Report Number: WUCS-87-12

1987-06-01

PRODB: An Experimental Generalized Database System User's Manual

Guillermo R. Simari

The following notes document in a succinct manner the use of the system PRODB. The system is still evolving and several new features are in the process of being added. PRODB is a prototype system that is being used as an exploration vehicle of the possible extensions to the relational model through logic programming. The system consists of a relational database system having a relational algebra type language as a query language. It is written in Prolog and it extends the capabilities of Prolog predicates with the relational algebra operators for handling the database structure. The database system currently... **Read complete abstract on page 2.**

PRODB: An Experimental Generalized Database System User's Manual

Guillermo R. Simari

Complete Abstract:

The following notes document in a succinct manner the use of the system PRODB. The system is still evolving and several new features are in the process of being added. PRODB is a prototype system that is being used as an exploration vehicle of the possible extensions to the relational model through logic programming. The system consists of a relational database system having a relational algebra type language as a query language. It is written in Prolog and it extends the capabilities of Prolog predicates with the relational algebra operators for handling the database structure. The database system currently provides the set theoretic operations (union, intersection, difference and product), join, project and select. The relations can be defined over any domain that can be defined using a Prolog predicate. Primary keys for the relations should be defined and their uniqueness is maintained. Facilities exist for defining assertions over the contents of the relations. Those assertions are predicates that constrain the set of legal tuples that can be part of a relation. The consistency of the database is checked and maintained with respect to the set of assertions and domains. Actions are daemons associated with the update operations and can be defined by the user as Prolog predicates that will be triggered by those operations. Actions can involve any Prolog (or PRODB) predicate and allow the definition of side effect behavior when an update operation is performed. A transaction can be started and all the operations executed inside of it can be started and all the operations executed inside of it can be rolled-back to the point when the transaction was started. Time is associated with the creation of the relations and with the insertion of tuples in the form of a time-stamp. That time-stamp is shown in the printouts of schemas and, by using the appropriate predicate, also in the printouts of relations. A predicate for selecting tuples according with their time of insertion is defined. There are several predefined comparison operators and predicates. The usual aggregate predicates are available including some handling time. A built-in help facility is available (see the Miscellaneous paragraph) and appropriate error messages are issues whenever an error condition is reached. The system is capable of handling several databases at the same time and all the relational operations can take arguments from different databases. Also the referential integrity can be enforced across databases. Different scenarios can be developed in that way and, in conjunction with the transaction facility, the seed for an exploration capability is in place. First the operation handling complete databases as objects are presented. Then, the relative level operations for handling tuples and the syntax of the implemented relational algebra operations for handing tuples and the syntax of the implemented relational algebra operators is introduced. After that, there is a brief discussion over how to define assertions and actions for the database. Finally the time related predicates, transaction facility and some miscellaneous predicates are described.

PRODB: AN EXPERIMENTAL GENERALIZED DATABASE SYSTEM USER'S MANUAL

Guillermo R. Simari

WUCS-87-12

June 1987

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Table of Contents

1. Overview

The following notes document in a succinct manner the use of the system PRODB. The system is still evolving and several new features are in the process of being added.

PRODB is a prototype system that is being used as an exploration vehicle of the possible extensions to the relational model through logic programming. The system consists of a relational database system having a relational algebra type language as a query language. It is written in Prolog and it extends the capabilities of Prolog predicates with the relational algebra operators for handling the database structure.

The database system currently provides the set theoretic operations (union, intersection, difference and product), join, project and select. The relations can be defined over any domain that can be defined using a Prolog predicate. Primary keys for the relations should be defined and their uniqueness is maintained.

Facilities exist for defining assertions over the contents of the relations. Those assertions are predicates that constrain the set of legal tuples that can be part of a relation. The consistency of the database is checked and maintained with respect to the set of assertions and domains.

Actions are daemons associated with the update operations and can be defined by the user as Prolog predicates that will be triggered by those operations. Actions can involve any Prolog (or PRODB) predicate and allow the definition of side effect behaviour when an update operation is performed.

A transaction can be started and all the operations executed inside of it can be rolled-back to the point when the transaction was started. Time is associated with the creation of the relations and with the insertion of tuples in the form of a time-stamp. That time-stamp is shown in the printouts of schemas and, by using the appropriate predicate, also in the printouts of relations. A predicate for selecting tuples according with their time of insertion is defined. There are several predefined comparison operators and predicates. The usual aggreagate predicates are available including some handling time. A built-in help facility is available (see the Miscellaneous paragraph) and appropriate error messages are issued whenever an error condition is reached.

The system is capable of handling several databases at the same time and all the relational operations can take arguments from different databases. Also the referential integrity can be enforced across databases. Different scenarios can be developed in that way and, in conjunction with the transaction facility, the seed for an exploration capability is in place.

First the operations handling complete databases as objects are presented. Then, the relation level operations for handling tuples and the syntax of the implemented relational algebra operators is introduced. After that, there is a brief discussion over how to define assertions and actions for the database. Finally the time related predicates, transaction facility and some miscellaneous predicates are described.

2. Database Handling

The system works over one or more databases loaded in memory. These databases are saved to and restored from secondary storage. A database is considered to be a set of relations together with their schemas, actions, assertions and foreign-key definitions. A relation is a set of tuples and this set is a subset of the Cartesian product of a finite number of domains. The schema, defined when the relation is created, gives the definition of the relation in terms of which domains are involved and in which order. This section also describes the subset of that set of domains that form the primary key. Schemas, actions, assertions and foreign-keys will be described later. Databases can be created, saved and restored from files. A brief description of the related commands follows.

2.1. default_db

Given that commands handling relations should specify to which database that relation belongs there is a way of implicitly refer to one database when no database name in mentioned. That implicit database name can be obtained using the predicate default_db with a variable as argument. For example,

```
default_db(Db_name).
```

the variable Db_name will be instantiated to the name of the default database. If there is no default database set the command will fail.

2.2. set_default_db

This command can be used for setting the default database. The command is

```
set_default_db(db_name).
```

where db_name is the desired new default database. The default database name will be set automatically to the name of the first database loaded, see the command load_db.

2.3. list_dbs

Prints a list of the currently loaded databases and the name of the default database. If given a parameter will return the list of loaded databases. The format is

```
list_dbs.
```

and the output will be,

```
Default Database : spj

Databases Loaded :

    spj
    new_spj
    parts_suppliers
```

and with a parameter,

```
list_dbs(Db_List).
```

will return the list of loaded databases in the variable Db_List, the first element of the list will be the default database if there is one set. For example,

```
list_dbs(Db_List).
```

will instantiate Db_list with the list [spj, new_spj, parts_suppliers].

2.4. create_db

It will create a new database. The format is

```
create_db(db_name).
```

where db_name is the name of the database (and of the main file associated with it). Also if there is no other database loaded db_name become the default database name. For example,

```
    create_db(my_db).
```

will create two files, my_db and my_db.pred where the database and the related assertions will be stored. If the database was already loaded or another database with the same name exists in the current directory the command will fail and an error message will be displayed.

