A HIERARCHICAL ASSOCIATIVE ARCHITECTURE FOR THE PARALLEL EVALUATION OF RELATIONAL ALGEBRAIC DATABASE PRIMITIVES

by

David Elliot Shaw

STAN-CS-79-778
October 19 7 9

D E P A R T M E N T   O F   C O M P U T E R   S C I E N C E
School of Humanities and Sciences
STANFORD UNIVERSITY

A Hierarchical Associative Architecture for the Parallel Evaluation of Relational Algebraic Database Primitives

David Elliot Shaw

Artificial Intelligence Laboratory
Stanford University

October 1979

Abstract

Algorithm6 are described and analyzed for the efficient evaluation of the primitive operators of a relational algebra on a proposed non-von Neumann machine based on a hierarchy of associative storage devices. This architecture permits an O(log *n)* decrease in time complexity over the best known evaluation methods on a conventional computer system, without the use of redundant storage, and using currently available and potentially competitive technology. In many cases of practical import, the proposed architecture may also permit a significant improvement (by a factor roughly proportional to the capacity of the primary associative storage device) over the performance of previously implemented or proposed database machine architectures based on associative secondary storage devices.

'Acknowledgements

The author gratefully acknowledges the substantial contributions of Bob Floyd, Don Knuth, Juan Ludlow, Luis Trabb-Pardo, Terry Winograd, and in particular, Gio Wiederhold, to the work reported in this paper.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract MDA903-77-C-0322.

Contents

1. Introduction

At the heart of the system which we are implementing as part of our thesis research is a process of description-based retrieval in which all documents in a collection which match a KRL-like description on the basis of information extracted from a domain-specific knowledge base. The limited size of the document collection which will be used to develop and test our thesis system should make the matching task computationally tractable within the context of our research effort. It is precisely in the case of a very large and conceptually heterogeneous collection of structured entities (in our application, documents), however, where the argument6 for our conceptual description matching techniques are strongest. If these techniques are to be seriously considered a6 promising tool6 for retrieval problems of practical significance, we must thus carefully consider the effect of an increase of several order6 of magnitude in the size of the target collection. The retrieval strategy which is perhaps most obvious, in which the search description is matched successively against each candidate description in the target collection (which would ordinarily reside on secondary storage), carries penalties in efficiency which, although probably manageable in the course of a research project, are likely to be prohibitive in the context of application6 which might involve several million target descriptions.

There are a number of possibilities for the efficient implementation on a very large scale of the sort of description matching with which we are concerned. One approach might involve the careful design of specially-tailored indexing and access schemes in software for execution on a "conventional" computer system architecture, at the cost of considerable software complexity and a somewhat inflexible attention to the details of the particular descriptive scheme under consideration. In this paper, however, we will consider certain *alternative* hardware architectures as a basis for a very general and efficient approach to large-scale meaning-based retrieval.

One of the most difficult issues which often seem6 to be encountered by the designer6 of complex special-purpose hardware, particularly in the research environment, involve6 the tradeoff between considerations of efficiency and economy for the initially conceived application, on the one hand, and issues of generality, flexibility and mutability in the face of incompletely specified and evolutionarily changing system requirements, on the other. In particular, any attempt to reduce conceptual matching to a small set of primitive operation6 for which efficient special-purpose hardware mechanisms seem appropriate necessarily involves careful consideration of the degree to which a given design decision may complicate the addition or modification of new descriptor types, matching criteria, and data representation schemes, should such characteristics subsequently be changed on the basis of early research results. In this paper, we have adopted a fairly conservative approach in this regard, foregoing the exploitation of certain special properties of our descriptive formalism which might in fact have been used to obtain further efficiencies if flexibility were not a concern, and providing in some instances mechanisms which are somewhat more general than those which would be strictly required for our immdediate implementation.

What has emerged is an architecture which, while quite different from that of a traditional von Neumann computer system, nonetheless speaks to a fairly wide class of problems outside the immediate province of conceptual matching. In particular, we have described a computer system for the parallel evaluation of the primitives of a relational *algebra,* a6 described by Codd [1971]—specifically, the operators project, *equi-join,* select, restrict, union, intersect, and set difference. The system architecture which we will describe is thus intended to be applicable to a wide range of problems of considerable current practical concern in the field of database management, and in particular, to the design of efficient systems based on the relational model of data.

The proposed architecture is organized a6 a two-level hierarchy of associative storage devices—the smaller and faster level of which will be called primary, and the larger and slower, secondary. In the course of evaluating each relational primitive, the entire database is associatively probed using logic associated with the secondary storage device6 themselves. In the case of all but two of the operators, selected segments of the relevant data are then transferred in succession from secondary to primary associative storage for further processing. This paper will outline the organization of the architecture we are proposing, and will describe and analyze algorithm@ for evaluation of each of the relational algebraic primitives, both in the case where the arguments can fit entirely within primary storage (which we call internal *evaluation),* and where they reside on secondary storage (external evaluation).

Section 2 of this paper review6 the essential element6 of a relational algebra. In Section 3, we survey previous work on the design of associative processors and database machines to provide a context for the introduction of our architecture. Section 4 provide6 a functional description of the central hardware components involved in our design. The notation to be used in analyzing the performance of our algorithms for evaluation of the primitive relational operator6 is introduced in Section 5. The algorithm6 for internal evaluation are presented in Section 6, along with an average-case analysis of their time complexity. The procedure6 for external evaluation (most of which make use of the internal evaluation routines) are described in Section 7. In Section 8, two alternative schemes are described for the partitioning of the argument relations into appropriate segments and their transfer into primary storage, a time-critical part of the process of external evaluation. The latter of these two schemes, which would appear to be preferable when appropriate hardware is available, is analyzed in **some** detail. Our results are summarized in Section **9;** the reader may wish to glance briefly at this section before proceeding with the body of the paper.

2. The Relational Algebra

The relational model of data, as typically formulated by researchers in database management systems, has its roots in two seminal papers by Codd [1970, 1972]. In this context, the term relation is used to denote a set of structured entities called tuples which, within a single relation, share a common attribute *structure.* More formally, we may define a normalized relation of degree *n* as a set of *tuples,* where each tuple is an element of the Cartesian product of *n* (not necessarily distinct) sets-called the *underlying* domains of the relation-of non-decomposable entities. (In some practical database applications, it may be useful to allow the appearance of "null values" in various attributes of certain tuples in the case where certain information is not available; apart from brief mention of one complication introduced by this convention, however, we will not be concerned in this paper with the problems of null values.) Since relations are sets, we may refer to the number of elements-in this case, tuples-in a relation as the ***cardinality*** of that relation.

Intuitively, relations may be thought of as "tables", in which each "row" represents one tuple and each column represents one of the **n** (simple) attributes of that relation, It is conventional to either name or number the attributes of a relation for convenience in referring either to a whole "column" of the relation, or to the value of the attribute in question within a particular tuple. In some discussions (and in particular, in much of this paper), it is also useful to group several attributes (some possibly repeated) together, referring to them jointly as a compound attribute. The term normalized reflects the "type distinction" between underlying domain elements, which may serve as the values of attributes, and tuples and relations, which may not. (A single tuple thus can not be used to directly represent a hierarchically nested data structure.)

The relational algebra which forms the central focus of this paper is based on a small set of algebraic operators enumerated by Codd [1972] which take one or more relations (along with certain "control" information) as arguments, returning a single new relation as their value. This set of primitives includes the ordinary set **operations**—union, *intersection* and *set* difference--which, with one restriction, are defined for relations in much the same way as for other sets, along with several structuredoperators, which make reference to the internal attribute structure of the constituent tuples. For completeness, the unstructured set operations are reviewed in Section 2.1. The two fundamental structured operators, *project* and join, are then introduced in Sections 2.2 and 2.3, respectively. Several other structured operations which may in fact be derived from project, join and the unstructured set operations, but which serve certain particularly important functions in many practical applications of relational algebraic systems, are then discussed in Section *2.4.*

2.1 The unstructured set operations

The three binary set operators ***union, intersection*** and ***set difference*** are defined in a relational algebraic system in the same way as for sets in general, with one exception: the relational version of each is defined only when the two relations which serve as its operands are ***union-compatible.*** Two relations are said to be union-compatible if and only if they are of the same degree n and the underlying domains of the i-th simple attributes of the two relations are the same for all i, $(1 \leq a \leq n)$.

We thus define the union of two union-compatible relations& and R_2, denoted $(R_1 \cup R_2)$, as a relation consisting of exactly those tuples which are an element of R_1, of R_2, or both. The intersection $(R_1 \cap R_2)$ is defined as that relation containing all tuples found in ***both*** R_1 and R_2. Finally, the set difference $(R_1 - R_2)$ is defined to consist of exactly those tuples of R_1 which are ***not*** present in R_2.

2.2 Project

In preparation for our definition of the projection operator, we first introduce some additional notation. First, we adopt the convention that a list of primitive domain elements enclosed by angle brackets ("$\langle$" and "$\rangle$") will designate a new tuple containing the specified elements as the values of its simple attributes, in the order listed. Futhermore, if r is a tuple of some n-ary relation R, we will define $r[j]$ to be the value of the j-th attribute of r, $(1 \leq j \leq n)$. It will be convenient to extend this notation to allow expressions such as $r[A]$, where A **is** a ***compound*** attribute of R consisting of the m (not necessarily distinct) simple attributes numbered $j_1, j_2, \ldots, j_m$, defined such that $\langle r[A] \rangle$ represents the new tuple $\langle r[j_1], r[j_2], \ldots, r[j_m] \rangle$.

We may now define the projection of a **relation** R over the compound attribute A **as the** set

$$\{\langle r[A] \rangle : r \epsilon R\} \quad .$$

Note that we have defined the projection operator in such a way that simple attributes within the compound attribute A may be replicated in the course of projection. Depending on certain details in the definition of the join operation (Section **2.3**), this convention may have important theoretical consequences affecting the expressive power of the resulting algebra.

The projection operator **may** be thought of as a sort of "vertical subsetting" operation, in which

1. the "non-projected" attributes of each tuple in the argument relation are eliminated,

2. the remaining attributes may be permuted and/or replicated, and

3. any duplicate tuples which result from the elimination of values which formerly distinguished different tuples are then removed,

In most implementations on a von Neumann machine, the first two functions can be implemented using a simple and computationally inexpensive procedure whose complexity is linear in the cardinality of the argument relation. The elimination of redundant tuples, on the other hand, may be surprisingly time-consuming, particularly when the argument relation is large. In fact, one common convention in some von Neumann implementations is to relax the requirement that relations be true sets, allowing the introduction of duplication during some or all projections. This approach introduces the following problems, however:

1. the maintenance of duplicate tuples may lead to combinatorially explosive growth in the cardinality of the intermediate results of a complex query, and

2. functions sensitive to the repetition of identical tuples-the calculation of numerical counts and statistical measures, for example-will not yield accurate results if redundant tuples are not first eliminated.

One of the goals of the architectures discussed in this paper is the implementation of true projection without the high cost of redundant tuple elimination.

2.3 Join

Definition of the join operation requires the definition of one additional construct: the concatenation of two tuples. If r_1 is a tuple of a relation R_1, having degree n_1, and r_2 is a tuple of relation R_2, having degree n_2, the concatenation $(r_1|r_2)$ of r_1 and r_2 is defined to be the new (n_1+n_2)-tuple

$$\langle r_1[1], r_1[2], \ldots, r_1[n_1], r_2[1], r_2[2], \ldots, r_2[n_2]\rangle \quad .$$

Several variations of the join operator are commonly discussed in the literature; we will begin by defining a particularly important variant known as the ***equi***-join. The equi-join of two relations R_1 and R_2 over the compound attributes A_1 and A_2, respectively (each assumed to be composed of the same number of simple attributes, with corresponding simple attributes having underlying domains which are comparable under the equality predicate) is defined as

$$\{(r_1|r_2): r_1 \epsilon R_1 \wedge r_2 \epsilon R_2 \wedge r_1[A_1] = r_2[A_2]\} \quad .$$

A_1 and A_2 are referred to as the (compound) ***join*** attributes. and will have special significance in the architectures introduced in this paper. In the case where A_1 and A_2 are the degenerate compound attributes containing ***no*** simple attributes, equi-join reduces to the ***extended*** Cartesian product of the tuples of R_1 and R_2—that is, to the set of all possible concatenations of one tuple from R_1 with one from R_2. The more general join operation may be intuitively thought of as a process of filtering the extended Cartesian product of R_1 and R_2 by removing from the result all conjoined tuples whose respective join attributes have different values. (The computational method suggested by this interpretation, of course, would in general be impractically inefficient.)

