A TECHNIQUE FOR IMPLEMENTING A SET PROCESSOR

W. T. Hardgrave
Department of Information Systems Management
University of Maryland

Abstract

Extended set theory is an effective method for describing the complex data structures needed to support large-scale data base applications. This paper describes the incorporation of the salient concepts of extended set theory into a general tool for the design and implementation of large-scale database systems. This tool is called a set processor. This implementation is based on the representation of sets of positive integers as bit strings and the application of the Cauchy/Cantor Diagonal Method.

1.0 Introduction

Several groups and individuals have suggested ways of providing data description; see, e.g. CODASYL's DDLC (8), Sibley (22) and Senko (20). Extended set theory, as developed by D. L. Childs, (4,5,6) augments classical set theory in such a way that set processing may be implemented more readily on computers. While a rigorous treatment of extended set theory may be found in Childs' papers, the essential concepts are restated in section 2.0. Extended set theory is an excellent formalism for data description: it provides a precise definition of "set based structures" in axiomatic mathematical terms. This paper will therefore use the term set-based structures instead of data structures. The term set-based structures will be used to include sets, tuples and intermixed nestings of these.

There are two basic reasons for attempting to implement a processor for extended sets. First, it would provide a basis for comparison of capability and performance of other data models such as data-structure-set (1,7,8), relational (9), and tree-structured (2). This comparison is possible because all basic structures found in data models can be expressed in terms of set theory. But, for a given model, the choice of representation in set theoretical terms is not always unique.

Second, a set processor would provide a tool with which implementations of large-scale data base systems (tailored to specific applications) could be constructed. In such cases, there would be relatively small cost, both in time and manpower, because the implementor need only specify the logical structures and operations in a set-based notation. As an example, the drudgery of performing classical data base operations (such as sorting, searching and updating) is handled by the set processor.

A description of the potential for using a set processor in a network having several mainframes and a very large (10^{12} bit) mass storage system is given in Hardgrave (16). An overview of the implementation of the set processor (SETP) is given in section 5.0, while the application of the Cauchy/Cantor diagonal method is given in section 4.0. At present, a prototype of SETP has been implemented to perform only the simplest operations. Development continues, but little can yet be deduced about the efficiency of set processors.

2.0 Extended Set Theory

In order to understand the notion of and the need for an extended set, it is first necessary to review the classical definition of an ordered pair as given by Kuratowski (12). He defines the ordered pair in the following manner:

$$\langle a,b\rangle = \{\{a\},\{a,b\}\} \qquad (2.1)$$

where we adopt the notation that braces {,} surround sets and angle brackets <,> surround tuples.

Certain problems arise when the Kuratorski definition is applied to computer data structures. First, the definition is not readily extensible to n-tuples; e.g. an ordered triple must be expressed in terms of nesting of ordered pairs. Second, these nestings require that a large number of unnecessary sets be generated in order to preserve structure.

This paper was prepared as a result of work performed under NASA Grant NGR 47-102-001 and Contract Number NAS1-14101 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

To amplify these points, let us first expand the definition of the ordered pair in a relatively simple fashion to give the following (apparently useful) definition of an ordered triple:

$$\langle a,b,c\rangle = \{\{a\}\ \{a,b\}\ \{a,b,c\}\} \qquad (2.2)$$

Unfortunately, this "intuitively obvious" definition results in a number of equivalent structures. As an example, one of these occurs for the triple <1,0,1>. Expansion using definition 2.2 gives:

$$\langle 1,0,1\rangle = \{\{1\}\ \{1,0\}\ \{1,0,1\}\} = \{\{1\}\ \{1,0\}\ \{1,0\}\}$$

$$= \{\{1\}\ \{1,0\}\} = \langle 1,0\rangle \qquad (2.3)$$

In other words <1,0,1> is indistinguishable from <1,0> which is of course unacceptable.

Because of this inability to produce a suitable definition for n-tuples, the traditional approach is to define n-tuples by nesting ordered pairs, either to the left or to the right. That is, either

$$\langle a_1,a_2,a_3,\ldots,a_n\rangle = \langle\langle\ldots\langle a_1,a_2\rangle,a_3\rangle,\ldots\rangle,a_n\rangle \qquad (2.4)$$

or

$$\langle a_1,a_2,a_3,\ldots,a_n\rangle = \langle a_1,\langle a_2,\langle a_3,\ldots\langle a_{n-1},a_n\rangle\ldots\rangle \qquad (2.5)$$

is considered to be the standard definition.

