Implementation of a Prototype for PRECI*

S. M. DEEN,* R. R. AMIN AND M. C. TAYLOR*
Department of Computing Science, University of Aberdeen, King's College, Old Aberdeen AB9 2UB

PRECI* provides a generalised architecture for a decentralised distributed database system capable of supporting a heterogeneous collection of pre-existing nodes. It permits data replication, subject to nodal access controls, and a number of user facilities, including both location-transparent and location-dependent queries in a relational language.

A subset of these features, dealing with distributed queries in a decentralised environment, has been implemented using two homogeneous PRECI nodes within the same computer. This system is written in C under Unix, as described in this paper.

Received February 1986

1. INTRODUCTION

PRECI* is a model for a generalised distributed database management system supporting heterogeneous, pre-existing databases as nodes with decentralised controls.[1–3] Its features include the following.

(1) Both 'inner nodes' (of which an integrated, location-transparent view is provided) and 'outer nodes' (which are not integrated).

(2) A relational language (PAL) for database integration and data retrieval. Each node must support at least a minimal subset of PAL commands.

(3) A subsidiary database is associated with each inner node and optionally with outer nodes. This may provide replicated data; meta data; additional PAL commands; and processing of external data (i.e. data sent from another node).

(4) It is capable of linking with other DDBs at peer level.

(5) Each node of whatever data model must provide a Participation Schema (PS), describing the data contributed by that node to the DDB. The PSs of the inner nodes are combined to form a Global Database Schema (GDS). Users access the PSs of the outer nodes directly, but access to the inner nodes can be made via a Global External Schema, which is an integrated view of the GDS. A version number is associated with each schema for checking purposes.

In this paper we shall describe the implementation of a subset of these features, as outlined in Section 2. The subsequent sections provide the implementation details, along with a Conclusion in Section 7 and an Appendix later.

2. REDUCED ARCHITECTURE FOR IMPLEMENTATION

Because of the limited time available for programming, not all the features of the design have been included in the implementation of a prototype. The prototype was implemented on the PDP/11 using 'C', with two PRECI/C databases as nodes.[4] These are both outer nodes. To allow fast implementation of the query parser, by recursive descent,[5] we restricted queries to a small subset of PAL. Each query is a combination of joins, unions and selections on single conditions of the form $x = y$ (where x and y are identifiers or constants). This restriction also saved the need to build subsidiary databases, the PRECI/C nodes being able to perform join and union operations on external data. The link to other DDBs has not been implemented, since no suitable DDB was available.

* Now at Department of Computer Science, University of Keele, Keele, Staffs ST5 5BG.

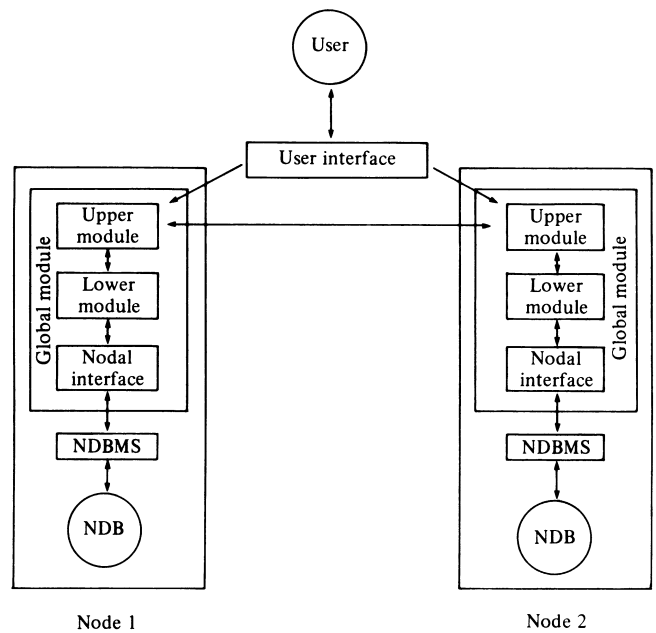

Fig. 1. Communications between modules

To avoid the need to write schema compilers, we store the schemas in compiled form. Schema version numbers are not maintained. Instead we re-initialise the DDB and re-compile queries, whenever a schema is changed.

Each node has an upper module and a lower module which run continuously and independently. The node-dependent part of the lower module is implemented separately as a nodal interface. The communications between modules are illustrated in Fig. 1.