It should be acknowledged that our definition of the equi-join operator leaves unanswered certain questions which, although not immediately relevant to the concerns of this paper, are commonly encountered in practical database applications. One such problem arises in the case where null values are allowed to appear in the join attributes, since it is generally not appropriate to treat two tuples as matching in the case where their join attributes are both null. Consideration of the various approaches which have been advanced for the accomodation of this case will not fall within the scope of this paper, however.

The join operation is in general extremely expensive on a conventional von Neumann machine, since the tuples of R_1 and R_2 must be paired for equality with respect to the join attributes before the extended Cartesian product of each group of "matching" tuples can be computed. In the absence of physical clustering with respect to the join attributes (whose identity may vary in different joins over the same pair of relations), or the use of various techniques requiring a large amount of redundant storage, joining is typically accomplished most efficiently on a von Neumann machine by pre-sorting the two argument relations with respect to the join fields. The ***order*** of the tuples following the sort is actually gratuitous information from the viewpoint of the join operation. From a strictly formal perspective, the requirements of a join-that the tuples be paired in such a way that the values of the join attribute match-are significantly weaker than those of a sort, which requires that the resulting set be ***sequenced*** according to the those values. The distinction is moot in the case of a von Neumann machine, where no better general solution to this pairing problem than sorting is presently known. One of the design goals of the architecture described in this paper, however, is to make use of the weaker constraints involved in the definition of the join operation to obviate the need for either pre-sorting or the extravagant use of redundant storage.

One common variant of the equi-join operator is the natural join, in which one of the two join attributes, which are redundantly represented in the result relation in the case of equi-join, is eliminated (as if by projection). Our architecture supports the natural join with a simple modification of the internal equi-join algorithm described in Section *6.2.* A more general form of join often discussed in the literature is the **θ-join,** whose definition is similar to that of the equi-join, but with the equality predicate replaced by a more general binary predicate θ. (In Codd's definition, θ is defined to be one of the arithmetic operations $=, \neq, <, \leq, >$ or $\geq$.) Consideration6 for the efficient evaluation of the general **θ-join** operator differ in several respects from those involved in evaluating the equi-join. We will not discuss this more general case in the present paper.

2.4 Other operations

Each of the relational algebraic operator6 described in this section can in fact be derived from the structured operators project and join and the three unstructured set operators, and are defined here for one or both of the following **reasons:**

1) The operator embodies a special case of one or more of the previously defined primitives which might admit the possibility of either a less complex, or a more efficient, hardware implementation

2) The operator represents an important and frequently encountered use of some composition of the primitives defined earlier

One derived operation which occurs frequently in both practical and theoretical discussions, and which has a special role in most of the hardware designs which we will discuss, is called ***selection.*** Most algorithms and architectures for "associative retrieval" are based closely on what is essentially a process of relational selection. The select operator also plays an important role in the architectures we propose in this paper, although unlike most associative processor designs, our architecture explicitly addresses the problems of efficiently implementing other relational operators as well. The select operator returns a subset of its single argument relation consisting of all tuples which satisfy a list of attribute/value pairs. The select operator may thus be regarded as a natural join of the argument relation with a ***singleton*** relation (a relation consisting of exactly one explicitly specified tuple) over all attributes of the singleton. More precisely, the result of a selection from relation ***R*** with compound attribute *A* and value tuple *V* is

$$\{r : r \epsilon R \wedge r[A] = V\} \quad ,$$

where the corresponding *A* and *V* domain6 are again assumed to be compatible with respect to equality.

Another important derived operation is known as ***restriction.*** While restriction, like the join operator, is sometimes defined in terms of a general θ, we will again be concerned only with the case where θ is the binary equality predicate. The restriction of a relation R over the compound attributes A_1 and A_2 (both composed of simple attributes of R) is defined as

$$\{r{:}r\epsilon R \wedge r[A_1] = r[A_2]\}$$

In its most common form, where the compound attributes A_1 and A_2 are each composed of exactly one simple attribute, the restriction operator returns all tuples of its argument relation in which the values of the two specified attributes are equal. Although restriction can be defined solely in terms of the join and project operators, an implementation based in a straightforward way on this derivation would be considerably more complex and inefficient than one specifically tailored to support the restrict operator. Restriction is an important enough operation in practice that we have treated the capacity for direct (and efficient) evaluation of restrictive expressions as a significant design objective.

Finally, we must acknowledge a derived operation which has considerable theoretical and practical importance in many applications, but to which we have devoted little special attention in our evaluation of alternative architectures. This operation, called division, is used to achieve the effects of universal quantification within the queries of a language based on the relational calculus (Codd [1972]) and may well be worthy of special attention in course of designing a generally-applicable relational database machine. Since it is not clear at this point that the design of our thesis system will require that this sort of operation be implemented efficiently in its full generality, however, the relational division operator will not be given the same sort of special consideration in this paper as the other two derived operators described above.

3. Relevant Previous Work

In this section, we will briefly survey certain areas in the literature of computer architecture which have central relevance to our own investigations. Because the great majority of the recent work on specialized architectures for database management applications-our own included-has drawn heavily on earlier work involving the design of content-addressable memories and associative processors, we will begin our review, in Section 3.1, with a rough taxonomy and description of the most important classes of associative processing devices. In Section 3.2, we will consider several of the best known proposals for, and actual prototype implementations of, what might be called true database machines: systems oriented toward fairly specific functions deemed relevant to actual database management applications.

3.1 *Associative* **processors**

At the risk of oversimplification, it is probably safe to say that virtually all existing and proposed database machine architectures have drawn their power from the utilization of a high degree of some variety of hardware ***parallelism*** at some level within the system. The different techniques for achieving such parallel computation are often distinguished according to a classification scheme suggested by Flynn [1972], which characterized the conventional sequential processor as a Single ***Instruction*** Stream ***Single*** Data Stream (SISD) machine, as contrasted with the most common mechanisms for parallel computation, among which he distinguished three different organizations:

1. ***Multiple Instruction Stream Single*** Data Stream (MISD) machines, typified by the ***pipeline*** processing approach.

2. ***Single Instruction*** Stream ***Multiple*** Data Stream (SIMD) machines, in which a single operation is performed in parallel by a number of independent processing units at any given time,

3. ***Multiple Instruction*** Stream ***Multiple*** Data Stream (MIMD) machines, which function as a number of independent, but communicating processors, each of which is capable of maintaining its own instruction and data streams.

Alternative classificatory schemes for parallel machines have also been proposed (eg., Murtha and Beadles [1964]). For a more thorough discussion of the taxonomy of parallel processors in general, the reader is referred to Thurber and Wald [1975].

Within the class of SIMD machines, two important subclasses are typically recognized. (Again, however, other classifications are possible; see, for example, Higbie [1973].) Members of the first subclass, exemplified by such machines as ILLIAC IV, are known as array processors, in which the data are processed in parallel using the conventional mechanism of coordinate addressing. The second subclass consists of the associative *processor&* which access their data in a content addressable manner. Although each of the varieties of parallel processing which we have described above may ultimately play an important role within practical database machines, our primary concern in this paper will be with the family of associative processors.

In general, we will define an associative processor to be a machine which is able to access selected items stored in memory in parallel on the basis of their contents. We will also require that items be accessible by partial match, so that selected elements of the "key" field can be "masked out" in the course of content-based addressing. (In many associative processors, the match criteria may be specified in using predicates c&her than equality-arithmetic comparison operators, for example; we will not require this capability as part of our definition of an associative processor, however.) While the parallel modification of content-selected responders is supported directly by a hardware multiwrite capability, output from an associative processor in the case where there is more than one responder presents a more complicated problem. A number of designs have been proposed and evaluated for reading out a single responder in the event of a multiple match. Typically (though not always) this responder is chosen arbitrarily on the basis of physical position within the associative memory. Although much work has been done in the area of multiple match resolution, we will not be concerned with such problems in this paper.

While the distribution of intelligence among memory elements is central to -the operation of all associative processors, the degree of that distribution-more specifically, the number of storage elements associated with each comparison logic unit— varies widely among the various classes of associative devices. In the remainder of this section, we will review the major categories of associative processor architecture, distinguished by the extent of distribution of the processing logic. A survey by Yau and Fung [1977] provides an outline of associative processor architecture in somewhat more depth than will be possible here. The area is treated even more thoroughly in an outstanding book by Foster [1976].

The greatest degree of distribution is found in the ***fully parallel*** associative processors, in which comparison logic is associated with every bit of storage. The fastest of these machines are the ***fully*** parallel word organized associative processors, whose hardware complexity, however, has resulted in their implementation only experimentally, and on a very small scale.

A second class of fully parallel designs is represented by the distributed *logic* associative processors. In the original distributed logic associative processor design introduced by Lee [1962], one comparison logic unit is associated with each character of storage. (In some variants, the comparison logic unit is instead associated with a small, fixed-size set of adjacent characters.) In all of the distributed logic associative processors based on Lee's design, each cell is capable of storing a small amount of "state" information in addition to the symbol data itself. The design includes a control unit which communicates with all cells in parallel using a common databus. Each cell, however, is connected not only to this public bus, but also to its immediate right and left neighbors, thus forming a rail along which control and state information can be propagated.

With some simplification and disregard for detail, a string of data stored in a contiguous set of character cells is retrieved as follows. Initially, the control system "broadcasts" a special "word header" character which precedes all strings stored in memory. Each matching header cell is then instructed to enable its right neighbor-for comparison against the first character in the search string. All matching first characters in turn enable their right neighbors for matching against the next character, and so on until the search string is exhausted. The set of matching strings is now easily identified, and may be modified or output. A number of variations on Lee's original distributed logic design have been proposed to deal efficiently with certain operations frequently required in the course of information retrieval, parallel arithmetic manipulations, etc. (eg., Lee and Paul1 [1963]; Gaines and Lee [1965]; Crane and Cithens [1965]). The content addressing mechanisms incorporated in PEPE, one of the first large-scale associative processor implementations, may also be regarded as derivative of Lee's design.

Among the numerous distributed logic designs which have been suggested, the Tree Channel Processor architecture proposed by Lipovski [1969; 1970] for the construction of very large primary associative processors is worthy of special mention. In Lipovski's design, the cells are themselves capable not only of passive comparison and simple propagation, but (in a particular mode of operation) of the active execution of certain control functions. The cells are organized in a tree structure, with "adjacent" cells connected by two separate rails and a "locally" common channel. In contrast to the strictly public bus used in the basic distributed logic design, this channel may be dynamically partitioned, thus isolating one or more subtrees which can then function as separate processing units. (In this respect, the Tree Channel Processor might in fact be considered a unconventional MIMD machine.) The Tree Channel Processor is designed to permit extremely high bandwidth parallel input and output and to greatly reduce certain propagation time bottlenecks associated with many applications of distributed logic processors.

Let us now turn our attention to a class of associative processors characterized by somewhat less extensive distribution of intelligence: the ***bit-serial*** associative processors. In this class of machines, first proposed by Shooman [1960], the content addressable memory is organized into (often fairly large) words, and comparison logic is associated with each word. In contrast with the fully parallel word organized processors, however, each logic unit is capable of manipulating only a single bit position within the word at a given time, resulting in a reduction in required processor logic roughly proportional to the number of bits per word, at the cost of a corresponding decrease in speed. At each point in time, one "bit slice" extending through all words is thus accessible for parallel processing. A small amount of storage associated with each word is typically used to retain state information between operations on successive bit slices in support of the primitive content search and multi-write capabilities of associative processing.

Since the introduction of Shooman's "orthogonal computer", bit-serial associative devices having a wide variety of characteristics have been proposed by a number of researchers, including Kaplan [1963], Ewing and Davies [1964] and Chu [1965]. The design of STARAN (Rudolph, 1972; Batcher, 1974], Goodyear Aerospace Corporation's large-scale associative processor, is based on a group of "multidimensional access memories" which implement both bit-slice (for associative processing) and ordinary word slice (for input and output) access capabilities -using standard random access memory chips. Among the other bit-serial associative processors which have been developed for practical use are the OMEN series [Higbie, 1972], designed by Sanders Associates, The Raytheon Associative/Array Processor (abbreviated RAP, but not to be confused with the Relational Associative Processor, a database machine bearing the same acronym which will be discussed in Section 3.2) [Couranz, Gerhardt and Young, 1974], the Extended Content Addressed Memory (ECAM) [Anderson and Kain, 1976], and the Hughes Aircraft Associative Linear Array Processor, (ALAP) [Finnila, 1977].

