A Query Facility to a network DBMS

N. PARIMALA, N. PRAKASH, B. L. N. RAO AND N. BOLLOJU
R and D Group, CMC Ltd, 115 Sarojini Devi Road, Secunderabad 50003, India

A query facility to a network DBMS is described wherein it is possible to retrieve and update the data in the database. Apart from the existential and universal quantifiers, some additional quantifiers, which are found useful in a database environment, are provided for. A facility, called the discourse facility, by which a set of related queries can be asked is also explained. In this, the data retrieved in a query can be used in subsequent queries.

Received April 1987, revised June 1987

1. INTRODUCTION

A query facility for a DBMS was first proposed for a relational model of data.[2] Also, a criterion was laid down to evaluate such a query facility. It was postulated that in order for such a facility to be meaningful it should have the power of the relational calculus. In other words, any formula expressible in the first-order calculus should be expressible in the query system.

The facilities available with a network DBMS, on the other hand, were for a long time restricted to the use of DML languages embedded in host languages. This is, perhaps, because the DBTG Report[1] gave a specification only for the DML and its embedding in Cobol. However, some database management systems have tried to provide a query facility with a network data model as well. For example there is a system called Seed, which is a CODASYL-based system and offers a query facility called Harvest. This system, however, provides only for retrieval and gives no language for updating the data base. Also, only the existential quantifier is available. The universal quantifier cannot be used in this system.

Admin[3] is a DBMS based on the network model of data. It provides a query facility, called Query, which allows one to perform retrieval as well as update to the data base. Both the existential and the universal quantifier are available in Query. Further, some more quantification forms are provided for in this system, as these are found useful in a database environment. For example, instead of using just the existential quantifier one might like to say 'there exists at least 3'. A discourse capability is also available within Query. Using this facility the user can ask a bunch of related queries in a conversational mode.

In this paper we shall explain the features of Query. A brief overview is given in Section 2. The retrieval commands are explained in Sections 3 and 4. The discourse facility is given in Section 5. The examples in these sections will refer to the schema of Fig. 1 unless mentioned otherwise.

During the course of this paper, reference is made to a corec type and a coset type. These terms correspond to a record type and a set type of CODASYL. The reason[3] behind departing from CODASYL terminology is that we make a fair number of departures from the CODASYL notions of a record type and a set type. Consequently, we chose the relatively neutral terms corec and coset to denote these two types.

2. OVERVIEW

When extracting information from the database, a query can ask for data from just one corec type or more than one corec type. We refer to the former as a single-variable query. A query which asks for information from more than one corec type will be referred to as a multi-variable query.

The general format of a retrieval query is

⟨command⟩ ⟨target list⟩ WHERE ⟨condition⟩

where *⟨command⟩* is either the *FETCH* or the *EXPLODE* command (see below),

⟨target list⟩ is a list of fields, belonging to one or more corec types, that has to be retrieved and

⟨condition⟩ is a boolean valued expression.

Functions like *AVG*, *SUM*, *MAX*, etc. can be used in the target list. The result of a query can be sorted and output in a pre-specified format. Also, tuples of the target list which have the same value in a certain field can be grouped together.[4]

The various commands that modify the contents of the database are[4]

Connect to create a link between an owner record and one or more member records of a coset type

Disconnect to remove a link from a number of member records and their respective owner records

Newowner to transfer one or more member records from one coset occurrence to another in the same coset type

Delete to delete one or more occurrences of a corec type

Store to store an occurrence of a corec type

Update to modify the fields of a corec type either to a new value or to an arithmetic expression involving the fields of the same corec type.

3. SINGLE-VARIABLE QUERY

Essentially such a query asks for data from one corec type and does not use the relationships defined across corec types.

Example. Get the names and age of all employees who are less than 30 years old and have a salary greater than 2000 or are greater than 30 years and have a salary of less than 2000.

```
FETCH employee.ename, age
  WHERE (E.salary > 2000 AND E.age < 30)
    OR (E.salary < 2000 AND E.age > 30)
```

3.1 Explode

The problem of explosion is handled by the explode command. Explosion takes place whenever (*a*) there is a relationship between elements of the same corec type; or (*b*) there is a relationship between two corec types, as for example in the parts explosion problem. We shall consider each of these in turn.

Consider Fig. 1.

