CPD: a useful tool for designing expert systems

Y. Zhang

This paper was presented at an IFIP WG 2.3 Workshop on "Security and Databases", Baltimore, September, 1987.

Abstract

The interface between logic programming language and databases is only beginning to be explored. Both fields have much to offer the other. This paper presents an approach of coupling Prolog with Db++ (CPD). This gives a user the advantages of both Prolog and Db++. Prolog is a logic programming language which has function of inference and pattern matching and is suitable for constructing knowledge-based systems. A prolog program consists of facts and rules. A fact in Prolog can be used to represent a tuple in a relational data base. A rule can be used to specify a view in a relational database, or to represent knowledge about how to use the facts. But existing versions of Prolog do not have the full data management functions of DBMS and Prolog can not use secondary memory to store its data, therefore it is very difficult for Prolog to treat very large applications. Much of our current research concerns the development of special computer software system to access Db++ from Prolog that mainly consists of inference, pattern matching and secondary storage management of facts stored as relations in Db++. The functions of inference and pattern matching come from Prolog, the functions of secondary storage management of facts come from Db++ (a DBMS).

Series Editor: M.J. Elphick

© 1988 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

ZHANG, Yuguo


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1988.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 265.)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Laboratory. Technical Report Series. 265

Abstract

The interface between logic programming language and databases is only beginning to be explored. Both fields have much to offer the other. This paper presents an approach of coupling Prolog with Db++ (CPD). This gives a user the advantages of both Prolog and Db++. Prolog is a logic programming language which has function of inference and pattern matching and is suitable for constructing knowledge-based systems. A prolog program consists of facts and rules. A fact in Prolog can be used to represent a tuple in a relational data base. A rule can be used to specify a view in a relational database, or to represent knowledge about how to use the facts. But existing versions of Prolog do not have the full data management functions of DBMS and Prolog can not use secondary memory to store its data, therefore it is very difficult for Prolog to treat very large applications. Much of our current research concerns the development of special computer software system to access Db++ from Prolog that mainly consists of inference, pattern matching and secondary storage management of facts stored as relations in Db++. The functions of inference and pattern matching come from Prolog, the functions of secondary storage management of facts come from Db++ (a DBMS).

About the authors

Mr. Zhang is a Ph.D. in the Computing Laboratory of the University of Newcastle upon Tyne.

Suggested keywords

DATABASES
EXPERT SYSTEMS
KNOWLEDGE-BASED SYSTEMS
LOGIC PROGRAMMING LANGUAGE
PROLOG

Suggested classmarks (primary classmark underlined)

Dewey (18th): 001.535 001.6442 001.6424
U.D.C. 007.52 519.682 681.322.06
CPD: A Useful Tool For Designing Expert Systems

Yuguo Zhang
Computing Laboratory
University of Newcastle Upon Tyne

ABSTRACT

The interface between logic programming language and databases is only beginning to be explored. Both fields have much to offer the other. This paper presents an approach of coupling Prolog with Db++ (CPD). This gives a user the advantages of both Prolog and Db++. Prolog is a logic programming language which has function of inference and pattern matching and is suitable for constructing knowledge-based systems. A Prolog program consists of facts and rules. A fact in Prolog can be used to represent a tuple in a relational data base. A rule can be used to specify a view in a relational database, or to represent knowledge about how to use the facts. But existing versions of Prolog do not have the full data management functions of DBMS and Prolog can not use secondary memory to store its data, therefore it is very difficult for Prolog to treat very large applications. Much of our current research concerns the development of special computer software system to access Db++ from Prolog that mainly consists of inference, pattern matching and secondary storage management of facts stored as relations in Db++. The functions of inference and pattern matching come from Prolog, the functions of secondary storage management of facts come from Db++ (a DBMS).

1. Introduction

One of the most rapidly growing fields of artificial intelligence is expert systems. Currently, expert systems exist or are being constructed for medicine, business and national defence. These systems are composed of a knowledge base of rules and facts and an inference mechanism that uses the knowledge base to respond to queries posed by users. The objective of such systems is to capture the knowledge of experts in particular domains (such as aircraft engine maintenance, financial analysis, computer configuration control and blood diseases) and make it generally available to nonexpert users. At present, such systems focus on narrow domains and have special small knowledge bases, they are thus limited in their application. In recent years, as Expert Systems [Chang 1983, Pereira and Porto 1982, Walker 1983, Santane-Toth and Szeredi 1982] are expanded, an increasing number of their applications, including CAD/CAM, office automation and military command and control, have a requirement for knowledge-directed processing of shared information, their knowledge base will become bigger, sometimes very big so that it will become more difficult to manage.