2.5. save_db

This command saves the contents of a database loaded in memory to a file. The format is

```
    save_db(db_name).
```

where db_name is the name of the database. If the database to be saved is the default database is not necessary to specify it, and it will suffice to use the command save_db without any parameter. For example, if the default database is my_db then

```
    save_db.
```

will write the contents of the default database in the file my_db, if the file already exists will be renamed to my_db˜, producing a backup copy of the previous state. Alternatively, if the file my_db˜ exists it will be deleted first.

The argument to the command can be a list of databases currently loaded in memory, and the effect will be the same as to apply the command save_db to each database sequentially. For example,

```
    save_db([db1, db2, db3]).
```

will save db1, db2 and db3 provided those databases where loaded in memory.

A different alternative for save_db is

```
save_db(db1, db2).
```

This command will save the database db1 to the database db2 as a file. The name db2 must be a new database name.

2.6. load_db

This predicate is used for loading a database from a file to memory. If the same database was loaded in memory it will first delete that database, and then will load the new one from the corresponding file. Predicates in the predicate file will not be deleted. If there is a collision, two predicates with the same name, it will be a redefinition. The format is,

```
load_db(db_name).
```

where db_name is the name of an existing database stored as a file in the current directory. For example,

```
load_db(my_db).
```

The system will check if my_db was previously loaded in memory and if that is the case will first delete it from memory and then will load my_db to memory. If no other database is loaded in memory my_db will become the default database.

The argument to the command can be a list containing databases names currently loaded in memory, and the effect will be the same as to apply the command load_db to each database sequentially. For example,

```
load_db([db1, db2, db3]).
```

will restore db1, db2 and db3.

2.7. copy_db

Copy_db is used for creating a duplicate, in memory, of an existing database. The format is,

```
copy_db(db_name1, db_name2).
```

where db_name1 is the source database and db_name2 is the target, i.e. the new database. If the user wants to save the newly created database he/she must save it using a save_db command.

2.8. destroy_db

Destroy_db will delete from memory the content of a given database. The file will remain as it was saved before issuing the command. The format is,

```
destroy_db(db_name).
```

where db_name is the name of the database to be removed from memory.
The argument to the command can be a list containing databases names currently loaded in memory, and the effect will be the same as to apply the command destroy_db to each database sequentially. For example,

```
destroy_db([db1, db2, db3]).
```

will destroy db1, db2 and db3.

The argument to the command can be a list containing databases names currently loaded in memory, and the effect will be the same as to apply the command destroy_db to each database sequentially. For example,

```
destroy_db([db1, db2, db3]).
```

will destroy db1, db2 and db3.

2.9. print_db

A complete print out of a database can be obtained using print_db. This command prints the relations as tables using the format information contained in the schema of the relations. The schemas can be printed using the command print_schemas_db, see bellow. The format is,

```
print_db(db_name).
```

Where db_name is the name of a database loaded in memory. The argument to the command can be a list of databases currently loaded in memory, and the effect will be the same as to apply the command print_db to each database sequentially. For example,

```
print_db([db1, db2, db3]).
```

will print the content of db1, db2 and db3. Any of the previous commands can be used with a second parameter representing the name of file where the output will be redirected, as

```
print_db(db_name, output).
```

2.10. print_schemas_db

This command prints the schema information of each relation in the requested database. The information includes the description of attribute names and domains, format, assertions, actions and foreign keys. The format is,

```
print_schemas_db(Db).
```

Where Db is the name of a database loaded in memory. For example, the command

```
print_schemas_db(spj).
```

will produce the following output,

```
Relation Schemas for spj

spj:suppliers (created : 07/09/86 22:50:16)
          [[s_nr,string]]
          [[sname,string],[status,int],[city,string]]
          [8,8,8,8]

spj:projects  (created : 07/09/86 22:50:16)
          [[j_nr,string]]
          [[jname,string],[city,string]]
          [4,10,7]

spj:equip     (created : 07/25/86 14:20:55)
          [[eq_nr,string]]
          [[eqname,string],[part_nr,string]]
          [6,10,8]

spj:parts     (created : 07/09/86 22:50:16)
          [[p_nr,string]]
          [[pname,string],[color,string],[weight,int],[city,string]]
          [4,8,7,6,7]
          weight>=10
```

see the paragraph covering print_schema for the description of the information printed.

The argument to the command can be a list of databases currently loaded in memory, and the effect will be the same as to apply the command print_schemas_db to each database sequentially. For example,

```
print_schemas_db([db1, db2, db3]).
```

will print the schema information for db1, db2 and db3. The command print_schemas_db will print the schema information for the default database.

3. Relation Handling

Relations are subsets of the Cartesian product of certain domains. These domains are arbitrary and the only restriction is that their definition can be associated with a Prolog predicate. That is, a domain is a set and a Prolog predicate must be defined for testing if a given value is a member of this domain.

A relation schema (or schema simply) is the device used to describe the relation characteristics and is a finite, ordered, set of attribute names. In correspondence with every attribute name there is a domain. The system provides a predicate for changing the attribute names, see the rename_attribute predicate, but the associated domains should remain the same. The order in which the attribute names appear is fixed at creation time, and cannot be changed, but see the reformat_rel operation ahead.

In PRODB, the set of attribute names is separated in two subsets. One, appearing first, includes the attribute names that form the key. This subset cannot be empty. The second includes the attribute names for the non-key attributes, and can be empty.

Several relation handling predicates are defined. The commands create_rel, delete_rel, print_rel, reformat_rel and rename_rel are provided. The commands print_schema, sort_rel, copy_rel_tuples, count_rel, count_rel_tuples, tprint_rel are also availables. A detailed explanation of them follows.

3.1. Notation

The name relation_name in the following commands must include the reference to the database which it is referring to, i.e. is a pair db_name : relation_name where the character ":" is mandatory. For example, to refer to the relation "parts" in the database "spj" the notation must be "spj : parts". If the reference is related to a relation in the default database, (see previous paragraph), it is enough to use just the name of the relation without the character ":".

When the command accepts a list of relations the list can take one of the following formats.

- [db_name_1:rel_name_1, db_name_2:rel_name_2, ..., db_name_n:rel_name_n]
- db_name:[rel_name_1, rel_name_2, ..., rel_name_n]

In the first case if the name of a database is omitted the name of the relation must correspond to one in the default database. In the second case all relations are contained in the database named "db_name".

3.2. create_rel

Adds a relation to the database. Its function is to give a description of which attribute names are involved and which domains are associated to them. Also establish the order in which the attributes should appear. The general form of the command is,

```
    create_rel(relation_name,
               Key_attributes_list,
               Non-Key_attributes_list,
               format_list).
```

where Key_attributes_list and Non_Key_attributes_list have the form

```
    [[att_name_1, domain_1],...,[att_name_n, domain_n]]
```

att_name_i is the name of the attribute (must be unique in the schema) and domain_i is the name of a data type (predicate) that will be used in checking the values of the attributes. Some data types are implemented as predicates in the system (string, real, natural, int, alpha, alpha_num). The format_list is optional and should be a list of positive integers, it must be of the same length as the sum of the lengths of the attribute lists. Both characteristics are tested and enforced. If the list is not included, a default format with eight spaces for each field is created and associated with the relation.