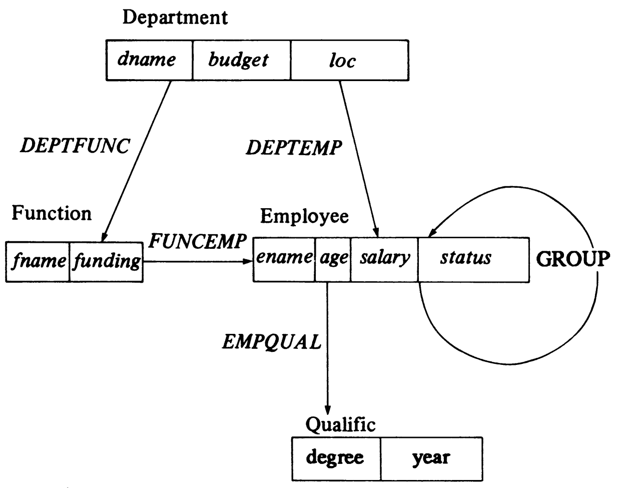

Figure 1.

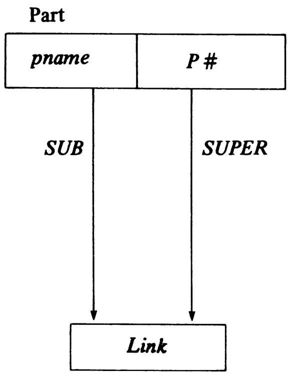

Figure 2.

Example. Get all the employees working under H. Ramesh

EXPLODE employee.ename WHERE ename = 'H. Ramesh'

This is executed, using the self-cycle GROUP as follows. First, the employee satisfying the condition is found. Thereafter, the coset occurrence in which this employee is the owner is identified. All members in this occurrence are to be output. Thereafter, the various coset occurrences with these members as owners are determined. Again, the members in these coset occurrences are output. This process repeats till such time as the coset occurrences constructed are all null.

It can be seen that explode command essentially outputs a hierarchy of corec occurrences rooted in the occurrence satisfying the condition. Whenever explosion takes place, the root occurrence, the member records connected to this, and recursively the member records connected to the records found are output.

In the foregoing we have considered an explode command with a condition. It is possible for no condition to be specified in this command. For example:

EXPLODE e.ename

In such commands, the roots of the hierarchies are those corec occurrences which do not take part in coset occurrence as members. The rest of the method of handling explode is the same as outlined earlier.

So far we have assumed that there is only one self cycle for a corec type. Whenever there is more than one self cycle *QUERY* lists out all these on the screen and asks the user to select one. Explosion, thereafter, takes place with the chosen self cycle.

Now, let us look at (*b*) above. For this, we shall refer to Fig. 2.

Example. Get the names of parts which are used in an engine

EXPLODE part.pname where pname = 'Engine'

This is executed using the coset types *SUB* and *SUPER*. First, a part named engine is found. Then the member records, *Link*, in the coset type *SUB* are found. For each link record thus selected a part is got by finding the owner record in the coset type *SUPER*. The above process is repeated for each part that is found. This process, for a part, terminates if there is no link record connected to it.

We have assumed, above, that (*a*) the cosets of traversal are *SUB* and *SUPER*; and (*b*) the order of traversal is first *SUB* followed by *SUPER*. In order to determine (*a*) and (*b*) the following is done.

First, all coset types with the corec in the explode command as owner and another corec type as member are found. If there are only two coset types than these are picked up. In case there are more than two the user is asked to select two among them. After the two coset types have been identified the user is always asked to specify the order in which they are to be used.

4. MULTI-VARIABLE QUERY

4.1. Path

A multi-variable query involves two or more corec types. In order to extract information across corec types the linking information provided by the coset types is to be used. It then becomes necessary to identify the cosets linking these corec types.

All the corec types in the target list together with those referenced in the condition are called the referenced corec types. A path is a traversal along coset types such that all the referenced corec types are included in the traversal. Also, there is a start corec type from which the traversal begins. This can be any one of the corec types of the target list. We have chosen the first corec type in the target list to be the start corec type.

Further, if more than one coset type is defined between two corec types, only one of these coset types is included for any given path.

It is possible that for a given query there exists more than one path, by using which the records can be found. Consider an example.

Example. Fetch names of departments and names of all employees in them.