Generally speaking, the knowledge base of an expert system encompasses both facts and rules and allows user to manage them in a unified way. For example, we have a course_management knowledge base which is presented in a formalism taken from the Prolog language. This simple example presents a database having three relations: student, course and sc having factual instances, and a base of rules representing a body of elementary knowledge about relationship between student, course and teacher.
The following are the three relations:

a) Relation student: has 3 attributes: sno, sname, and dept standing for the number of student, the name of student and the department in which student is studying.

b) Relation course: has 4 attributes: cn, cname, tname, and precno standing for the number of course, the name of course, the name of teacher who teaches this course and the number of precourse.

c) Relation sc: has three attributions: sno, cno, and grade standing for the number of student, the number of course taken by student and the grade of the course.

There are five rules in the rule base:

a) Rule1 defines a quest for the grade of course having name Cname, taken by the student whose number is Sno. In order to satisfy this goal, the system has first to find the number of course Cno from relation course according to Cname, second it finds the grade from relation sc according to Sno and Cno, and finally displays the grade on screen or printer. Here write and nl are system predicates for outputing result and "new line".

b) Rule2 defines a quest for the students who take course Cname. In order to satisfy this goal, first system finds the Cno from relation course according to the Cname, second it finds the first student number Sno from relation sc according to Cno, finally, it finds the student name from relation student according to Sno and displays/prinits student name, finally, predicate fail results in failure and causes system to backtrack to goal student(Sno,Sname,Dept), then resatisfies this goal to find second student and so on.

c) Rule3 defines a quest for all the course names which are the precourses of course Cname. In order to satisfy this goal, first system finds the precourse number Precno from relation course according to Cname, second it calls the predicate p_precourse to finds all the precourse names and display/print them.

d) Rule4 and rule5 define a quest just for outputting the course name and all the precourse names according to Cno. Rule5 is a recursive rule.

*Rules*

\[\text{grade(Sno,Cname);-course(Cno,Cname,Tname,Precno),}\]
\[\quad\text{sc(Sno,Cno,Grade),}\]
\[\quad\text{write(Grade),nl.}\]

\[\text{students(Cname);-course(Cno,Cname,Teacher,Precno),}\]
\[\quad\text{sc(Sno,Cno,Grade),}\]
\[\quad\text{student(Sno,SName,Dept),}\]
\[\quad\text{write(Sname),nl,fail.}\]

\[\text{precourse(Cname);-course(...,Cname,..,Precno),}\]
\[\quad\text{p_precourse(Precno).}\]
\[\text{p_precourse(0).}\]
\[\text{p_precourse(Cno);-course(Cno,Cname,..,Precno),}\]
\[\quad\text{write(Cname),nl,p_precourse(Precno).}\]
*Facts*

student(4001,John,computer).
student(4002,Alfred,mathematics).
student(4003,Mary,electricity).
student(4004,Peter,chemistry).

.....

course(100,calculus,david,90).
course(110,computer_organization,ted,80).
course(120,database,alfred,110).
.....

sc(4001,100,a).
sc(4001,110,c).
sc(4002,120,b).

.....

Now if we suppose that a university has 5000 students and every student has to take at least 20 courses, the above Prolog program must have at least 5000 student assertions and 100,000 sc assertions. This is a very big Prolog program and it is difficult to run such a program on a limited space.

For sake of conciseness we are not going to further develop the example. In addition to simple queries as: "Who is studying in the computer Dept.?", the system has enough knowledge about the course_management to answer queries like:

-Who is the teacher of Mary?

-Who is the student of david?

-How many course does Ted have?

.....

We see above that the important features of managing in the same context both the rules and the facts is supplied in a natural way within a Prolog system. The Prolog system[Battani and Meloni 1973, Kowalski 1979, Roberts 1977,Sowa 1981, Warren 1977] supplies the formalism (the language itself) to represent the two forms of knowledge (facts and rules) and the mechanisms to process that knowledge, the mechanism includes an inference engine and a pattern matching mechanism [Clocksin and Mellish 1984, Missikoff and Wiederhold 1984]. But for very large knowledge bases a Prolog system has two principal problems:

-How can the Prolog system store very large knowledge base as shared information in secondary memory?
-How can the Prolog system obtain the desired facts from secondary memory in the shortest time possible?

