Data language requirements of database machines

by DAWEI LUO
Harbin Institutes of Technology
Harbin, China
DAOZHONG XIA
Shanghai Shipbuilding Technology Research Institute
Shanghai, China
and
S. BING YAO
University of Maryland
College Park, Maryland

ABSTRACT

The selection of a proper set of operations for a database machine has a great influence on the performance of the overall system. In this paper, we will discuss the data language requirements of database machines. The data language for a proposed relational database machine is presented. Methods for supporting hierarchical and network interfaces using a relational database machine language set are introduced. The relative performance for data access using relational and network interfaces is compared through the evaluation of cost functions. It is shown that the support of a network interface in a relational database machine is feasible, but expensive.
1. INTRODUCTION

The development and application of database technology has made a great impact on data processing. It has made possible the sharing of common data resources and relieved the end user’s burden of managing the data. The data independence provided by many database systems has made the maintenance of program and data more effective.

The advantages of database systems have encouraged the growth of the size and complexity of many database system applications. In many applications the efficiency and reliability of the system are becoming problems. A database management system is a fairly complex piece of software. It competes with other application programs for memory and computing resources. It also requires a certain effort for maintenance.

Database machines have been proposed as hardware solutions to these problems. The objective is to offload database management functions from the host computer to a specially designed, dedicated back-end computer whose sole function is to maintain the database and process database requests. A database machine interfaces with the front-end host computer through a high-level-language interface. By fixing such an interface, a database machine can communicate simultaneously to multiple host computers. Depending on the size and complexity of a system, the host computer could be a mini- or a microcomputer, or even, in some situations, an intelligent terminal.

To permit a database machine to communicate with various types of front-end processors, it is important to select a proper language interface. The front-end processors may vary in processing capabilities, type of applications, or even data models. This paper reports the development of a database machine language interface intended to support multiple types of processing effectively.

The end user interface in the host computer usually consists of a terminal handler, a communication processor, and a language parser and translator. The major function of the host computer is to translate end user queries into the database machine language, send it to the database machine, receive results from the database machine, and finally organize and display the result. Depending on the application, it is generally accepted that user interfaces to database systems can be functionally classified into three categories:

1. Self-contained query language. The end user requests are formulated directly in a query language and are translated into the database machine language for further processing.
2. Predefined commands. Certain frequently processed transactions can be predefined and stored. The end user simply activates a stored command and supplies a few required parameters. The predefined command can be stored in the host computer or in the database machine in an internal form. The advantage of this approach is that a stored command can be compiled only once and is therefore more efficient. However, this gives the database machine an additional burden of recompiling the stored commands when the access paths are changed by database reorganization.
3. Embedded query language. Transactions written in a host language, such as FORTRAN or COBOL, may access the database by using embedded queries. A language preprocessor replaces the embedded queries with procedure calls to a run-time system, which sends the embedded queries to the database machine during transaction execution. Alternatively, the embedded queries can be preprocessed and stored in the database machine as predefined commands. During execution the transaction simply activates appropriate stored commands through calls to the run-time system. The tradeoff between these two approaches is similar to that of the stored commands.

It is clear that the self-contained query language interface is the most fundamental, and it is the focus of this paper. Our design for the database machine language interface is based on a previous experience of implementing a SEQUEL (SQL) interface for a rational database machine. Although both the query language and the database machine are based on the relational data model, discrepancies in their detailed operations have made the implementation nontrivial. Next section discusses the general database machine language requirements for effective query languages implementation. The two sections following the next section consider issues for the implementation of hierarchical and network interfaces for database machines.

2. DATABASE MACHINE LANGUAGE REQUIREMENTS

Database applications have many varieties. They differ in type of operations and data models. A database machine cannot possibly support all these access requirements directly. It is only feasible to provide a basic set of operations from which other operations can be derived. To select a proper set of operations, several requirements must be satisfied:

1. Completeness. The set of operations should be, in some sense, complete. This means that all the operations performed on the database can be derived from the basic set of operations in a straightforward manner. Otherwise,
some database operations will have to be performed in the host processor, thus making the system less effective.

