A Relational Database Machine Architecture

David Elliot Shaw

Computer Science Department
Stanford University

October 1979

Abstract

Algorithms are described and analyzed for the efficient evaluation of the *project* and *join* 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.

Accepted for publication in the *Proceedings of the 1980 Workshop on Computer Architecture for Non-Numeric Processing*, March 11-14, 1980, and for distribution by SIGARCH, SIGIR and SIGMOD.

1. Introduction

The past several years has seen a dramatic growth of interest in specialized machine architectures oriented toward the problems of data base management. During the same period, the relational model of data [Codd, 1971] has gained considerable attention as a powerful framework, from both a practical and theoretical perspective, for the design of database management systems. It is thus not surprising that several hardware approaches to the efficient evaluation of the primitive operations of a relational algebra have already been suggested in the literature.

This paper proposes an approach to the large-scale parallel execution of relational algebraic primitives on a specialized non-von Neumann machine. The design is based on a content-addressable primary storage unit called PAM and a rotating logic-per-track associative device called SAM, each of which might be realized in several ways using existing, and in the near future, potentially competitive technologies. We have developed and analyzed algorithms for the very rapid evaluation of most of the commonly used relational algebraic operators—specifically, *project*, *equi-join*, *select*, *restrict*, *union*, *intersect* and *set difference*—on the proposed machine [Shaw, 1979]. Relational selection and restriction are performed in much the same way on our proposed system as on several proposed and already-implemented database machines; our design, however, appears to offer substantial performance advantages for the evaluation of equi-join, project and the unstructured set operators under many circumstances of significant practical import. We will limit our attention in the current paper to the algorithms for projection and equi-join, two of the most important, and in general, computationally expensive of the relational algebraic operators.

The paper will briefly outline the organization of the architecture we are proposing, and will describe and analyze algorithms for evaluation of equi-join and project, both in the case where the argument relations fit entirely within primary storage (which we call *internal evaluation*), and where they reside on secondary storage (*external evaluation*). The section which follows introduces those elements of a relational algebra essential to our discussion. Relevant previous work in the area of associative processors and database machines is briefly noted in Section 3. The fourth Section outlines our proposed architecture in general terms. Following the introduction of notation in Section 5, the algorithms for internal evaluation are presented, along with an average-case analysis of their time complexity. Section 7 describes the corresponding external evaluation procedures, considering in some detail the complexity of the most time-critical aspects of the external operations, which involve partitioning the argument relations into appropriate segments for transfer into primary storage. Our results are summarized in Section 8.

2. The relational algebraic primitives

We may define a *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. Since relations are sets, we may refer to the number of tuples in a relation as its *cardinality*. 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. In some discussions, it is useful to consider several attributes (some possibly repeated) as a single *compound attribute*.

In this paper, a list of primitive domain elements enclosed by angle brackets ("⟨" and "⟩") 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 $\langle r[j_1], r[j_2], ..., r[j_m] \rangle$. Finally, 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

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

The *projection* of a relation R over the compound attribute A may be defined as the set

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

The projection operator may be thought of as defining 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 in a simple and computationally inexpensive manner. The elimination of redundant tuples, on the other hand, may be surprisingly time-consuming, particularly when the argument relation is large. One of the goals of the architecture discussed in this paper is the minimization of the high cost of redundant tuple elimination.

The equi-join of two relations R_1 and R_2 over the compound attributes A_1 and A_2, respectively (each assumed 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*. The join operation may be intuitively thought of as a process of filtering the *extended cartesian product* of R_1 and R_2—the set of all possible concatenations of one tuple from R_1 with one from 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.)

The join operation may in general be 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.

3. Associative processors and database machines

Most proposals for specialized database machines have involved the use of *associative processing* hardware. 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. The greatest degree of distribution is found in the *fully parallel* associative processors, including word organized and distributed logic designs, in which comparison logic is associated with every bit of storage. A somewhat less extensive distribution of intelligence is found in the *bit-serial* associative processors, in which one "bit slice" extending through all words is accessible for parallel processing at any given time. Another family of less-than-fully-parallel content-addressable devices is populated by the *word-serial* associative processors, in which all bits of a *single* word are compared in parallel, but the *set* of words is examined sequentially.

