An associative search language for data management

by AMAR MUKHOPADHYAY and ALIREZA HURSON

The University of Iowa
Iowa City, Iowa

INTRODUCTION

Recent years have witnessed a widespread and intensive effort to develop systems to store, maintain and access data bases of varied size. Such systems are referred to as DBMS—Data Base Management Systems. In different areas, such as artificial intelligence, management information systems, military and corporate logistics and medical diagnosis, a wide variety of DBMS exist. All these systems have generally been implemented on conventional computers, which are based on the von Neumann design. In this design, operations will be performed on the information in the memory by means of their addresses. Because of the size of typical data bases and costs of memory, we cannot hold all information in the main memory and swapping converts the search problem to a transportation problem. Present-day systems have to transfer large sets of data from their mass storage to the CPU, where simple compare-functions are performed in order to separate relevant data from irrelevant data. The transfer channels with their limited capacity form the main bottleneck of this system and as a result, great efforts have been made to reduce the necessary data flow by means of sophisticated software systems and additional redundancy such as index tables and inverted files. By these techniques, address of information will be obtained from a directory. Although directory partially solved the bottleneck problem, it nevertheless created some problems. The directory should logically be kept in the main memory. If we are dealing with large data bases, naturally we are dealing also with large directories, and large directories occupy a large portion of the memory. Also, the use of directories will create some complexity in the search, update and delete algorithms. Conventional computers are all based on numerical operations. The necessity of designing new hardware based on non-numerical operations has been discussed in detail by one of the authors.1 In contrast, use of associative or content addressable memories and hardware design based on non-numerical operations as well as numerical operations causes information stored at unknown locations to be processed efficiently on the basis of some knowledge of its content.

In developing new architectures for future machines some of the most important trends in hardware and software technology must be brought into focus. On the hardware side, the significant trends are the development of LSI and VLSI technology which allow increased functionality of hardware components coupled with a drastic reduction in cost per function. This implies that new architectures must exploit this trend to incorporate more software functions by hardware. The second important trend in hardware is the development of serial access secondary storage devices such as CCD or magnetic bubble memory (MBM). It is forecast that by 1980/81 a 1Mbit memory will be available for both CCD and MBM, which will emerge as powerful competitors for disks and tapes.2 We could make the assumption that a major part of future data base systems will reside in such shift-register type memories. The proposed architectures in this paper will, of course, work for other kinds of rotating secondary devices such as disks. A common feature of all these secondary devices is that they are block-oriented, that is, access to a block is slow but data flow rate is high. If the processing of information has to be done by CPU, blocks of information need to be transferred back and forth from the CPU to secondary devices. The new machine architectures, therefore, should attempt to provide processing capabilities outside of CPU and along with secondary devices in the form of associative hardware which will operate on information on the fly. This idea was originally suggested by Slotnick,3 and in recent years a significant amount of effort has been expended in the development of similar architectures in data base applications.4–17

There are, however, several disadvantages to an associative approach—first, hardware for associative processing still has to prove its cost-effectiveness compared to the existing software implementations: second, future innovations in computer systems design and architecture have to confront inertia and large investments on the existing systems. Although the principles of associative processing are as old as second generation computers (and there is an extensive body of literature dealing with many aspects of this phenomenon in computer design), the major bottleneck for the development of associative systems has been the degradation of speed (the memory cycle time) for the required size of the memory. The demand on the size of the memory can be reduced by intelligent design of the architecture. For example, the design of the Lee machine is impractical because the design assumes the main memory system to be associative. In an earlier paper one of the authors1 showed that a small associative memory could em-
ulate the functions of the Lee machine if information could be passed over it. In the data base context, we are looking for similar architecture and therefore the size of the associative memory will be reduced. This, coupled with the development of VLSI technologies, shows good promise for the use of associative hardware for future machines. A relatively recent associative processor ALAPE demonstrates the feasibility of making large associative processors practical.

This paper will introduce a high-level data base language, ASL—Associative Search Language—which is suitable for direct hardware implementation by associative hardware. The architecture for ASL and its proposed hardware implementation will be described in future papers. This paper is concerned with a preliminary definition and specification of ASL.

