operation against the occurrences in a knowledge base; this approach has been the basis of some proposals for implementing production systems in DBMS environments such as in [4] and [6].

Figure 2 shows the general architecture of DIPS. As in OPS5, the Match/Maintenance process (MMP) is triggered by one (or more) changes to WM relations. The **WMRead** operation in the figure represents this dependency. Functionally, the MMP is similar to the Match phase of the Rete algorithm. This process identifies productions whose antecedents are satisfied. The process makes use of the special COND relations that store production definitions as well as information on partially satisfied productions, as discussed in the next section. It comprises **insertion** and **deletion** algorithms that update the COND relations, as shown by the **CONDUpdate** operation. The MMP identifies **matching patterns** corresponding to productions whose antecedents are satisfied and are ready for execution. The patterns are placed in the conflict set and this is reflected by the **CSUpdate** operation in Figure 2. The difference between DIPS and OPS5 is that in the latter, the maintenance of the Rete network precedes the update of the conflict set. In DIPS, the update of the conflict set (**CSUpdate**) precedes the maintenance process (**CONDUpdate**). This implies that productions whose antecedents are satisfied are identified at an



WMUpdate:	Working Memory Update
WMread:	Working Memory Read
CONDUpdate:	Condition Relations Update
CSUpdate:	Conflict Set Update
CSRead:	Conflict Set Read

Figure 2 DIPS Architecture

earlier instant (and can be executed earlier).

It is with respect to the Select and Act phases (of the Rete algorithm) that DIPS differs significantly. As mentioned earlier, in OPS5, productions placed in the conflict set are executed in a serial order. In DIPS, this serial execution strategy is replaced by a concurrent one (Concurrent Execution process in Figure 2). This process concurrently executes a set of productions in the conflict set. Within the Concurrent Execution process, all the tasks associated with the execution of a candidate production will be defined as a single transaction. Within such a transaction, the first task is a retrieval from the WM relations; this is reflected by the **WMRead** operation of the Concurrent Execution process. The next task is executing the corresponding RHS (right-hand side) actions. These actions represent changes to the WM classes and include insertions, deletions and updates of the WM elements. This is reflected by the **WMUpdate** operations in the figure. Details of the Concurrent Execution process (CEP) are in section 5.

4. The Match/Maintenance Process

In DIPS, each WM class is simulated using a WM relation which stores occurrences of this WM class. The antecedents of productions are compiled and stored using a relational data structure, the COND relation. There is one COND relation for each WM relation in the system. The COND relation associated with a WM relation R, stores the descriptions of conditions found in all productions that involve R in their antecedents. Partial match information about the occurrences of other WM classes involved in the same production as R, linking those occurrences to the occurrences of R, are also stored in the COND relation for R.

Insertion and deletion of occurrences of WM classes (tuples in the WM relations) will affect the antecedents of productions that test conditions against these classes. Insertion of tuples may cause the antecedents to become satisfied and deletion will make them no longer satisfied. The antecedents could also include negative conditions, i.e., they verify the absence of some occurrences of a WM class. Thus, insertion or deletion into a WM class may also effect productions with negative conditions.

The details of the COND relational structures and the insertion and deletion algorithms of the MMP are discussed in [14]. Here, we use an example to illustrate the operation of the algorithm that handles insertion of tuples into WM relations. Assume three relations `E(name,salary,dno)`, `W(name,mgr,job)`, `D(dname,mgr,dno)` (standing for `Emp`, `WorksFor` and `Dept` respectively). The following defines a production which is used to fire all Clerks working in the Toy department and making 20K:

```
(p Rule-1
   (E ↑name <x> ↑salary 20K ↑dno <z>)
   (W ↑name <x> ↑mgr <y> ↑job 'Clerk')
   (D ↑dname 'Toy' ↑mgr <y> ↑dno <z>)
→
   ( remove 1 ))
```

This production is initially stored as a single tuple within three COND relations, which correspond to the three WM classes in the LHS of the production. Suppose now that we insert the tuples `W(Mike,John,Clerk)`, `D(Toy,Tom,8)`, `E(Mike,20K,8)` and `W(Mike,Tom,Clerk)` in the sequence given. The contents of the various COND relations will be as follows (for brevity we use `E` for `E`, `W` for `W`, and `D` for `D`)

COND-E

CEN	name	salary	dno	RCE	WD
1	<x>	20K	<z>	(W,2),(D,3)	00
1	Mike	20K	<z>	(W,2),(D,3)	10
1	<x>	20K	8	(W,2),(D,3)	01
1	Mike	20K	8	(W,2),(D,3)	11

COND-W

CEN	name	mgr	job	RCE	ED
2	<x>	<y>	Clerk	(E,1),(D,3)	00
2	<x>	Tom	Clerk	(E,1),(D,3)	01
2	Mike	<y>	Clerk	(E,1),(D,3)	10
2	Mike	Tom	Clerk	(E,1),(D,3)	11