For example, the following command will create the relation parts in the database spj.

```
    create_rel(spj : parts,
               [ [p_nr, string] ],
               [ [pname, string], [color, string],
                 [weight, int], [city, string] ]).
```

i.e., creates the schema for the relation named parts in the database spj. If spj is the default database it would be enough to issue the following command,

```
create_rel( parts,
          [ [p_nr, string] ],
          [ [pname, string], [color, string],
            [weight, int], [city, string] ]).
```

This command also creates a default printing format [8,8,8,8,8]. Optionally the create_rel command can contain information about the format as in

```
create_rel(spj : parts,
          [ [p_nr, string] ],
          [ [pname, string], [color, string],
            [weight, int], [city, string] ],
          [4,8,7,6,7]).
```

the format can be modified (see the modify_format predicate)

3.3. delete_rel

The command delete_rel eliminates from the current database both the schema and the values of the relation specified. The format is,

```
delete_rel(Arg).
```

where Arg is a single relation name or a list of them (see the Notation paragraph). For example,

```
delete_rel(spj : parts).
```

or, if spj is the default database

```
delete_rel(parts).
```

will eliminate the relation parts from the database. The predicate alternatively can take a list of relation names as argument.

For example,

```
    delete_rel(spj : [parts, suppliers]).
```

or, if spj is the default database

```
    delete_rel([parts, suppliers]).
```

will delete both relations from the database spj.

3.4. print_rel

The print_rel command prints, formatted, the contents of relations. The format is,

```
    print_rel(Arg).
```

where Arg is a single relation name or a list of them (see the Notation paragraph). For example,

```
    print_rel(spj : parts).
```

or, if spj is the default database

```
    print_rel(parts).
```

will produce the following output.

```
DB: spj  Relation : parts (created : 07/09/86 22:50:16)

--------------------------------------------
p_nr |    pname |    color | weight |    city
-----+----------+---------+--------+---------
  p1 |      nut |      red |     12 | london
  p2 |     bolt |    green |     17 |  paris
  p3 |    screw |     blue |     17 | athens
  p4 |    screw |      red |     14 | london
  p5 |      cam |     blue |     12 |  paris
  p6 |      cog |      red |     19 | london
--------------------------------------------

6 tuples in the relation
```

using the format [4,8,7,6,7] specified before for this relation when was created. Formats can be changed using the command modify_format.
The command,

```
print_rel(spj : [parts, suppliers]).
```

or, if spj is the default database

```
print_rel([parts, suppliers]).
```

will print the contents of the relations parts and suppliers from the database spj. A related command is print_db. This command will print all the relations in a given database.

3.5. tprint_rel

This command allows to print the time stamp associated with each tuple at the insert moment. The format is

```
tprint_rel(rel_name).
```

where Arg is a single relation name or a list of them (see the Notation paragraph). For example,

```
tprint_rel(spj : projects).
```

or, if spj is the default database

```
tprint_rel(projects).
```

will print the following information,

```
DB: spj  Relation : projects (created : 07/09/86 22:50:16)

-------------------------------------------------
 time of insertion | j_nr |      jname |    city
------------------+------+------------+---------
 07/08/86 21:32:24 |   j1 |     sorter |   paris
 07/09/86 22:54:33 |   j2 |      punch |  athens
 07/09/86 22:55:21 |   j3 |     reader |  london
 07/09/86 22:56:17 |   j4 |    console |  athens
 07/09/86 22:56:56 |   j5 |   collator |  london
 07/09/86 22:56:59 |   j6 |   terminal |   paris
 07/09/86 22:57:19 |   j7 |       tape |  london
-------------------------------------------------

7 tuples in the relation
```

3.6. print_schema

This command prints the complete information about a relation, or a list of them. The information includes the description of attribute names and domains, format, assertions, actions and foreign keys. The format is,

```
print_schema(Arg).
```

where Arg is the name of a relation in the database, or a list of them, see the paragraph about notation at the beginning of this section. A related command is print_schemas_db, see description in the Database Handling section.

3.7. rename_rel

The rename_rel command takes two arguments, the present name of a relation and the name to which should be changed, and renames the relation in that way. The format is,

```
rename_rel(Old, New).
```

the schema, tuples, actions, assertions and foreign keys are also renamed. In relation named Old becomes renamed New in every sense. Old and New can be composed of a pair db_name : relation_name and the relations can be in different databases. Both names must be homogeneous, i.e. both referring to the default database implicitly or both specifying the databases explicitly. For example,

```
rename_rel(parts, auto_parts).
```

will rename the relation parts as auto_parts in the default database. The same command in different databases will be,

```
rename_rel(spj : parts, auto_spj : auto_parts).
```

will rename the relation parts as auto_parts from the spj database to the auto_spj database. The relation parts is removed from memory in both cases.

3.8. reformat_rel

This predicate is used for changing the distribution of the attributes among the key and non–key attributes in a relation schema. It can only modify the distribution but not the order of them.

Note that some of the tuples can now be rejected because of key-attribute value duplication. This is only possible if the new key–attributes are a subset of the key-attributes in the previous definition.

The general form of the command is,

```
reformat_rel(relation_name,
          Key_attributes_list,
          Non-Key_attributes_list).
```

where Key_attributes_list and Non_Key_attributes_list have the same form as in create_rel. The relation_name is as above in the create_rel predicate. The format will not be changed. For example,

```
reformat_rel(spj : parts,
        [ [p_nr, string] ], [pname, string] ],
        [ [color, string],
          [weight, int], [city, string] ]).
```

assuming the example in the paragraph for create_rel as starting point, the command will change the schema for the relation parts, including the pname attribute as part of the key. Again the database reference can be omited if the relation resides in the default database.

3.9. sort_rel

It produces a sorted relation out of another relation. The ordering is produced following the criteria given in the list Comp_List (see below). The general format is

```
sort_rel(rel_name1, rel_name2, Comp_List).
```

where rel_name1 is the relation to be sorted, rel_name2
is the name of a non—existing relation where the result will be stored and Comp_List has the form

```
[ [att_1, c_op1], ...,[att_n, c_opn] ]
```

where att_i is an attribute name in the schema of rel_name1 and c_opi is a order comparison operator defined over the members in the domain of such attribute. The order in which the attributes appear is the order in which are considered, i.e., if the att_1 are equal then the att_2 are compared and so on (lexicographic order). The schema for rel_name2 will be the same as for rel_name1. The relations names used must follow the restrictions mentioned before in the Notation paragraph. For example,

```
    sort_rel(spj : parts, auto_spj : parts_ordered,
             [ [p_nr, > ], [p_name, *> ] ]).
```

or , if parts reside in the default database and parts_ordered should be stored in the same database

```
    sort_rel(parts, parts_ordered,
             [ [p_nr, > ], [p_name, *> ] ]).
```

the command will produce, in parts_ordered, in the specified database, a sorted version of parts, coming from the requested database, using the attributes p_nr and p_name and the operators > and *> (this is the system defined grater than operator for the system defined string data type).

3.10. copy_rel

The copy_rel command takes two arguments, the name of a relation and a new relation name, the existing relation will be copied to the new relation. The format is,

```
    copy_rel(Old, New).
```

only the schema and tuples are copied, actions, assertions and foreign keys are not copied. Both names must be homogeneous, i.e. both referring to the default database implicitly or both specifying the databases explicitly. For example,

```
    copy_rel(parts, auto_parts).
```

will copy the relation parts to auto_parts in the default database. The same command in different databases will be,

```
    copy_rel(spj : parts, auto_spj : auto_parts).
```

will copy the relation parts to auto_parts from the spj database to the auto_spj database.