A more complete discussion of the above, as well as some other possibilities for defining the ordered n-tuple, is given in Skolem (23). The essential points made in his paper are:

(1) n-tuples defined in terms of nestings of ordered pairs are inconvenient in a set theory with types ("type" here is used as a measure of the level of nesting of a set).

(2) At that time (1957) a suitable definition for the n-tuple in set theoretic terms was still lacking.

In view of the popularity of a relational approach, which relies entirely on nesting n-tuples within sets, it is most desirable to have satisfactory definitions (in computational terms) of both the set and the n-tuple.

2.1 Definition of an Extended Set

Childs (4,5,6) proposed an extended set theory which could support both classical sets and n-tuples at the same hierarchical level; i.e. neither being defined using the other.

A rigorous axiomatic definition of extended set theory is given in Childs' paper (5), and the reader is referred to this for a full mathematical treatment. However, the essence of extended set theory for our purposes is the basic definition of a set (E), which we will call an extended set:

$$E = \{a_1^{i_1},a_2^{i_2},\ldots a_n^{i_n}\} \qquad (2.6)$$

Here, the a_k are element identifiers and the i_j are position indicators; the superscripts are of course a notational convenience, which should not be confused with exponentiation.

Elements can be atoms or other extended sets. For our purposes, the position indicators are taken from the positive integers.

It is now easy to show that finite classical sets and tuples can be defined using extended sets; first for the classical set:

$$\{a_1,a_2,a_3,\ldots,a_n\} = \{a_1^1,a_2^1,a_3^1,\ldots a_n^1\} \qquad (2.7)$$

while, for the tuple:

$$\langle a_1,a_2,a_3,\ldots a_n\rangle = \{a_1^1,a_2^2,a_3^3,\ldots,a_n^n\} \qquad (2.8)$$

In summary, extended set theory has two distinct computational advantages over the previous definitions. First, each of the structures

$$\langle a_1,a_2,a_3,\ldots,a_n\rangle \qquad (2.9)$$

$$\langle a_1,\langle a_2,\langle a_3,\langle\ldots\langle a_{n-1},a_n\rangle\rangle\ldots\rangle\rangle \qquad (2.10)$$

$$\langle\langle\ldots\langle\langle a_1,a_2\rangle,a_3\rangle,\ldots\rangle,a_n\rangle \qquad (2.11)$$

may be uniquely defined: designers of data base systems may wish to use all three in one implementation. Second, an implementation of extended set theory involves less nesting in representing n-tuples than do other definitions. It appears that efficient implementations of extended set theory can be found, whereas the nesting required to represent the nested definitions are bound to cause inefficiency.

2.2 Operations on Extended Sets

There is a kernel of operations on extended sets which are similar to those found in classical set theory: obviously, they should not conflict with, but may tend to expand on, the operators in classical set theory. We discuss some of the operations in this section, but first it is necessary to review classical set theory and the extended set theory concepts of the membership criterion.

In classical set theory:

$$X \in A \qquad (2.12)$$

states that X is a member of the set A. A rigorous treatment is given in Suppes (24). In extended set theory:

$$X \in_i A \qquad (2.13)$$

states that X is a component of the i^{th} position of A. Childs (5) gives a rigorous treatment of this point.

It is now possible to define the union of two extended sets A and B as:

$$A \bigcup B = \{x^i \mid \forall i \in N(x \in_i A \text{ or } x \in_i B)\} \qquad (2.14)$$

where N is the set of positive integers. Similarly intersection may be defined as

$$A \bigcap B = \{x^i \mid \forall i \in N(x \in_i A \text{ and } x \in_i B)\} \qquad (2.15)$$

Other classical set operations, such as relative complement and symmetric difference, may be defined in a similar manner.