Codd\(^6\) has shown that relational algebra is equivalent to relational calculus and has defined the notion of relational completeness. A query language is said to be complete if it can perform all the relational calculus operations. The completeness discussed here is not limited to the relational completeness. By completeness we mean the ability to perform all database operations effectively. Such completeness property is difficult to define formally, as data models and data languages are still continuously being developed. New data models and data languages are constantly being proposed, and it is difficult to make a list of all possible database operations. On the other hand, it seems reasonable to base the completeness requirements on the ability of performing all operations found in present database systems.

2. Efficiency and flexibility. Although most of database operations can be supported by a minimum set of basic operations, the performance of some of the operations can be greatly improved if a slightly larger set of basic operations is used. Appendix A provides some examples to illustrate the relative inefficiency of processing some derived operations by using a small set of basic operations. Some of these derived operations can be very easily implemented in the database machine.

3. High-level operations. The objectives of using a high-level-language interface are to suppress the irrelevant details of storage and access path information. This is appropriate, since the storage and access paths aspects are handled exclusively by the database machine. There are two principal advantages: (a) A high-level interface makes the processing in the front-end processor much simpler; (b) A high-level interface minimizes the number of communications between the front-end processor and the database machine and thus reduces the communication overhead.

4. Extensibility. Database machines should provide facilities for defining new operations. The new operation can be defined in terms of existing ones and included in the database machine as a stored command. These stored commands can be defined by the front-end processor system or by the end user.

A database machine language could be defined on the basis of a particular data model, such as the relational, hierarchical, or network model. In view of the above requirements, we have chosen to base the proposed database machine language on the relational data model. The advantage of the relational model is its simplicity and high level of operation. In fact, it is not surprising to see that most of the recently developed database machines are based on relational data models.

Typical relational data languages, however, do not have sufficient operations to implement the operations of other data models efficiently. Our design enhances the basic relational interface by additional operations that we feel are necessary in order to build an efficient interface for multiple data models. The proposed set of operations is given in Appendix B. We have investigated the problem of supporting several important languages, including the following:\(^1,4,8,10,11,12\) (1) Relational: QUEL, SQL, QBE; (2) hierarchical: DL/1 (IMS), FQL; and (3) network: DML (DBTG).

Table I shows the database management language requirements for these query languages.

### Table 1—Database management language requirements for various query languages

<table>
<thead>
<tr>
<th>operation</th>
<th>operand</th>
<th>result</th>
<th>QUEL</th>
<th>SOL</th>
<th>OBE</th>
<th>DL/1</th>
<th>FQL</th>
<th>DBTG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) project</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>2) select</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>3) order</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>4) group</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>5) unique</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>6) select</td>
<td>relation</td>
<td>tuple</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>7) get next</td>
<td>two relations</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>8) subtract</td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>9) intersect</td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>10) join</td>
<td></td>
<td></td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>11) join</td>
<td></td>
<td>tuple</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>12) aggregate function</td>
<td>relation</td>
<td>value</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>13) delete</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>14) insert</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>15) update</td>
<td>relation</td>
<td>tuple</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>16) assign</td>
<td>relation</td>
<td>relation</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

the following operations are for expanding select condition:

| 1) contains | two relations| boolean| V    | V   | V   | V    | V   | V    |
| 2) does not contain |         |         | V    | V   | V   | V    | V   | V    |
| 3) equal | tuple | boolean | V    | V   | V   | V    | V   | V    |
| 4) in | relation |         | V    | V   | V   | V    | V   | V    |
| 5) not in |         | boolean | V    | V   | V   | V    | V   | V    |

3. THE DATA LANGUAGE FOR A HIERARCHICAL DATABASE

Two problems must be addressed when developing a hierarchical data language for a relational database machine. The first is to transform a hierarchical data structure into relations; the other is to transform the operations on the data into the basic operations. We will illustrate these transformations by using as an example a hierarchical data language.

FQL is a form language for the manipulation of data stored in a hierarchical database.\(^8\) A hierarchical file containing nested repeating groups can be represented by a form, which is also called an unnormalized relation. More precisely, a form can be defined as follows:

\[
F = R(K,A,(F_i))
\]
where

$K$ is the set of key attributes

$A$ is the set of non-key attributes

$F_i = R_i(K_i, A_i, \{R_j\})$ are repeating groups which are also forms, defined recursively

**Example 1.**

A DEPARTMENT form can be illustrated as in Figure 1.