On the software side, as far as the data base management systems are concerned, an overall recent trend is characterized by the phrase data independence. Data independence means that the user is able to perform retrieval and storage operations on the data base depending on the information content without having to deal with representational details. Codd proposed an important data model, the now-famous relational model, which provides the user with a model of data based on content or data value only. Based on Codd's model, a number of families of data sub-languages and query languages have been proposed—one class based on relational calculus (viz., DSL, ALPHA, INGRES, DAMAS), another class based on relational algebra (PRTV, SQUIRAL), and a third class which adopts a "mapping" approach based on the relational algebra (SEQUEL, Query-by-example). The reader is referred to the text by Date or the review article by Chamberlin for more details. In the late 60's, an associative language, ASP, was developed in the context of list and language processing.

The conceptual framework of ASL is very similar to that of SQUARE, a data sub-language which formed the basis for the development of SEQUEL. The similarities of these two languages are reflected in the definition of the basic operations which brings forth explicitly the associative nature of processing, and also in avoiding the use of quantifiers required by languages based on relational calculus. Both ASL and SQUARE are relationally complete, provide facilities for query, insertion, deletion and update operations, and are meant for non-professional programmers who do not possess a high degree of mathematical sophistication. However, the originators of SQUARE did not emphasize the associative nature of the primitive operations since the language was not considered for hardware implementation. The authors are of the opinion that for hardware implementations of data base operators, development of a language like SQUARE or ASL is essential. The relationship between a high-level user language such as SEQUEL or Query-by-example with SQUARE or ASL is similar to the relationship between FORTRAN or PL/1 and assembly language except that the definition of ASL is independent of actual hardware implementation. We also claim that ASL has several advantages over SQUARE, as follows: the structure of the language is more precise and the expressions in the language can be handled by parsing algorithms that are available for well formed arithmetic expressions in programming languages which, in the hardware context, define a precise sequence of controlling operations by the hardware: ASL allows parallel computations of a set of relations and is based on variable size information fields with complete independence: ASL is also easily extendable to multi-dimensional as well as non-relational data bases; and finally, ASL provides potential for the development of a FORTRAN-like language for data base applications.

**ASL—AN ASSOCIATIVE SEARCH LANGUAGE FOR DATA BASE MANAGEMENT**

The language ASL—Associative Search Language—is a high-level data base language designed for information retrieval and storage operation on data bases using associative principles for basic operations. The language has been defined based on the relational model of data presented by Codd, although extensions to non-relational models are possible. A partially complete syntax for ASL is included in the Appendix.

The fundamental operation in ASL is a search on a data base with respect to criteria (search arguments). The result is a relation yielding the retrieved information. This operation can be thought of as an assignment statement $W = X$, where $W$ is a relation and $X$ is composed of three parts.

**HOW WHERE WHAT.**

where "HOW" specifies the search arguments (search criteria), "WHERE" specifies the relation name and "WHAT" specifies the output domains of the relation. If $R$, $C$, $D$ stand for WHERE, HOW and WHAT respectively, we can say $R$ is a binary transformation operation whose operands are $C$ and $D$, and can be expressed as

$$C \circ D.$$  

$C$ is a set of sets $(C = (c_0, \ldots, c_k))$ where each element in this set $(c_i, 0 \leq i \leq k)$ is an unordered list of search arguments; for $i=0$, $c_i = \phi$, denotes an empty set of search. $D$ is also a set of sets $(D = (d_0, \ldots, d_n))$ where each $d_i (0 \leq i \leq n)$ is a non-empty unordered list of domains over $R$. When $d_i$ includes all the domains of $R$, it will be denoted by $\delta$. The result of operation $(C \circ D)$ on $C$ and $D$ is a set of relations $(W_0, \ldots, W_n)$ where the domains of $W_0 (0 \leq i \leq n)$ are the same as those of $d_i (0 \leq i \leq n)$. In our implementation we assume $C$ and $D$ as well as $W$ are singleton sets.

The informal presentation of ASL will be with respect to the employee-department data base as follows:

**E(ENO, ENAME, DEGREE, LOCATION)**  
**D(DNO, DNA, DHEAD, LOCATION)**  
**ED(ENO, DNO, HOUR)**

where $ENO$, $DNO$ stand for Employee and Department number, $ENAME$, $DNAME$ stand for Employee and Department name, and $HOUR$ shows how many hours each
employee works in each department. Each query will be expressed in two forms, first as assignment statements $W := X$, where $X$ is specified using conventional notations in data base literature, and then in the form of ASL statements.

**RETRIEVAL OPERATIONS**

*Simple retrieval*

- Get department number of all departments
  $W := \phi DNO$

  According to our current implementation of ASL, this query will be presented as below in ASL language:

  $D.
  W = [D]DNO$

  where $D.$ declares the name of the relation. Note the set of rules in the appendix shows that each program will be started with the declaration of all relations used in that program. Henceforth, in all the examples, we will omit the declarative statement.