In addition to these operations, which are similar to those found in classical set theory, there is another class of operations that can be defined by taking advantage of position

indicators. For example, the k^{th} element(s) of an n-tuple (M) may be extracted into an extended set L by the operation:

$$L = EXT(M,K) = \{x^1 | x\varepsilon_k M\} \qquad (2.16)$$

For example if

$$M = \langle 12,34,97,15\rangle = \{12^1,34^2,97^3,15^4\}$$

then

$$L = EXT(M,3) = \{97^1\} = \{97\}$$

Another similar operation is

$$L = DOM(M,K) = \{x^K | x\varepsilon_k M\} \qquad (2.17)$$

Unlike EXT, DOM preserves the original position indicator; for example, for M defined as above

$$L = DOM(M,3) = \{97^3\}$$

The names for these two operations were arbitrarily assigned by the author. Numerous other similar operations may be defined as needed.

3.0 A Review of the Cauchy/Cantor Diagonal Method

Periodically, a mathematical idea which was born in a past century finds new application in the area of information systems; such is the case with the Diagonal Method of Cauchy (1789-1857) and Cantor (1845-1918). The Cauchy/Cantor Diagonal Method, discussed herein, was originally used to demonstrate that the rational numbers were countable, but here it is applied to a computer implementation of a system based on extended set theory.

The Cauchy/Cantor Diagonal Method is a reversible mapping from the ordered pairs to the integers. The following discussion clarifies this: Let N denote the set of positive integers and Q the set of non-negative integers; then

$$Q = N \bigcup \{0\} \qquad (3.1)$$

Let QxQ be defined as $QxQ = \{\langle K,L\rangle : K\varepsilon Q, L\varepsilon Q\}$ (3.2)

That is, QxQ is the set of all ordered pairs each element of which is a member of Q.

The domain and range for the Cauchy/Cantor mapping (as used in this paper) may be defined as:

$$C: QxQ \rightarrow N$$

This mapping is both one-to-one and onto:

There exists an inverse function $C^{-1}: N \rightarrow QxQ$ with the following property: For each $M\varepsilon N$ there exists exactly one pair $\langle K,L\rangle \varepsilon QxQ$ such that $C^{-1}: M \rightarrow \langle K,L\rangle$.

The importance of this mapping (henceforth called KOSH) and its inverse (henceforth called UNKOSH) is that each ordered pair <K,L> may be uniquely associated with a single integer M. Thus given K and L, M is known, whereas given M both K and L are uniquely defined.

This method of associating pairs with integers is generally called the Cauchy/Cantor Diagonal Method; it was originally used to demonstrate that the rational numbers were countable (see Fraenkel (12)). Figure 3-1 is a table demonstrating the mapping for the pair <K,L>, K increases down the left side column and L increases across the top (row). The value (M) of the mapping KOSH may be found in the table at row K, column L.

Using the Cauchy/Cantor mapping, M can be directly calculated from K and L, and vice versa. Thus the table need not be stored. Figure 3-2 gives Fortran Subroutines for KOSH and UNKOSH.

4.0 Use of the Diagonal Method

As shown in the last section, the Diagonal Method provides a reversible mechanism for representing an ordered pair of (non-negative) integers as a single integer. Also, since the mapping is onto, the integers are used as efficiently as possible without bounding either K or L.

The basic problem of implementing extended set theory is one of finding a suitable machine representation for the extended set (E), see definition 2.6, which is repeated here for clarity:

$$E = \{a_1^{i_1}, a_2^{i_2}, a_3^{i_3}, \ldots a_N^{i_N}\} \qquad (4.1)$$

Methods of representing classical sets of positive integers by bit strings are easily derived and well known (e.g. Hardgrave (15)). The essence of such techniques is that when a bit string is used to represent a set (S), a '1' in bit position I implies that $I\varepsilon S$.

4.1 Operations on Bit Strings Representing Sets

Set theoretic operations (e.g. union, intersection) may be implemented using bit strings to represent the sets. Logical bit-by-bit operations may be invoked to perform the respective set operation (that is, union and intersection, respectively.) Figure 4-1 shows the general process. The two sets A and B are input to the processor that performs the set operation. The output from the processor is the single set (i.e. bit string) C. The processor performs set operations by scanning and comparing the current bit position of both input sets, at which time the truth table for the selected set operation is used to generate the 'output' bit.

The above process is satisfactory for bit strings in a logical (i.e. uncompacted form). However, uncompacted (logical) bit strings would, in general, require too much storage space. There is obviously a trade-off between storage and processor complexity. When the bit strings representing sets are compacted, the processor is more complex, as it must insure correct positioning of all three strings. The processor may, however, perform the set operation without expanding the compacted bit strings into the logical bit strings; furthermore, the output string may be generated directly in its compacted form.