FETCH dname, ename

If we look at the network of Fig. 1 there exist two paths

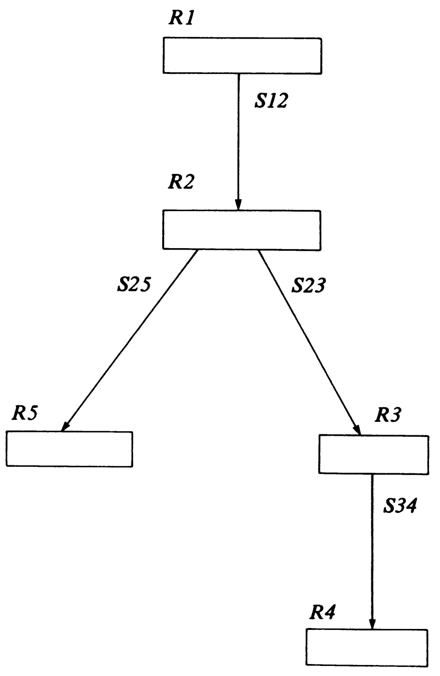

Figure 3*a*.

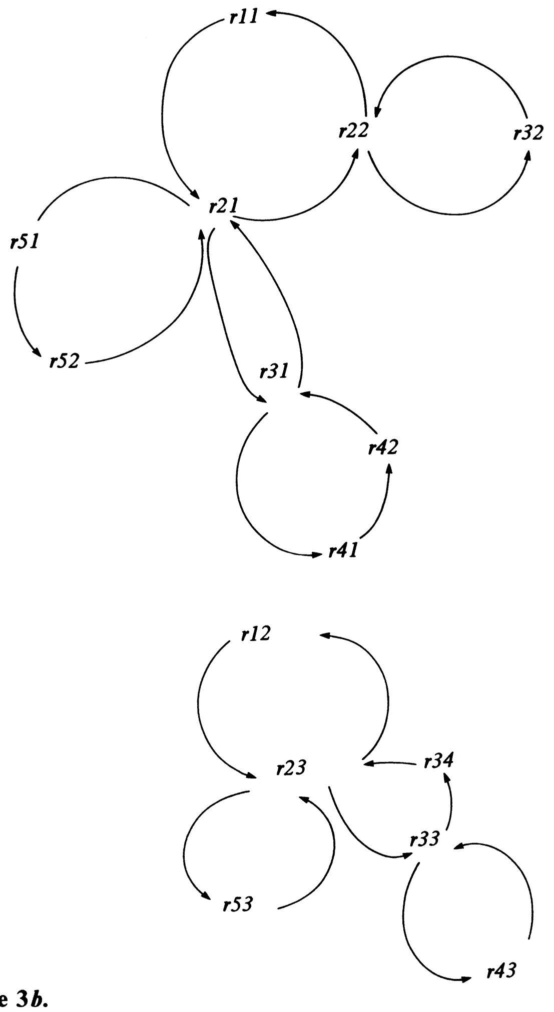

Figure 3*b*.

between department and employee. One of these is *DEPTEMP* and the other path is along the two coset types *DEPTFUNC* and *FUNCEMP*. Each of these paths will give a different answer. If the former is chosen then we would get all the employees connected to a department. Traversing via the two cosets *DEPTFUNC* and *FUNCEMP* would give the employees who are connected to the department via their functions. If there is an employee who is working in a department but has no function assigned to him the path *DEPTEMP* would list the employee, whereas the latter path would not.

Whenever there is more than one path which can be used to answer the query, the path of interest has to be identified. One method of identification of the path of interest would be to specify the path in the query itself. Instead, the system tries to help the user to identify the path rather than insist on the user specifying the information. In order to do so the system first determines all the paths that exist among the referenced corec types. If there is only one path, that path is used for fetching the records. In case of more than one path all the paths are displayed on the screen and the user is asked to make his choice.

When the system was first released to the users all the paths were flashed on the screen. It was later found out that users tend to choose the shorter (in terms of number of cosets) paths. Also, in most cases, the path which had cosets directly linking the corecs to be retrieved was chosen. In the above example *DEPTEMP* was chosen more often than the second path. Taking into account the user reaction, changes were made to the manner of displaying paths. To start with, all the paths were arranged in an ascending order, where the number of cosets contained in a path was the ordering criterion. Thereafter, all paths having the same number of cosets were flashed, starting with the lowest number.

4.2. Evaluation

In this section we shall consider the processing strategy that is adopted to answer a query. For this purpose consider the schema given in Fig. 3*a* and the instance diagram as given in Fig. 3*b*.

Let *R1*, *R2*, *R3*, *R4*, *R5* be the referenced corec types and *R1* the start corec type in a query.

Let the path be *S12*, *S23*, *S34*, *S25* where *Sij* is a coset type between *Ri* and *Rj*.