Each module has a mail file for receiving messages from other modules (see Fig. 2). Reading from, and writing to, mail files is controlled by a locking mechanism. If a module has nothing to do, it periodically claims the lock and scans its mail file until a message is received. If a message is found, it is dealt with fully and any appropriate messages are sent to other modules before the lock is released. This ensures that two processes never write to the same mail file at the same time,

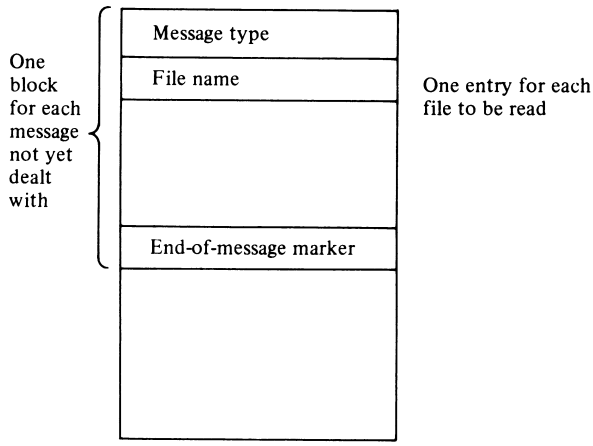

Fig. 2. Mail file

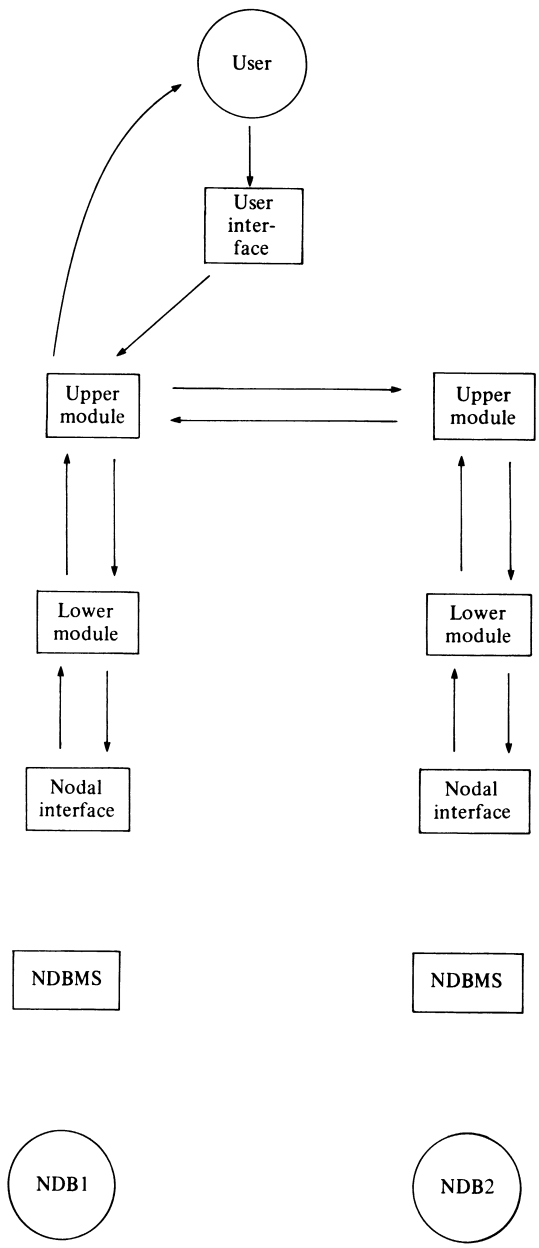

Fig. 3. Compile-time communications for a query involving two subqueries

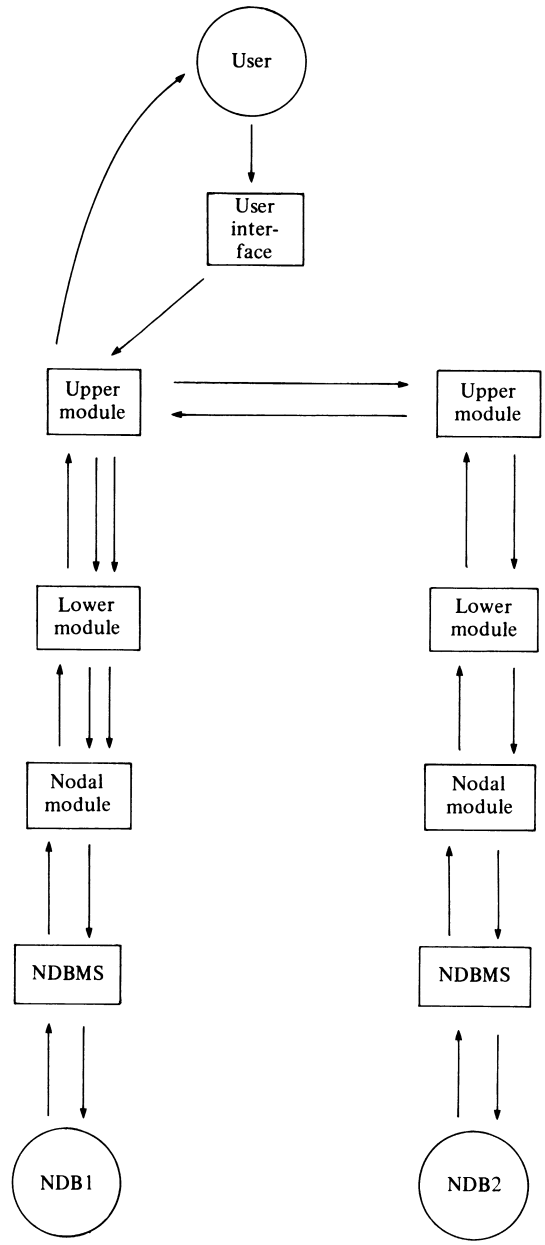

Fig. 4. Execution-time communications for query of Fig. 3

and that a module never reads its mail file while it is being written to. Each module handles only one message at a time, dealing with it fully before looking for any further messages.

A user interacts with the system via a user interface. This is a program which prompts the user for instructions, and translates these instructions into a form which can be understood by the appropriate upper module.

All queries are compiled and executed separately but, since the PRECI/C nodal database systems interpret queries directly, there is no compilation problem at the nodal level. In the compilation phase, the upper module performs the function of the GQP and the lower module performs the function of the GSP.[2] In place of the NQP, the nodal interface returns a message to indicate that the compilation was successful, without invoking the NDBMS. During the subsequent execution of a query, the upper and lower modules perform the function of the GQE and GSE respectively. This time, however, the