COND-D

CEN	dname	mgr	dno	RCE	EW
3	Toy	<y>	<z>	(E,1),(W,2)	00
3	Toy	John	<z>	(E,1),(W,2)	01
3	Toy	<y>	8	(E,1),(W,2)	10
3	Toy	John	8	(E,1),(W,2)	11
3	Toy	Tom	<z>	(E,1),(W,2)	01
3	Toy	Tom	8	(E,1),(W,2)	11

For each production, the `RCE` list in each tuple of the COND relations indicates which conditions (involving other WM relations) of the same production are affected by insertions or deletions in the current relation being examined. There is one `Mark` bit for each `RCE`, which if set indicates that the pattern is created by a tuple that satisfies the corresponding condition element. These `Mark` bits are stored in the last column of the COND relations. For example, the insertion of tuple `W(Mike,John,Clerk)` is recorded in both `COND-E` and `COND-D`. Thus, when a tuple is inserted later in the current WM relation which matches that pattern, we know immediately that there are matching occurrences of related WM classes, without having to examine the other WM relation(s). Notice that when `W(Mike,Tom,Clerk)` is inserted, the last tuple in `COND-W` causes `Rule-1` to be put in the conflict set because all `Mark` bits are set (indicating the existence of matching tuples `E` and `D`).

Comparing the DIPS Approach with OPS5 we can see that matching is very fast in DIPS because only a single search over a COND relation is necessary. The propagation cost, is the same as the cost incurred by a Rete Network. The method is easily *parallelizable*, since propagation of changes can be performed in parallel to all the COND relations. In contrast to that, the Rete Network method is highly sequential.

5. Concurrent Execution of Productions

There are many sources of concurrency in executing productions. If several combinations of the WM elements satisfy a single production in the conflict set, then this results in *intra-production* concurrency (within a single production). In DIPS, a set-oriented selection will retrieve all possible combinations of the WM tuples satisfying each antecedent. Thus, a selected production can be simultaneously applied to all possible combinations of the WM tuples, that are retrieved. Similarly, there is a potential for *inter-production* concurrency; i.e., for executing several different productions from the conflict set.

In the rest of this section, we investigate the concurrent execution of productions in the conflict set. First, we show the equivalence of a serial and a concurrent execution. Next, we define the requirements for a *correct* concurrent execution strategy, in DIPS. Our notion of correctness is based on the serializability criterion; WM relations will not be updated until a transaction commits, and we guarantee a correct execution that is serializable, based on 2 Phase Locking mechanisms. Details on the estimated performance benefits of a concurrent execution are in [13].

Consider an initial set Ψ_1 of transactions in the conflict set. In a serial production system, in each step i, a single transaction, T_i is *arbitrarily* selected from the conflict set and applied. We use the term "arbitrarily", because the OPS5 conflict resolution strategies are syntactic. Subsequently, the production system will determine if, as a result of applying T_i, some other transactions in this conflict set are no longer applicable; if so, these transactions will be deleted from the set. Let the set of transactions deleted in step i be del_i. Also as a result of applying T_i, the production system will determine if some additional transactions are now applicable as well. Let the set of transactions added in step i be add_i. The new set of candidate transactions in step $i+1$ is $\Psi_{i+1} = \Psi_i - \{T_i\} - del_i \bigcup add_i$. This process will continue until in step F, the set Ψ_F is empty.

The selection of each T_i is arbitrary; thus, it is entirely possible that in step 2, T_2 is selected from the set $\Psi_1 - \{T_1\} - del_1$ which is the set $\Psi_2 - add_1$. In other words, T_2 could also be selected from the initial set Ψ_1 and not from the added set of transactions add_1. Similarly, in subsequent steps i, T_i can be selected from the set $\Psi_1 - \bigcup_{j=1}^{i-1} (\{T_j\} \bigcup del_j)$ After some f_1 steps the serial production system will have executed a sequence of f_1 transactions $T_1, T_2, ..., T_{f_1}$, where each T_i happens to be an element of the initial set Ψ_1. After step f_1, all transactions in Ψ_1 are either executed or deleted and the set of applicable transactions for step (f_1+1), Ψ_{f_1+1} is the set $\bigcup_{j=1}^{f_1} add_j$, i.e. all the transactions added in the f_1 previous steps, which were not selected previously.

Given this same initial set Ψ_1, a concurrent execution strategy would interleave the execution of this set of transactions. If an appropriate protocol is used, and the resulting schedule is serializable, then it must be equivalent to some serial schedule $T_1, T_2, ...,$ etc., where each T_i must be from the initial set Ψ_1. In other words, the concurrent production system will execute an equivalent serial schedule which will be the same as some serial schedule arbitrarily selected by the serial production system.