	0	1	2	3	4	5	6	7	8	9	10
0	1	3	6	10	15	21	28	36	45	55	66
1	2	5	9	14	20	27	35	44	54	65	
2	4	8	13	19	26	34	43	53	64		
3	7	12	18	25	33	42	52	63			
4	11	17	24	32	41	51	62				
5	16	23	31	40	50	61					
6	22	30	39	49	60						
7	29	38	48	59							
8	37	47	58								
9	46	57									
10	56										

Figure 3-1

The Cauchy/Cantor Mappling for KOSH and UNKOSH

```
SUBROUTINE KOSH (M,K,L)
INTEGER DIAG. SUM
DIAG = K + L + 1
SUM = (DIAG * (DIAG + 1)  ) /2
M = SUM - K
RETURN
END

SUBROUTINE UNKOSH (M,K,L)
INTEGER DIAG, SUM
DIAG = (-1.0 + SQRT (1.0 + 8.0 * M) ) /2.0 + .999
SUM = (DIAG * (DIAG + 1) ) /2
K = SUM - M
L = DIAG - K - 1
RETURN
END
```

Figure 3-2

Fortran Programs for KOSH and UNKOSH

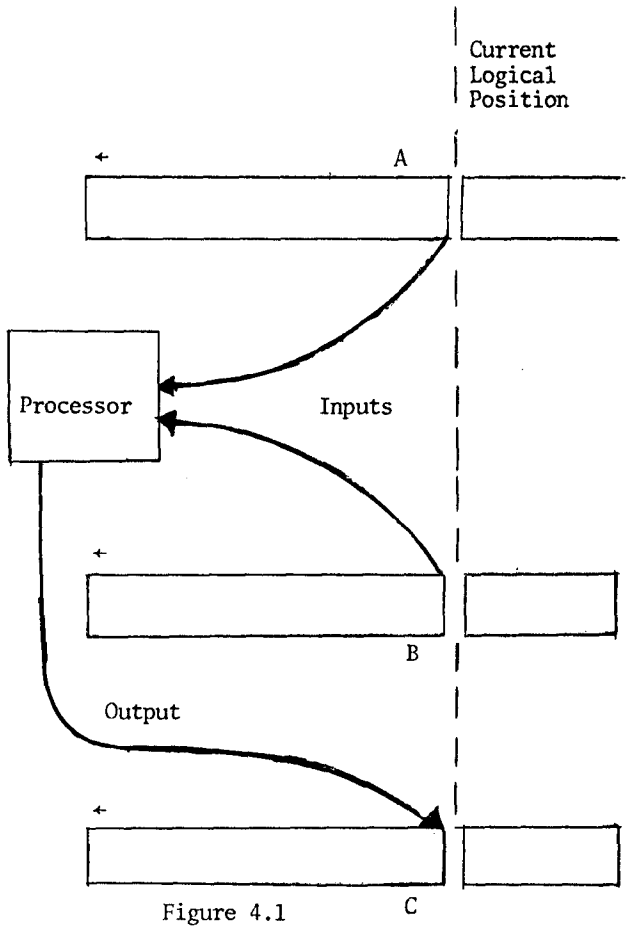


Figure 4.1

Logical Set Operations

4.2 Representing Extended Sets as Bit Strings

One solution to the representation of extended sets can be achieved if E can be transformed into:

$$E = \{M_1, M_2, M_3, \ldots M_N\}$$

where the problem of representing extended sets has been reduced to representing the pair $a_i{}^{i}j$ as a single integer M_i. If the element a_i is stored in a "table of elements" (as discussed in detail in section 5), then the ordinal index (K) to the entry in the table represents the element.