The process starts by finding the first occurrence, *r11*, of *R1*. Now, *r11* is established in *S12* and *r21*, the first in the coset occurrence of *S12* is found. Similarly, *r31* and *r41* are found. Further, *r21* is established in *S25* and *r51* is found. These records contain the values of the target list as well as those required for evaluating the boolean valued condition. We shall refer to this bunch of records as result tuple.

Using the result tuple the condition is first evaluated, and if found true, all the values of the target list obtained from these records are output. If there is no condition, the values of the target list obtained from these records are listed.

Now, the next result tuple is found. This is done by moving to the next record in the occurrence of last coset type of the path. In our example, the next result tuple is

r11   r21   r31   r41   r52

Now, the end of the coset occurrence of *S25* is reached. Therefore, the next record *r42* in *S34* is found. It is established in the coset type and navigation is resumed.

As a general rule, the process of moving back in the path is repeated till either (*a*) a next record in some coset type is found; or (*b*) the end of the occurrence of the first coset type in the path is reached. When this happens, an attempt is made to find the next record of the start corec type. If it exists the process outlined above is repeated; if not, all the records have been found and the process halts.

In our example the result tuples are

r11	*r21*	*r31*	*r41*	*r51*
r11	*r21*	*r31*	*r41*	*r52*
r11	*r21*	*r31*	*r42*	*r51*
r11	*r21*	*r31*	*r42*	*r52*
r12	*r23*	*r33*	*r43*	*r53*

4.3. Quantification

In *QUERY* there are a total of five quantifiers.

ANY [*n*]
ALL
JUST n
UPTO n
RALL

where *n* is a positive integer. A default value of *1* is assumed for *n* in *ANY*.

The meaning of the first four quantifiers is given below. Here, whenever we talk of occurrences we shall be referring to the occurrences of the quantified corec type. Also, the condition is that one which applies to this corec type.

Example. FETCH dname WHERE ANY E (*salary* > *2000*)

In this example, *E* is the quantified corec type and (*salary* > *2000*) is the condition applying to it. The explanation of the quantifiers is as follows:

ANY n	at least *n* occurrences should satisfy the condition
JUST n	exactly *n* occurrences should satisfy the condition
UPTO n	*0* or more but not more than *n* occurrences should satisfy the condition
ALL	all the occurrences should satisfy the condition.

The basic processing strategy for handling quantifiers will be explained with the help of examples.

(1) *ANY*

FETCH dname WHERE ANY 3 E(*salary* > *2000*)

Let the path be *DEPTEMP*. Let there be a counter initialised to zero. The process starts by picking up the first occurrence of the start corec type, department. Thereafter, the first employee linked to it in *DEPTEMP* is found, the result tuple constructed and the condition, *salary* > *2000*, is checked. If it holds, the counter is set to *1*; if not, the counter remains at zero. In either case, the next occurrence of employee linked to *DEPTEMP* is found. The process of checking the condition, operating the counter and finding the next employee is repeated.

As this process continues the system keeps monitoring the counter. If the counter becomes *3*, it is clear that '*at least 3 employees having salary* > *2000*' are connected to this occurrence of department. Hence the field, *dname*, can be output. Clearly, there is now no point in picking up the next occurrence of employee in *DEPTEMP* linked to this occurrence of department. Therefore, the next occurrence of department is picked up, the counter is reset to zero and the process repeats.

On the other hand, it is possible that as the new occurrences of employee in *DEPTEMP* are found, the counter never reaches *3*. In such a case the particular occurrence of *DEPTEMP* will be exhausted. Since the condition on at least *3* employees will not be satisfied, the *dname* value of the department occurrence will not be output. As before, the next occurrence of department is picked up, the counter is reset to zero and the process repeats.

The foregoing continues till the occurrences of department are exhausted.

(2) *JUST*

FETCH dname WHERE JUST 3 E (*salary* > *2000*)

The basic method of processing this is the same as that outlined for *ANY*. The only difference lies in the manner in which the counter is handled.

It is necessary to pick up employee occurrences in *DEPTEMP* till one of the following holds.

(*a*) The counter becomes *4* and the coset occurrence may or may not be exhausted. In such a case the department does not have '*exactly 3 employees with salaries greater than 2000*' attached to it. Hence, *dname* is not output. Additionally, there is no point in continuing further processing of this coset occurrence. Therefore, a new occurrence of the start corec type, department, is picked up; the counter is reset to zero and the process repeats.

(*b*) The counter is less than *3* and the coset occurrence is exhausted. Again, the department does not have '*exactly 3 employees with salaries greater than 2000*' attached to it. The same action as in (*a*) above is taken.