3.11. count_rel

It counts the number of tuples in a relation. The format is

```
    count_rel(db_name : rel_name, Tuple_nr).
```

or if rel_name is the name of a relation in the default database

```
    count_rel(rel_name, Tuple_nr).
```

where rel_name is the name of an existing relation in the specified database and Tuple_nr is a variable which will contain the number of tuples.

3.12. copy_rel_tuples

This predicate is used for copying the tuples in a given relation into another. The second relation can exists previously and can be non empty. Conflicting tuples are not inserted and in that respect behaves as the insert predicate, see the tuple handling paragraph. A possible application of this predicate is to change the schema of a given relation. In such case, the user should create a relation with the schema desired and then the old one can be copied to it. Then, the old relation can be deleted and the new renamed. The format is,

```
    copy_rel_tuples(Rel1,Rel2).
```

where Rel1 is the source and Rel2 is the target. As in the case of the rename_rel predicate Rel1 and Rel2 can be composed of a pair db_name : relation_name and the relations can be in different databases. Both names must be homogeneous, i.e. both referring to the default database implicitly or both specifying the databases explicitly. For example,

```
copy_rel(parts, auto_parts).
```

will copy the relation parts to auto_parts in the default database. Or, if the copy is across different databases spj and auto_spj,

```
copy_rel(spj : parts, auto_spj : auto_parts).
```

3.13. count_rel_tuples

This predicate counts the number of tuples matching a generic tuple. A generic tuple is one which has some variables included. The format is

```
count_rel_tuples(db_name : rel_name, Tuple_nr, Generic).
```

or if rel_name is the name of a relation in the default database

```
count_rel_tuples(rel_name, Tuple_nr, Generic).
```

where rel_name is the name of an existing relation in the specified database , Tuple_nr is a variable which will contain the number of matching tuples and Generic is a generic tuple. For example, if we apply the predicate to the relation parts

```
count_rel_tuples(spj : parts,
                 Red_parts_nr,
                 [X, Y, "red", Z, W]).
```

will return Red_parts_nr instantiated to 3 (see previous example in print_rel).

3.14. modify_format

This command can be used for changing the format information. The general form is

```
    modify_format(db_name:rel_name, new_format).
```

or, if the relation is in the default database

```
    modify_format(rel_name, new_format).
```

where "rel_name" is a relation name and "new_format" is a list of positive integers of the same length as the previous one. For example,

```
    modify_format(spj:parts, [6,8,7,7,7]).
```

will change the format [4,8,7,6,7], the previous value, to [6,8,7,7,7] for the relation parts.

3.15. rename_attribute

This predicate is used for changing the names of attributes in the schema of a relation. The format is,

```
    rename_attribute(db_name:rel_name, old, new).
```

or, if the relation is in the default database

```
    rename_attribute(rel_name, old, new).
```

where "rel_name" is the name of a relation in the database "db_name", "old" is the name of an existing attribute in the schema and "new" is the new name for that attribute. For example, if the following is the schema for the relation parts,

```
parts       (created : 07/09/86 22:50:16)
            [[p_nr, string]]
            [[pname, string], [color, string], [weight, int]...]
            [4,8,7,6,7]
            weight>=10
            if_removed : write_del
```

and the following command is issued,

```
rename_attribute(parts, pname, part_name).
```

the schema will be changed to,

```
parts       (created : 07/09/86 22:50:16)
            [[p_nr, string]]
            [[part_name, string], [color, string], [weight, int]...]
            [4,8,7,6,7]
            weight>=10
            if_removed : write_del
```

4. Tuple Handling

After creating a database, and having the relation schema defined, the tuples can be stored in the relation. Four primitive commands can be used for handling the tuples. These commands are insert, remove, modify and get_tuple. Assertions, actions and foreign keys are related concepts described elsewhere.

4.1. insert

The predicate insert_tuple adds tuples to a given relation. The format is,

```
insert(db_name : rel_name, tuple).
```

or, if db_name is the default database

```
insert(rel_name, tuple).
```

where db_name : rel_name form the pair that indicates the database and the relation where the tuple will be stored, and "tuple" is the tuple to be inserted. For example,

```
insert(spj : parts,
       [p3, screw, blue, 17, athens]).
```

or, if spj is the default database

```
insert(parts,
       [p3, screw, blue, 17, athens]).
```

adds the tuple to the relation parts in the database spj. The uniqueness of the tuples in the relation is maintained. The components of the tuple are checked using the domain information provided in the schema i.e. every component of the tuple should belong to the domain with it associated.

If there are assertions defined, the tuple is tested against them and if any of them is violated the tuple is rejected.

If there is an action specified it will be executed after the insertion. Assertions and actions will be described later.

4.2. get_tuple

The predicate get_tuple will return the value of non-key attributes for a tuple with a given key. The format is,

```
get_tuple(db_name : rel_name, key, Tuple).
```

or, if db_name is the default database

```
get_tuple(rel_name, key, Tuple).
```

where rel_name is the name of a relation in the database db_name, "key" is a list containing the values corresponding to the key-attributes of a tuple stored in that relation, and "Tuple" is the complete tuple that is identified by Key. For example,

```
get_tuple(spj : parts, [p1], Tuple).
```

or, if spj is the default database

```
get_tuple(parts, [p1], Tuple).
```

will instantiate Tuple to [p1, nut, red, 12, london] from the relation parts in the spj database. If there is an action specified it will be executed and can be defined in such a way that the recovered tuple will contained calculated attributes using the actually contained in the relation.

4.3. modify

The modify command is used for making changes in a given tuple. The format is,

```
    modify(db_name : rel_name, key, tuple).
```

or, if db_name is the default database

```
    modify(rel_name, key, tuple).
```

where rel_name is the name of a relation in the database db_name, "key" is a list containing the values corresponding to the key-attributes of a tuple stored in that relation, and "tuple" is the tuple that will replace the identified by "key". If there is an action specified it will be executed after the completion of the command. For example,

```
    modify(spj : parts,
            [p1], [p1, nut, black, 12, london]).
```

or, if spj is the default database

```
    modify(parts,
            [p1], [p1, nut, black, 12, london]).
```

will replace the tuple [p1, nut, red, 12, london] for the tuple [p1, nut, black, 12, london] in the database spj.

The modified tuple is handled as in the insertion case, this implies that all the checking done at that time is also performed. If the modified tuple is rejected the original one remains unchanged. Note again that the key-attribute for the tuple being modified is the only one included.

4.4. remove

The remove command permits to eliminate a tuple from the relation and has the form

```
    remove(db_name : rel_name, key).
```

or, if db_name is the default database

```
    remove(rel_name, key).
```

where rel_name is the name of a relation in the database and "key" is a list containing the values corresponding to the key-attributes of a tuple stored in that relation. If there is an action specified it will be executed after the completion of the command. For example,

```
    remove(spj : parts, [p1]).
```

or, if spj is the default database

```
    remove(parts, [p1]).
```

will remove the tuple [p1, nut, red, 12, london] from the database spj. Note that only the key-attribute is specified.