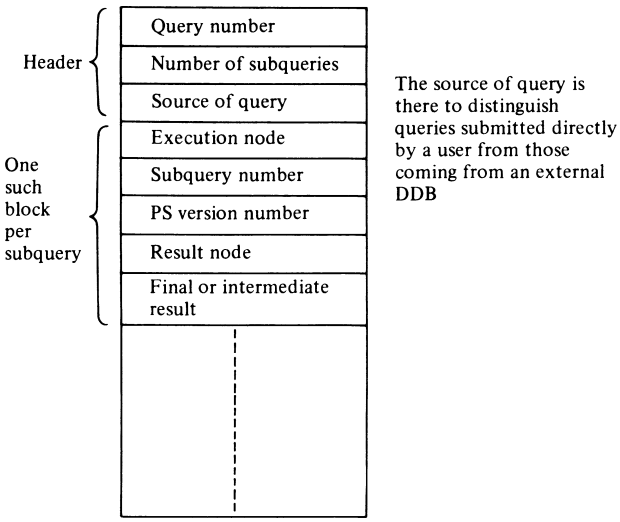

Fig. 5. Query plan file

NDBMS is invoked by the nodal interface. Fig. 3 and 4 illustrate the communications involved for compilation and execution.

Sections 3 and 4 describe the upper and lower modules respectively. Section 5 presents the nodal interface which includes the external interface to PRECI/C. The user interface is described in Section 6 and the steps involved in running queries are given in the Appendix.

3. UPPER MODULE

An upper module may receive messages of broadly five types:

(1) to compile a query (from user interface);
(2) to execute a query (from user interface);
(3) to receive a subquery packet (from the other upper module);
(4) to receive a result packet (from its lower module);
(5) to receive a result packet (from the other upper module).

The packets received correspond to those described in Ref. 6. The action taken on receipt of each of the five types of message is described below.

3.1 To compile a query

The user interface passes to the upper module the name of a file containing a query tree. This tree includes an indication of the vertices at which it should be split into subtrees (i.e. the breakpoints of the query.[7, 8]) The upper module reads the tree from the file, and decomposes it into subtrees. Each subtree (subquery) is allocated to a node in a straightforward way (there is no replication, and all non-local subqueries are assigned to the result node, hence the upper module has no optimisation to do).