(*c*) The counter is *3* and the coset occurrence is exhausted. In this case, the department has exactly *3* employees with a salary larger than *2000*. Hence the name of this department is output. The next occurrence of department is picked up; the counter is reset to zero and processing resumes.

(3) *UPTO*

FETCH dname WHERE UPTO 3 E (*salary* > *2000*)

Again, the basic processing scheme is that outlined for *ANY* except that the counter is handled in a different way. It is necessary to pick up new employee occurrences in *DEPTEMP* till one of the following holds:

(*a*) The counter becomes *4* and the coset occurrence may or may not be exhausted. The department has more than *3* employees with salary greater than *2000*. Hence, it is not a candidate for output. The next occurrence of department is picked up, the counter is reset to zero and the process continues.

(*b*) The counter is less than or equal to *3* and the coset occurrence is exhausted. In this case, the department has up to *3* employees with *salary* > *2000*. Hence, the *dname* value for this occurrence of department is output. The counter is reset to zero, the next occurrence of department is found and the process repeats.

(4) *ALL*

FETCH dname WHERE ALL E (salary > 2000)

In this case there is no need to maintain a counter. As each new occurrence of employee in *DEPTEMP* is found, it is checked, as before. If *salary > 2000* holds, the next occurrence of employee is found. If, however, even one occurrence is found for which this condition does not hold, '*all employees of the department do not have a salary > 2000*'. Hence the department is not a candidate for output. In this case the next occurrence of department is found and the process repeats.

If, on the other hand, the coset occurrence is exhausted, this must be because each employee occurrence in it satisfied the condition on salary. Hence the *dname* value of department can be output. Again, the next occurrence of department is found and the process repeats.

(5) *RALL*. We shall now consider the quantifier *RALL*. *RALL* is used to indicate that all record occurrences of the corec type, and not just the ones connected to an occurrence of the start corec type, should be used in the evaluation of the condition. Let us take an example for the schema of Fig. 4.

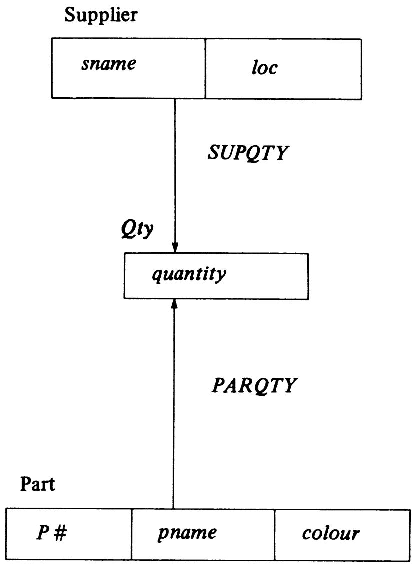

Figure 4.

Example. Get the names of suppliers who supply all the red coloured parts.

FETCH sname WHERE RALL P (colour = 'red')

The query can be rephrased as 'Print the names of the suppliers provided all the parts in the data base having the colour red are connected to the suppliers'. Therefore, first, all occurrences of part which satisfy the condition of colour equal to red are found. Thereafter, the first occurrence of start corec type, supplier, is found. If all the part occurrences selected earlier are also connected to the supplier occurrence the name of the supplier is output. On the other hand, if even one occurrence of the selected occurrences is not connected no value is output. As before, in both cases the next occurrence of the start corec type, if found, is picked up for further processing.

Whenever more than one *RALL* is present in a condition each *RALL* is evaluated separately and then the query itself is evaluated. Consider Fig. 1.

Example

FETCH dname WHERE RALL E (salary > 2000) and RALL Q (degree = 'M.S.') and budget > 10000

Here *RALL E* would be first evaluated to find out all the employees who earn more than *2000*. Thereafter, all the qualifications with the *degree M.S.* would be fetched. Now the fetch command is executed. The name of a department is printed out if its budget is greater than *10000* and all the above selected employees and qualifications are linked to it.

5. DISCOURSE

When a database is queried it is possible that there is a sequence of related queries which finally give the result the user is looking for. Consider the following sequence.

Find the employees whose salary is greater than *2000*.

Get their qualifications.

For all these employees who have an *M. Tech.* degree, find their functions.

Find the funding sources for these functions.

The above sequence is dictated by a thought process which is a natural way of seeking information. If a system does not support such a sequence of querying, the user may first have to write down the sequence and then frame a long and complicated query.

In *QUERY* a facility by which a sequence of queries can be asked is said to be a 'Discourse' facility.