- Get all the information of all employees:
  $W := \phi E$

  which in fact is equivalent to $W := E$. This query in ASL will be written as

  $W = [E]$.

*Qualified retrieval*

- Get employee numbers for all employees in LOCATION L1 with DEGREE>3:
  $W := (LOCATION = 'L1') \wedge (DEGREE > 3) [E] ENO$

  In ASL it is $W = ((LOCATION EQ 'L1') \& \& (DEGREE GT 3))[E] ENO$.

  This example illustrates that $C$ could be specified by a predicate. The output $W$ is a relation whose tuples are $ENO$ of relation $E$.

*Complex retrieval*

Up to now, retrieval operations with simple search arguments and over one relation which yields a sub-relation of the original relation have been discussed. A sub-relation is a relation derived from a given relation by the selection of a row, the projection on columns, and then removing the redundant tuples. We will now consider more complex retrieval operations.

Retrievals that yield sub-relations by using more than one relation:

- Get employee names for employees who work in department D1:

  $W := (((DNO = 'D1') [ED] ENO) [E] ENAME)

  The nested nature of operations has been illustrated by this example—first, the relation $ED$ will be searched for all department numbers equal to D1 yielding a subrelation of $ENO$. This sub-relation will then be used as an argument to obtain $ENAME$. In fact, this sub-relation will be searched in associative fashion. The operation can be seen as:

  $X := (DNO = 'D1') [ED] ENO$

  $W := V[ED][ENAME]$

  In ASL we have $W = ((DNO EQ 'D1') [ED][ENO][E][ENAME]$.

  The right-hand side of an assignment statement is called a set expression. A set expression, like arithmetic expressions, can be written in infix notation, as previously, or in postfix notation as

  $W := (DNO = 'D1') [ED ENO [ENAME]$.

  defining the same relation $W$. The use of postfix notation avoids redundant use of parentheses and also has the advantage with respect to compilation and the design of hardware controller that implements the language. We will write the set expressions in infix notation. The predicate expressions will always be parenthesized.

- Get employee numbers for employees who work in department managed by 'SMITH':

  $W := (((DHEAD = 'SMITH') [D] DNO) [ED] ENO)$

  and in ASL:

  $W := ((DHEAD EQ 'SMITH') [D] DNO) [ED] ENO$.

- Get employee names for employees who work in department managed by 'SMITH':

  $W := (((DHEAD = 'SMITH') [D] DNO) [ED] ENO) [E] ENAME$

  and in ASL:


A complicated set expression like the previous one could be parsed starting from the postfix equivalent expression using dynamic relation variables such as $A$ and $B$:

$W := (DHEAD = 'SMITH') DNO [ED ENO [ENAME]$.

The parsing will proceed from left to right and the computation will proceed as:

$A := (DHEAD = 'SMITH') [D] DNO$

$B := A [ED ENO$

$W := B [ENAME]$

We will now consider retrievals that do not necessarily
yield sub-relations and use several construction operators like tuple-compatible union (U), intersection (∩), complementation (¬), cartesian product (×), join, division (÷) and restriction (|), along with Boolean operations AND (∧), OR (∨), NOT (¬), and predicates using these and relational binary operators <, >, =, ≤, ≥, =. The normal set of arithmetic functions and ‘library’ functions of counting and averaging, etc. could be super-imposed on these. In many of the expressions written that follow, more than one statement, sometimes with implicit loops, will be used. Although our language is relationally complete, it does not satisfy the single statement requirement. The data sub-languages using relational calculus that satisfy the single sentence constraint but use existential and universal quantifiers do not seem to be so easily understood by even reasonably sophisticated users. Furthermore, our motivation also has roots in the hardware implementation and provides us with a simple interface design for translating queries into hardware control signals. Only the first forms of the queries (viz., W := X) are specified; ASL statements are omitted.

- Get a list of all employees with the amount of hours they work for each department:
  \[ X := ED \]
  \[ W := \bigcup_{x \in X} [x \otimes (\text{ENO} = x(\text{ENO})) \otimes \text{ENAME}] \]
  where \( x(\text{ENO}) \) denotes the \text{ENO} field of \( x \in X \).