Fortunately, on the other hand, Data Base Management Systems (DBMS)[Date 1981] have been developed to manage large amounts of shared information in secondary storage. A DBMS can make multiple users (application programs) access to the same collection of information for purposes of retrieval and/or update. Under these circumstances, the DBMS also provides a separate software facility required between the users and the database to protect the shared information. This facility provides consistency control, recovery control, concurrency control and security control. In addition, a DBMS can provide other advantages in controlling data redundancy and distribution, and in making application programs
easier to develop and maintain.

Many computer scientists have suggested representing the facts in Prolog by relations stored in a relational database [Brodie 1984, Dahl 1984, Deering 1984, Sciore 1984]. The Db++ system comprises a family of programs which form an efficient, flexible and reliable relational database system [Ward 1984]. My research work is trying to resolve these problems by coupling Prolog and Db++ for developing applications requiring both a Db++ and one or more Expert Systems, as shown in Fig. 1. We call the system which is formed by coupling Prolog with Db++ CPD.

![Diagram](image_url)

**Fig. 1**

In addition, today the majority of computer researchers believe that by the 1990's Expert Database Systems are expected to become one of the most important application development products [Smith 1984]. Thus the integration of Expert Systems and DBMS Systems is an emerging research area and attracts many computer researchers' attention.

2. The Architecture and Implementation of a CPD system

The following diagram shows the architecture of a CPD system:
We can see from this architecture that the CPD system consists of 5 parts:

a) CPD program: running user’s CPD program.
b) Transforming program of predicate (TPP): transforming the retrieval predicate to standard Db++ command and sending it to Db++ system.
c) Db++ program: acting as a database server to CPD program, managing the facts in relation form.
d) Transforming program of tuples (TPT): transforming the tuples to standard CPD facts and sending to CPD program.
e) Data file: secondary storage for stored facts in relation form.

CPD program and Db++ run on two independent virtual machines on the same physical machine, running under the UNIX operating system. Communication consists of sending messages in file. Each invocation of the retrieval predicate in a CPD program is a request to Db++. The retrieval predicate is sent to TPP and is transformed to standard Db++ command. Then the system sends this command to Db++ through the shell command in Unix environment. When Db++ finishes its retrieval to Data file, it sends the tuples to TPT. The TPT transforms these tuples to standard CPD facts and sends to CPD program.

In implementation of CPD system, we complete the transforming program of predicate using the Prolog language and complete the transforming program of tuples using the C language.

In CPD you can use all standard Prolog predicates and expanded predicates which can be used to interface with Db++ to retrieve the related facts from Db++. Now we call the
language which consists of Prolog predicates and expanded predicates defined by us as CPD. The details of the CPD program will be described in following section.

3. Structure of CPD programs

A CPD program is a Prolog program that uses the special predicates, which can be used to obtain the desired facts or store the facts as tuples into a relation. We now describe these special predicates as follows:

3.1. Data definition and removal in secondary memory

Because all facts in CPD program should be stored as a relation in secondary memory, CPD should have predicates to tell the system what relation to be defined and put in secondary memory. (Obviously, you can directly use Db++ to define a relation). CPD provides two predicates for setting up a relation in the particular format or removing a relation from secondary memory.

a) Defining the relation on secondary memory

Format: create( S ).

The predicate create is meant for this situation when you want to set up a new relation file, structure it in the particular format required by the Db++ programs and place initial data into the new relation file. The argument must be an atom representing a relation definition statement. It may be used to create a relation or a view. Once a relation or a view is defined, CPD can get facts(tuples) from the relation or view on Db++, or insert tuples into a relation on Db++.

For example:

?- create('student -df/ sno/u sname/S dept/S').

| >47001/smith/computer/ |
| > ... |
| > 'D |
| ?- |

Here is a list of the available type characters and their corresponding data types(NB: this relation is a subset of Db++):

<table>
<thead>
<tr>
<th>TYPE CHARACTERS FOR CREATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Type</td>
</tr>
<tr>
<td>Short Integer</td>
</tr>
<tr>
<td>Unsigned Short Integer</td>
</tr>
<tr>
<td>Long Integer</td>
</tr>
<tr>
<td>Floating Point</td>
</tr>
<tr>
<td>Double Precision Floating Point</td>
</tr>
<tr>
<td>Packed Decimal(15 Digit)</td>
</tr>
<tr>
<td>Packed Decimal(31 Digit)</td>
</tr>
<tr>
<td>Character String</td>
</tr>
</tbody>
</table>

In Prolog, a fact is composed of objects and their relationship. For example, we want to tell Prolog the fact that "John likes Mary", this fact consists of two objects, called "Mary" and "John", and a relationship, called "Likes", we need to write it in a standard form:
likes(john, mary).