The hierarchical schema given on the top of the form can also be represented as follows:

**DEPARTMENT (DD#, DNAME, CLERK (CE#, CAGE), ENGINEER (EE#, EAGE), EXPERIENCE (ED#, DATE))**

where DD# is the key attribute of the form and CLERK, ENGINEER, and EXPERIENCE are repeating groups.

![Figure 1 — A department form](https://www.computerhistory.org)

In order to provide an FQL interface to a relational database machine, it is necessary to represent forms as relations. The following algorithm to perform this transformation is well known:

**Algorithm 1 (Codd)***

Decomposition of a form into relations.

**INPUT:** A form $F = R(K, A, \{F_i\})$

**OUTPUT:** A set of equivalent relations $R, R_1, R_2, \ldots$

**METHOD:** For each repeating group, define a relation consisting of the repeating group’s key and nonkey attributes and the key attributes of all its parent repeating groups. Therefore,

for $F$ we have $R(K, A)$,

for $F_i$ we have $R_i(K_i, K_{i,0}, A_i)$,

and in general, for $F_j, \ldots, k$ we have

$$R_j, \ldots, k(K_{j,0}, K_{j,1}, \ldots, K_{k,0}, A_{j,0}, \ldots)$$

**Example 2**

The form in Example 1 can be decomposed into the following four relations:

**DEPARTMENT (DD#, DNAME)**

**CLERK (DD#, CE#, CAGE)**

**ENGINEER (DD#, EE#, EAGE)**

**EXPERIENCE (DD#, EE#, ED#, DATE)**

The extensions of the relations are given in the Figure 2.

We will now show the transformation of an FQL query to relational queries. An FQL query can be defined as follows:

**SELECT** $P_1, \ldots, P_k$

**FROM** $F = R(K, A, \{F_i\})$

**WHERE** $<\text{condition}>$

Here $P_1, \ldots, P_k$ are attribute names in the form $F$ and $<\text{condition}>$ is specified in the following:

$<\text{condition}> ::= <\text{condition}> \text{ AND } <\text{condition}> |$

$\text{ NOT } <\text{condition}> | (((<\text{condition}>)) |$

$<\text{elementary phrase}> ::= R.a \ op \ v | \{R.a\}_u \ sop \ S$

$R.1.a \ op \ R.2.a | \{R.1.a\}_u \ sop \ \{R.2.a\}_u$

$\text{ op } ::= < | \leq | | \neq | | \geq | >$

$\text{ sop } ::= = C | = | = C | \subseteq | <$

$a$ is an attribute,

$R$ is a repeating group which contains $a$,

$U$ is a parent repeating group of $R$,

$\{R.a\}_u$ represents a set of values of a group by $U$,

$v$ is a value,

$S$ is a constant set.

Our strategy is to translate the FQL query into an equivalent relational query on the decomposed relations. After the relational query is executed, the resulting relation is then transformed into a form using a special database machine instruction.

**Algorithm 2**

Translation of an FQL query.

**INPUT:** a form query $Qf$

$Qf = \{ \text{SELECT: } P_1, \ldots, P_k \}$

**FROM** $F = R(K, A, \{F_i\})$

**WHERE** $<\text{condition}>$

**OUTPUT:** a result form $Fr$

**METHOD:**

(1) Construct a query on the decomposed relations:

**SELECT** $P_1, \ldots, P_k$

**FROM** $R, R_1, R_2, \ldots$

**WHERE** $<\text{link condition}>$

$\text{ AND } <\text{projection condition}>$

*For better readability, we use a SEQUEL-like syntax to describe the relational query. The translation of this query into the proposed database machine language is obvious.
The relations \( R_1, R_2, \ldots, R_m \) are the decomposed relations referenced in the \(< \) condition of the FQL query. Since we use the same name for the repeating groups and the decomposed relations, this mapping is straightforward.

Two types of relations are included in the FROM list: (1) select relation—relations required to perform the \(< \) link condition; and (2) target relation—relations required to perform the \(< \) projection condition. The \(< \) link condition is the joins created by the decomposition of a form into relations. Let \( K_x \) indicate the key of relation \( x \) and \( K_{xy} \) indicates the intersection of the keys of relations \( x \) and \( y \). The select relations and \(< \) link condition are derived from Table II.