Every query in a discourse mode gets data from the environment given to it by the previous commands. This environment is referred to as a context. A context is defined to consist of:

(*a*) a set of chosen corec types and sets of selected occurrences of these;

(*b*) non-chosen corec types and all occurrences of these;

(*c*) chosen path;

(*d*) non-chosen coset types.

Example

FETCH ename, degree WHERE salary > 2000

Here the chosen corec types are employee and qualific. The selected occurrences of employee are those who earn more than *2000* and the selected qualific occurrences are those which are connected to the selected employee occurrences. The non-chosen corec types are department and function. The chosen path is *EMPQUAL*. The set of non-chosen coset types contains *DEPTFUNC*, *FUNCEMP* and *DEPTEMP*.

The user enters into discourse using the command *SET DISCOURSE*. Once this command is executed the context is defined to be database context or DB context. When the context is DB context:

(*a*) the set of chosen corec types is empty;

(*b*) all the corec types of the subschema belong to the set of non-chosen corec types, hence all occurrences of all corec types are available;

(*c*) there is no chosen path; and

(*d*) all the coset types of the subschema belong to the set of non-chosen coset types.

The first query after the *SET DISCOURSE* command operates in the DB context. Once this query gets successfully executed a new context gets defined. Normally, this new context is the context for the query to follow (see later). The context available at any instant of time during a query session is referred to as the current context.

Only the occurrences of corec types available in the current context are used for evaluating the query. That is, for any referenced corec type, which is also in the set of chosen corec types, only the selected occurrences of the corec type are used. Also, if a corec type is not a chosen one, all the occurrences of the corec type are used in the evaluation process.

Consider now the path that will be chosen for evaluating the query. If all the referenced corec types of the query are included in the chosen path, the chosen path of the current context is used, and no new path is constructed. On the other hand, if there are corec types in the referenced corec types which are not included in the chosen path, a new path is constructed using the coset types from the set of non-chosen coset types. This path must include the chosen path as a part of the path. In other words, the new path is an extension of the path in the current context. It is possible that more than one path can be found which has the chosen path as a part of it. In this case the path-asking mechanism is invoked and the user is asked to make a choice.

Consider now the manner in which a query which is successfully executed updates the current context.

(*a*) All corec types which appear in the target list of the query are added to the set of chosen corec types.

(*b*) If a corec type has been newly added to the set of chosen corec types, the set containing all the occurrences of the corec type that qualified for being output is added to the sets of selected occurrences of chosen corec types. On the other hand, if a corec type of the target list already existed in the set, the old set of occurrences of this corec type is replaced with the set of occurrences which were selected in this query.

(*c*) If any corec type of the target list was in the set of non-chosen corec types, that corec type is removed from the set.

(*d*) For each corec type which was newly added to the set of chosen corec types all the occurrences are removed from the set of occurrences of non-chosen corec types.

(*e*) The path used to answer this query becomes the chosen path.

(*f*) If any coset type from the set of non-chosen coset types was used in constructing the path, this coset type is removed from the set. Further, in case more than one path existed, the coset types of each of these paths are removed from the set. This is because for all the corec types of the chosen path which may appear in a query to follow, the chosen path is always used. Therefore, even though these corec types may be linked to each other via many paths, to retain the coset types of all the alternate paths in the set of non-chosen coset types is not meaningful. These paths can never be chosen later on during the discourse.

It must be noted that even though only some fields of a corec type may be asked for in the target list, each occurrence with all its fields is available in the context. Therefore, in a subsequent query it is possible to ask for fields which are difficult from the ones in the previous query. Consider an example.

Example

FETCH ename where salary > 2000

FETCH employee.status

The second fetch command gives the status of those employees whose salary is greater than *2000*. It must also be noted that the above method of specification of a context does subsetting operation on the corec occurrences in the database.

In some cases it may be desired to digress from the current context and move to another one, but resume the interrupted discourse from the point of interrupt. Therefore, it is necessary to provide a means to save and restore contexts. The current context can be saved by using the command

MARK CONTEXT ⟨*context-name*⟩

Context-name is the name by which the current context is saved. The current context itself does not change. A saved context can be asked for using the command

SET CONTEXT ⟨*context-name*⟩

Once this command is executed ⟨*context-name*⟩ becomes the current context. The current context which existed at the time *SET CONTEXT* ⟨*context-name*⟩ was executed is lost unless it has been explicitly saved.

It is possible to move from a discourse mode to a non-discourse mode during a query session. This is done by using the command

SET NO DISCOURSE.

When such a movement occurs the saved contexts are not destroyed. These contexts can be used in another discourse session within the same query session.