Another class of less-than-fully-parallel content-addressable devices is comprised of the ***word-serial*** associative processors [Young, 1962; Crofut and Sottile, 1966; Rux, 1969], in which all bits of a single word are compared in parallel, but the *set* of words is examined sequentially. Word-serial machines thus function in much the same way as a program loop on a conventional von Neumann machine in which each word in memory is examined in turn for partial match and modified or output as appropriate. The word-serial associative architecture, however, obviates the need to fetch and decode the instructions which would be required to perform such functions in software on an ordinary von Neumann machine. While the word size of a word-serial machine might in principle be chosen large enough to make word-serial techniques competitive in speed with bit-serial schemes, the number of words is generally much larger than the number of bits per word, thus typically making word-serial techniques much slower in practice. Although this speed disadvantage has thus far tended to discourage practical applications of the word-serial approach in favor of distributed logic and bit-serial techniques, current prospects for inexpensive, high density, noninertial circulating storage devices (future generations of bubble memories or charge-coupled devices, for example) may make the word-serial approach worthy of serious consideration for large-scale associative processing applications.

At the low end of the associative logic distribution spectrum is the class of ***block-oriented,*** or segment sequential associative processors (also sometimes called partially associative devices), which offer much larger capacities than the devices discussed thus far, but at a significant penalty in speed. Most commonly, such devices are based on a rotating storage device (a disk, for example) having one or more heads per track of storage, so that each piece of stored information passes under some head exactly once during each revolution of the device. In the simplest such designs, one search and modification logic unit is associated with each head (and thus with each track), permitting one associative operation to be performed on each revolution.

The first logic-per-track associative devices of which we are aware were proposed by Slotnick [1970] and Parker [1971]. Parhami [1972] designed an associative processor called RAPID (for Rotating Associative Processor for Information Dissemination), which functioned in much the same way as a slow distributed logic memory, but with only one search operation possible per revolution, and with information propagation in one direction only. Different block-oriented associative processor designs have been proposed by Minsky [1972], Healy, Lipovski and Doty [1972], and others. Because the need for large-scale storage is essential to data base management applications, parallel head-per-track disk devices, or their equivalent, have a central role in the majority of the database machine designs discussed in Section 3.2.

While the block-oriented associative processors are usually regarded as rep resenting the "low end" of the logic distribution spectrum within the family of associative processor architectures, certain system designs based on an even lower degree of distribution, but nonetheless sharing some of the features of an orthodox associative processor, might be worth mentioning in passing. One such approach is illustrated by the modified head-per-track disk drives incorporated in the DBC architecture (discussed in Section 3.2), in which the contents of one cylinder may be associatively probed in a single revolution. An even lower degree of logic distribution which nonetheless speaks to some of the concerns of associative processing is represented by the design proposed by Lang, et al, [1977] for the evolutionary enhancement of conventional disk-based systems—the authors' proposal was in fact presented in the context of an architecture like that of the IBM System/370—for increased efficiency'in database applications. Their scheme involved the association of one small, intelligent unit called a "DASD processor" with each direct access storage device. Each such processor would be capable of searching for records based on the values in arbitrarily specified (as opposed to fixed-position) fields when so instructed by a special channel command. Analysis predicted significant performance improvement over more conventional system architectures, particularly in the case of heavy transaction traffic.

In the following section, we will review some of the ways in which the various classes of associative processors have been applied to the specific problems of database management.

3.2 Database machines

Several authors have surveyed the emerging field of database machine architecture from various perspectives, and adopting various scopes, within the past several years. Linde, Gates, and Peng [1973] were among the first to point out the potential advantages of associative processor-based architectures for real-time database management applications. Berra [1974] reviewed the state of the art as of 1974, and critically examined the potential for such applications, pointing out a number of positive and negative aspects of the application of associative processors to database management. Anderson [1976] and Baum and Hsiao [1976] provided later overviews of trends in the field, the latter predicting the emergence of hierarchically organized systems, with each level containing functionally specialized search and data manipulation modules. Lowenthal [1977] offered a taxonomy for distinguishing three different kinds of processors specialized for data base management in distributed environments, which he called intelligent controllers, *backends* and *datacomputers*. Hsiao and Madnick [1977] and Berra [1977] also survey and provide references to the field of data base machine architecture. In this section, we will review the best known efforts to date in the area of database machine architecture.

One of the earliest actual implementations of an associative processor-based system geared toward database management applications was IFAM (DeFiore and Berra [1973]), developed on an experimental prototype basis for the Rome Air Development Center. This implementation of IFAM was based on a 2048-word, 48-bit, word parallel, bit serial associative processor called AM, developed by Goodyear. The capabilities of IFAM were closely tied to the primitive associative operations; in relational terms, tuples could be retrieved only by selection (although with inequality and "within-range" comparisons in addition to simple equality). Although limited in storage capacity by comparison to later block-oriented associative processor-based database machines, IFAM served to concretely illustrate the potential utility of associative operations in database management applications.

Moulder [1973] described an implementation of a hierarchical database management system based on STARAN (described in Section 3.1) and a parallel head-per-track disk drive. Using a technique similar to that described by DeFiore, Stillman and Berra [1971], the hierarchical data structures chosen for data representation were converted into a single level data base to permit the use of the associative processing capacities of the hardware. Retrieval was again by selection based on equality or inequality (but not ranges) over various attributes. The database was partitioned into a number of physical disk sectors which were sucessively read into the STARAN memory arrays in a high speed parallel fashion, where they were searched using the associative capabilities of STARAN. The time required in the case of typical queries to perform these associative searches within the STARAN arrays was found to be small enough that every other sector could be searched in the course of one revolution of the disk, so that the whole data base could be searched in two revolutions (about 78 msec in the prototype system).

One of the first large-scale research efforts directed toward the development of a specialized system containing many of the features critical to database management is represented by the CASSM project, active at the University of Florida since 1972. CASSM [Su, Copeland and Lipovski, 1973; Copeland, Lipovski and Su, 1973; Lipovski, 1978] is a block-oriented design oriented specifically toward a hierarchical data model, providing a direct hardware implementation of hierarchical data structures, which are linearized in a top-down, left-to-right manner; CASSM is capable of supporting the relational and network (Codasyl DBTG) models as well, however.

In the terminology of CASSM, the system architecture includes a collection of identical cells, each consisting of a processing element and a circulating sequential memory element (a disk track or a circularly organized CCD or bubble memory device, for example). Each processing element can communicate with its two immediate neighbors, in support of the storage of files and records which overlap physical segments of the secondary storage device. Associated with each cell are two heads: one used for reading, and one for writing data. After being read by the first head, data is pipelined through a chain of processing logic, each portion of which serves a specialized function. CASSM includes special features for searching complex data structures such as sets, ordered sets, trees, variable length character strings and directed graphs. Among the distinctive features of CASSM is the fact that both programs and data are stored on the associative secondary storage device. Both an assembly language [Su, Chen and Emam, 1978] and a high-level nonprocedural language [Su and Emam, 1978] have been developed for programming the CASSM system. A single cell prototype system was completed in 1976. Since that time, efforts have concentrated on the implementation of and experimentation with a software simulation of a multi-cell CASSM system.

The best-known database machine designed specifically for efficient support of the relational model of data is probably RAP (for Relational Associative Processor), developed at the University of Toronto [Ozkarahan, Schuster and Smith, 1974, 1975; Schuster, Ozkarahan and Smith, 1976; Ozkarahan, 1976]. RAP is designed as a backend database processor for a general purpose computer, accepting from the latter a set of primitive commands relevant to the evaluation of relational queries. Like CASSM, the RAP architecture is organized around a set of identical cells, each consisting of a processor and a block of circulating memory, and capable of various retrieval, insertion, deletion and update functions. All cells are connected to a common controller, which includes a statistical arithmetic unit. Simple inter-cell communication facilities are provided for priority polling in the course of output. The front end computer is used to translate different query languages into RAP primitives, to handle various input/output processes, for query scheduling, and for various functions related to the maintenance of protection, security and data integrity. The RAP language interface is described by Kerschberg, Ozkarahan and Pacheco [1976] and Ozkarahan and Schuster [1976].

An analytical comparative performance evaluation [Ozkarahan, Schuster and Sevcik, 1977] revealed advantages in speed ranging between one and three orders of magnitude by comparison with a hypothetical conventional system using inverted lists-with the very important exception of the join operation, where only a slight improvement was found. (Note that it is just this sort of operation for which our own architecture offers the greatest potential advantages.) Specific aspects of the performance of of RAP are examined by Nakano [1976] and Ozkarahan, Schuster and Sevcik [1977]. The RAP system has now evolved for several years, with the latest version, called RAP.2 [Schuster, Nguyen, Ozkarahan, and Smith, 1979], embodying several significant changes. First, a general purpose microprocessor has been employed for implementation of the previously hardwired controller. Second, the RAP.2 design is strongly oriented toward the use of CCD memories instead of head-per-track disk devices. The instruction set has also been modified somewhat in RAP.2 to make it more uniform and flexible, and to add certain additional capabilities. Enhancements based on analogues of multiprogramming and virtual memory organizations have been proposed by Ozkarahan and Sevcik [1977].

Another architecture specifically oriented toward the relational database model is embodied in a proposed database machine called RARES [Lin and Smith, 1975; Lin, Smith and Smith, 1976]. The RARES design is distinguished primarily by the adoption of an orthogonal storage layout, in which individual tuples are distributed across (and not along) the tracks of the parallel head-per-track secondary storage device, with one byte stored on each track. In the orthogonal storage scheme, a given relation thus occupies all tracks within a particular sector of the disk device (whose extent depends on the size of the relation), rather than completely filling a corresponding number of tracks. One motivation for the orthogonal scheme adopted in the RARES design is to reduce the incidence of contention in cases where more than one tuple is identified in parallel for output. Among the other advantages cited for this scheme are a reduction in the amount of storage necessary to hold each tuple in the course of associative comparison and certain efficiencies in the execution of operations on relations in which a sorted order must be maintained.

The relational database machine architectures which we have thus far considered have primarily addressed the problems of evaluating a single relational primitive operation. An organization called DIRECT [DeWitt, 1979], on the other hand, is directed to a broader set of problems, dealing with such questions as intra- and inter-query concurrency and database integrity in a multiple-process relational database environment. DIRECT is a virtual-memory, MIMD (see Section 3.1) system, currently being implemented using a number of DEC LSI-11/03 microprocessors, along with O-based associative storage units. The microprocessors and CCD modules are connected using a special cross-point switch design, with the number of processors assigned to the evaluation of a given query determined dynamically based on certain statistics of the query and the relations involved.

At Ohio State University, an architecture has been proposed for a very-large-scale database system based on the use of a number of interconnected subsystems specialized for different aspects of the process of database managcmcnt. This systern, called DBC (for database computer) [Baum and Hsiao, 1976; Hsiao, 1977; Banerjee, Baum, Hsiao and Kannan, 1979], is designed to support all three data models, communicating with a general-purpose computer through a very high level language oriented toward the data base management functions for which DBC is intended. The design of DBC was strongly influenced by several kinds of data protection concerns, and includes specialized mechanisms for the imposition of related constraints.

The system is composed of two sets of processor and memory components, configured as closed loops, and interconnected (both to each other and to the general purpose computer to which DBC is subordinated) by a database command and control processor. The first, called the data loop, contains a mass *memory* based on a number of modified moving-head disk drives, along with a specialized processing unit called the security filter processor. The second, called the structure loop, is comprised of a block-oriented associative storage unit (envisioned to be constructed using CCD or bubble technology) called the *structure memory,* another specialized processing unit called the structure memory information processor, and two other specialized modules called the keyword transformation *unit* and the index translation unit.

The moving-head disk drives are modified to provide for simultaneous output from all tracks in a given cylinder in parallel. (Such drives have in fact recently been announced by Ampex Corporation [1978], and are apparently not expected to be priced far above the cost of unmodified moving-head drives.) Associated with each track is a ***track information*** processor, capable of associative comparison operations. Thus, a single cylinder can be searched associatively by DBC in much the same way as were the full contents of secondary storage in the associative head-per-track devices discussed earlier. Information which is used to locate the relevant cylinders to search is stored in the structure memory unit, which is designed for very fast access and processing by the structure memory information processor, in conjunction with the keyword transformation and index translation units. A more detailed description of the structure memory, structure memory information processor, keyword transformation unit and index translation unit are provided by Hsiao and Kannan [1976] and Hsiao, Kannan and Kerr [1977]. The design of the mass memory, security filter and associated units are detailed in Hsiao and Kannan [1976a].

Other proposals for specialized database architectures include XDMS [Canady, et al., 1974] a network-oriented SISD architecture originating at Bell Laboratories, and an approach to the implementation of a relational database system suggested by McGregor, Thompson and Dawson [1976].

4. The Proposed Architecture

As noted in the introduction, our proposed architecture is configured as a hierarchy of associative storage devices. At the top of this hierarchy is the primary associative *memory* (PAM), a fairly fast content-addressable memory of relatively limited capacity. (For concreteness, the reader might imagine a PAM containing between 10K and 1M bytes, and requiring somewhere between 100 nanoseconds and 10 microseconds per associative probe.) PAM might be realized with a large-scale distributed logic memory, or with a suitable bit-serial or word-serial design. There is reason to believe that recent progress in distributed logic architectures, device-level fault-tolerant designs and wafer-scale integration could soon make such a memory unit feasible for wide application.