4.5. match_tuple

This command will instantiate a variables contained in generic tuple with values coming from tuples stored in a given relation from a given database. (A generic tuple is a tuple that contains some variables in it, i.e. a pattern). Note that can be multiple tuples matching the generic tuple. In the first call the command will return the values corresponding to the first asserted matching tuple. Further values can be obtained by failing the command using the PROLOG fail predicate. Warning: no error message is produced when no matching tuple exists, the predicate just fail. The format is,

```
    match_tuple(db_name : rel_name, Tuple_Patt).
```

or, if the relation is in the default database

```
    match_tuple(rel_name, Tuple_Patt).
```

where rel_name is the name of a relation in the database db_name, and Tuple_Patt is the generic tuple. For example,

```
    match_tuple(spj : parts, [p1, X, red, Y, london]).
```

or, if spj is the default database

```
match_tuple(parts, [p1, X, red, Y, london]).
```

will instantiate X = nut and Y = 12.

4.6. do_insert

This predicate, is a rudimentary facility provided for helping in the task of loading the relations saving some typing work. The format is,

```
do_insert.
```

The predicate begins asking the name of the database and the relation as a pair "db_name : rel_name", see the Notation paragraph, and then the tuples to be inserted. When no more tuples are to be inserted, the word exit ends the insertion in that relation. Again the pair "db_name : rel_name" is asked. If no more tuples are to be inserted in any relation the word exit completes the command execution. The following example should make that interaction more clear.

```
do_insert.

Database : Relation? : spj : parts.
tuple > [p7, nut, black, 11, buenos aires].
tuple > [p8, bolt, blue, 30, cordoba].
tuple > exit.
Database : Relation? : exit.
```

the dots ending the lines are mandatory.

4.7. do_modify

This predicate, is a rudimentary facility provided for helping in the task of modifying the tuples loaded in the relations saving some typing work. The format is,

```
    do_modify.
```

The predicate begins asking the name of the database and the relation as a pair "db_name : rel_name", see the Notation paragraph, and then key of the tuple that will be modified. The old tuple, if exists, is printed and a prompt for the new one is printed. When no more tuples are to be inserted, the word exit ends the modifications in that relation. Again the pair "db_name : rel_name" is asked. If no more tuples are to be modified in any relation the word exit completes the command execution. The following example should make the interaction more clear.

```
    do_modify.

    Database : Relation? : spj : parts.
    key > [p7].
    old tuple > [p7, nut, black, 11, buenos aires].
    new tuple > [p7, nut, black, 11, bahia blanca].
    key > [p8].
    old tuple > [p8, bolt, blue, 30, cordoba].
    new tuple > [p8, bolt, blue, 28, mar del plata].
    key > exit.
    Database : Relation? : exit.
```

the dots ending the lines are mandatory.

5. Relational Algebra Operators

Given one or more databases composed of relations we are about to introduce operators which will give us a way of obtaining new relations out of the already existing relations. Relational Algebra Operators take one or two relations as operands and produce a new relation as result of the operation, i.e. the closure under these operations is maintained and the objects in the databases remain of the same type.

The following operators are implemented : union, intersection, difference, Cartesian product, join, select and project. First the set theoretic operators are introduced and then the special relational operations are presented. The result of the operations will be printed or not depending on the state of the system. If the system is in mute state (see Miscellaneous section) no printing is done. In loud state the resulting relation will be printed using the same format as in the print_rel predicate.

5.1. Notation

The relations involved as operands can reside in the same or different databases. If all relations belong to the default database is not necessary to indicate that explicitly and the command format will be

```
operator(rel_name1, rel_name2, rel_result).
```

where operator is one of union, intersection, difference and product, rel_name1, rel_name2 and rel_result are relations in the default database. If the relations reside in different databases, all must be explicitly referred in the command, even the default database, and the command format will be,

```
operator(db_name1 : rel_name1,
         db_name2 : rel_name2,
         db_name_result : rel_result).
```

The syntax extends accordingly for project, select and join.

5.2. Union, Intersection, Difference and Product

These are the usual set operations. The union of two relations is a third relation containing the tuples in both operands. The intersection of two relations is the relation containing the tuples common to both operands. The difference of two relations is the relation containing the tuples that appear in the first operand and do not appear in the second. The product (Cartesian product) is the relation whose tuples are formed by concatenation of every tuple in the first operand with every tuple in the second. They have the same syntactic form,

```
    operator(db1 : r1, db2 : r2, db3 : result).
```

or, if all three relations are in the default database

```
    operator(r1,r2,result).
```

where operator is one of union, intersection, difference or product (for the Cartesian product). "r1" and "r2" are two existing relations and "result" is a new relation to be created as the result of the operation, i.e., the meaning is,

```
    result  :=  r1 operator r2
```

order is important in the case of the non-commutative operator difference.

Both relations must be compatible (sometimes referred as union-compatible in the literature) in the sense that they should have the same schema, i.e. the domains should be the same and should appear in the same order, attribute names can be different. The resulting relation will have the same attributes names as the first operand, but now all the attributes are part of the key. This is the most conservative position possible, but the schema can later be changed using the reformat_rel predicate, see the Relation Handling paragraph. The format for the result will also be the same as the format for the first operand, and can be changed afterwards using the modify_format command.

The schema for the product is discussed below in the paragraph corresponding to the join operator with which share the definition of the resulting schema.

5.3. Select

This predicate selects the tuples from one relation such that all the predicates in a given list evaluate to true. The command has the following syntax,

```
select(db1: rel, db2 : result, Selection_list).
```

or, if both relations are in the default database

```
select(rel, result, Selection_list).
```

where "rel" is the relation from where the tuples are selected, "result" is la relation to be created for storing the tuples selected, and Selection_list has the following general form,

```
[ Selector_1, Selector_2, ..., Selector_n ]
```

where Selector_i, for i=1,..,n will have the form

```
[ Pred_i, [ai1, ..., air], [ci1, ..., cis] ]
```

where Pred_i is a system or user defined predicate, the a's are attribute names in the schema for Rel, and the c's are constants involved in the evaluation of Pred_i. The subscripts r and s are arbitrary for each Pred.

The operator obtains the values of the attributes for a given tuple and perform a call to Pred_i using them and the constants provided. If Pred_i succeed for all i the tuple is stored in Result. The schema and format for Result are the same as for Rel.

For example, selecting the red or blue parts from the relation parts in the spj database, default database in this case, will give in the color_parts relation the following result.

```
select(parts,
       color_parts,
       [ [member, [color], [[red, blue]] ] ]).
```

will produce,

```
DB: spj  Relation : color_parts (created : 08/29/86 08:10:50)

 --------------------------------------------
 p_nr |    pname |    color | weight |    city
 -----+----------+---------+--------+---------
   p1 |      nut |      red |     12 |  london
   p3 |    screw |     blue |     17 |  athens
   p4 |    screw |      red |     14 |  london
   p5 |      cam |     blue |     12 |   paris
   p6 |      cog |      red |     19 |  london
   p8 |     bolt |     blue |     30 | cordoba
 --------------------------------------------

 6 tuples in the relation

 CPU : 0.133335 secs.
```

5.4. Join

The join of two relations "r1" and "r2" is a selection over the Cartesian product of them. The schema for the result in the join operator (and for the product operator) is a schema formed concatenating the schema of "r1" and the schema of "r2". All the attributes will be considered as forming the Key_attribute_list and the Non_key_attribute_list will be considered empty ("[]"). The uniqueness of attribute names in the join operator (and in the product operator) is preserved.