From each subquery, the upper module forms an S-plan according to the format described in Ref. 6. The major task in preparing S-plans lies in determining what will be the format of the result of each subquery. This is done by analysing the subquery tree, and looking up the schema file (Fig. 12) for the formats of base relations. The formats of intermediate results are stored in an intermediate result file (Fig. 13) since they determine the formats of the results of non-local subqueries. Each S-plan is despatched to the appropriate module for further processing. Queries for its own node are sent to the lower module, whereas queries for the other node must be sent to the upper module of that node, which then passes them on to its lower module.

The overall query plan is stored in the query plan file (see Fig. 5) for future reference. The reasons for this are twofold. First, the upper module must store the number of subqueries involved, so that it knows how many compilation reports to expect back. Secondly, it must store sufficient information to be able to send appropriate instructions for the subsequent execution of the query, in the event of the compilation being successful.

3.2 To execute a query

The upper module reads from a file the query number of the query to be executed. (Each query is assigned a unique number at compile time and this number is returned to the user.) The query plan file is then read, in order to find the query plan of the relevant query. From this the upper module is able to form packets to be sent to the appropriate modules as for compilation.

3.3 To receive a subquery packet

Whenever a packet is received, its header is analysed in order to determine the packet type. If the packet is a subquery (for either compilation or execution) it must have been sent by the other upper module and is to be passed on to the lower module at the receiving node. The upper module therefore has only to send a message to the lower module.

3.4 To receive a result packet from the lower module

The action to be taken on receiving a result packet from the lower module will depend on the nature of the packet. The first thing to do is to check the destination node. If the packet is destined for the other node it is simply passed to the upper module of that node. If it is destined for the upper module's own node, however, the packet has to be analysed further. If it is found to be a compilation result, the next action is to look up the query plan file to find the number of compilation results expected for that query (i.e. the number of subqueries). The compilation result file (Fig. 6) is then checked, to find

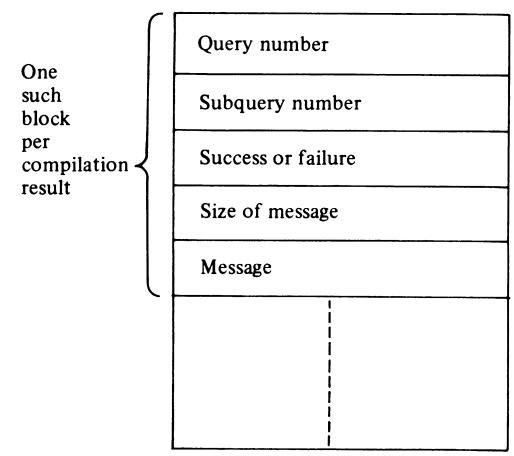

Fig. 6. Compilation result file

how many results have already been received for that query. If the newly arrived result is the last one expected, a final compilation report is returned to the user. Otherwise the result is written to the compilation result file.

Execution results may be either final results or intermediate results, and the packet header indicates which type applies. An intermediate result which is already at its destination node is just passed to the lower module, since it will be required as input for another subquery. A final result is written, in an appropriate form, to the user.

3.5 To receive a result packet from the other upper module

Result packets are dealt with in exactly the same way, regardless of the sender. The only difference between this case and 3.4 is that in this case the packet will never be destined for another node. Nevertheless, the check has to be made because the receiving node is unaware of the sender of the message.

4. LOWER MODULE

The lower module receives query packets from the upper module and results from the nodal interface. In each case the procedure is different at compile time from execution time. At execution time results may also be received from the upper module, so five cases may be distinguished in all.

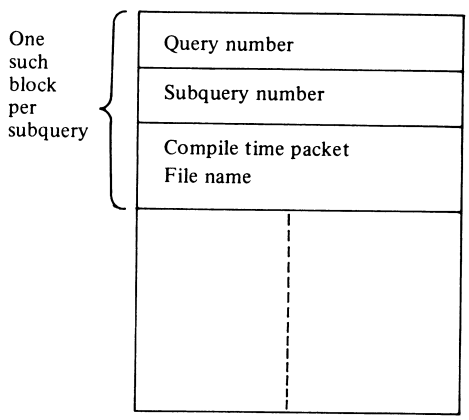

Fig. 7. Subquery file

4.1 Query for compilation

The lower module reads the query packet and produces a program file in the form required by the nodal interface (in general this will be node-dependent). A subquery file is used for storing information that will be required during any subsequent execution of the query (see Fig. 7).