Two primitive PAM operations, each requiring a single associative probe, will be involved in our analysis: mark all and retrieve and mark first. In both cases, all tuples of a specified relation for which the value of a selected compound attribute is found equal to a particular constant are associatively identified. The mark *all* operation writes a one or zero in a specified Aag *bit* of each such matching tuple using a parallel hardware multiwrite. The *retrieve* and mark *first* operation sets a specified flag bit within a single tuple chosen arbitrarily from among the responders and copies the value of that tuple to storage external to PAM, but accessible to the controlling processor.

As an alternative to physical content-addressability, the algorithms which we will describe could be modified to accomodate a "pseudo-associative" PAM, constructed using, say, random access memory and high-bandwidth special purpose hash coding hardware. In general, however, argument and intermediate result relations would have to be re-hashed (on different attributes) prior to every algebraic evaluation, adding a significant (and less predictable, as seen from the wide discrepancy between average and worst case hashing behavior) amount of time to the algorithms presented in Chapter 6.

The secondary associative memory (SAM) is intended to be a larger, slower content-addressable device. (A capacity of between 1 and 100M bytes and an associative operation time of between 1 and 100 milliseconds should adequately exemplify our design.) Physically, SAM might be realized using an intelligent circulating storage device such as a parallel head-per-track disk with a modest amount of logic associated with each track, or a non-inertial circulating storage device constructed using CCD or bubble memory storage technology, and having similar logic associated with each storage loop. (The ability to temporarily suspend circulation in individual storage loops in the latter class of device could in fact be utilized to improve somewhat on the external evaluation results reported in this paper, although such enhancements are not within the scope of our present discussion.) It is assumed that the relative speeds of PAM and SAM are such that a quantity of data sufficient to fill PAM can be transferred from PAM to SAM in the course of a single SAM revolution. Although the combined potential bandwidth of the set of intelligent heads associated with the SAM device could in principle be extremely high, the average bandwidth in the course of a external evaluation will ordinarily be much lower. Given adequate buffering capabilities, the algorithms which we will describe should thus present no unusually stringent requirements on the communication channel between SAM and PAM.

Among the specific capabilities assumed for the "per-track" logic of an acceptable SAM device is the ability to output or mark all tuples for which the values of selected attributes are found equal to a constant or to the value of some other attribute within that tuple, or to be within some specified range of values. Note that SAM is thus capable of evaluating the select and restrict operators directly, without recourse to "internal evaluation" within PAM. Each per-track logical unit is also assumed to have a sufficient quantity of random access buffer memory to hold the tuple currently passing under its head until a determination can be made, on the basis of selective or restrictive criteria, as to whether it satisfied the current match criteria.

The specifications which we have thus far considered for SAM are quite similar to those of such actual rotating associative processors as those used in the RAP and CASSM systems. One of the techniques we will describe (in Section 8.1), however, also requires that each per-track processing unit have a small amount of random access memory dedicated to the tabulation of a "domain histogram". In addition, this algorithm requires that the unit be capable of determining whether each tuple satisfies one of a set of (not more than a small fixed number of) range specifications. In the alternative algorithm (described in Section 8.2), the per-track logic unit is instead assumed to have the capability of sequentially computing a hashing function on selected attribute values of each tuple which "passes under" the associated head (or its functional equivalent), and of outputting all tuples for which the resulting hashed value falls within a specified range.

The analytic portion of this paper assumes a fixed time for an associative probe of the entire contents of SAM, as is the case for the sort of block-oriented associative processors discussed in Section 3.1 and employed in such database architectures as CASSM, RAP and RARES (Section 3.2). Our external evaluation algorithms are also applicable, however, to the kind of modified moving-head disk devices employed in the DBC design (Section 3.2), thus supporting very large data base applications. In order to adapt the complexity results reported in Section 7 to a SAM of this sort, in which only part of the database is associatively accessible on each rotation, a constant term would be added to the external evaluation times of each of the seven primitives. In addition, the complexity of these results would be increased (by a formally linear, although in practice probably quite small) factor in the event the argument relation(s) were allowed to exceed the capacity of the cylinder or cylinders capable of simultaneous parallel examination (at least in the absence of a significant modification of our algorithm). To simplify our discussion, however, the remainder of this paper will assume that SAM is a fixed probe time associative device of sufficient capacity to store both argument relations.

The seven relational algebraic primitives with which we are concerned may be evaluated most quickly when the argument relation(s) can fit into PAM- the case we have referred to as internal **evaluation.** (Similarly, we will use the terms ***internal projection,*** internal ***equi-join,*** etc., to refer to the evaluation of specific relational operators in the case where their argument relation(s) fit entirely within SAM.) External evaluation is performed whenever the argument relation(s) fit in SAM, but not in PAM, and in most cases involves the reading into PAM of successive segments of the argument relation(s), each of which is (are) processed according to the corresponding internal evaluation algorithms, Note that this implies that each tuple of the argument relations is processed only once in primary storage, in contrast with the best currently known general techniques for the external evaluation of most of the algebraic primitives under consideration on a conventional non-associative system.

In addition to the two associative devices involved in our design, we assume the existence of a general purpose processor serving as a controller for the evaluation process, and responsible for the performance or delegation to other specialized units of all collateral functions (input language translation, input/output control, etc.) which would be involved in a practical implementation. Adequate buffering would also be required at several points within the design we are proposing. Although we will give little explicit attention to such issues in the present paper, it should be acknowledged that the detailed design of a useful realization of the architecture we propose would require careful consideration of the nature and capacities of these resources.

5. Notation

The following notation will be **used** in our analysis of the algorithms for the internal and external evaluation of the relational algebraic primitives:

Fixed system parameters:

P	Size in bytes of the primary associative memory (PAM)
S	Size in bytes of the secondary associative memory (SAM)
T_p	Time for an associative probe (**returning** one matching tuple) in PAM
T_r	Time for one revolution of SAM

Functions of the argument relation(s):

$c(R)$	cardinality of the relation R
$t(R)$	(fixed) size of the tuples of R in bytes
$d(A,R)$	number of distinct values of the (compound) attribute A in R
r	cardinality of the result relation

Because the quantity $P/t(R)$ (roughly speaking, the 'tuple capacity' of PAM) plays an important role in our analysis, we will also define a derived function a(R) with this value.

It should be noted that r is being treated as an independent variable, although it is in fact determined by the composition of the argument relations. There are several ways in which this functional dependence might have been explicitly embodied in our analysis if we had chosen to do so. We might have used, for example, a fixed value estimating the average number of occurrences of any given join attribute value, or for a more careful analysis, a particular statistical distribution of such values might have been assumed. While such an analysis might well help to identify certain interesting properties of the proposed algorithms when applied to argument relations having various properties, we have chosen in the present analysis to forego the considerable added complexity involved in explicitly examining such relationships, treating the cardinality of the result relation as a constant and indicating verbally its relationship to the arguments where appropriate.

When there is no danger of confusion, we will sometimes omit the relation argument R.

6. Internal Evaluation

Our algorithms for internal evaluation of the project and join operators will be expressed in a hypothetical parallel programming language having a Pascal-like format, but extended to include four high-level associative processing primitives. The first is the **parallel set** command, used to set a specified flag to ***true*** in each tuple satisfying certain conditions; all flags are set in parallel using a single mark ***all*** operation, requiring one associative probe. This command has the form

parallel set ⟨*flag*⟩ in all (tuple variable) **of** (relation) with (conditions) ,

where (conditions) is a Boolean combination of predicates involving the variable (tuple variable). The format of the parallel clear command **is** identical to that of **parallel set,** but sets the specified flags to false.

The third **associative** processing primitive is the **for** each control structure, which has the form

for each (tuple variable) with ***(conditions)*** **[set** (flag) **and] do** ***(statement)*** ,

where the **"set.. . and"** clause is optional. Unlike the **parallel** set and parallel clear statements, execution of a for **each** loop is sequential (although each iteration of the loop involves the performance of parallel associative probes). During each iteration, a single ***retrieve*** and mark ***first*** operation is performed, during which **⟨*tuple*** variable) is instantiated with an arbitrarily chosen tuple satisfying ***(conditions).*** If a "set . . . **and"** clause is specified, the appropriate **⟨*flag*⟩** is set within this tuple; (statement), which may be either a simple statement or a "begin . . . end" block, and which may set flags affecting the value of (conditions), is then executed with the current binding of (tuple variable). Iteration terminates when no further tuples of the specified relation satisfy ***(conditions).***

The final primitive is a conditional statement, which has the form

if [not] **exists** (tuple variable) with (conditions) **[set** (ffag) and] do (statement) ,

where **not** is optional. This statement executes a ***retrieve*** and mark ***first*** operation, executing (statement) if any tuple satisfies ***(conditions)*** **(or in** the case where not is specified, **if** no tuple **satisfies** (conditions)).

```
procedure project (R, A);
  for each t of R
    with not flag do              (r + 1 probes)
      begin                       (r times)
      output  t[A];   .
      parallel set flag           (r probes)
        in all t' of R
        with t'[A] = t[A];
      end;
```

Algorithm 1. Internal Project

6.1 Project

The procedure for internally projecting a relation R over a compound attribute A is detailed in Algorithm **1.**

From the execution counts, it can be seen that internal projection requires time

$$(2r + 1)T_p$$

in addition to the time required to extract the projected compound attribute of, and output, each of the r result tuples, both non-associative functions which could be overlapped with the following associative probe. **As** noted in Section 2.2, projection can be quite expensive on a von Neumann machine, particularly in the case where the argument relation is large. The utility of the proposed architecture for the evaluation of the relational project operator thus lies not only in the fact that it requires time independent of the size of the the argument relation (being proportional only to the cardinality of the result relation, which can never be larger, and is often much smaller), but also that it implicitly eliminates the possibility of tuple duplication, obviating the need for sorting, for example, to remove redundant result tuples.

```
procedure join($R_1, R_2, A_1, A_2$);
      for each $t_1$ of $R_1$
        with not flag
        set flag and do                              ($d(A_1,R_1) + 1$ probes)
            begin                                    ($d(A_1,R_1)$ times)
            distribute($t_1,R_2,A_1,A_2$);
            for each $t'_1$ of $R_1$
              with $t'_1[A_1] = t_1[A_1]$
                and not flag
              set flag and do                        ($c(R_1)$ probes)
                  distribute($t'_1, R_2, A_1, A_2$); ($c(R_1) - d(A_1, R_1)$ times)
            end;

procedure distribute($t_1,R_2,A_1,A_2$);
      begin                                          ($c(R_1)$ times)
      for each $t_2$ of $R_2$
        with $t_2[A_2] = t_1[A_1]$
          and not flag
        set flag and do                              ($r + c(R_1)$ probes)
            output ($t_1[A_1] | t_2[A_2]$);
      parallel clear flag
        in all $t_2$ of $R_2$
        with $t_2[A_2] = t_1[A_1]$;                  ($c(R_1)$ probes)
      end;
```

Algorithm 2. Internal Join

6.2 Join

Algorithm 2 computes the equi-join (or with the indicated modification, the natural join) of relations R_1 and R_2 over the compound attributes A_1 and A_2, respectively.

Intuitively, the join algorithm functions as follows: First, an arbitrary R_1 tuple is retrieved and marked. The extended Cartesian product of all (associatively retrieved) R_1 and R_2 tuples having the same value in their respective compound join attributes as this arbitrarily selected tuple is then output. Another arbitrary R_1 tuple is then arbitrarily selected from among those which have not yet been processed, and the above procedure repeated until all R_1 tuples have been exhausted, at which point the equi-join is complete. The process of forming the extended Cartesian product involves a nested iteration over all matching R_1 (in the outer loop) and R_2 (in the subfunction ***distribute)*** tuples, each of which is retrieved in a fixed amount of time, without regard to its position in memory, by virtue of the content-addressibility of PAM. Excluding the time required for concatenation and output,

$$(r + 3c(R_1) + d(A_1,R_1) + 1)T_p$$

is required for internal evaluation of the join operator.

Note that the asymmetry of this algorithm with respect to the roles played by the two argument relations permits a (possibly quite significant) increase in efficiency in the case where the relative sizes of the two argument relations is known or inexpensively computable. Some of the existing designs which might be chosen for a particular physical implementation of PAM are in fact capable of providing, in a single associative operation, a count of the number of responders to an associative probe. When this capability is provided, the above algorithm may be preceded by two counting probes (on all R_1 and R_2 tuples) to determine the smaller relation, When such relative size information is available, R_1 should in practice generally be chosen to be the smaller of the two relations in order to minimize the size of the $c(R_1)$ and $d(R_1)$ terms, since $d(R_1)$ might in practice be expected to be directly related (or at least not strongly inversely related) to $c(R_1)$. This observation is particularly significant in the common special case where the two argument relations are of very different sizes. As it happens, our external algorithm for the evaluation of two very large relations A and B admits the possibility of assigning A segments to R_1 during some of the internal cycles, and B segments during others. At the cost of a very minor complication of the procedures for transfer from SAM into PAM, the algorithm can thus in some cases be made to perform more efficiently than would be the case if either A or B were "bound" to R_1 for the duration of the join, yielding a modest improvement on the above results.