### Table II—Select relations and \(< \) link condition

<table>
<thead>
<tr>
<th>FQL &lt; condition</th>
<th>select relations</th>
<th>(&lt; ) link condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R.a \ op \ v )</td>
<td>( R )</td>
<td>( R.a \ op v )</td>
</tr>
<tr>
<td>{( R.a )_1} sop ( S )</td>
<td>( U, R )</td>
<td>( R.K_u = U.K_v )</td>
</tr>
<tr>
<td>( R.a \ op S.b )</td>
<td>( R, S )</td>
<td>( R.K_{RS} = S.K_{RS} ) AND ( R.a ) group by ( K_u ) sop ( S )</td>
</tr>
<tr>
<td>{( R.a )_1} sop {( S.b )_w}</td>
<td>( R, S, U, W )</td>
<td>( R.K_u = U.K_v ) AND ( S.K_w = W.K_w ) AND ( U.K_{RW} = W.K_w ) AND {( S.b ) group by ( K_u )} sop {( S.b )_w}</td>
</tr>
</tbody>
</table>

The \(< \) projection condition are the joins between the target relations and certain select relations. Let \( Z \) denote a target relation. The \(< \) projection condition are derived as shown in Table III:

### Table III—Projection condition

<table>
<thead>
<tr>
<th>FQL &lt; condition</th>
<th>(&lt; ) projection condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R.a \ op v )</td>
<td>( R.K_{RS} = Z.K_{RS} )</td>
</tr>
<tr>
<td>{( R.a )_1} sop ( S )</td>
<td>( U.K_{uw} = Z.K_{uw} ) AND ( S.K_{uw} = Z.K_{uw} )</td>
</tr>
<tr>
<td>( R.a \ op S.b )</td>
<td>( R.K_{RS} = Z.K_{RS} ) AND ( S.K_{uw} = Z.K_{uw} ) AND ( W.K_{uw} = Z.K_{uw} )</td>
</tr>
</tbody>
</table>

(2) Execute the transformed query and obtain a result relation \( B_f \).

(3) Group \( B_f \) into form using the keys of \( R_1, R_2, \ldots, R_m \).

**4. THE DATA LANGUAGE FOR THE NETWORK APPROACH**

To support a DBTG interface on a relational database machine, it is essential to solve the following two problems:

1. Transformation of DBTG data structures into relational data structures.
2. Transformation of DBTG data manipulations into queries for the database machine language.

The network data structure of a DBTG system is more complex than the hierarchical approach. It has two basic constructs: RECORD type and SET type. A RECORD type is a hierarchical structure consisting of data items and repeating groups. It is similar to the concept already detailed in the previous section. The SET type represents links among RECORD types. Each SET consists of an OWNER record type and one or more MEMBER record types. This is typically implemented in DBTG as a pointer chain (one direction or binary direction) that starts from one owner instance and links up all member instances. In a relational database machine, SET types can be removed by a normalization procedure similar to Algorithm 1. Any data access traversing a SET instance can be performed by an equivalent join operation on the common attributes of the OWNER relation and the MEMBER relation. To make this operation possible in general, however, it is necessary to introduce an attribute in the MEMBER relation to represent the rank of each record within a SET instance.

The performance of the data access can be enhanced by additional data structures. It was indicated in Yao\(^{13}\) that three types of additional storage organizations can be defined: (1) indexing—defining a tree structure to provide random access to records in a relation, (2) linking—defining a pointer chain similar to that of the SET implementation, and (3) clustering—storing the member records of a set close to its owner. In another proposal, an index structure is designed to store or-
ordered (owner, member) pairs of set instances. All of these data structures may be employed in a relational database machine to enhance data access performance. The use of these features, a database design problem, obviously depends on system tradeoffs and their availability in a particular database machine.

Example 1

Given a DBTG data structure for a supplier-parts database (Figure 3).