If we consider i_j, the position indicator, we once again have a non-negative integer (L). Thus the element may be identified by its index (K), in conjunction with its position indicator (L). Moreover this pair <K,L> is uniquely representable by mapping

$$C : \langle K,L \rangle \rightarrow M$$

and its inverse:

$$C^{-1} : M \rightarrow \langle K,L \rangle$$

Thus the transformation of formula 4.2 is possible. The coding of the extended set E therefore involves:

i) Placing the element, which may itself be an extended set, and consequently involves recursion, into a table of elements; this returns the resulting index K
ii) Noting the positional indicator L
iii) Calling KOSH (M,K,L) to determine the integer M
iv) Change the M^{th} bit of the E string to 1 to show that:

$$a_j{}^{ij} \in E$$

For the reverse process:

i) Examine the string representing E, finding that the M^{th} bit is 1.
ii) Call UNKOSH (M,K,L) to determine K and L.
iii) Enter the element table to find the K^{th} entry; note that this may be an atom or an extended set.
iv) This element is in position L of E.

4.3 Operations on Extended Sets

Operations on extended sets may be performed using the same techniques that were used for operations on classical sets (see section 4.1). Below is an example of the intersection of two tuples, an operation found but not generally useful in classical set theory.

Consider the tuples

$$A = \langle 21, 36, 91.2 \rangle$$

$$B = \langle 36, 21, 91.2 \rangle$$

Using the definition in section 2, the tuples may be represented as extended sets in the following manner:

$$A = \{21^1, 36^2, 91.2^3\}$$

$$B = \{36^1, 21^2, 91.2^3\}$$

For simplicity we will assume that the atoms 21, 36, 91.2 were stored on the table of elements with the ordinals 1, 2, 3 respectively, as described in the table below

Atom	Element Table Ordinal
21	1
36	2
91.2	3

Using the mappings (see Figure 3-1), the logical bit strings (A and B) representing the extended sets can be created:

A = 000010000000100000000000001...

B = 000000011000000000000000001...

The intersection may be derived by performing the logical product on the two strings.

C = 000000000000000000000000001...

By using the reverse Cauchy mapping and accessing the element table, we decide that

$$C = \{91.2^3\}$$

This may be considered as a tuple with data only in position three. Finally, it should be noted that the set operation itself is performed without accessing the table of elements.

A potentially more relevant example is in the extraction of selected domains using set intersection. In Figure 3-1, each column of the table is the set of possible values that may appear for that domain. For example, the ordinals of the sequence, 15, 20, 26, 33, 41, 50, 60, ... are the only bits that indicate a value in position four of an extended set. Therefore, a standard "Domain four" set represented by the bits 15, 20, 26, 33, 41, 50, 60, ... may be used to extract elements of position four from a given extended set. This may be accomplished by intersecting the set in question with the standard "Domain four" set. The example below clarifies this point.

In order to extract positions 1 and 3 from tuple A, as defined above, the standard sets (as bit strings) for domains one and three (SD1 and SD3) are

SD1 = 001010010001000010000010000001...

SD3 = 000000000100010000100000100000...

Their union is

SD1 ⋃ SD3 = 00101001010101001010001010000 1...

A = 00001000000010000000000100000...

The intersection of the above is

D = 00001000000000000000000100000...

Thus

$$D = \{21^1, 91.2^3\}$$

5.0 An Overview of the Set Processor Implementation

A prototype set processor (SETP) is currently being developed for the Prime 300 Computer at the Institute for Computer Applications in Science and Engineering (ICASE) and also for the UNIVAC 1108 Computer at the University of Maryland. Several other groups are considering the use of SETP as a tool for supporting applications such as graphics. This section briefly discusses the implementation methods.

SETP is principally designed as a tool for designing and implementing systems that require large volumes of data with an elaborate data structure. Such data must be managed in auxiliary storage as well as main memory. Thus SETP is not an end product, but a basis upon which a large class of application systems may be constructed.

SETP supports atoms and extended sets. This allows implementation of classical sets, sequences (i.e. tuples) and arbitrarily deep nestings of potentially intermixed sets and sequences. As discussed later, the extended sets are maintained as bit strings that reference a table of elements. The bit strings are managed in memory buffers and on auxiliary storage media (e.g. disks) by the set processor. Thus, the implementor is relieved of the drudgery of considering auxiliary storage characteristics (such as access times and transfer rates) and performance problems are handled within the set processor.

5.1 The Integer Set Processor

The extended set processor (SETP) is supported by an improved integer set processor similar to that previously described in Hardgrave (15). The Integer Set Processor (ISP) is not concerned with extended sets but supports classical sets of positive integers from a predefined universe:

$$U = \{i \mid i \in N \text{ and } i \leq 2^L\}$$

Where N is the set of natural numbers and L depends on the word size of the computer.