As in the case of projection, it is instructive to compare the proposed associative equi-join algorithm to the best known general algorithms for this operation on a- conventional von Neumann machine, which, as seen from the discussion in Section 2.3, are of $O(n \log$ n) complexity in the absence of physical clustering with respect to the join attributes or the use of extensive storage redundan[illegible] On the machine we have described, on the other hand, tuples can be set in correspondence using a procedure of lower computational complexity than sorting, yielding a joining time which is linear in the cardinality of the smaller argument relation, the number of distinct join attribute values in this relation, and the size of the result relation. (As we shall see in Section 7.3, linear complexity is preserved in the external algorithm for equi-join as well.)

Lest these results be misinterpreted, it should be emphasized the worst case behavior of this algorithm (or indeed, of any algorithm involving sequential output, regardless of the underlying architecture) may still be quite bad when the ***result*** relation is very large. Specifically, if for all $t_1 \epsilon R_1$ and $t_2 \epsilon R_2$,

$$t_1[A_1] = t_2[A_2] = t_c$$

for some single constant tuple t_c, the cardinality of the result relation will be equal to the product of the cardinalities of the two input relations. Given reasonable assumptions reflecting the typical **use of** the join operation, however, the architecture and algorithm which we have described offer a very significant increase in efficiency.

It is worth noting that the algorithm we have described assumes access only to a structural model of the data, and not to any of the semantic characteristics of the stored relations (both terms being understood in the senses applied in the relational database literature). In fact, such semantic information, if available, could be used to significantly improve on several important special cases of the above join algorithm. As an example, consider the case where the compound join attribute is in fact a primary key of R_1, R_2 or both- that is, where the value of the join attribute uniquely identifies a single tuple of the relation. In this case, the associative probe used to terminate the above the for each control structures is unnecessary, resulting in a saving of roughly half of the necessary probes within the innermost two loops of the algorithm. In many problems, the availability of domain-specific knowledge might permit certain other kinds of improvements on these results. Although an adequate analysis of the manner in which such additional sources of information might be profitably integrated into our approach is unfortunately beyond the scope of our present discussion, it is worth noting that the very general case of evaluation on the basis of purely structural characteristics, to which our attention is currently directed, may often in practice ignore information sources which might lead to increased efficiencies.

```
procedure select (R, A, V);
    for each t of R
      with t[A] = V
       and not flag
      set flag and do              (r + 1 probes)
          output t;
```

Algorithm 3. Internal Select (with sequential output)

6.3 Select

The algorithm for selection is quite straightforward within the architecture we have specified, since the associative retrieval primitive which defines the behavior of PAM itself serves what is essentially a selective function. If, in a particular application, it is not necessary to sequentially enumerate and output the result of a selection, but only to mark the included tuples, (as may in fact be the case in the evaluation of many complex queries), the operator in fact requires only a single probe, and takes exactly time T_p, independent of the size of the argument relation. When sequential output is required, the selection from relation R with compound attribute A equal to value tuple V is defined as in Algorithm 3. It is easily seen that $r + 1$ probes are required, so that the time required for a single selection with sequential output is simply

$$(r+1)T_p$$

It should be noted that the time required for selection with sequential output is again independent of the size of the argument relation.

```
procedure restrict($R, A_1, A_2$);
    for each $t$ of $R$
     with not flag do                         ($d(A_1,R_1)$ + 1 probes)
        begin                                 ($d(A_1,R_1)$ times)
        for each $t'$ of $R$ .
          with $t'[A_1] = t[A_1]$
            and $t'[A_2] = t[A_1]$
            and not flag
          set flag and do                     ($r$ + $d(A_1,R_1)$ probes)
              output $t'$                     ($r$ times)
        parallel set flag
          in all $t''$ of $R$
          with $t''[A_1] = t[A_1]$ of $R$     ($d(A_1,R_1)$ probes)
        end;
```

Algorithm 4. Internal Restrict

6.4 Restrict

The procedure for internal restriction is detailed in Algorithm 4. Initially, an arbitrary tuple is chosen from the argument relation. If the A_1 and A_2 values of this tuple are equal, one tuple having this value for both A_1 and A_2 is output during each successive probe until exhaustion. At this point, *all* tuples having that value for their A_1 attribute are flagged, and the process is repeated on all unflagged tuples. The total time required for internal restriction is

$$(2d(A_1,R_1) + r)T_p .$$

It is worth mentioning that the addition of certain hardware capabilities to the PAM device may substantially decrease the complexity of internal restriction. If the hardware permits the associative retrieval of all tuples in which a Boolean disjunction of attribute-value pairs is specified, for example, the **parallel set** instruction can be changed to flag all tuples in which the value of either A_1 *or* A_2 is equal to $t[A_1]$; the elimination of such tuples may exclude from consideration some of the $d(A_1,R_1)$ tuples having distinct A_1 values without the need for a separate associative probe in the outer loop. A more significant improvement may be possible if the PAM device itself supports the associative retrieval of tuples having identical values in specified fields; in this case, internal restriction has the same complexity as selection.

```
procedure union($R_1$, $R_2$);
      begin
      for each $t_1$ of $R_1$
        with not flag
        set flag and do                              ($c(R_1) + 1$ probes)
            begin                                    ($c(R_1)$ times)
            output $t_1$;
            parallel set jlag                        ($c(R_1)$ probes)
              in all $t_2$ of $R_2$
              with $t_2 = t_1$
            end;
      for each $t_2$ of $R_2$
        with not flag          set flag and do       ($r - c(R_1) + 1$ probes)
            output ($t_2$;
      end;
```

Algorithm 5. Internal Union

6.5 Union

The algorithm for the union of relations R_1 and R_2, assuming as usual the requirement for sequential output of the result relation, has two stages. First, each tuple of R_1 is output in succession, and each one which also occurs in R_2 is associatively marked to avoid duplication in the result relation. Second, all unmarked R_2 tuples are output. The procedure is detailed in Algorithm 5. From the execution counts, it may be seen that the algorithm requires time

$$(r + c(R_1) + 2)T_p \quad .$$

It should be noted that, as in the case of the join operator, this algorithm is asymmetric with respect to R_1 and R_2, and is more efficient when R_1 is chosen to be the smaller of the two argument relations. The techniques discussed in Section 6.2 may thus be employed to optimize the efficiency of the evaluation of union on the basis **of the relative** sizes of its argument relations.

```
procedure $intersect(R_1, R_2)$;
    begin
    for each $t_1$ of $R_1$
      with not $flag$
      set $flag$ and do              $(c(R_1) + 1$ probes)
          if exists $t_2$ in $R_2$     $(c(R_1)$ probes)
            with $t_2 = t_1$ do
                output $t_1$;
```

Algorithm 6. Internal Intersect

6.6 Intersect

In Algorithm 6, which computes the intersection of relations R_1 and R_2, each tuple of R_1 is examined in turn, and an associative probe is performed to determine whether the tuple in question is also a member of R_2. It is easily seen that

$$(2c(R_1) + 1)T_p$$

is required to intersect two relations in PAM. Again, the dependence of our result on the choice of R_1 should be noted. Selection of the smaller argument relation for R_1 is in fact somewhat more important in the case of set intersection than set union because of the larger relative contribution of the cardinality of R_1 to total execution time.

It is interesting to compare our algorithm for set intersection with the one presented for the join operator. Note that set intersection may be regarded as a special case of natural join in which the compound join attributes are exactly the set of *all* attributes of the argument relations. In the case of intersection, though, we know that no two tuples in an argument relation can have the same value for this compound join attribute, since relations are in fact sets, and are thus prohibited from containing duplicate tuples as elements. This is precisely the sort of "additional information" discussed earlier which must, in the case of the general join, be determined by reference to the semanticsof the particular database at hand. Because this information is available on purely structural grounds in the case of intersection, our intersect algorithm avoids the probe which is always necessary to detect exhaustion of all R_1 tuples having the current join attribute value. In the case of those join attribute values which match some R_2 tuple, an additional probe is saved over the case of the general join, for much the same reason.

Recent work by Trabb-Pardo [1978] on the complexity of set intersection on a von Neumann machine suggests another interesting perspective on our associative algorithm for intersection. Trabb-Pardo considered two strategies for representing and intersecting sets of unstructured elements (as distinguished from tuplcs having internal attribute-value structure, as in the relational algebra). The first involves the representation of sets as tries, which are intersected through a process of parallel traversal. The second approach, which permits extremely fast intersections, is closely related to our own algorithm, but uses hashing functions to approximate the process of associative retrieval on a von Neumann machine. Like our algorithm, Trabb-Pardo's hashed intersection algorithm searches for the presence of each R_1 tuple, in turn, within R_2, and intersects in time linearly proportional to the smaller argument relation.

In the case of intersection (as opposed to the more general join operator), Trabb-Pardo's "pseudo-associative" intersection algorithm in fact appears to offer comparable performance to the associative scheme described here. It is in the more general case of the natural join, where result tuples may be generated based on a partial match between the corresponding attributes of the argument relations, that the argument for a non-von Neumann architecture is strongest. Extending the use of hashed search to the case of the general natural join in the most obvious way would require that each tuple be hashed in more than one way to provide for natural joins over different compound attributes. Since the set of compound attributes on which a join might be based is equivalent to the powerset over the simple attributes, the number of such hashings is in fact exponential in the number of simple attributes. At the cost of a non-standard, but economically feasible, hardware design, the architecture and algorithms which we have described permit the straightforward and efficient generalization of the associative approach to set intersection to the more general case of the relational join.

```
procedure setDijjerence($R_1$, $R_2$);
    begin
    for each $t_1$ of $R_1$
      with not flag
      set flag and do                ($c(R_1)$ + 1 probes)
          if not exists $t_2$ in $R_2$    ($c(R_1)$ probes)
            with $t_2 = t_1$ do
                output $t_1$;
```

Algorithm 7. Internal Set Difference

6.7 Set difference

The algorithm for set difference (Algorithm 7), where R_1 is the set minuend and R_2 is the set subtrahend, is quite similar to that for intersection: As in the case of intersection, evaluation of the set difference operator requires time

$$(2c(R_1) + 1)T_p ,$$

but does not offer the freedom to choose R_1 for maximum efficiency.

7. External Evaluation

In this section, we will describe the algorithms for evaluating the relational algebraic primitives in the case where the argument relation(s) exceed the capacity of PAM. The seven relational operators may be divided into three categories according to the general manner in which they are externally evaluated. The first category includes the two unary operators select and restrict, whose external evaluafion algorithms are the least complex (both in the sense of perspicacity and efficiency) of the seven. The second category contains the single remaining unary operator, project, whose external evaluation is made more complex by the need to avoid duplicate result tuples. The final category is comprised of the four binary operators, equi-join, union, intersection and set difference, whose tuples are set into correspondence using a generalization of the category two algorithm.

The algorithms for evaluation of theoperators in the second and third categories are each based on the partitioning of the argument relation (or in the case of category three, relations) into disjoint buckets (or disjoint shared buckets, in category three). Typically, one such bucket (which, in the case of the category two operators, will in general include tuples from both argument relations) is transferred into PAM during each successive revolution of SAM, and the corresponding internal operation performed. In each case, partitioning is accomplished by associatively examining the values of some (compound) key attribute in the argument relation(s), defined as follows for each of the category two and three algorithms. In the case of projection, the key is the (compound) projected attribute of the single argument relation. For an external join, the (compound) join attribute in each of the two argument relations are defined as fhe keys. In the case of the three conventional set operators (union, intersection and sef difference), all attributes -in the argument relations are included in the key.

In this section, we will describe and analyze the algorithms for transferring successive segments of large argument relations from SAM into PAM in the case of the operators belonging to the first, second and third external evaluation categories, respectively.

7.1 Select and Restrict

Selection and restriction differ from the other relational algebraic operations in that they can be evaluated using only the per-track logic of the SAM device, and hence do nof require that successive segments of their argument relation be read into PAM for internal evaluation. Very little need be said about the external evaluation of fhe select operator, since the retrieval of all tuples of a given relation having values from selected attributes which match explicitly specified constants is in fact the central primitive operation characterizing a SAM device. As in the case of CASSM, RAP and RARES, our architecture thus performs external selection in constant time, independent of the size of the argument relation, assuming only that the argument relation is no larger than the capacity of the secondary associative storage device, and that the size of the result relation is does not exceed the bandwidth and buffering limitations of the system. Under these assumptions, a single selection requires time T_s, the time for one revolution of SAM. Indeed, our assumptions raise a number of interesting practical questions which must be considered by the designer of a practical system; these issues have been raised by other database machine researchers, however, and will not be given further attention in the current paper.