Record types:
- S(S#, SNAME, STATUS, CITY)
- P(P#, PNAME, COLOR, WEIGHT, CITY)
- SP(DATE, QTY)

Set types:
- S-SP
- P-SP

The following equivalent relations may be created:

\[
\begin{align*}
S(S#, SNAME, STATUS, CITY) \\
P(P#, PNAME, COLOR, WEIGHT, CITY) \\
SP(S#, P#, DATE, QTY, SRANK, PRANK)
\end{align*}
\]

The following additional data structures can also be created:

UNIQUE CLUSTERING INDEX INDS ON S# OF S
UNIQUE CLUSTERING INDEX INDP ON P# OF P
LINK S-SP FROM S(S#) TO SP(S#)
ORDER BY P#
LINK P-SP FROM P(P#) TO SP(P#)
ORDER BY S#

TABLE IV—The transformation of SET traversal operations

<table>
<thead>
<tr>
<th>DML of DBTG system</th>
<th>Database Machine Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE 'New York' TO CITY IN S</td>
<td>SELECT S.ALL FROM S WHERE S.SNAME = 'New York'</td>
</tr>
<tr>
<td>FIND ANY S USING CITY IN S</td>
<td>FROM S</td>
</tr>
<tr>
<td>WHERE S.SNAME = 'New York'</td>
<td></td>
</tr>
<tr>
<td>FIND &lt;position&gt; SP WITHIN S-SP</td>
<td>SELECT NEXT SP.ALL FROM SP WHERE S.S# = SP.P# AND S.SRANK = &lt;position&gt;</td>
</tr>
<tr>
<td>FIRST</td>
<td></td>
</tr>
<tr>
<td>NEXT</td>
<td></td>
</tr>
<tr>
<td>PRIOR</td>
<td></td>
</tr>
<tr>
<td>LAST</td>
<td></td>
</tr>
<tr>
<td>N-TH</td>
<td></td>
</tr>
<tr>
<td>FIND OWNER WITHIN S-SP</td>
<td>SELECT NEXT S.ALL FROM S WHERE S.S# = SP.P#</td>
</tr>
<tr>
<td>MOVE 'p1' TO P# IN SP</td>
<td>SELECT NEXT SP.ALL FROM SP WHERE SP.P# = 'p1' GROUP BY SP.S#</td>
</tr>
<tr>
<td>FIND SP WITHIN S-SP CURRENT USING P# IN SP</td>
<td>WHERE SP.P# = 'p1' GROUP BY SP.S#</td>
</tr>
<tr>
<td>ADD 20 TO STATUS IN S</td>
<td>UPDATE S SET S.STATUS = S.STATUS + 20</td>
</tr>
<tr>
<td>MODIFY S</td>
<td></td>
</tr>
<tr>
<td>CONNECT SP TO S-SP</td>
<td>UPDATE SP SET S.SRANK = MAX(S.SRANK) + 1 GROUP BY SP.S#</td>
</tr>
<tr>
<td>DISCONNECT SP FROM S-SP</td>
<td>UPDATE SP SET S.SRANK = 0</td>
</tr>
</tbody>
</table>

The data manipulation language of DBTG system is a procedure-oriented language. Its operands are RECORD and SET instances. The operations on a hierarchical RECORD type (i.e., RECORD that contains repeating groups) have been discussed in the previous section. The transformation of SET traversal operations is given in Table IV.

Data manipulation using a back-end database machine requires additional communication time between the host and the back end. A data traversal interface can be very ineffective because of the number of times communications are initiated. Although the additional communication cost is unavoidable in this type of data access, it can be significantly reduced if enough buffer space is provided in the back-end and host systems.

To consider the relative costs of SET traversal operations, let us define a few basic access cost equations. The following figure shows the two communication links in a host/back-end system.

\[
f(x) = (c_1 + c_2 * w) * \lceil x/b \rceil + r * x
\]

where \(x\) is the number of bytes transferred, \(b\) is the buffer size, \(\lceil x/b \rceil\) is the number of transmissions required, \(w\) is the average waiting time, \(c_1\) is the number of packets sent in each transmission, \(c_2\) is the transmission overhead, and \(r\) is the transmission speed. In our implementation of a host interface using a 9600-baud connection, the parameter values are \(c_1 = 20, c_2 = 4, w = 20\ \text{ms}, \text{and } r = 0.83.\)

The transmission time between the back-end and the disk can be similarly estimated:

\[
g(y) = wd * \lceil y/b \rceil + y/s
\]
where \( y \) is the number of bytes transferred, \( wd \) is the average disk access time, and \( s \) is the disk transfer rate. For 1 MB disk transfer rate, \( s = 1000 \text{ byte/ms} \). \( wd \) is assumed to be 50 ms.