As discussed in section 4.1, the technique used to represent a subset S of the universe, U, is a bit string signifying membership. If the i^{th} bit of the bit string is "i", then:

$i \in S$

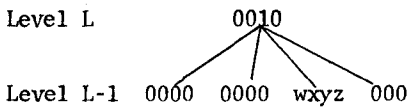
If the i^{th} bit is "0", then:

$i \notin S$

The size of the logical bit string is the cardinality of the universe, which is too large to store. However, a suitable compaction technique (e.g. the QUATREE method) may be used to reduce the size of the bit string as discussed later.

The QUATREE compaction method described in Hardgrave (15) is based upon the notion that compaction can be performed by creating multi-level directories of four-bit packets. This size packet was chosen over larger packet sizes because large sets whose elements are randomly distributed over the absolute universe tend to have smaller representations when smaller packet sizes are used. Each bit in a given directory of level L corresponds to four bits in the bit string at level L-1. Level 1 corresponds to the bit string representation of the set in uncompacted form. For CDC 6000 implementation, level 23 is the highest level directory and consists of only four bits. The null set is represented by a single zero packet. All other sets have at least one non-zero packet at each level. Levels between one and twenty-three are intermediate directories; each referring to the next lower level directory.

The following example illustrates the directory concept:



Each of the zero bits in the level L packet imply that their corresponding packets in the level L-1 directory are all zero. The 1-bit in the level L packet implies that at least one of the bits w, x, y, or z is set to 1. In addition, all zero packets are omitted from the actual representation of the bit string; thus accomplishing the desired compaction.

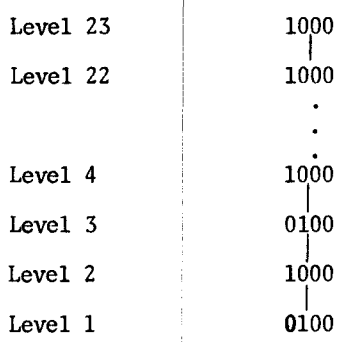
A set having only a single element may be represented as a trace in the hierarchical structure of directories. A trace consists of one packet at each level in the hierarchy and each packet in the trace has one and only one bit set to 1. For example, the representation for the set {18} is as shown below.

Although the actual representation is linear, we also give the hierarchical form for the sake of clarity. The numbers given to the left of the hierarchical representation and below each packet in the linear representation are the level indicators. Level 1 packets are from the original bit string and the level 23 packet is the highest level directory.

Linear form:

1000	1000	...	1000	0100	1000	0100
23	22		4	3	2	1

Hierarchical form:



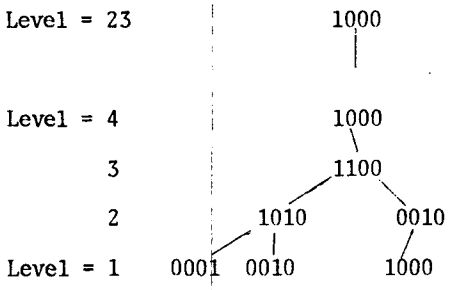
Multi-element sets may be represented by combining these sets accordingly. These representations are not presented here in detail. However, an example of the multi-element set {4,11,25} is given below:

Linear form:

1000	1000	...	1000	1100	1010	0001
23	22		4	3	2	1

0010	0010	1000
1	2	1

Hierarchical form:



One advantage of the QUATREE technique is that there is one and only one compacted representation of a given logical bit string.

The preliminary data reported in Hardgrave (15) suggested that to store sets of positive integers having a cardinality of 10,000 would require at least 13,400 bits and no more than 662,000 bits. Furthermore, the real time required to perform a set operation (e.g. union or intersection) on two sets of that magnitude was less than 10 seconds on a CDC 6400. Using a minicomputer, data is currently being collected at ICASE from several practical experiments of a lesser magnitude. We suspect that in practice, storage requirements for sets will be about 10 bits per element and set operations of the magnitude mentioned above may be performed on larger minicomputers in under a minute (real time). These preliminary indications will be confirmed (or denied) in a future paper.