Therefore the following things are important in CPD:

1. The name of all relations to be defined must begin with a lower-case letter. For example, `likes`. This is different from `Db++` which permits relation to have name beginning with a capital letter, because a name with a capital letter stands for a variable in Prolog.

2. The name of all domain in the relation also must begin with a lower-case letter.

b) Removing the relation from secondary memory

Format: `remove(S)`.

The predicate `remove` enables a program to remove a relation which is not needed in future from secondary memory. The argument must be an atom representing the name of relation file which is to be removed from secondary memory.

For example:

```
?- remove(student).
```

After this query, the system deletes this file on secondary memory, therefore you must be careful in using this predicate.

3.2. Data insertion and deletion

When you want to tell the system to insert some tuples into or delete some tuples from a relation, CPD provides two predicates to deal with this situation.

a) Inserting the tuples into relation on secondary memory

Format: `addition(S)`.

The predicate `addition` is meant for inserting some tuples into the relation. The argument must be an atom giving the name of relation file which must be first defined either by CPD, or by separately on `Db++`.

For example:

```
?- addition(student).
>47002/jamas/geology/
>...
>^D
?-.
```

b) Deleting the tuples from relation on secondary memory

Format: `delete(S)`.

The predicate `delete` cause the system to delete some tuples which match the given qualification from a relation on `Db++`. The argument must be an atom representing a qualification.

For example:
?- delete('student::dept = "computer").

After this query, the tuples that match the given qualification (dept = "computer") are removed from relation student.

3.3. Clear the facts from main memory

When system runs a program, so many facts may have been sent to main memory from Db++, but there can be some facts which will be not useful in inference. The best way is to clear these facts from main memory as soon as possible when these facts become no use, because main memory is very important for running a program. There is a predicate to deal with this problem.

Format: clear(S,N).

The predicate clear has two arguments S and N. The S gives the name of relationship of facts and the N gives the number of arguments of facts because same relationships but different number of arguments represent different facts. The predicate clear enables the system to delete all facts which have the name S and N arguments.

For example:

?- clear(student, 3).

After the system executes this predicate, all facts having the name student and 3 objects are cleared from main memory.

3.4. Retrieve the facts

When you want to send a relation or select some tuples from a relation and send them into main memory to answer the question, CPD provides seven predicates to deal with seven different situations according to different demands of program.

a) Inputing the relation or selecting tuples

Format: select(S).

The predicate select with one argument is meant for asking Db++ to send a relation or select some tuples from a relation and send into main memory. The argument must an atom giving the qualification and name of relation file which must be first defined either by CPD, or by separately Db++.

For example:

?- select(student).

or

?- select('student::dept = "computer").

System executes these predicates, the relation student or tuples selected from student are transformed to CPD facts, then sent into CPD program.

b) Selecting the facts from relation

Format: select(S,L).

The predicate select with two arguments is meant for selecting some tuples from a relation in secondary memory according to given qualification and sending into main memory. The first argument is a string representing relation select_statement, second argument is a list giving a group of variables which will be instantiated when the system executes this
predicate. This predicate is defined for being inserted to rules (Obviously you can use this predicate as a goal).

For example:

\[
\text{students}(\text{Cname})\text{-select('course::cname} = = \text{Cname}',\{\text{Cname}\}),
\text{course}(\text{Cno, Cname, Tname, Prcno}),
\]

When system executes the predicate \textit{students}, first it executes the predicate \textit{select(...)}, all tuples which match the qualification are loaded from secondary memory and transformed to facts, then sent into CPD program, second it can execute the following predicate \textit{course} and so on.

c) Selecting facts and setting up a temporary relation file

Format: \text{select(S,L,F)}.

The predicate \textit{select} with three arguments is just like \textit{select} with two arguments, except that the predicate will build a new relation file with name \textit{F}, which will be used to select another relation. After you used it, it should be removed.