We further assume that the SET type has an OWNER record type \( O \) and one MEMBER record type \( M \). The record size of \( O \) is \( p \), the record size of \( M \) is \( q \), and the mapping between owner and member is \( 1:n \). In a relational system the time required to access all the member records for a given owner record can be estimated by

\[
t_1(p, q) = t_0 + f(p) + f(n^*q) + g(p) + g(n^*q)
\]

where \( t_0 \) is the communication initiation time, assuming \( t_0 = 863 \text{ ms}, p = 128b, q = 256b \). In the case of DBTG SET traversal, one owner and one member record are transmitted each time. The transmission time required is therefore:

\[
t_2(p, q) = t_0 + n^* (f(P) + f(q) + g(p) + g(q))
\]

Assuming that the buffer size in the back-end is \( b \). Since there are a total of \( n \) records of size \( q \) to be accessed, the number of times the disk must be accessed is \( m = (n^*q)/b \).

The transmission times required for each access are illustrated as follows:

1st: \( f(p) + f(q) + g(p) + g(b) \)
2nd: \( f(p) + f(q) + g(p) + 0 \)
\ldots
mth: \( f(p) + f(q) + g(p) + g(b) \)
\( m + 1th: f(p) + f(q) + g(p) + 0 \)

The total time required is therefore

\[
t_3(p, q) = t_0 + n^* (f(P) + f(q) + g(p) + g(q)) + m^*g(b)
\]

The results of these cost equations are plotted in Figures 4 and 5. It is evident that the relational data access is much more efficient (see \( t_1 \)). Network data access without using a buffer can be very inefficient, since multiple accesses are required. The curve \( t_2 \) gives the access cost without buffer and \( t_3 \) gives the access cost with a 256-byte buffer. Figure 5 shows the sensitivity of access time to various buffer sizes. It is interesting to note that even the worst case of relational data access is still far better than the best case of network data access. Therefore the support of a network data model interface on a relational database machine is feasible, but relatively inefficient.

V. CONCLUSION

The choice of a data model and set of operations for a database machine has a great effect on the efficiency of its user interface implementation. Such data models and interfaces must be carefully determined during the initial design stage. A properly designed database machine language will make it feasible and efficient in supporting multiple data languages.

This paper has presented a set of database machine language requirements. These requirements are based on our experience in implementing the XQL system as a user interface to a database machine. The XQL system provides the SEQUEL query language, menu system, interactive form processor, and report writer. Host language interface is provided by imbedding SEQUEL statements in application programs.

The algorithms for supporting hierarchical and network interfaces by using a relational database machine are introduced. The relative performances for supporting relational
and network user interfaces are compared. Our analysis shows that it is possible to support multiple user interfaces by using a relational database machine. The support for a network user interface is less efficient when compared with the relational approach. The performance can be improved by providing additional buffer space. However, even with a very large buffer size, the performance of the network user interface is still significantly worse than that of the relational interface.

APPENDIX A

If the set of basic operations in the database machine is not sufficient, it can sometimes be difficult to support certain data language. Examples are provided here of our implementation of the system XQL/IDM based on a database machine. For better readability, we use a QUEL-like language to describe the object database machine language code.

**Example 1**

XQL: select name
from Employee
where salary > 2000
minus
select name
from Department
where manager = "D. Smith"

The object code:
range of e is Employee
retrieve into Temp1(e.name)
where e.salary > 2000

range of d is Department
retrieve into Temp2(d.departno)
where d.manager = "D. Smith"

range of t1 is Temp1
range of t2 is Temp2
delete t1
where t1.departno = t2.departno

retrieve(e.name)
where e.departno = t1.departno

destroy Temp1
destroy Temp2

This query can be more efficiently handled if the database machine can support a set operation in search conditions.

**APPENDIX B—A PROPOSED SET OF OPERATIONS FOR THE RELATIONAL DATABASE MACHINE**

1. **Projection:** Project \((a_1, \ldots, a_p)\) from \(R\)
   
   where \(a_i\) is an attribute name of the relation \(R\).
   
   The result is a new relation obtained from \(R\) by deleting from each tuple all attributes not listed in \((a_1, \ldots, a_p)\).

2. **Selection:** Select from \(R\) where <condition>
   
   Here the <condition> is a Boolean predicate which specifies the search condition. Each term in the predicate contains an arithmetic comparison operator and two operands that are constants or attribute names of \(R\). The terms in the predicate are linked together by the logical operators AND, OR, and NOT. The result is a new relation which consists of all tuples from \(R\) satisfying the given <condition>.