4.2 Query for execution

The packet is checked to see whether the query can be executed immediately or whether it needs to wait for external data to be sent. If it has to wait, the query is written to the wait file (see Fig. 8), from where it can be retrieved when the required data has all been received.

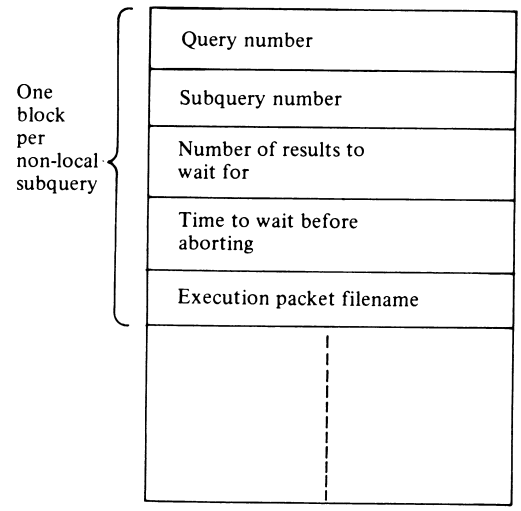

Fig. 8. Wait file

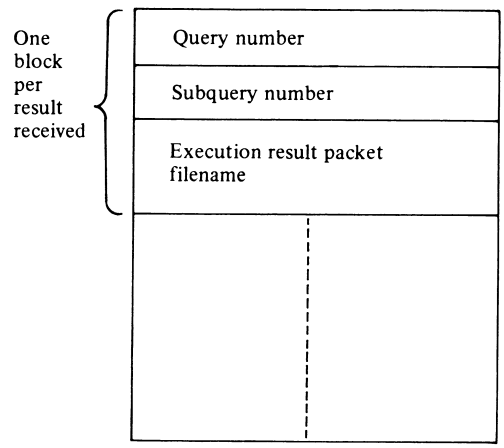

Fig. 9. Wait result file

4.3 Execution result from upper module

The upper module may pass on a result packet which it has received from another node. This packet will contain external data to be used in processing a subquery. The lower module looks up the subquery file to see how many such result packets are expected for that subquery and then looks up the wait result file (see Fig. 9) to see if any other result packets have already arrived. If there are still more result packets to come, the newly received packet is written to the wait result file. Otherwise the execution of the subquery can go ahead, so an appropriate file is prepared and a message is sent to the nodal interface.

4.4 Compilation result from nodal interface

From the compilation result and information stored earlier, the lower module forms a result packet which it sends to the upper module.

4.5 Execution result from nodal interface

From the execution result, the lower module forms a packet which it passes to the upper module.