If there are common names, the new schema for the result will contain the common names renamed to the same names with a "1" and a "2" appended. For example, if both schemas contain the name part_nr the join schema (and the product schema) will contain part_nr1 and part_nr2 in the corresponding places.

The join operator has the following syntactic form,

```
join(db1 : r1, db2 : r2, db3 : result, Selection_list).
```

or, if all three relations are in th e default database

```
join(r1, r2, result, Selection_list).
```

where "r1", "r2" are two existing relations, "result" is the relation where the result will be stored, and Selection_list has the same general form as for the select operator with the only difference that the second list is an arbitrary one instead of only constants. For example,

```
join(autos : auto_parts,
     trucks : truck_parts,
     vehicles : parts,
      [ [ ==, [part_nr], [part_nr] ],
        [ approx_eq_weight, [weight], [weight] ] ] ).
```

the symbol "==" is the Prolog predicate, but approx_eq_weight is user defined, that could be defined as

```
approx_eq_weight(X,Y) :-
            ( X =< Y + 2, X >= Y - 2 ).
```

meaning that the value of X cannot execeed the value of Y in more than 2 nor be lower than Y by more than 2.

The join selects from the Cartesian product of auto_parts, in the autos database and truck_parts, in the trucks database, those tuples for which both predicates succeed, that tuples will be stored in the parts relation of the vehicles database.

In the resulting schema the common names part_nr and weight will be renamed to part_nr1 (for part_nr in auto_parts) and part_nr2 (for part_nr in truck_parts), similarly for weight. That attributes names can be changed using the rename_attribute predicate.

5.5. Project

The command project select the columns specified in a given list. Redundant tuples will be eliminated, if necessary. Note that this operator gives the possibility of reordering of the columns in a relation. The general form is,

```
project(db1 : rel, db2 : result, Attribute_list).
```

or, if both relations are in the default database

```
project(rel, result, Attribute_list).
```

where "rel" is the relation from where the columns will be taken and "result" is the new relation where the new tuples will be stored. Attribute_list has the form,

```
[a1, a2, ..., an]
```

where the a's are attribute names appearing in the schema of "rel". The new schema for "result" will have all the attributes specified, and in the specified order, as forming the Key_attributes and the non_key_attributes list will be empty. The domain specification will be associated automatically as in the original schema. The ordering specified in the Attribute_list will be the order in the new schema. For example,

```
project(parts, s_parts, [p_nr,pname,city]).

DB: spj  Relation : s_parts (created : 08/29/86 08:33:34)

  -------------------------
  p_nr |    pname |    city
  -----+----------+---------
    p1 |      nut |  london
    p2 |     bolt |   paris
    p3 |    screw |  athens
    p4 |    screw |  london
    p5 |      cam |   paris
    p6 |      cog |  london
    p7 |    brake |  madrid
    p8 |     bolt | cordoba
  -------------------------

  8 tuples in the relation

  CPU : 0.199997 secs.
```

6. Actions

Actions are predicates that are are associated with the relations and are executed in one of the following situations:

- when a tuple is inserted,
- when a tuple is removed,
- when a tuple is modified,
- when a tuple is requested via get_tuple.

Actions are shown as part of the schema by the predicates print_schema and print_schemas_db.

The predicate should be predefined and if is not a part of the system, should be in the database predicate file, i.e. if the database file is named my_db, then the predicate file is my_db.pred and is created empty by the predicate create_db(db_name) when the database is created.

When the predicates are defined by the user, he/she should do it taking the tuple as argument, i.e. only one argument, in the first three cases. In the if_requested case the predicate should take two arguments, the first is the the tuple stored in the relation and the second is a place where the tuple modified will be returned. If no modification is done is not necessary to return the original tuple. This provides a way of returning

values calculated using the values in the original tuple.

6.1. define_action

This command adds an action to be executed in one of the mentioned situations above. The format is,

```
define_action(db_name : rel_name, Action, Pred).
```

or, if the relation is in the default database

```
define_action(rel_name, Action, Pred).
```

where "rel_name" is the name of a relation in the database "db_name", Action is one of the keywords if_inserted, if_removed, if_modified or if_requested, and Pred is the predicate that will be executed in the related command, i.e. if_inserted for insert and do_insert, if_removed for remove, if_modified for modify and do_modify, if_requested for get_tuple. The predicate must be predefined. For example,

```
define_action(spj : parts, if_inserted, print_message).
```

will associate the predicate print_message with the insertion command for the relation parts in the spj database, and every time a tuple will be inserted that predicate will be executed. For example, the print_message action (predicate) could be like,

```
print_message(X) :-
            write('Tuple '), write(X),
            write(' inserted.'), nl.
```

where X is a tuple inserted in the relation parts in the spj database.

6.2. remove_action

This predicate is intended for removing the relationship between an action and a predicate. The predicate is no removed. The format is,

```
remove_action(db_name : rel_name, Action).
```

or, if the relation is in the default database

```
remove_action(rel_name, Action).
```

where "rel_name" is the name of a relation in the database "db_name" and Action is one of the keywords if_inserted, if_removed, if_modified or if_requested. For example,

```
remove_action(spj : parts, if_inserted).
```

will remove the association between the insertion action and the predicate print_message, assuming that the command in the previous example for define_action

was in fact performed.

6.3. modify_action

This predicate can be used for changing the predicate associate with an action. The format is,

```
    modify_action(db_name : rel_name, Action, Pred).
```

or, if the relation is in the default database

```
    modify_action(rel_name, Action, Pred).
```

where "rel_name" is the name of a relation in the database "db_name", Action is one of the keywords if_inserted, if_removed, if_modified or if_requested, and Pred is the new predicate associated with the action. For example,

```
    modify_action(spj : parts, if_inserted, emit_signal).
```

in the context of the previous example in define_action, will terminate the association between insert and print_message and will associate it with emit_signal.

7. Assertions and Consistency

In the present version, assertions are specified as Prolog predicates that every tuple in a relation should conform to. These predicates can be system or user defined. User defined predicates should be included in the predicate file before the assertion is defined. The relevant assertions are checked every time a tuple is inserted in a relation and can be also checked using the consistency predicates (see below).

The file containing the predicates can be modified using the command modify_predicates. The command invokes the editor "vi" over the assertions file allowing its modification.

Clearly, after modifying an assertion or adding a new one to the file, some tuples may no longer conform to all of them. The consistency is automatically checked upon exiting the editor and the offending tuples printed out if the system is in the "loud" state (see the Miscellaneous section). In any case the command fails if the consistency is not maintained. The printing is done using the same format as the one specified in the corresponding relation.

Assertions are printed along with the schema by the predicates print_schema and print_schemas_db. The following predicates are provided for handling assertions.