As noted in Section 4, the our specifications for the SAM device also permit the restriction operator to be performed entirely within the SAM device, since the per-track logic is itself capable of testing for equality among the attributes of a single tuple. IR contrast with the case of internal evaluation, external restriction thus has the same complexity as external selection, requiring time T_s under the assumptions specified above.

7.2 Project

As we have noted earlier in this paper, it is the problem of redundant tuple elimination which makes projection a substantial computational task in most applications. In the case where the argument relation is no larger than the capacity of PAM, redundant tuples are implicitly eliminated in the course of the internal projection algorithm. In order to extend the projection algorithm to the problem of external evaluation, however, we must first partition the large argument relation into a set of key-disjoint buckets. Buckets are defined as non-intersecting subsets of tuples from a given relation; a set of buckets is called key-disjoint if no bucket contains any tuple whose key-which in the case of projection is the value of the projected compound attribute-is the same as that of some tuple belonging to a different bucket.

In most cases, the partitioning and transfer algorithms described in Section 8 will tend to produce buckets no larger than the capacity of PAM. Given such a partitioning of the argument relation, external projection is effected by reading each bucket into PAM in succession and using the fast associative capabilities of PAM to project the tuples over the key. In the case where a bucket exceeds the capacity of PAM, the procedure is complicated somewhat, although the aggregate effect of such "PAM overflows" on the efficiency of the external evaluation algorithm will be negligible under most conditions.

To illustrate the notion of key-disjoint buckets, let us consider a projection over the second attribute of the following binary, integer-valued relation, which we will assume to be stored on SAM:

$$\begin{array}{c}\langle 2 \quad 7\rangle \\ \langle 3 \quad 1\rangle \\ \langle 4 \quad 7\rangle \\ \langle 8 \quad 7\rangle \\ \langle 9 \quad 3\rangle \\ \langle 3 \quad 2\rangle \\ \langle 4 \quad 1\rangle \\ \langle 2 \quad 3\rangle\end{array}$$

Extracting the second attribute without removing duplications yields two instances of the value 1, one of the value 2, two of the value 3 and three of the value 7. Supposing (unrealistically, of course) that PAM has a capacity of five such two-attribute tuples, we might bring all tuples having a key value of either 1 or ‘7 into PAM during a single cycle for internal projection. It is significant that the values represented in a given PAM load need not be contiguous; indeed, the values 1 and 7 are non-contiguous within the projected domain of our example. It is required only that if any tuples having the key 1 are brought into PAM on some given cycle, then—in the absence of PAM overflow-4 such tuples are in fact collected on the same cycle.

Let us now consider the modifications necessary to this algorithm in order to accomodate any instances of PAM overflows, occuring when a single bucket exceeds the capacity of PAM. The simplest (and by far the most common) case is that of a partition which exceeds the size of PAM by less than 50%, and can thus be divided into three sub-buckets A, B and C, any two of which can fit into PAM at a given time. During one SAM revolution, sub-buckets A and B are transferred into PAM and projected over the attribute in question. During the next SAM revolution, the tuples of sub-bucket B are replaced in PAM by those of sub-bucket C, and following another internal evaluation phase, those of sub-bucket A are replaced by those of sub-bucket B. In this manner, all possible pairs of sub-buckets, and hence, all possible pairs of tuples, are submitted to internal projection in PAM at some point. Generalizing this procedure to the case where x tuples are assigned to a given bucket $(x > a)$, a total of

$$\frac{n(n-1)}{2}$$

SAM revolutions are found to be required, where

$$n = \left\lfloor \frac{2x}{a} \right\rfloor \quad (a < x < \infty) \quad .$$

In the worst case (corresponding to the situation where all key values fall within a given segment, and must thus be assigned to the same partition), external projection thus has a complexity of $O(n^2)$ (albeit with very small constants). In most cases, however, the partitioning and transfer algorithms which we will consider should insure that the effects of PAM overflow are dominated by the lower-complexity terms.

It should be clear that the operation of partitioning the argument relation into key-disjoint buckets on the basis of matching values of the compound key attribute is at the heart of the process of efficient external projection. Because a similar partitioning process, based on the key attributes of both argument relations, is involved in the external evaluation of the equi-join, union, intersection and set difference operators, we have chosen to consider the details of partitioning as a separate topic in Section 8.

7.3 Join, union, intersect and set difference

As is the case for projection, external evaluation of the equi-join, union, intersection and set difference operators can not be performed efficiently within the SAM device alone. Again, it is necessary to transfer successive portions of the argument relations into PAM for internal evaluation on the basis of associatively identified characteristics of the key attributes. In the case of category three evaluation, though, each bucket is in general comprised of tuples from both of the two argument relations whose keys satisfy the current criteria for that bucket. The two argument relations are thus partitioned into what we shall call ***key-disjoint sharcd buckets,*** which may be regarded as a variant of the notion of key-disjoint buckets introduced in the previous section.

Specifically, a shared ***bucket*** is defined as a set of tuples from either or both of the argument relations& and R_2. Again, the partitioning and transfer algorithms described in Section 8 insure that the size of the great majority of such buckets (including both R_1 and R_2 tuples) will not exceed the capacity of PAM. A set of shared buckets is called key-disjoint if no bucket contains any tuple whose key is the same as that of some tuple belonging to a different bucket. It should be recalled that the key of an equi-join is the (possibly compound) join attribute, while in the case of the unstructured set operators, the key is comprised of all attributes taken together. In the latter case, the key-disjointness condition thus reduces to the requirement that no bucket contain a tuple of ***one*** argument relation which is also present within some other bucket (necessarily as part of the other argument relation.)

As an example, consider the case of an equi-join of the following two integer-valued relations R_1 and R_2 over the second attribute of R_1 and the first attribute of R_2:

R_1:

$\langle 2 \quad 7\rangle$
$\langle 3 \quad 1\rangle$
$\langle 4 \quad 7\rangle$
$\langle 8 \quad 7\rangle$
$\langle 9 \quad 3\rangle$
$\langle 3 \quad 2\rangle$
$\langle 4 \quad 1\rangle$
$\langle 2 \quad 3\rangle$

R_2:

$\langle 7 \quad 8\rangle$
$\langle 2 \quad 5\rangle$
$\langle 6 \quad 3\rangle$
$\langle 2 \quad 6\rangle$
$\langle 1 \quad 5\rangle$

Assuming again a PAM capacity of 5 binary tuples, one possible partitioning would assign to the first bucket all R_1 tuples whose second attribute has either 1 or 3 as its value and all R_2 tuples whose first attribute has either 1 or 3 as its value—specifically, the R_1 tuples $\langle 31\rangle, \langle 93\rangle$, (41) and $\langle 23\rangle$, together with the R_2 tuple (15). (It is perhaps worth mentioning at this point that the identity of the relation to which each such tuple belongs must be included as part of its representation within PAM.) The second bucket might contain all tuples having 7 as the value of the join attribute, with a final bucket for keys of value 2 or 6. Again, it should be noted that the key values included within a given bucket need not fall within a single contiguous range. Indeed, the efficiency of one of the two partitioning algorithms described in the following section is dependent on the admissability of non-contiguously defined buckets.

The procedure for recovery from PAM overflows in the course of external joining is somewhat different from that employed in external projection. The algorithm'divides both R_1 and R_2 into sub-buckets, each no larger than half the capacity of PAM; each pair of sub-buckets, one chosen from R_1 and one from R_2, is then transferred into PAM in succession. If x_1 tuples from R_1 and x_2 tuples from R_2 are assigned to the bucket in question $(x_1 + x_2 >$ a), this recovery procedure requires exactly $n_1 n_2$ SAM revolutions, where

$$n_1 = \left\lceil \frac{2x}{a(R_1)} \right\rceil$$

and

$$n_2 = \left\lceil \frac{2x}{a(R_2)} \right\rceil \quad .$$

8. Partitioning and Transfer

Since the external evaluation of each of the relational algebraic operators with the exception of selection and restriction is dependent on the partitioning of the argument relation into key-disjoint (possibly shared) buckets, we now turn our attention to the manner in which this process may be efficiently executed. We will consider **two** techniques for partitioning large argument relations into key-disjoint buckets. The two schemes, which we call the domain histogram and hashing methods, impose somewhat different requirements on the logic and memory which must be associated with each functional head, and differ slightly in efficiency. Independent of its merits as a practical algorithm for incorporation in an actual system, the domain histogram method is of interest by virtue of its relationship to previous work on associative sorting techniques. The process of domain histogram partitioning will be considered in this context in Section 8.1. When supported by the available per-track hardware, however, the hash partitioning scheme, described in Section 8.2, should generally be somewhat faster, and is also more amenable to statistical analysis.

8.1 Domain histogram partitioning

The domain histogram method is closely related to a technique introduced by Lin [1977] for sorting external files stored on an associative head-per-track disk device such as SAM. Lin's bucket sort algorithm assumes, as does our scheme, that the file can be stored entirely on the associative disk device, and thus that each data entity passes under an intelligent processing unit exactly once per revolution. The scheme functions in a manner analogous to that we have described for external evaluation of the relational algebraic operators, reading one bucket from the external file into a primary random access memory during each successive revolution of the disk. By contrast with the relational operators, however, the task of sorting requires that each partition be comprised of tuples whose sort domain-the compound attribute whose value is to determine the sorted order-contains contiguous values. Note that if the sort domain values were known a priori to be uniformly distributed over some range $[x_{\min}, x_{\max}]$, the file could be divided into $h =$ (c/a) buckets, each containing a tuples of relation R (ignoring **a few** boundary conditions), with the i-th inter-bucket boundary being

$$\frac{x_{\min} + (x_{\max} - x_{\min})i}{h}$$

Each such bucket would correspond to one contiguous range of sort domain values, so that successive buckets could be read into primary storage in a monotonic sequence of their sort key ranges, then internally sorted, and the resulting filc output in fully-sorted order.

Unfortunately, most files of practical interest deviate substantially from this assumption of uniform sort domain distribution. Lin's solution involves dividing the domain into a large number of equal-sized intervals whose size is small by comparison with P. During a single preliminary revolution of the associative disk device, a count is taken of the number of tuples of R whose sort domain values fall within the bounds of each of these smaller intervals, forming what is called a domain histogram. The lowest-valued k intervals are then combined to form the first bucket, with k chosen as large as possible such that the resulting bucket would fit within available primary storage (based on the counts of each such interval and the fixed tuple size). This first bucket is then tranferred into primary storage for internal sorting. On each successive revolution of the associative disk device, another such bucket is identified in a similar manner and read into primary storage.

As an example, consider the case of a file of IO-byte tuples whose integer-valued sort domain is bounded by the values 0 and 99. We might first divide the domain into ten equal intervals, and obtain the following counts for the number of tuples whose sort domain values fall within each interval:

$[x_l, x_u]$:	*wunt*
[0, 9]:	53
[10, 19]:	81
[20, 29]:	27
[30, 39]:	59
[40, 49]:	2
[50, 59]:	14
[60, 69]:	11
[70, 79]:	28
[80, 89]:	36
[90, 99]:	91

Assuming 2000 bytes of available storage, the first three intervals, together occupying (53 + 81 + 27) 10 = 1610 bytes, would constitute the first bucket. Thus on the first revolution (following the one required for histogram creation), all tuples whose sort domain values fell between 0 and 29 would be read into primary storage in the order they were encountered on the disk. The algorithm for internal projection would then be applied to this first bucket of tuples, and the result output. On the next revolution, all tuples in the five intervals bounded by (30, 89) would be read and processed internally; the third bucket would consist of all tuples within the bounds of the final interval.

The associative bucket sort algorithm can be applied to the problem of key-disjoint partitioning and transfer by using the key of the argument relation (in the case of projection) or relations (in the case of join, union, intersection and set difference) in the same way as the sort domain is used in the bucket sort algorithm. As it happens, though, the loosening of the contiguity constraint in favor of the weaker requirements of key-disjoint partitioning makes possible a modest refinement of this technique when applied to the relational algebraic operators. Note that if the interval bounded by [80, 89] is added to the first partition (which, unlike the interval immediately following the first partition, would not result in a PAM overflow), and the entire third partition then merged with the second (which would now have enough room), only two SAM revolutions (plus the one for histogram construction) would be required to pass the relation through PAM. The contiguity requirement thus makes it necessary to expend one extra SAM revolution by comparison with a different assignment of intervals to buckets which would be possible in the absence of this requirement. Indeed, Lin has observed that the average bucket size obtained using the bucket sort algorithm may be as small as half the capacity of the available primary store in a worst case situation, resulting in up to twice the optimal number of disk revolutions.