In a DBMS environment, a transaction commits all its changes after it has terminated its execution normally. Once the transaction commits, these changes are physically made to the database. In a concurrent environment, appropriate locks must be obtained to satisfy the following: First, the interleaved execution of a set of productions must maintain consistency of the database, i.e. two transactions that update the same WM relation must be serializable. Second, transactions that are inter-related and affect each other's execution, that is transactions that delete each other's matching-pattern tuples from the conflict set, must interact correctly. For example, when a transaction T_i executes, the selected commit point must be chosen to enforce a delay in the execution (and commit) of the transactions in the set del_i, i.e. transactions that are deleted as a result of applying T_i. Transactions in this set must either not be executed or, if executed, their changes must not be committed to the database.

A locking protocol is used in DIPS to produce serializable execution schedules. Consider a simple case where the antecedent of a production only tests positive conditions on WM relations.

- Each transaction must obtain a read lock (**R lock**) for specific tuples of the WM relations that satisfy the LHS of the production. This prevents the deletion or update of these tuples by other transactions. Note that if any of these retrievals returns an empty set of tuples, then the transaction is aborted. This may happen if T_i is in the delete set of a previously committed transaction. T_i is aborted since it is in some delete set.
- Each transaction must obtain a write lock (**W lock**) for specific tuples of the WM relation that it deletes or updates. These would necessarily be tuples for which it would have previously obtained an **R lock**; consequently, these tuples must exist and could not have been deleted. However, it could wait indefinitely in case of a deadlock. In the case of an insertion, a W lock needs to be obtained for the whole relation.
- Once these locks are obtained, the transaction can modify the WM relations, corresponding to the RHS actions and then commit all of its changes and release all locks.

In [13], we have proved the correctness of our requirements for serializability by examining all cases of inter-dependencies among productions (through the WM relations).

6. Conclusions

In this short paper we have described the basic approach taken by the DIPS system. DIPS provides ways for storing, maintaining and using large production rule bases. DIPS uses a novel approach to screen updates to the database for faster detection of applicable productions. This approach also achieves localization of the match procedure in the sense that a single relation has to be checked in order to decide if an inserted or deleted tuple renders a production applicable for firing. Thus, our approach is not only suitable for relational DBMS's but in addition makes our method easily parallelizable. Finally, we use a new way to process applicable productions based on the notion of transactions. Applicable productions can be processed concurrently, assuming that the DBMS will serialize RHS actions (insertions and deletions) through its concurrency control mechanism.

Our current work focuses on the implementation and optimization of the proposed approach. First, we examine ways in which multiple query processing and optimization algorithms can be applied to provide optimal Rete Networks. Although our approach does not assume any global execution strategy, we are currently conducting a performance analysis of the original Rete Network and our approach. In addition, we study the properties and performance of a fully concurrent system where transactions are used to implement the actions of the various productions. Finally we plan to investigate the parallel execution of the matching algorithms in a SIMD environment such as the Connection machine, where the COND relational data structures will be partitioned.

7. References

[1] Blakeley, J.A., Larson, P., and Tompa, F.W., Efficiently Updating Materialized Views, *Proc. of the ACM-SIGMOD Conf.*, Washington, DC (1986).

[2] Blakeley, J.A., Coburn, N., and Larson, P., Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates, *Proc. of the 12th VLDB Conf.*, Kyoto, Japan (1986).

[3] Buneman, O.P., and Clemons, E.K., Efficiently Monitoring Relational Databases, *ACM TODS* (4) 3 (1979).

[4] de Maindreville, C., and Simon, E., A Production Rule Based Approach to Deductive Databases, *Proc. of the 4th Data Engineering Conf.*, Los Angeles, CA (1988).

[5] Dayal, U. et al, The HiPAC Project: Combining Active Databases and Timing Constraints, *SIGMOD Record* (17) 1 (1988).

[6] Delcambre, L.M.L, and Theredge, J.N., The Relational Production Language: A Production Language for Relational Databases, in *Expert Database Systems: Proceedings From the First International Conference* (Kershberg, L., Ed.), Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).

[7] Eswaran, K.P. et al., The Notions of Consistency and Predicate Locks in a Database System, *CACM* (19) 11 (1976).

[8] Forgy, C.L., OPS5 User's Manual, Tech. Report CMU-CS-81-135, Carnegie-Mellon University (1981).

[9] Forgy, C.L., Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence* (19) (1982).

[10] Hayes-Roth, F., Rule Based Systems, *CACM* (28) 9 (1985).

[11] Kershberg, L., Editor, *Expert Database Systems: Proceedings From the First International Workshop*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1986).

[12] Kershberg, L., Editor, *Expert Database Systems: Proceedings From the First International Conference*. Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1987).

[13] Raschid, L., Sellis, T., and Lin, C–C., Exploiting Concurrency in a DBMS Implementation for Production Systems, *Proc. of the Intern. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX (1988).

[14] Sellis, T., Lin, C–C., and Raschid, L., Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms, *Proc. of the ACM-SIGMOD Conf.*, Chicago, IL (1988).

[15] Stonebraker, M., Sellis, T., and Hanson, E., Rule Indexing Implementations in Database Systems, In [12].