All saved contexts are maintained till either (*a*) the query session is over, or (*b*) the context is explicitly deleted using the command

DESTROY ⟨*context-name*⟩.

If an update command is given, the current context is updated; it is possible that the record being updated is in one or more saved contexts. However, no change is made to any of the saved contexts that might get affected by this command.

In some cases it may be desired that the chosen path should not be used during a discourse. Consider an example. Let the schema be as shown in Fig. 5.

Suppose we are interested in finding out persons and buildings, distinguishing between those who live in buildings and those who own buildings. Therefore, we first find the buildings and the persons living in them. The context at this time consists of Person, Building, the path *LIVES* and the collection of Person and Building records which are linked via the coset type *LIVES*. But now it is not possible to know the persons who also own buildings, as *OWNS* is not available in the context. The only method of doing this is by discarding the chosen path and selecting the new path *OWNS*. This is possible in *QUERY* by specifying the command

ALL PATHS

This command tells the system that the chosen path is to be made nil, and all the coset types of the subschema are to be made available as non-chosen coset types. However,

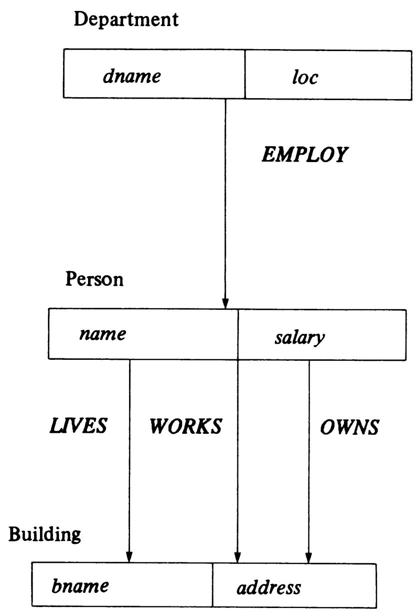

Figure 5.

the rest of the information in the context remains unchanged. In the above example, *ALL PATHS* would make both *LIVES* and *OWNS* available as non-chosen coset types of the context. Therefore, it is now possible to ask for persons and buildings. The system will display the two paths, namely *LIVES* and *OWNS*. The user can choose *OWNS*. It must be noted that since the selected occurrences of chosen corec types are maintained in the context after the command *ALL PATHS*, choosing *OWNS* will give the names of people who live in buildings and who own buildings.

5.1. Constructing new contexts

It may sometimes be desired to construct new contexts out of previously available ones. For this purpose the union (+), intersection (*) and difference (−) operators can be used.

These operators are binary in nature and are defined on compatible contexts. Two contexts *C1* and *C2* are said to be compatible if the chosen corec types of *C1* are identical to those of *C2*. The binary operations are performed with the occurrences of chosen corec types and will be explained later.

Since it is possible that the chosen path of *C1* is different from the chosen path of *C2*, the question that naturally arises is regarding the path that is defined for the newly constructed context. In general there is no way by which this path can be determined by the system. To understand this, consider each operation by which a new context can be constructed.

If the operator is union, the choice of either the chosen path of *C1* or that of *C2* can create difficulties. This is because if the path of *C1* is chosen the records of *C2* may not get accessed, and vice versa. This would make the union operator ineffective.

If the operator is intersection, either the path of *C1* or the path of *C2* can be chosen, because the records arising out of the intersection operator are in both the contexts.

If the operator is difference, *C1–C2*, the path of *C1* only can be chosen as the path of the new context, since all the records in the new context are also in *C1*.

It is also possible that the user wishes to choose an altogether new path for the newly constructed context. Consider an example. Let the schema be as shown in Fig. 5. Let it be required to find the names of people and buildings such that these people own, live and work in these buildings.

Fetch the persons and buildings where they live in. Choose the path *LIVES*

FETCH person.name, building.bname
MARK CONTEXT living

SET DISCOURSE

Fetch the persons and buildings which they own. Choose the path *OWNS*

FETCH person.name, building.bname
MARK CONTEXT owning

Get the persons who live in the building they own

MAKE CONTEXT (*owning * living*)

Get the persons and buildings which they own, live and work in. Choose the path *WORKS*

FETCH person.name, building.bname

In the foregoing, first, those people and buildings are identified in which the people live. The context is saved and marked as '*living*'. Thereafter, the context is set to database context and those people and buildings are identified which the people own. Again, the context is saved and marked as '*owning*'. Now a new context is created, which is the intersection of the two saved contexts. Thus those buildings are known in which people live and which they own. Now the next fetch command asks for information about people and buildings where these people work. It must inherit the data constructed out of the Make operation but must traverse the *WORKS* relationship to answer the query. Clearly, the user requires facilities to choose an altogether new path after a new context is constructed. The system flashes all paths between person and building and the user chooses *WORKS*. The data retrieved give the desired result.