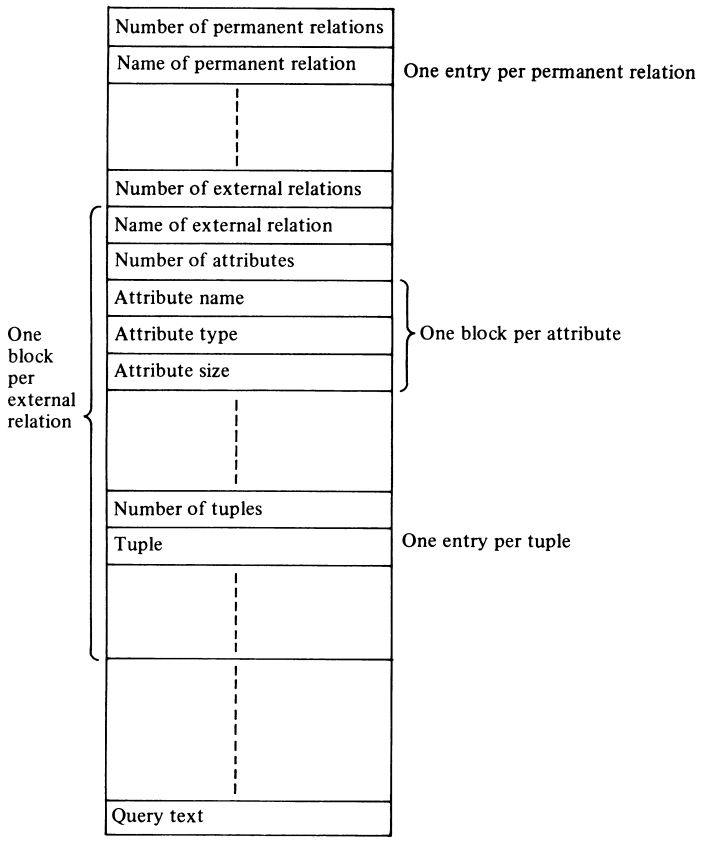

Fig. 10. Input file for NDBMS

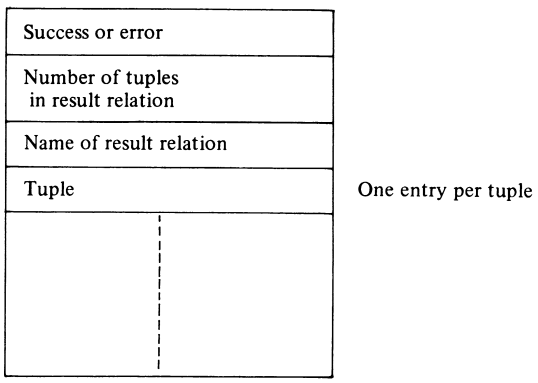

Fig. 11. Output file from NDBMS

5. INTERFACE TO NDBMS

The external interface to PRECI/C allows processing on external data, taking an input file in the form of Fig. 10 and returning a result file in the form of Fig. 11. Communications with the NDBMS are handled by the nodal interface. As explained in Section 2, at compilation time the nodal interface does not invoke the NDBMS but simply returns a message to the lower module.

At execution time it first converts the request from the lower module into the form required by the NDBMS. The NDBMS is then invoked by forking so that, on completion of the execution, control returns automatically to the nodal interface. The NDBMS is instructed to return the result to a specified file. This file is then passed (by the nodal interface) to the lower module.

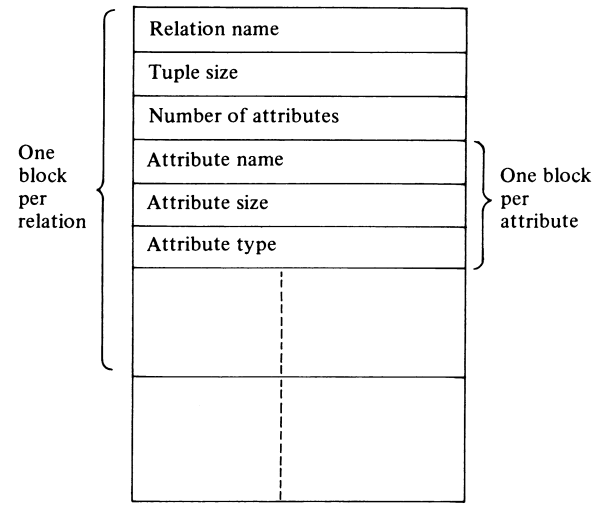

Fig. 12. Schema file

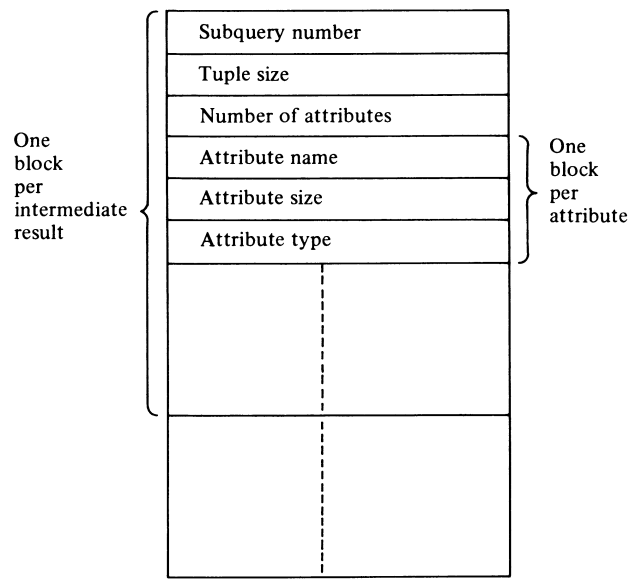

Fig. 13. Intermediate result file

6. USER INTERFACE

Requests are submitted to the DDBMS via a user interface. This interface allows a user one of four options:

(1) initialise ddb;
(2) display schema;
(3) compile query;
(4) execute query.