For example:

\[
\text{students}(\text{Cname})\text{-select('course::cname} = = \text{Cname}',\{\text{Cname}\}, \text{f}),
\text{intersect('sc..f')}, \ldots
\]

When system executes the predicate \textit{students}, first it executes the predicate \textit{select(...,f)}, causes the system to select some tuples according to given qualification from source relation, transform them to facts and send to CPD program. In addition, the system set up a new relation file with the name \textit{f} which will be used to select another relation in following predicate \textit{intersect}.

d) Selecting the facts through intersection of two relation

Format: \text{intersect(S)}.

The predicate \textit{intersect} with one argument is meant for setting a new relation which is composed of the tuples from one relation correspond to those from another. In both relations there must be at least one domain with the same name. The argument must be an atom representing an expression of intersection of relations using \textit{Db++} syntax. The result relation is then transformed to CPD facts and sent to CPD program.

For example:

\[
[?- \text{intersect('student..sc')}.]
\]

After this query, the system sends a group of facts from relation \textit{student} to CPD program.

e) Selecting facts through intersection and setting up temporal file

Format: \text{intersect(S,F)}.

The predicate \textit{intersect} with two arguments is just like the predicate \textit{intersect} with one argument, except that the predicate will set up a new relation file with the name \textit{F} on secondary memory. The new file just is used to select another relation.
For example:

\[ ?- \text{intersect('student..sc', st)}. \]

After this query, the system sends a group of facts from relation \textit{student} into CPD program and sets up relation file \textit{st} on secondary memory.

f) Selecting the facts through difference of two relations

Format: \text{differ(S)}. 

The predicate \text{differ} with one argument is just like the predicate \text{intersect} with one argument, except that selecting the facts from the relation is through difference of two relations. The argument \text{S} represents expression of difference of two relations.

For example:

\[ ?- \text{differ('student--sc')}. \]

After this query, system produces a new relation which is the result of difference of \textit{student} and \textit{sc}, transforms the new relation to CPD facts and sends them to CPD program.

g) Selecting the facts through difference and setting up temporary file

Format: \text{differ(S,F)}. 

The predicate \text{differ} with two arguments is just like the predicate \text{differ} with one argument, except that the predicate will set up the new relation file with name \text{F} on secondary memory. The new file is just used to select another relation.

For example:

\[ ?- \text{differ('student--sc',st)}. \]

After this query, the system produces a new relation which is subset of \textit{student} relation, transforms it to CPD facts and sends them to CPD program; besides, the system sets up the new relation file with name \textit{st} on secondary memory.

3.5. Other fundamental operation of relations

The following predicates are used to deal with a new relation produced through operation of relation(s).

a) Projection of relation

Format: \text{project(S,F)}. 

The predicate \text{project} causes the system to form the new relation from source relation through projection of relation. The name of new relation is \text{F}. Then, the system transforms this relation to CPD facts, and sends them to CPD program. The arguments \text{S} and \text{F} separately represent expression of projection of relation and the name of new relation.

For example:

\[ ?- \text{project('student%%%no sname', newstu)}. \]

When system executes this predicate, a new relation \textit{newstu} which just consists of two columns from relation \textit{student} is produced, transformed to CPD facts and sent to CPD
program.

b) Union of two relations
   
   Format: union(S).
   
   The predicate union is meant for concatenating two relations together, transforming the result to CPD facts and sending them to CPD program. Both relations must have the same degree. The domains need not have the same names in both relations, but the data types must be compatible with one another. The argument must be an atom representing the expression of union of two relations.
   
   For example:
   
   ?- union('student + + newstudent').
   
   After the system executes union, the result of union of two relations student and newstudent is transformed to CPD facts and sent to CPD program. The result is a relation with the same name as that of student.

c) Join of two relations
   
   Format: join(S,F).
   
   The predicate join enables the system to cross-reference two relations, transform the result to CPD facts with the name F and send them to CPD program. The argument S represents the expression of join of two relations, F represents the name of facts.
   
   For example:
   
   ?- join('course**sc', cs).
   
   After this query, the system cross-reference relation course and sc, transform the result to CPD facts and send these facts with the name cs to CPD program.

d) Product of two relations
   
   Format: product(S,F).
   
   The predicate product is meant for forming new tuples from two relations by concatenating every tuple from the first relation with each tuple in turn from the second relation, transforming these tuples to CPD facts, and sending these facts to CPD program.
   