- Get employee names for employees who do not work in department D1:
  \[ W := (\phi \otimes DNO) \]
  \[ -((\text{DNO} = 'D1') \otimes \text{DNO}) \otimes \text{ENAME} \]

- Get all employee and department names. Pair so that the indicated employee and department are located in the same place:
  \[ X := \bigcup_{x \in x} [(\text{LOCATION} = x) \otimes \text{ENAME}] \]
  \[ \otimes [(\text{LOCATION} = x) \otimes \text{DNAME}] \]

- Get employee names for employees who work in all departments:
  \[ X := \phi \otimes \text{ENO} \]
  \[ Y := \phi \otimes \text{DNO} \]
  \[ W := \{y \otimes x(\text{DNO}) = y\} \otimes \text{ENAME} \]

**JOIN OPERATION**

Let \( \theta \) be any relation =, <, >, ≤, ≥. Then the \( \theta \) join of a relation \( R \) on domain \( D_R \) with relation \( S \) on domain \( D_S \) is defined (using Codd’s notation) as:

\[ R[D_R \theta D_S] = \{(rs) | r \in R \land s \in S \land (r(D_R) \theta s(D_S))\} \]

where \( r(D_R) \) and \( s(D_S) \) are assumed to be \( \theta \)-comparable. The quantity \( (rs) \) denotes a concatenation of tuples. It is possible to specify a subset of domains to be concatenated without repeating the comparand domains. Let us denote these domains by \( D_R^* \) and \( D_S^* \). Then we can define our set expression equivalent of \( \theta \)-join as:

\[ Y := \phi \otimes D_R \]
\[ Z := \phi \otimes D_S \]
\[ X := \{(y,z) | y \in Y, z \in Z \land y(\theta) z\} \]
\[ W := \bigcup_{\alpha \in x} \{y \otimes (D_R^*) \otimes \alpha \otimes \{z \otimes (D_S^*)\} \]

where \( \theta \) could be either \( y \) or \( z \) or \( (y,z) \).

**Example**

Let

\[
\begin{array}{ccc}
A & B & C \\
\hline
a & 1 & 1 \\
b & 1 & 2 \\
c & 2 & 5 \\
c & 3 & 3 \\
\end{array}
\quad
\begin{array}{ccc}
R & S & D \\
\hline
a & 2 & u \\
b & 3 & u \\
\end{array}
\]

and let \( D_R = D_s = C \), \( D_R^* = (A,B) \), \( D_s^* = (D) \), \( r = y \) and \( \theta \) be equality (=) so that we are computing equijoin. Then

\[ Y := \phi \otimes C = (1,2,3,5) \]
\[ Z := \phi \otimes C = (2,3,4) \]
\[ X := ((2,2), (3,3)) \]
\[ W := \bigcup_{\alpha \in x} \{y \otimes (A,B) \otimes \alpha \otimes \gamma \otimes \{z \otimes (D_s^*)\} = \begin{bmatrix} b & 1 & 2 & u \\
                        c & 3 & 3 & u \end{bmatrix} \].

The computations of the set \( X \) do not follow the format of a set expression although it is a well defined set. Similar well defined set computation procedures must be part of ASL facilities.

**DIVISION OPERATION**

Let \( D_R, D_S \) denote domains in relation \( R \) such that \( D_R \cap D_S = \phi \). Let \( D_S \) be a domain in relation \( S \) such that it is comparable with \( D_R \). Then the division operator

\[ R[D_R \div D_S] = \{D_S \}
\]

is defined by the set expressions

\[ X := \phi \otimes D_S \]
\[ W := \bigcap_{\alpha \in x} [y \otimes (R \otimes D_S] \]
Example

Let $D_1 = C$ and $D_2 = D$. Let $D_3 = (B)$. Then

$X := \psi \left( D = (x, y) \right)$

$W := (x \left( R \right) \cap (y \left( R \right)) = (11, 12) \cap (11) = (11)$

RESTRICTION OPERATION

The restriction operation is simple to model in our scheme. The search arguments are defined in terms of a binary relation $\theta$ between comparable subsets of domains $D_1$ and $D_2$ of the relation $R$, and an output domain $D_3$ is specified. Thus

$W := ((D_1 \theta D_2) \left( R \right) D_3)$

Example

Let $D_1 = B$, $D_2 = C$, $D_3 = (A, B, C)$, $\theta = >$. Then

$W := ((B > C) \left( R \right) (A, B, C)) = \begin{bmatrix} p & 2 & 1 \\ q & 2 & 3 \\ q & 5 & 4 \\ r & 3 & 3 \end{bmatrix}$

STORAGE OPERATIONS

Storage operations such as "update," "insert," and "delete" can be performed easily by accessing appropriate tuple(s) from the appropriate relation followed by modification of information. They can be specified in two forms:

1. Set Expression $:= \text{new-data}.$

   This is a simple store operation that changes the existing information to new values without regard to previous value and does not alter the relation otherwise.