6.1 Initialise ddb

This option creates the necessary files so that the upper modules can run. The only file that is assumed to exist already is the schema file (Fig. 12).

6.2 Display schema

This lists the relations and their attributes available at the two nodes of the system. The information is found from the schema file (Fig. 12).

6.3 Compile query

The user is prompted to type in a query as PAL text, e.g. (node1 . . r(a1) * (a1) node2 . . s): b = 'fred'

The query is parsed, and is then passed on to the upper module in the form of a parse tree with the breakpoints marked. After invoking a compilation, it is necessary to exit from the interface and look up the results file for the outcome.

6.4 Execute query

After successful compilation of a query, a message is returned to the results file, giving the number which has been assigned to the query. It is then possible to invoke an execution of the query by specifying the query number.

Again it is necessary to exit from the interface to look up the results file for the outcome. If the execution was successful the result is returned in a tabular form, giving attribute names and a list of tuples.

7. CONCLUSION

The PRECI* project was fairly large and carried out between 1982 and 1985. Its architecture is very general and can be subsetted for specific implementation. A modified version of this architecture has been successfully applied for an international directory database system, as specified by CCITT, in a joint project with British Telecom.

In a large research project of this kind, it is not always feasible or advisable to implement a prototype with all aspects of the design, not only because of the shortage of resources (manpower included) but also due to the absence of any real end users, noting at the same time that research prototypes are not commercial products. This is the approach we adopted, and we found the implementation worthwhile. Finally, the authors would like to thank the UK Science and Engineering Research Council for supporting this work.

REFERENCES

1. S. M. Deen, R. R. Amin, G. O. Ofori-Dwumfuo, M. C. Taylor, D. A. Bell, J. Grimson and G. O'Brien, The PRECI* project and its data communication links. *Proceedings of Euteco, Varese, Italy, 1983*, (edited T. Kalin). North-Holland, Amsterdam.
2. S. M. Deen, R. R. Amin, G. O. Ofori-Dwumfuo and M. C. Taylor, The architecture of a generalised distributed database system – PRECI*. *The Computer Journal* **28**(3), 282–290 (1985).
3. S. M. Deen, R. R. Amin and M. C. Taylor, Data integration in distributed databases. *IEEE Transactions on Software Engineering* (to be published).
4. S. M. Deen, R. Carrick and D. Kennedy, A flexible DBMS for research and teaching (PRECI/C). *Proceedings, 4th British National Conference on Databases, Keele 1985.*
5. A. V. Aho and J. D. Ullman, *Principles of Compiler Design.* Reading, Mass.: Addison-Wesley (1977).
6. S. M. Deen, R. R. Amin and M. C. Taylor, *Standard Communication Packets for Distributed Databases.* Aberdeen University Internal Report (1984).
7. S. M. Deen, R. R. Amin and M. C. Taylor, Query decomposition in PRECI*. *Proceedings of the Third International Seminar on Distributed Data Sharing Systems, Parma, Italy*, 1984, edited F. Schreiber and W. Litwin, North-Holland, Amsterdam.
8. S. M. Deen, R. R. Amin and M. C. Taylor, A strategy for decomposing complex queries in a heterogeneous DDB. *Proceedings of VLDB, Singapore 1984.*

APPENDIX: STEPS IN RUNNING QUERIES

(1) Initialisation of Files
Log in at terminal 1 as 'pstar'
Type inter (to call the user interface)
When prompted for node, type 1
When prompted for option, type 4 (initialise ddb)
When prompted for next option, type 0 (exit)

(2) Starting Modules Running
Log in at terminals 2 to 7 as 'pstar'
Type the following (one command at each terminal):
(i) unode1
(ii) unode2
(iii) lnode1
(iv) lnode2
(v) nnode1
(vi) nnode2

(3) To Compile a Query
At terminal 1, type inter (re-enter user interface)
When prompted for node, type 1
When prompted for option, type 2 (compile query)
When prompted for next option, type 0 (exit)
After query has compiled, type cat results and note query number returned

(4) To Execute a Query
At terminal 1, type inter (re-enter user interface)
When prompted for node, type 1
When prompted for option, type 3 (execute query)
Then type in query number (returned with compilation result) e.g. 0
When prompted for next option, type 0 (exit)
After query has executed, type cat results and note name of result file (e.g. U1R0)
Type cat U1R0 (if that is name of result file)