7.1. define_assert

This predicate permits the association of assertions and relations. The format is,

```
   define_assert(db_name : rel_name,Selector).
```

or, if the relation is in the default database

```
   define_assert(rel_name, Selector).
```

where "rel_name" is the name of a relation in the database "db_name", Selector has the same for as described in the select operation, i.e.,

```
   [ Pred, Att, Cons]
```

where Pred is a user or system predifined predicate, Att is a list of attribute names occurring in the relation schema and Cons is a list of constants.

The testing of the assertions is done in the following way. The values of the attributes in Att are extracted from the tuples and a list is formed appending the constants in Cons. Then Pred is executed taking the elements in that list as arguments. For example,

```
define_assert(spj : parts,[ >=, [ weight], [ 10 ] ).
```

define an assertion telling that the weight attribute in every tuple of the relation parts in the database spj should be greater than or equal to 10.

7.2. remove_assert

This command allows the elimination of an assertion. The predicate involved is not deleted from memory nor from the predicate file. If the user wants to eliminate the assertion from the database, the assertion must be removed from the predicate file. The format is,

```
remove_assert(db_name : rel_name, Selector).
```

or, if db_name is the default database

```
remove_assert(rel_name, Selector).
```

with the same meaning assigned to the arguments as in define_assert.

7.3. consistency

Its function is to check the consistency of the relations against the set of defined assertions. The consistency check is performed without the user intervention when a database is restored using the command load_db(db_name), and when the predicate file is modified via the modify_predicates command. Also is performed, over the tuple involved, each time a tuple is added in a relation or an existing tuple is modified.

It has three general forms,

```
consistency(db_name).

consistency(Arg).

consistency(db_name : rel_name, Offending_list).
```

in the first case the whole database named "db_name" is checked out, in the second Arg can be the name of a relation in a database or a list of them, with the format db_name : [r1,r2, ...,rn]. In the third "rel_name" is the name of a relation in the database "db_name" and Offending_list is the list of the tuples which do not conform the assertions for the given relation, "db_name" can be omitted if the database involved is the default database. With proper handling this list can be used for eliminating, or modifying, those tuples from the relation. If the system is in loud state (see Miscellaneous paragraph) the inconsistent tuples are printed and the predicate fails, in the mute state it just fails.

8. Referential Integrity

The term referential integrity in relational databases refers to the problem of ensuring that a certain set of referential constraints among the relations are satisfied. A referential constraint is defined whenever a subset of the attributes in a relation are considered as equivalent to the set of attributes forming the key attributes in another relation. That is, the values stored in a tuple of a given relation are used as the values of the key in another relation. In PRODB is possible to define these referential constraints among relations contained in the same or different databases. For example, consider the part_nr attribute in the relation equip.

```
spj:equip      (created : 07/25/86 14:20:55)
          [[eq_nr,string]]
          [[eqname,string],[part_nr,string]]
          [6,10,8]
          [part_nr] -> [p_nr] in spj:parts
                    Daemon : warning_daemon

spj:parts      (created : 07/09/86 22:50:16)
          [[p_nr,string]]
          [[pname,string],[color,string],
           [weight,int],[city,string]]
          [4,8,7,6,7]
          weight>=10
          parts_assertion(pname,color,city)
```

This attribute must have a corresponding value in the relation parts, otherwise there will be a dangling reference. A part of the attributes in a relation that form the key attributes of another relation are referred as a foreign key. In the example above the attribute part_nr in the relation equip form a foreign key pointing to the relation parts as shown in the schema for equip.

In updating relations involved in the definition of referential constraints two different problems can arise. Lets assume the existence of two relations R1 and R2, and that an attribute Att in the schema of R1 form a foreign key pointing to R2. When inserting a new tuple in the relation R1, the value of the attribute Att in the new tuple can contain a foreign key value without a counterpart in the relation R2. When deleting a tuple from the relation R2 which is pointed to by a tuple in the relation R1, it is possible to produce a dangling reference, that is, a tuple in R1 can now contain a value without a counterpart in the pointed to relation R2. The modification of a tuple in either relation can produce any one, or both of the anomalies mentioned.

The approach taken in PRODB to handle this problem is to provided a way to express a referential constraint and to associate with it a daemon that gets fired when the constraint is violated in any of the cases mentioned above. The daemon is an user defined Prolog predicate. A postcondition for it should be that the referential integrity is reestablished. The deamon will be called with the following parameters:

```
    daemon(Db1:Rel1, Db2:Rel2, Tuple, Foreign_Key).
```

where Db1:Rel1 is the relation pointing to Db2:Rel2, Tuple is a tuple in Db1:Rel1 and Foreign_Key is the value of the foreign key contained in Tuple pointing to Db2:Rel2.

In the example above the schema of the relation equip in the database spj shows a referential constraint, and the deamon, named warning_deamon, associated with it,

```
                [part_nr] -> [p_nr] in spj:parts
                             Daemon : warning_daemon
```

There are two other commands related to foreign keys, define_fkey and delete_fkey. The first for establishing a referential constraint and the second to remove it.

8.1. define_fkey

The purpose of this command is to establish a referential constraint through foreign key definition. The command has the following format,

```
    define_fkey(Db1:Rel_Base, Db2:Rel_Target, Corr_List, Daemon)
```

The pair Db1:Rel_Base determines the relation that contains the attributes involved in the referential constraint with the relation determined by the pair Db2:Rel_Target. The correlation list (Corr_List) is a list composed by pairs of attribute names [[a1,b1],[a2,b2],...,[an,bn]], where ai belongs to the relation Db1:Rel_Base and bi belongs to the target relation Db2:Rel_Target and uniquely determines the referential constraint defined. Daemon is a user defined predicate (see above). For example,

```
    define_fkey(spj:equip, spj:parts,
                    [[part_nr,p_nr]], warning_deamon).
```

will define an referential integrity constraint between the relation equip and the relation parts of the database spj through the association of the attribute part_nr in spj:equip with the key attribute p_nr in the relation spj:parts. If that constraint is violated the deamon warning_deamon will be fired, with the following parameters,

```
warning_deamon(spj:equip, spj:parts, Tuple, FK).
```

where Tuple must be instantiated to a tuple in spj:equip and FK is instantiated to the foreign key pointing to spj:parts.

8.2. delete_fkey

The purpose of this command is to eliminate an existing referential constraint. The command has the following format,

```
delete_fkey(Db1:Rel_Base, Db2:Rel_Target, Corr_List)
```

The pair Db1:Rel_Base determines the relation that contains the attributes involved in the referential constraint with the relation determined by the pair Db2:Rel_Target. The correlation list (Corr_List) is as defined in define_fkey, and uniquely determines the referential constraint being eliminated. For example,

```
delete_fkey(spj:equip, spj:parts, [[part_nr,p_nr]]).
```

will delete the referential constraint between the relation equip and the relation parts of the database spj defined by [[part_nr,p_nr]].

9. Aggregate Predicates

Several built-in aggregates are defined:

- max
- min
- sum
- avg
- count_rel
- count_rel_tuples
- get_oldest
- get_newest

Max (Min) returns the maximum (minimum) value of an attribute in a relation. Sum returns the total sum of the values corresponding to an attribute in a relation. Avg returns the average value corresponding to a set of values of an attribute in a relation. Count_rel and count_rel_tuples are described in the Tuple Handling paragraph. Get_oldest and get_newest are described in the Time Handling paragraph.