2. $W := \text{a modified relation expressed by appropriate transformation of the set expression}.$

We give some examples below:

- **Update**
  change the location of department $D1$ to "NEW YORK":

  $(DNO='D1') \left( R \right) LOCATION := 'NEW YORK'$

- **Delete** all employees whose degree is less than 4.

  $E := E \left(-\text{DEGREE} < 4\right) \left( R \right) D_3$

  In ASL it is $E := E \left(-\text{DEGREE LE '4'}\right) [E]$.

- **Delete** relation $E$.

  $E := E - E$

  In ASL it is simply $E -$.

- **Add** a new employee to EMPLOYEE relation.

  $E := E \cup (ENO = '123'; ENAME = 'JOE', DEGREE = '5', LOCATION = 'ROME'))$

  and in ASL we have

  $E \cup (ENO = '123'; ENAME = 'JOE', DEGREE = '5'; LOCATION = 'ROME'))$

- **Add** 5 to Hour for employee number $N$:

  $x := N \left( R \right) HOUR$

  $x := x + 5$

  $(ENO=N) \left( R \right) HOUR := x$

  In ASL it is given by

  $(ENO=N)[ED]HOUR := HOUR + 5$

CONCLUSION

A preliminary definition of a data base management language based on associative primitive operations along with its BNF specification has been presented in this paper. Detailed implementation of ASL as a language processor and hardware architecture based on ASL will be reported in future papers. Because of the upsurge of recent interest in nonnumeric computation, development of specialized language processors will result in simplification of software systems and will lead to the development of specialized hardware processors which will exploit the current trends in technology.

ACKNOWLEDGMENT

The work is supported by National Science Foundation grant MCS76-04763.

REFERENCES

APPENDIX

The following syntax, which is based on BNF form, gives an intuitive understanding about the structure of ASL. This is not a complete set of rules—the syntax has been simplified for ease of understanding.

\[
\begin{align*}
\text{(Program)} & : = (\text{Dec list}) (\text{Re list}) \\
\text{(Dec list)} & : = \text{empty} | \\
& \quad \text{(Dec list)} (\text{Relation Id}) \\
\text{(Re list)} & : = (\text{Re list}) (\text{Relation}) \\
\text{(Relation)} & : = (\uparrow \text{Relation}) | \\
& \quad (Q \text{ Relation}) | \\
& \quad (T \text{ Relation}) \\
\text{(Up Relation)} & : = (\text{Ch Relation}) \\
& \quad (\text{Add Relation}) | \\
& \quad (D \text{ Relation}) \\
\text{(Add Relation)} & : = \text{(Relation Id)} \cup (\text{Domain-value list}) \\
& \quad (\text{Relation Id}) \cup (\text{Domain-type list}) \\
\text{(D Relation)} & : = (\text{Relation Id}) \setminus (\text{Set expression}) \\
\text{(Ch Relation)} & : = (\text{Search set}) (\text{Relation Op}) (\text{Domain-value list}) \\
& \quad (\text{Set expression}) (\text{Relation Op}) (\text{Domain-value list}) \\
\text{(Q Relation)} & : = (\text{Set expression}) \\
& \quad (\text{Relation Id}) \bowtie (\text{Relation Id}) \\
& \quad (\text{Relation Id}) (\text{Join}) (\text{Relation Id}) \\
& \quad (\text{Relation Id}) (\text{Divide}) (\text{Relation Id}) \\
& \quad (\text{Relation Id}) (\text{Restriction}) (\text{Output Set}) \\
& \quad (\text{Relation Id}) (\text{Log-set Op}) (\text{Relation Id}) \\
\text{(T Relation)} & : = (\text{Relation Id}) (\text{Equal}) (\text{Output Set}) \\
& \quad (\text{Relation Id}) = (\text{Up Relation}) \\
\text{(Set Expression)} & : = (\text{Search Set}) (\text{Relation Op}) (\text{Output Set}) \\
& \quad (\text{Set expression}) (\text{Relation Op}) (\text{Output Set}) \\
& \quad (\text{Relation Id}) \\
\text{(Search Set)} & : = \text{empty} | \\
& \quad (\text{expression}) \\
& \quad (\text{Search Set}) (\text{Log-set Op}) (\text{expression})
\end{align*}
\]

* (Expression) is any usual arithmetic expression.