The task of finding an optimum partitioning of the relation from the viewpoint of minimizing the required number of SAM revolutions is an example of a bin packing problem, whose exact solution is unfortunately NP-complete. In practice, however, one of several known linear time heuristic algorithms for non-optimal, but typically reasonably effective, bin packing can be used to improve the performance of the partitioning of relations using domain histograms. (As these algorithms seem to constitute a separable and fairly well reported area of work, they will not be given further attention in this paper.)

Having identified a group of interval sets which do a reasonable job of controlling the number of buckets, all tuples whose key falls within the *set* of interval ranges which define a particular bucket must be retrieved during a single revolution of SAM. This imposes stronger requirements on the capabilities of the per-track logic than those required by the unoptimized algorithm, since more than one range specification must be checked for each tuple which passes under the head. Although there are several possible ways in which this operation might be performed, most depend on a fixed limit on the number of non-contiguous ranges used to define each bucket, thus constraining the bin packing problem in an interesting way.

Three factors are worth mentioning with regard to the choice of interval size. First, the expected amount of wasted PAM space after bin packing (manifested in a larger number of buckets, and hence, additional SAM revolutions) is directly related to the size of the intervals. Second, the likelihood that those tuples whose keys fall within a single interval will exceed the capacity of PAM (thus causing a PAM overflow regardless of the chosen partitioning) varies inversely with interval size. Note, however, that there is no interval size small enough to guarantee that no overflow will occur; the recovery procedures outlined in Section 7 are necessary to provide for the occurence of PAM overflow, however unlikely.

Finally, we note that the choice of an extremely small interval size is not without substantial cost, as each per-track logical unit would almost certainly (at least within the context of the designs we have considered) require a quantity of random access memory bearing an inverse linear relationship to interval size. To see why this is the case, note that the total number of interval count increments required during the first (histogram creation) revolution of SAM is exactly c (assuming, for simplicity, a single argument relation). The bandwidth necessary to perform all of these increments directly on one single-ported random access memory could easily be several orders of magnitude too great in a typical practical application. All of the solutions which we have considered seem to be essentially equivalent to the provision of a number of random access words within each per-track unit which is equal to the maximum number of intervals into which the domain can be divided. The individual subtotals from each per-track logical unit may then be summed to obtain the final counts for each interval. (Although the time required for this final summation is proportional to the number of SAM heads in the absence of n-argument adding hardware, this delay, which occurs only once per operator evaluation, should ordinarily be insignificant by comparison with the cost of associative retrieval.)

In attempting to rigorously evaluate the average case behavior of the domain histogram method, we are faced with the need to make fairly strong (and problematic, given our limited current understanding of the actual and potential use of such systems) assumptions about the incidence of PAM overflow. In the case of the hash partitioning method, on the other hand, a much weaker set of assumptions yields an analytically tractable model for use in computing the average case cost—which in fact turns out to be linear and small-of PAM overflows. Since the hash partitioning technique is probably superior in most applications to the method currently under discussion (at least under the assumption of suitable per-track logical capabilities), we will thus omif a detailed average case analysis of the overflow scheme as applied to domain histogram partitioning, but include such a treatment in our analysis of the hash-based scheme.

8.2 Hash partitioning

Let us now turn our attention to the hash-based scheme for partitioning and transferring the argument relation, The intent of this algorithm is to manage the problem of non-uniform distribution of the key by assigning tuples to PAM-sized buckets using a hashing function. The algorithm requires that each per-track unit be capable of sequentially computing a hashing function on the compound attribute in question, and of outputting all tuples for which the resulting hashed value falls within a specified range. Because the algorithm does not require the ability for a dynamic choice of the range of the hash function, the requirement for real-time hashing is well within the capabilities of the sort of simple and inexpensive hardware which would be required in a practical per-track logical unit. One implementation, for example, would combine the entire compound attribute into a single, fixed length "signature word" (of, say, 16 bits), by computing the exclusive or of each two-byte segment with the current accumulated signature word as it passes under the head. In the discussion which follows, we assume that the hashing function maps all keys onto a range $[0, H_{\max}]$.

In the interest of simplicity, we will first consider the case of a single relational argument. In the first step of the algorithm for category two hash-based partitioning, the range of the hash function is divided into h equal hash *intervals*, where

$$h = \left\lceil \frac{(1+W)c}{a} \right\rceil \quad .$$

W (for "waste factor") is a fixed system parameter, ordinarily much smaller than one. The number of hash intervals is thus chosen to be slightly larger than the size of the relation in "PAM-fulls", (We assume that the size in bytes of each stored relation is immediately available or easily determinable, so that this operation requires negligible time.) During each SAM revolution, all tuples whose keys hash to a value within a single hash interval are transferred into PAM, providing their combined size does not exceed the capacity of PAM.

In the absence of overflows, exactly h SAM revolutions, requiring time hT_s, are necessary to transfer all buckets of the argument relation(s) into PAM. Whenever x, the number of tuples assigned to the current bucket, is greater than (c/h) by a factor of more than W, however, an overflow occurs, resulting in the expenditure of more than one SAM revolution for the bucket in question; the exact number of revolutions depends on the ratio of x to (c/h). In the general case where an average of v extra "'overflow revolutions" are required per bucket, the time required is exactly

$$(1+v)hT_s \quad .$$

The central concern of our analysis is the derivation of an upper bound on the average case value of v.

By comparison with the domain histogram algorithm, the randomizing property of the hashing scheme permits a relatively accurate statistical evaluation of the number and extent of PAM overflows to be expected in the course of hash partitioning without excessively stringent assumptions regarding the distribution of the key values. (Our analysis is dependent, of course, on the assumption that the distribution of hash values, given a large set of keys, will be close to uniform over the range $[1, H_{\max}]$; this may not in fact always be the case.) The analysis is based on the treatment of the partitioning process as a set of c independent Bernoulli trials, one for each tuple in the relation, with each trial defined as successful if the tuple in question falls within the current hash interval, and as unsuccessful otherwise. The number of tuples which will be assigned to any given bucket is thus a binomially distributed random variable whose probability of being equal to some particular value k is exactly

$$\binom{c}{k}\left(\frac{1}{h}\right)^k\left(1-\frac{1}{h}\right)^{(c-k)} .$$

Unless there is-a very small number of tuples per PAM load, this function is well approximated by the Gaussian distribution

$$\phi\left(\frac{x-\eta}{\sigma}\right)=\frac{1}{\sqrt{2\pi}\,\sigma}e^{-\frac{(x-\eta)^2}{\sigma^2}}$$

having mean

$$\eta=\frac{c}{h}$$

and variance

$$\sigma^2=\left(1-\frac{1}{h}\right)\eta .$$

Furthermore, both η and σ^2 approach

$$\frac{a}{1+W}$$

as c grows large, and are thus asymptotically independent of the size of the argumen t relations.

Note that this approximation differs from that most commonly employed in analyzing hash coding behavior in database management applications (see Wiederhold [1977], for example). In the more common use of hashing, the expected value of x is typically quite small, so that the corresponding function is better approximated by a Poisson distribution. When the η is reasonably large, however, a normal distribution provides a better approximation. As a practical rule of thumb, the Gaussian approximation, which is justified in the limit by the **DeMoivre-Laplace** theorem, is very good whenever the quantity

$$\left(\frac{1}{P}-\frac{1}{S}\right)t$$

is less than about 0.1, which should be true in most conceivable practical cases.

The expected number of overflow revolutions may thus be estimated by

$$v=\sum_{i=2}^{\infty}\frac{i(i+1)}{2}\int_{ia/2}^{(i+1)a/2}\phi\left(\frac{x-\eta}{\sigma}\right)dx \quad .$$

For purposes of obtaining a simple upper bound, the discrete summation may be eliminated by substituting $2x/a$ for i within each term, so that a constant expression equal to--the lower limit of integration is replaced by the variable of integration within that range, which must necessarily be larger. This yields

$$v<\int_{a}^{\infty}\frac{x}{a}\left(\frac{x}{a}+\frac{1}{2}\right)\phi\left(\frac{x-\eta}{\sigma}\right)dx$$

$$=\left(\frac{1}{2a}\right)^{2}\left\{\left(2\sigma^{2}+\eta(2\eta+a)\right)\left(1-\Phi\left(\frac{a-\eta}{\sqrt{2}\,\sigma}\right)\right)+(2\eta+3a)\phi\left(\frac{a-\eta}{\sqrt{2}\,\sigma}\right)\right\} ,$$

where

$$\Phi(x)=\int_{-\infty}^{x}\phi(y)\,dy \quad ,$$

which has no closed form solution, but whose values for specific x are available in tabular form.

v is thus independent of the size of the argument relations, and since h varies linearly with argument size, the time

$$(1+v)hT_{s}$$

for partitioning and transfer is of linear complexity in the size of the argument relations. (Since the algorithm for internal projection is also linear, the corresponding external algorithms are linear.) The time required is, however, inversely related to W, the waste factor, and directly related to *a,* the capacity in tuples of PAM.

Calculations using a range of typical c, t, P and h values suggest that a very modest W (say, on the order of 0.1) should generally suffice to make the cost of overflow recovery negligible by comparison with the complexity component due to the transfer of non-overflowing buckets.

The algorithm for hash-based external evaluation of the join, union, intersect and set difference operators is analogous to the one described for external projection. In the case of the category two operators, the number of hash intervals, h, is set equal to

$$h = \left\lceil (1+W)\left(\frac{c(R_1)}{a(R_1)} + \frac{c(R_2)}{a(R_2)}\right)\right\rceil .$$

Analysis of the average case time complexity of the category two operators is similar to that presented above for projection, the primary differences being due to substitution of $n_1 n_2$ for $n(n-1)/2$ as the number of SAM revolutions required for recovery from PAM overflow. As in the case of projection, such overflows make only a linear contribution to the cost of category two evaluation.

In practice,. the time required for evaluation of both the category one and category two operators should ordinarily be quite close to the sum of

1. the time required for a number of SAM revolutions equal to the size of the argument relation (or in the case of category two, the combined size of the two argument relations) in "PAM-fulls", and

2. the time required for internal evaluation of the operator in question.

In the case where the argument relation(s) are large, this may represent a very substantial improvement on the results attainable using a database machine based on an associative secondary storage device alone, as in the RAP, CASSM and RARES designs.

9. summary

In this paper, we have proposed a non-von Neumann machine architecture for the efficient large-scale evaluation of relational algebraic database primitives. The design is based on a content-addressable primary storage unit called PAM and a rotating logic-per-track associative device called SAM, both based on existing, and in the near future, economically feasible, technology. The machine we have described functions in much the same way as several proposed and already-implemented database machines for the operations of selection and restriction, but appears to offer a significant performance advantage in the case of project, join, and the unstructured set operators.

Specifically, the time required for external selection and restriction is independent of the size of the argument relation, being equal to the time for one revolution of SAM under the assumptions enumerated in the paper. This result substantially improves upon the best known general algorithms for evaluating these operations on a von Neumann machine, but is essentially equivalent to those obtained on most of the database machines reviewed in Section 3.2. The time required for external projection, join, union, intersection and set difference, on the other hand, is roughly that required for a number of SAM revolutions equal to the combined size of the argument relations in 'PAM-fulls" plus the (also linear) time required for internal evaluation of the operator in question. This latter result represents an $O(\log n)$ improvement over the best presently known methods on a von Neumann machine, and appears to offer a large linear factor improvement (roughly proportional to the capacity of PAM) over the best reported results involving a specialized database machine architecture having comparable hardware complexity.

It must be acknowledged, however, that we have left many details unspecified, have made a number of assumptions which ought to be carefully examined, and have not yet performed the sorts of detailed comparisons that would justify a confident claim that the architecture we have described is in fact more suitable for practical application than those already proposed in the literature. It is hoped that the readers of this paper will contribute to the process of critical review necessary to adequately assess the merit of the approach we have suggested.

References

Ampex Corporation, "9300 Parallel Transfer Disk Drive", product announcement, Redwood City, 1978.

Anderson, Donald R., "Data base processor technology", Proceedingsof *theNational* Computer Conference, 1976.

Anderson, G. A. and Kain, R. Y., "A content-addressed memory design for data base applications" , Proceedings of *the* 1976 **International** Conference *on* Parallel ***Processing,*** IEEE, New York, pp. 191-195, 1976.

Banerjee, Jayanta, Baum, Richard I., **Hsiao,** David K. and Kannan, Krishnamurthi, "Concepts and capabilitiesof a databasecomputer", to appear in ACM Transactions ***on*** Database ***Systems,*** 1979.