3. **Selection of next tuple:** Select next from \(R\) where <condition>
   
   Same as 2 except only one tuple is returned. When a relation is ordered this tuple is the one logically next to the last selected tuple. The first execution of this operation returns the first tuple in \(R\) satisfying <condition>.

4. **Ordering:** Order \(R\) by \(a\)
   
   Where \(a\) is an (or a set of) attribute name of the relation \(R\). The result is a new relation containing all the tuples from \(R\) sorted on the attribute (set) \(a\).

5. **Grouping:** Group \(R\) by \(a\)
   
   Where \(a\) is an (or a set of) attribute name of the relation \(R\). The result is a new relation containing all the tuples from \(R\) "grouped" on the values of the attribute (set) \(a\). This is useful when aggregate functions are to be performed.

6. **Unique:** Unique \(R\)
Relation is sorted on its key and tuples with identical keys are removed.
7. Union: \( R \cup S \)
   The relations \( R \) and \( S \) are identically defined. The result
   is a new relation containing all tuples from both \( R \) and \( S \).
8. Subtraction: \( R \setminus S \)
   \( R \) and \( S \) again must be identically defined. The result is
   a new relation containing all tuples that are in \( R \) but not in \( S \).
9. Intersection: \( R \cap S \)
   \( R \) and \( S \) again must be identically defined. The result is
   a new relation containing all tuples that are in both \( R \) and \( S \).
10. Joining: \( R \bowtie_S S \)
    Each term in the join \( <\text{condition}> \) is of the form \((a \text{ op } b)\) where \( a \) is an attribute name of \( R \), \( b \) is an attribute
    name of \( S \), and \( \text{op} \) is an arithmetic comparison operator.
    The result is a relation containing all possible concatenations of tuples
    from \( R \) and \( S \) satisfying the given \( <\text{condition}> \).
11. Get next join tuple: Get next \( R \bowtie_S S \) where \( <\text{condition}> \)
    The join is defined as before. The result returned is the next
    tuple from the join result.
12. Aggregate functions: \( F (R, a) \)
    \( F \) is one of the aggregate functions: average, sum, min,
    max, count. The function is performed on the column
    \( a \) of the relation \( R \). The result returned is a single scaler
    quantity. If the relation is grouped, then the aggregate
    function is performed for each group.
13. Delete: Delete from \( R \) where \( <\text{condition}> \)
    All the tuples from \( R \) satisfying the condition are deleted.
14. Insert tuple: Insert \( T \) into \( R \)
    The tuple as given in \( T \) is inserted into the relation \( R \).
15. Update: Update \( <\text{list}> \) in \( R \) where \( <\text{condition}> \)
    \( <\text{list}> \) is a list of update actions. Each action is of
    the form \( A_i = u_i \) where \( A_i \) is an attribute name of \( R \)
    and \( u_i \) is the new value to be assigned. The update list is
    performed for all tuples satisfying the given \( <\text{condition}> \).
16. Assignment: \( S := R \)
    The action is to store \( R \) into database as a new relation
    named \( S \).

The following operations are for expanding select condition. The result of these operations is a Boolean value that can be served as a condition.

1. Contains: \( R \) contains \( S \)
   \( R \) and \( S \) are two relations of arity \( k \). The result is True
   if all tuples in \( S \) are tuples in \( R \), False otherwise.
2. Does not contain: \( R \) does not contain \( S \)
   \( R \) and \( S \) are two relations of arity \( k \). The result is True
   if there is any tuple in \( S \) is not tuple in \( R \), False otherwise.
3. Equal: \( R \) equals to \( S \)
   The result is True if \( R \) and \( S \) have the same tuples, False
   otherwise.
4. In: \( T \) is in \( R \)
   \( R \) is a relation, \( T \) is a tuple. The result is True if there
   is a tuple in \( R \) equal to \( T \), False otherwise.
5. Not in: \( T \) not in \( R \)
   The result is True if there is no tuple in \( R \) equal to \( T \),
   False otherwise.

These operations are the most common ones. The database machine should contain all of these operations. On the basis of these operations, the queries expressed in most data models can be easily handled.

REFERENCES