   For example:
   
   ?- product('student`course', sc).
   
   After this query, the system produces the tuples through Product of relations student and course, transforming these tuples to the facts with the name sc and sends them to CPD program.

4. Examples of CPD programs

   In this section, we use the following example to illustrate how CPD programs can be written.
Consider the CPD program:

\[
\text{grade}(\text{Sno},\text{Cname}) :- \text{course}(\text{Cno},\text{Cname},\text{Tname},\text{Precno}), \\
\quad \text{sc}(\text{Sno},\text{Cno},\text{Grade}), \\
\quad \text{write}(\text{Grade}), \text{nl}. \quad \text{(rule 1)}
\]

\[
\text{students}(\text{Cname}) :- \text{course}(\text{Cno},\text{Cname},\text{Tname},\text{Precno}), \\
\quad \text{sc}(\text{Sno},\text{Cno},\text{Grade}), \\
\quad \text{student}(\text{Sno},\text{Sname},\text{Dept}), \\
\quad \text{write}(\text{Sname}), \text{nl}, \text{fail}. \quad \text{(rule 2)}
\]

\[
\text{precourse}(\text{Cname}) :- \text{course}(\_,\text{Cname},\_,\text{Precno}), \\
\quad \text{p_precourse}(\text{Precno}). \quad \text{(rule 3)}
\]

\[
\text{p_precourse}(0). \\
\text{p_precourse}(\text{Cno}) :- \text{course}(\text{Cno},\text{Cname},\_,\text{Precno}), \\
\quad \text{write}(\text{Cname}), \text{nl}, \text{p_precourse}(\text{Precno}). \quad \text{(rule 4)}
\]

\[
\text{student}(4001,\text{John},\text{computer}). \\
\text{student}(4002,\text{Alfred},\text{mathematics}). \\
\text{student}(4003,\text{Mary},\text{electricnic}). \\
\text{student}(4004,\text{Peter},\text{chemistry}). \\
\quad \text{.....} \quad \text{(facts)}
\]

\[
\text{course}(100,\text{calculus},\text{david},90). \\
\text{course}(110,\text{computer\_organization},\text{ted},80). \\
\text{course}(120,\text{database},\text{alfred},110). \\
\quad \text{.....} \quad \text{(facts)}
\]

\[
\text{sc}(4001,100,a). \\
\text{sc}(4001,110,c). \\
\text{sc}(4002,120,b). \\
\quad \text{.....} \quad \text{(facts)}
\]

Now, suppose we store the facts in Db++. That is, we create relations \text{student}, \text{course} and \text{sc}, and insert the tuples (facts) into the relations. The column names of relation \text{student} are \text{sno}, \text{sname} and \text{dept}, the column names of relation \text{course} are \text{cno}, \text{cname}, \text{tname} and \text{precno}, the column names of relation \text{sc} are \text{sno}, \text{cno} and \text{grade}. Then we can write CPD program in different ways:
CASE 1

(1)  ?- select(student).  (**)
(2)  ?- select(course). (**)
(3)  ?- select(sc). (**)

(4)  grade(Sno,Cname):-course(Cno,Cname,Tname,Precno),
    sc(Sno,Cno,Grade),
    write(Grade),nl.

(5)  students(Cname):-course(Cno,Cname,Tname,Precno),
    sc(Sno,Cno,Grade),
    student(Sno,Sname,Dept),
    write(Sname),nl,fail.

(6)  precourse(Cname):-course(_,Cname,_,Precno),
     p_precourse(Precno).

(7)  p_precourse(0).
     p_precourse(Cno):-course(Cno,Cname,_,Precno),
     write(Cname),nl,p_precourse(Precno).

In this CPD program, statements (1)-(3) are requests from CPD to the Db++. Each is treated as a distinct transaction by Db++. (We note that clauses beginning with the question mark "?" are treated as goals.) Db++ executes the select calls in (1)-(3) and sends the results to CPD which stores them as assertions in its work space. Then we can ask some questions:

(8)  ?- grade(4001, database).

Prolog executes this goal, answers our question and displays the results on screen:

b
yes
?-

Then we can continue our questions:

(9)  ?- students(database).

Prolog executes this goal, displays all student names who take database course on screen:

john
mary
...
no
?-

Then we can continue our questions.

We can use following form to write above program:
CASE 2

(1) grade(Sno,Cname):-select(course), select(sc),
    course(Cno,Cname,Tname,Precno),
    sc(Sno,Cno,Grade), write(Grade), nl.