Batcher, K. E., **"STARAN** parallel processor system hardware", ***Proceedings of the AFIPS*** 1974 National Computer ***Conference,*** vol. 43, AFIPS Press, Montvale, New Jersey, pp. 405-410, 1974.

Baum, Richard I. and Hsiao, David K., "Data base computers-a step towards data utilities" , *IEEE* Transactions on Computers, vol. C-25, December, 1976.

Berra, P. Bruce, "Some problems in associative processor applications to **data** base management", ***Proceedings of the*** National ***Computer*** Conference, 1974.

Berra, P. Bruce, 'Data Base Machines", ACM ***SIGIR Forum,*** Winter, 1977.

Canady, et al,, "A back-end computer for database management", Communications ***of the*** Association ***for*** Computing Machinery, vol. 17, pp. 575-582, October, 1974.

Chu, Y. H., "A destructive-readout associative memory", ***IEEE*** Transactions on ***Computers,*** EC-14, pp. 600-605, August, 1965.

Codd, E. F., "A relational **modelof** data for largeshared data banks", Communications ***of the*** ACM, vol. 13, no. 6, pp. 377-387, June, 1970.

Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus", ***Proceedings of the*** 1971 **ACM *SIGFIDET*** Workshop on Data ***Description,*** Access and ***Control,*** Association for Computing Machinery, 1971.

Codd, E. F., "'Relational completeness of data base sublanguages", in **Rustin,** Randall (ed.), Courant ***Computer Science Symposium 6:*** Data Base ***Systems,*** Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1972.

Copeland, G. P., Lipovski, G. J. and Su, S. Y. W., "The architecture of CASSM: A cellular system for nonnumeric processing", *Proceedings of* the First *Annual Symposium on Computer Architecture,* 1973.

Couranz, G. R., Gerhardt, M. S., and Young, C. J. "Programmable radar signal processing using the RAP", Proceedings *of -the* Sagamore *Computer Conference on Parallel Processing,* Springer-Verlag, New York, pp. 37-52, 1974.

Crane, B. A. and Githens, J. A., "Bulk processing in distributed logic memory", *IEEE* Transactions *on Computers,* EC-14, pp. 186-196, April, 1965.

Crofut, W. A. and Sottile, M. R., "Design techniques of a delay-line content-addressed memory", *IEEE* Transactions on Computers, pp. 529-534, 1966.

DeFiore, Casper R. and Berra, P. Bruce, "A data management system utilizing an associative memory", *Proceedings of the AFIPS National Computer Conference,* vol. 42, 1973.

DeFiore, Casper R., Stillman, C. R., and Berra, P. Bruce, "Associative techniques in the solution of data management problems", *Proceedings of the* ACM, pp. 28-36, 1971.

Dewitt, David J., "DIRECT-A multiprocessor organization for supporting relational database management systems", *IEEE Transactions on Computers,* vol. c-28, no. 6, June, 1979.

Ewing, R. G. and Davies, P. M., "An associative processor", *Proceedings of the AFIPS 1964* Fall *Joint Computer Conference,* Spartan Books, Inc., Baltimore, Maryland, pp. 147-158, 1964.

Finnila, C. A., "The associative linear array processor", *IEEE Transactions on -Computers,* February, 1977.

Flynn, M. J., "Some computer organizations and their effectiveness", IEEE Transactions *on Computers,* pp. 948-960, September, 1972.

Foster, Caxton *C., Content* Addressable Parallel *Processors, New* York, Van Nostrand Reinhold, 1976.

Gains, R. S., and Lee, C. Y., "An improved cell memory", *IEEE* Transactions *on Computers,* pp. 72-75, February, 1965.

Healy, L. D., Lipovski, G. J. and Doty, K. L., "The architecture of a context addressed segment-sequential storage" *Proceedings of the AFIPS 1972 Fall Joint Computer Conference,* AFIPS Press, Montvale, New Jersey, pp. 691-701, 1972.

Higbie, L. C., "The OMEN computers: Associative array processors", *IEEE*

COMPCON, pp. 287-290, 1972.

Higbie, L. C., "Supercomputer architecture", *Computer,* pp. 48-58, December, 1973.

Hsiao, David K., "The architectureof a database computer-A summary", Proceedings *of the Third Workshop on Non-Numeric* Processing, May, 1977.

Hsiao, David K., and Kannan, Krishnamurthi, "The architecture of a database computer-part II: The design of the structure memory and its related processors", Technical Report OSU-CISRC-TR-'76-2, Ohio State University, October, 1976.

Hsiao, David K., and Kannan, Krishnamurthi, "The architecture of a database computer-part III: The design of the mass memory and its related components", Technical Report OSU-CISRC-TR-76-3, Ohio State University, December, 1976a.

Hsiao, David K., Kannan, Krishnamurthi, and Kerr, D. S., "Structure memory designs for a data base computer", Proceedings of *the* ACM Annual Conference, pp. 343-350, 1977.

Hsiao, David K. and Madnick, Stuart E., 'Data Base Machine Architecture in the Context of Information Technology Evolution", Proceedings *of the Third International Conference on Very* Large Data Bases, October, 1977.

Kaplan, A., "A search memory subsystem for a general-purpose computer", *Proceedings of the AFIPS 1963 Fall Joint Computer Conference,* vol. 24, Spartan Books, Inc., Baltimore, Maryland, pp. 193-200, 1963.

Kerschberg, L., Ozkarahan, Esen A., and Pacheco, J. E. S., "A synthetic English query language for a relational associative processor", Proceedings of the Second *International Conference on Software Engineering,* October, 1976.

Lang, T., Nahouraii, E., Kasuga, K. and Fernandez, E. B., "An architectural extension for a large database system incorporating a processor for disk search", Proceedings of the Third International Conference *on Very* Large Data *Bases,* -Tokyo, Japan, October, 1977.

Lee, C. Y., "Intercommunicating cells as a basis for a distributed logic computer", *Proceedings of the AFIPS 1962* Fall *Joint Computer Conference,* Spartan Books Inc., Baltimore, Maryland, pp. 130-136, 1962.

Lee, C. Y., and Paull, M. C., "A content adressable distributed logic memory with applications to information retrieval", *Proceedings of the IEEE,* pp. 924-932, June, 1963.

Lin, Chyuan Shiun, "Sorting with associative secondary storage devices", *Proceedings* of the National Computer Conference, pp. 691-695, 1977.

Lin, Chyuan Shiun, and Smith, Diane C. P., "The design of a rotating associative array memory for a relational data base management application", Proceedings *of* the International Conference *on* Very Large Data Bases, vol. 1, no. 1, September, 1975.

Lin, Chyuan Shiun, Smith, Diane C. P., and Smith, John Miles, "The design of a rotating associative memory for relational database applications", *ACM* Transactions on Database *Systems* vol. 1, no. 1, pp. 53-65, March **1976.** (Revised version of Lin and Smith **[1975],** cited above).

Linde, Richard R., Gates, Roy and Peng, Te-Fu, "Associative processor applications to real-time data management", Proceedings of the National Computer Conference, 1973.

Lipovski, G. J., "The architecture of a large distributed logic associative memory", National Technical Information Service, AD 692195, July, 1969.

Lipovski, **G. J.,** "The architecture of a large associative processor", ***Proceedings of the AFIPS Spring Joint Computer Conference,*** pp. 385-396, 1970.

'Lipovski, G. J., "Architectural features of CASSM: A context addressed segment sequential memory", Proceedings of ***the*** Fifth Annual Symposium ***on Computer Architecture,*** Palo Alto, California, pp. 31-38, April, 1978.

Lowenthal, Eugene I., "A survey-The application of data base management computers in distributed systems", ***Proceedings of the Third*** International ***Conference on Very*** Large Data Bases, Tokyo, Japan, October, 1977.

McGregor, D., Thompson, R. and Dawson, W., "High performance hardware for 'database systems", in ***Systems for*** Large Data Bases, **Lockmann** and **Newhold,** eds., Amsterdam, The Netherlands, North Holland, pp. 103-116, 1976.

Minsky, N., "Rotating storage devices as partially associative memories", ***Proceedings of the AFIPS 1972 Fall Joint Computer Conference,*** AFIPS Press. Montvale, New Jersey, pp. 587-595, 1972.

Moulder, Richard, "An implementation of a data management system on an **as-socia** t ive processor" , ***Proceedings of the National Computer Conference,*** 1973.

Murtha, J. C., and Beadles, R. L., "Survey of the highly parallel information processing systems", Office of Naval Research Report No. 4755, November, 1964.

Nakano, R., **"A** simulator for **a RAP virtual** memory system", M.S. thesis, University of Toronto, **1976.**

Ozkarahan, Esen A., “An associative processor for relational data bases-RAP”, Ph.D. Dissertation, University of Toronto, 1976.

Ozkarahan, Esen A., and Schuster, Stewart A., “A high level machine-oriented assembler language for a data base machine”, Technical Report CSRG-74, Computer Systems Research Group, University of Toronto, October, 1976.

Ozkarahan, Esen A., Schuster, Stewart A., and Sevcik, K. C., ‘Performance evaluation of a relational associative processor”, ACM TODS, vol. 2, pp. 175-195, June, 1977.

Ozkarahan, Esen A., Schuster, Stewart A., and Smith, Kenneth C., “A data base processor”, Technical Report CSRG-43, Computer Systems Research Group, University of Toronto, Sept. 1974.

Ozkarahan, Esen A,, Schuster, Stewart A., and Smith, Kenneth C., “RAP--An associative processor for data base management”, Proceedings ***of the AFIPS*** National ***Computer Conference,*** vol. 44, pp. **379-387,** 1975.

Ozkarahan, Esen A., and Sevcik, K. C., “Analysis of architectural features for enhancing the performance of a data base machine”, ACM TODS, vol. 2, pp. 297-316, December, 1977.

Parhami, B., “A highly parallel computing system for information retrieval”, ***Proceedings of the AFIPS 1972 Fall Joint Computer*** Conference, AFIPS Press, Montvale, New Jersey, pp. 681-690, 1972.

Parker, J. L., “A logic per track retrieval system”, Proceedings of the ***IFIP 1971 Congress,*** vol. 1, North-Holland Publishing Co., Amsterdam, The Netherlands, pp. 711-716, 1971.

Rudolph, J. A., “A production implementation of an associative array processor: STARAN", ***Proceedings of the AFIPS 1972*** Fall ***Joint Computer Conference, vol.*** 41, pt. 1, AFIPS Press, Montvale, New Jersey, pp. 229241, 1972.

Rux, P. T., “A glass delay line content-addressable memory”, *IEEE* Transactions ***‘on Computers,*** pp. 512-520, 1969.

Schuster, Stewart A,, Nguyen, H. B., Ozkarahan, Esen A., and Smith, Kenneth C., “RAP.2-An associative architecture for data bases and its applications”, ***IEEE Transactions on Computers,*** vol. c-28, no, 6, June **1979.** (Revised version of “RAP.2-an Associative Processor for Data Bases”, ***Proceedings of the Fifth Computer*** Architecture ***Symposium,*** May, 1978.)

Schuster, Stewart A., Ozkarahan, Esen A., and Smith, Kenneth C., “A virtual memory system for a relational associative processor”, ***Proceedings of the AFIPS***

National *Computer Conference,* vol. 45, pp. 855-862, 1976.

Shooman, W., "Parallel computing with vertical data", Proceedings of *the 1960* Eastern Joint Computer Conference, New York, pp. 393-400, 1960.

Slotnick, D. L., 'Logic per track devices", Advances *in Computers, vol.* 10, Academic Press, New York, pp. 291-296, 1970.

Su, Stanley Y. U., Chen, W. F. and Emam, Ahmed, "CASAL: CASSM's assembly language", Technical Report 7778-7, Computer and Information Sciences Department, University of Florida, March, 1978.

Su, Stanley Y. U., Copeland, George P., and Lipovski, G. J., 'Retrieval operations and data representations in a content-addressed disc system", Proceedings of *the* International *Conference on* Very *Large Data* Bases, Framingham, Massachusetts, September, 1975.

Su, Stanley Y. U., and Emam, Ahmed, "CASDAL: CASSM's data language", *ACM* Transactions on Database *Systems,* vol. 3, no. 1., pp. 57-91, March, 1978.

Thurber, Kenneth J. and Wahl, Leon D., "Associative and parallel processors", *Computing* Surveys, vol. 7, No. 4, December, 1975.

Trabb-Pardo, Luis, *Set* Representation and *Set* Intersection, Ph.D. thesis, Report STAN-CS-78-681, Computer Science Department, Stanford University, December, 1978,

Wiederhold, Gio, Database Design, McGraw-Hill, pp. 292-294, 1977.

Yau, S. S., and Fung, H. S., "Associative processor architecture-a survey", *Computing Surveys,* vol. 9, no. 1, March, 1977,

Young, F. H., "Circulating associative memories", Department of Mathematics Report, Oregon State University, 1962.