Max, min, sum and avg work over a single attribute of numeric type, and the value returned is also numeric. The general syntax for max, min, sum and avg is the following,

```
op(db_name: rel_name, Attribute, Value).
```

or, if rel_name is the name of a relation in the default database

```
op(rel_name, Attribute, Value).
```

where op is one of max, min, sum or avg, rel_name is the name of an existing relation in the database named db_name, Attribute is the name of an attribute in the schema, and Value is the resulting value.

Given the following relation, shown by the output of tprint_rel,

```
DB: spj  Relation : parts (created : 07/09/86 22:50:16)

   ------------------------------------------------------------------------
    time of insertion | p_nr |    pname |   color | weight |     city
   -------------------+------+----------+---------+--------+---------
    07/09/86 22:50:17 |   p1 |      nut |     red |     12 |  london
    07/09/86 22:50:19 |   p2 |     bolt |   green |     17 |   paris
    07/09/86 22:50:23 |   p3 |    screw |    blue |     17 |  athens
    07/09/86 22:50:37 |   p4 |    screw |     red |     14 |  london
    07/09/86 22:50:51 |   p5 |      cam |    blue |     12 |   paris
    07/09/86 22:50:56 |   p6 |      cog |     red |     19 |  london
    08/26/86 20:59:59 |   p7 |    brake |   black |     30 |  madrid
    06/27/87 08:00:39 |   p8 |     bolt |    blue |     30 | cordoba
   ------------------------------------------------------------------------
```

The following queries will produce the results indicated by Value.

```
avg(spj:parts, weight, Value).

Value = 18.875

max(spj:parts, weight, Value).

Value = 30

sum(spj:parts, weight, Value).

Value = 151
```

9.1. time_select

This predicate selects tuples based in a date provided by the user and the date when the tuples were inserted

```
time_select(db1 : rel_name, db2 : result, Date, Comp).
```

or, if both relations belong to the default database

```
time_select(rel_name, result, Date, Comp).
```

where "rel_name" is the name of a relation, in the database "db1", over which operate, result is the name of a new relation, to be created in the database "db2" where to deposit the tuples selected. Date has the form MM/DD/YY hh:mm:ss or MM/DD/YY hh:mm or MM/DD/YY hh or MM/DD/YY (should be quoted, see example). Comp is one of: *> (after), *>= (in or after), *< (before), *=< (in or before), *== (in the date), *<> (in a different date). For example,

```
time_select(spj : parts,
            auto_parts : new_parts,
            '07/11/86 10:30', *> ).
```

selects all the tuples from parts in the database spj inserted after the date given and stores them in the newly created relation new_parts in the database auto_parts. The result will be,

```
DB: auto_parts  Relation : new_parts (created : 07/02/87 09:02:12)

  --------------------------------------------
  p_nr |    pname |    color | weight |    city
  -----+----------+----------+--------+---------
    p4 |    screw |      red |     14 |  london
    p5 |      cam |     blue |     12 |   paris
    p6 |      cog |      red |     19 |  london
    p7 |    brake |    black |     30 |  madrid
    p8 |     bolt |     blue |     30 | cordoba
  --------------------------------------------

  5 tuples in the relation

  CPU : 0.0999908 secs.

Tuples inserted after 07/11/86 10:30
```

9.2. get_newest, get_oldest

These two predicates return the newest and oldest tuple in a relation respectively. The formats are,

```
get_newest(db_name : rel_name, Tuple).

get_oldest(db_name : rel_name, Tuple).
```

where rel_name is the name of a relation in the database db_name and Tuple is the place where the tuple will be returned. As always if the relation is in the default database the db_name can be omitted. For example, the query

```
get_oldest(spj : parts, Tuple).
```

will instantiate Tuple = [p1,nut,red,12,london].

9.3. older, newer and same_date

This set of predicates provide a way of comparing the time of insertion of two tuples. The formats are,

```
older(db_name : rel_name, tuple1, tuple2).

newer(db_name : rel_name, tuple1, tuple2).

same_date(db_name : rel_name, tuple1, tuple2).
```

where rel_name is the name of a relation in the database db_name and Tuple1 and Tuple2 are tuples in that relation. For example, the query

```
older(spj : parts,
      [p7,brake,black,30,madrid],
      [p4,screw,red,14,london]).
```

will succed.

A second format is possible and it is used for comparing tuples stored in different relations from different databases. The formats are,

```
older(db1 : rel_name1, tuple1, db2 : rel_name2, tuple2).

newer(db1 : rel_name1, tuple1, db2 : rel_name2, tuple2).

same_date(db1 : rel_name1, tuple1, db2 : rel_name2, tuple2).
```

where rel_name1 is the name of the relation in the database db1 where the tuple tuple1 is stored and rel_name2 is the name of the relation in the database db2 where the tuple tuple2 is stored. For example, the query

```
newer(auto_parts:new_parts,
          [p7,brake,black,30,madrid],
       spj:parts,
            [p3,screw,blue,17,athens]).
```

will succed.

10. Transactions

A transaction is considered as one operation. Inside a transaction the referential integrity checking is suspended. At present time this capability is useful in connection with the roll_back predicate. Transactions cannot be nested. The following predicates are implemented.

10.1. open_transaction

This predicate starts a transaction. The format is,

```
open_transaction.
```

10.2. close_transaction

This command ends a transaction. The format is,

```
close_transaction.
```

10.3. roll_back

This predicate cancels the effect of the commands issued since the open_transaction command. The transaction remains open. The format is,

```
roll_back.
```

11. Miscellaneous

11.1. loud

The system can operate in one of two states : loud and mute. If the system is in loud state the result of some operations (all the relational algebra operators and the consistency command), will be printed. In the mute state no printing occurs.

Also the emission of error messages depends on the state of the system. In the mute state no error messages are printed. The user can switch from one to the other using the commands:

```
loud.

mute.
```

with the obvious meanings.

11.2. help

PRODB provides an on-line help facility, which is basically an on-line version of this manual. The starting page of the manual gives the topics over which help can be obtained by,

```
??help.
```

That command will display the following text,

```
                PRODB User Manual

           Help Available in the following subjects


           o  introduction
           o  databases
           o  relations
           o  tuples
           o  operations
           o  actions
           o  aggregate
           o  assertions
           o  time
           o  transactions


           use ??topic. to get more help
```

For example,

```
    ??operations.
```

will give the help for the operations item.

```
| ?- ??operations.

                         PRODB User Manual


                    Relational Algebra Operators

Given one or more databases composed of relations we are about to
introduce operators which will give us a way of obtaining new
relations out of the already existing relations.  Relational Algebra
Operators take one or two relations as operands and produce a new
relation as result of the operation, i.e. the closure under these
operations is maintained and the objects in the databases remain of
the same type.

     The following operators are implemented : union, intersection,
difference, Cartesian product, join, select and project.  First the
set theoretic operators are introduced and then the special
relational operations are presented.  The result of the operations
will be printed or not depending in the state of the system.  If the
system is in mute state (see Miscellaneous section) no printing is
done.  In loud state the resulting relation will be printed using the
same format as in the print_rel predicate.


  More Help on Level : operations

          o  notation
          o  set
          o  join
          o  select
          o  project


yes
| ?-
```