(2) students(Cname):-select(student), select(course),
    select(sc),
    course(Cno,Cname,Tname,Precno),
    sc(Sno,Cno,Grade),
    student(Sno,Sname,Dept),
    write(Sname), nl, fail.

(3) precourse(Cname):-select(course),
    course(_,Cname,_,Precno),
    p_precourse(Precno).

(4) p_precourse(0).
    p_precourse(Cno):-course(Cno,Cname,_,Precno),
    write(Cname), nl, p_precourse(Precno).

CASE 2 is functionally the same as CASE 1.

Although the above CPD program is functionally equal to Prolog program and relation files student, course and sc which are shared data file can be used by the other users, we think the above CPD program is not good because CPD work space can be overflowed by very big relation sc or student. In order to resolve this problem, we should use some qualifications to limit the size of relation which will be sent to CPD work space. Then we can have following CPD program:

CASE 3

(1) grade(Sno,Cname):-select('course::cname = = Cname', [Cname]),
    course(Cno,Cname,Tname,Precno),
    select('sc::sno = = Sno', [Sno]),
    sc(Sno,Cno,Grade),
    write(Grade), nl.

(2) students(Cname):-select('course::cname = = Cname', [Cname]),
    course(Cno,Cname,Tname,Precno),
    select(sc),
    sc(Sno,Cno,Grade),
    select(student),
    student(Sno,Sname,Dept),
    write(Sname), nl, fail.
(3) precourse(Cname):-select(course),
    course(_,Cname,_,Precno),
    p_precourse(Precno).
    p_precourse(0).
    p_precourse(Cno):-course(Cno,Cname,_,Precno),
    write(Cname),nl,p_precourse(Precno).

(4) ?- grade(4001,database).
...
(5) ?- students(database).
...
(6) ?- precourse(pascal).
...

In this CPD program, because we use some qualifications in predicate select, the size of CPD work space taken by facts becomes smaller than CASE 1 or CASE 2. But in rule(2), the relations student and sc are still wholly sent to CPD work space and must take so much CPD work space because the argument of predicate students is not a argument of student and sc and we can not directly use predicate select to select facts and limit CPD work space. Fortunately, we have some predicates for operation of relation, which can be used to limit the size of CPD work space taken by student and sc.

CASE 4

(1) grade(Sno,Cname):-select('course::cname == = Cname',[Cname]),
    course(Cno,Cname,Tname,Precno),
    select('sc::sno == = Sno',[Sno]),
    sc(Sno,Cno,Grade),
    write(Grade),nl.

(2) students(Cname):-select('course::cname == = Cname',[Cname],file),(**)
    course(Cno,Cname,Tname,Precno),
    intersect('sc..file',file1),
    sc(Sno,Cno,Grade),
    intersect('student..file1'),
    student(Sno,Sname,Dept),
    write(Sname),nl,fail.

(3) precourse(Cname):-select(course),
    course(_,Cname,_,Precno),
    p_precourse(Precno).

(4) p_precourse(0).
    p_precourse(Cno):-course(Cno,Cname,_,Precno),
    write(Cname),nl,p_precourse(Precno).

(5) ?- students(calculus).
...

We think this is a very elegant CPD program, it saves so much CPD work space and allows a very big CPD program to be run in a limited CPD work space. We also want to point
that the rule(4) is a recursive rule which is not permitted in Db++. Obviously, CPD has
the advantages of Db++ and Prolog.

5. Conclusion

In this paper we have described the architecture and implementation of CPD system
and have introduced the structure of CPD program and special predicates which take the
responsibility for interface between Prolog and Db++. The syntax of an argument of the
predicates is like the syntax of the Db++ language. This is because the tools for managing
data bases are already available in Db++.

Because CPD has the advantages of both Prolog and Db++, the CPD system can
handle many different problems that involve recursive or non-recursive views, transactions,
or different data distributions among Prolog and Db++, etc. It will be a practical and
powerful tool for designing very large knowledge-based systems which can store shared
data (facts base) on secondary memory.

Now I am implementing and improving the CPD system; much work has been done
but more work remains to be done.
References

[BaM73] Battani, G., and Meloni, H., "Interpreter du Langage de Programmation PROLOG", Groupe d'Intelligence Artificielle, Marseille-Luminy, Manual for the first implementation of PROLOG.