The Integer Set Processor has a number of operations that work directly with compacted bit strings and (when applicable) produce compacted bit strings. Thus, for these operations the logical bit string is never generated. The Integer Set Operations include set union, intersection, and relative complement, and a collection of utilities such as set traversal, which allows the set to be examined one element at a time.

The Integer Set Processor was designed to deal with large and complex sets, which may grow to be too large to be maintained in the central memory of even the largest computers. The I/O Processor maintains sets (as compacted bit strings) on back-up store and in main memory as requested.

5.2 The Tables

The Set Processor utilizes three tables, the Element Table, the Alias Table, and the Text Table to store and manipulate data: the Element Table contains the occurrences of data items; the Alias Table contains user defined names; the Text Table contains overflow from the Element and Alias Tables. All table managers and all transactions concerning a table utilize their relevent table manager.

The Element Table and the Alias Table are designed to be fixed length entry tables. This allows the table managers easy conversion from from entry number to displacement (i.e. address) within the respective tables. The Text Table is of variable length; it handles any necessary overflow from the other (two) tables.

Each fixed length entry in the Element Table contains one element, which is either an atom or an extended set. Atomic element types currently supported by SETP include integers, single precision floating point numbers and character strings. Each element in the table is unique; thus when an integer (e.g. "10") or an extended set (e.g. $\{1^1, 2^1\}$) is stored on the table, all references to it reference that particular entry. A simple form of hash-coding is used to implement element table access. The efficient management of such elements, which involve large storage volumes, is an interesting implementation problem which has not yet been properly investigated.

The purpose of the Alias Table is to provide a mechanism for binding user defined names to specific data items or structures. These names are stored on the Alias Table and bound to specific elements in the element table by means of a pointer. The names found in an Alias Table are an integral part of the database, and are not the same as the temporary names assigned to "working structures" for an interactive session. The user may specify an alias for any particular element and bind it accordingly.

5.3 Efficiency Considerations

In the initial implementation, the set processor was made as simple as possible, the guiding philosophy being to create, as quickly as possible, a modular system that performs according to a flexible set of specifications. Care was taken to insure that an efficient implementation is not precluded. However, time has not yet been devoted to producing efficient code, or even to the investigation of better algorithms. The next phase will analyze the system by performance measurement techniques.

At present a trace-driven modeling technique is being developed.

In order to be a viable tool for use in data management system development, the set processor should be competitive with today's generalized data base management systems in two major areas:

- Data base size
- Accessing speed

Investigations are currently under way to provide exact formulae for these parameters. For example, it is possible to estimate the size of a data base consisting of a single relation (e.g. as in Codd (9)) as follows:

$$S = Q * D(R) * C(R) + Q * C(R) + U * L$$

where:

- S is the size in bits.
- C(R) is the cardinality of the relation R (i.e. the number of tuples to be stored).
- D(R) is the degree of the relation R (i.e. the number of elements in each tuple).
- Q is the average number of bits per element required by the compaction technique.
- U is the number of unique element values in the data base.
- L is the average length, in bits, of the unique element values.

The size of the Alias Table is ignored since the assumption is that it is negligible compared with the element table size. The values D(R), C(R), U, and L may all be determined from the data base in question. However, Q is dependent on the compaction technique. As noted previously by the author (15) the values for Q using the QUATREE method range from 1.3 to 92 bits per element. As a rule of thumb, a value of ten is used for Q in the QUATREE method.

6.0 Concluding Remarks

Set Processors are potentially valuable as tools for data base design and implementation. Large-scale data base systems implemented using set processors appear to be less complex than traditional data base systems.

Extended set theory is an excellent vehicle for implementing set processors. This concept supports both classical sets and tuples using an unparalleled notation and lends itself readily to computer implementation. This article demonstrates that an implementation is possible for extended set theory in its purest form.

However, the fact that an implementation exists does not guarantee any measure of efficiency, and this represents the most important question about set processing. If implementations can be found that compare favorably with state-of-the-art systems based on other data models, then set processing will certainly succeed. Further effort is necessary in determining methodologies for implementing existing data models using set processors. Concepts such as "data independence," "integrity," "security," etc. will be as important in those studies as they are today.

Finally, more analysis is necessary for configurations that include mass storage systems, networks, multiple mainframes, etc. as well as set processors. This will help provide comprehensive "data sharing" and eliminate unnecessary redundancy and updating overhead.