It must be noted that the foregoing could also have been achieved by saving three contexts – living, owning and working respectively. Thereafter, an intersection of all these would have answered the query.

From the foregoing it is clear that the specification of the chosen path of the newly constructed context is best left to the user. Therefore, at the time the command to make a new context is issued, the system assumes that all the coset types of the subschema are available as non-chosen coset types and the chosen path is null. When the first query is issued in the new context, all paths that can be used to answer the query are flashed on the screen and the user is asked to choose one of these. This path then becomes the chosen path and the process continues as explained earlier.

There is only one question remaining about the new context. This is regarding the way the occurrences of the chosen corec types are arrived at. Consider the example given above.

Let the result tuples for the context *living* be ⟨*P1*, *B2*⟩ and ⟨*P2*, *B1*⟩. Further, let the result tuples for the context *owning* be ⟨*P1*, *B3*⟩ and ⟨*P2*, *B1*⟩. Now, it is clear that there is only one person living in the building he owns. This is *P2* and the building in question is *B1*. Thus, it is necessary that at the time the new context is created, the intersection operator should be performed over the result tuples of the two contexts.

It is important to notice that at the time the '*mark context living*' command is issued, the result tuples of that context are not available. Instead, the path, *LIVES* and the occurrences of Person, namely *P1*, *P2* as well as the occurrences of Building, namely *B1*, *B2* are available. Similarly, at the time the '*mark context owning*' command is issued, the path *OWNS* and the occurrences of Person and Building respectively are available. Therefore, at the time *Make context* is issued, the result tuples of the two contexts '*living*' and '*owning*' are to be generated. Section 4.2 specified the manner of creating result tuples by a process of navigation which used the knowledge of occurrences of the various corec types and the path selected for this purpose. Applying the same principle, result tuples of the two contexts *living* and *owning* can be created.

Once the result tuples of the two contexts are created, it is possible to apply the construction operators over these as follows.

If the operator is union, then the union operation is performed on the result tuples of *C1* and *C2*. Similarly, intersection gives the tuples which are both in *C1* and *C2* and difference gives the result tuples which are in *C1* and not in *C2*.

Once the required operation is performed over the result tuples and the result tuples qualifying for the new context have been arrived at, the qualifying occurrences of the chosen corec types of the new context are known. Hence, the sets of selected occurrences of chosen corec types are known. In the foregoing example the chosen corec types are Person and Building. The qualifying occurrences of these in the new context are *P1* and *B1* and the sets of selected occurrences are {*P1*} and {*B1*} respectively.

Summarising, then, the context which is constructed out of previously defined contexts inherits the chosen corec types of these and sets of selected occurrences are created as explained above. Further, the new context inherits the non-chosen corec types and the occurrences of these. There is no chosen path, and all the coset types in the subschema are available as non-chosen coset types.

6. CONCLUSION

In this paper we have explained the query facility for a network DBMS. Since, in a network model, the relationships across corec types are predefined, these can be used to answer the query. This reduces the complexity of a query and also shifts the burden of identifying these relationships from the user to the system. The user can thus only ask for the information he wishes to seek and have a neater interface. The problem of explosion can also be handled in a simple way.

In the system described here, we have introduced a number of additional quantifiers like *JUST*, *UPTO* and *RALL*. A discourse facility is also given where the user retrieves related information by a sequence of queries. This obviates the need to specify a single long complicated query to extract the same information. Further, since operations can be performed on contexts, more information can be obtained than would be possible in a non-discourse facility. For example, in the schema of Fig. 5, the query 'get the names of persons and buildings where the persons live in the building they own' cannot be answered in a non-discourse facility.

Acknowledgements

We would like to thank R. Krishnan and V. V. R. S. S. N. R. Kumar for helping with the implementation.

REFERENCES

1. CODASYL DDLC Report, *Information Systems*, pp. 247–320.
2. E. F. Codd, *A Data Base Sublanguage Founded on the Relational Calculus*. Research Report RJ 893, IBM Research Laboratory, San Jose.
3. Naveen Prakash *et al.*, Data definition facilities in admin. *The Computer Journal*, **26** (4), 329–335.
4. N. Parimala *et al.*, *QUERY – A User Interface*. Technical Report 1/87, CMC Ltd, Secunderabad.