ACKNOWLEDGEMENT

The author is grateful to E. H. Sibley, D. L. Childs, and J. R. Rothnie for their assistance and encouragement in preparing this paper. In addition, the author is indebted to NASA and ICASE for their support of this area of research.

REFERENCES

(1) Bachman, C. W., The Data-Structure-Set Model, Workshop on Data Description, Access and Control, Vol. 2, ACM SIGMOD, May 1974.

(2) Bleier, R. E., Treating hierarchical data structures in the SDC Time-shared Data Management System (TDMS). Proc. ACM 22nd National Conference, MDI Publications, Wayne, PA, 1967, 41-49

(3) Bleier, R. E., Vorhaus, A. H., File organization in the SDS Time-Shared Data Management System (TDMS). IFIP Congress 1968, North Holland, 1968, 1245-1252.

(4) Childs, D. L., Description of a set-theoretic data structure. AFIPS Conf. Proc., Vol. 33, Part 1, AFIPS Press, Montvale, NJ 1968, 557-564.

(5) Childs, D. L., Extended Set Theory: A Formalism for the Design, Implementation, and Operation of Information Systems. Volume IV, Current Trends on Programming Methodology, edited by R. T. Yeh, Prentice Hall, expected Publication in early 1977.

(6) Childs, D. L., Feasibility of a set-theoretic data structure: a general structure based on a reconstructed definition of relation. IFIP Congress 1968, North Holland, Amsterdam, 1968, 420-430.

(7) CODASYL Data Base Task Group. ACM, New York, April, 1971.

(8) CODASYL Data Description Language, Journal of Development 1973. Published by the National Bureau of Standards: NBS Handbook 113.

(9) Codd, E. F., A relational model of data for large shared data banks. Comm. ACM, 13 (June 1970), 377-387.

(10) Codd, E. F. and Date, C. J., Interactive Support for Non-programmers: The Relational and Network Approaches. Workshop on Data Description, Access and Control, Vol. 2, ACM SIGMOD, May 1974.

(11) Date, C. J. and Codd, E. F. The Relational and Network Approaches: Comparison of the Application Programming Interfaces. Workshop on Data Description, Access and Control, Vol. 2, ACM SIGMOD, May 1974.

(12) Fraenkel, A. A., Abstract Set Theory, North Holland, 1966.

(13) Hardgrave, W. T., Accessing technical data bases using STDS: a collection of scenarios. ICASE Report 75-8, Hampton, VA 1975.

(14) Hardgrave, W. T., BOLTS: A retrieval language for tree-structured data base systems. Information Systems, COINS IV, Plenum, New York, 1974.

(15) Hardgrave, W. T., The prospects for large capacity set support systems inbedded within generalized data management systems. International Computing Symposium - 1973, North-Holland, Amsterdam, 1974.

(16) Hardgrave, W. T., Set Processing in a Network Environment. ICASE Report 75-7, Hampton, Virginia, March 1975.

(17) Schwartz, J. T., Abstract and concrete problems in the theory of files. Courant Computer Symposia 6: Data Base Systems, Prentice-Hall, Englewood Cliffs, N.J., 1971.

(18) Schwartz, J. T., On Programming: An Interim Report on the SETL Project. Courant Institute of Mathematical Sciences, New York University, New York, 1973.

(19) Senko, M. E., Altman, E. B. Astrahan, M. M., and Fehder, P. L. Data Structures and Accessing in Data Base Systems. IBM Systems J 12, 1 (1973).

(20) Senko, M. E. Information Systems: records, relations, sets, entities, and things. Information Systems 1, 1 (Jan. 1975).

(21) Sibley, E. H., Taylor, R. W., Data definition and mapping language. Comm. ACM, 16 (December 1973) 730-759.

(22) Sibley, E. H. On the equivalences of data based systems, Workshop on data description, access and control, Vol. 2, ACM SIGMOD, May 1974.

(23) Skolem, T., Two remarks on set theory. Math Scand. 5, 1957, 40-46.

(24) Suppes, P., Axiomatic Set Theory, Dover, New York, 1972.

(25) Whitney, K. M., Fourth generation data management systems. AFIP Conf. Proc. 42, AFIPS Press, Montvale, N.J., 1973, 239-244.