At the low end of the associative logic distribution spectrum is the class of *block-oriented* associative devices, in which search and modification logic unit is associated with each head of a head-per-track rotating storage device (or its non-inertial equivalent), permitting one associative operation to be performed on each revolution. The reader is referred to a survey by Yau and Fung [1977], or to an outstanding book by Foster [1976], for a more complete taxonomy and discussion of associative processing devices.

Although small scale associative processing devices characterized by comparatively extensive logic distribution were employed in some early experimental investigations of the potential applicability of associative processors to database management applications (see DeFiore and Berra [1973], for example), the large amount of data involved in most such applications has given the class of block-oriented associative devices a prominent place in most database machine designs. Moulder [1973] has described an architecture for hierarchical database applications which uses a head-per-track disk drive in combination with the STARAN machine [Rudolph, 1972], which supports both bit-slice (for associative processing) and ordinary word slice (for input and output) access capabilities. 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. The CASSM system [Su, Copeland and Lipovski, 1973] is a block-oriented design providing a direct hardware implementation of hierarchical data structures, but supporting the relational and network data models as well. CASSM includes special features for searching complex data structures such as sets, ordered sets, trees, variable length character strings and directed graphs.

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]. Like CASSM, RAP is based on a block-oriented associative processor, and is intended to serve as a backend database processor for a general purpose computer. Advantages in speed ranging between one and three orders of magnitude over conventional systems have been obtained for many relational database operations,

although certain operations of considerable practical importance, such as the equi-join of two large relations, are still quite computationally expensive [Ozkarahan, Schuster 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]. RARES differs from the design of RAP primarily in its 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, offering the possibility of a decrease in the incidence of output contention and a reduction in buffer storage requirements.

At Ohio State University, a very-large-scale architecture called DBC (for Database Computer) has been proposed which is based on the use of a number of interconnected subsystems specialized for various aspects of database management [Baum and Hsiao, 1976]. One of the components is a *mass memory*, based on a number of moving-head disk drives, each modified to provide for simultaneous associative operations on all tracks in a given cylinder. Information which is used to locate the relevant cylinders to search is stored in another block-oriented associative unit called the *structure memory*. The design of DBC was strongly influenced by several kinds of data protection concerns, and includes specialized mechanisms for the imposition of related constraints. Another organization, called DIRECT [DeWitt, 1979], is oriented toward the problem of achieving intra- and inter-query concurrency and database integrity in a multiple-process relational database environment.

4. The proposed architecture

Our proposed architecture is configured as a hierarchy of associative storage devices under the control of a general purpose processor. 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 one of several bit-serial or word-serial designs. 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 *flag 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.

The *secondary associative memory* (SAM) is a block-oriented associative device, considerably larger, but slower, than PAM. (A capacity of between 1 and 100M bytes and an associative operation time of between 1 and 100 milliseconds should adequately exemplify our design.) As in the CASSM, RAP and RARES designs, SAM might be realized using a parallel head-per-track disk or a non-inertial circulating storage device constructed using CCD or bubble memory technology; in either case, a modest amount of logic is assumed associated with each storage loop. While the analysis presented in this paper assumes a fixed time for an associative probe of the entire contents of SAM, the algorithms themselves are also applicable to the kind of modified moving-head disk devices employed in the DBC design, thus supporting very large data base applications.

The following capabilities are assumed for the "per-track" logic associated with each head (or its functional equivalent) of the SAM device. First, it must be possible to examine an arbitrary compound attribute in each tuple which "passes under" the associated head. As in the case of most of the database machines which we have discussed, our proposed secondary associative device is able to collect all such matching tuples for output; the SAM device, however, is also assumed 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 described in Section 7 does not require the ability for a dynamic choice of the *range* of the hash function, this 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 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 project and join operators:

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.

While we have chosen to treat the size of the result relation as an independent variable in our analysis, it should be noted that the value of r is in fact determined by the composition of the argument relations.

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 three 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*.

```
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

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 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. The utility of the proposed architecture for the evaluation of the relational project operator 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

```
procedure join(R1, R2, A1, A2);
      for each t1 of R1
        with not flag
        set flag and do                          (d(A1,R1) + 1 probes)
            begin                                (d(A1,R1) times)
            distribute(t1, R2, A1, A2);
            for each t1' of R1
              with t1'[A1] = t1[A1]
                and not flag
              set flag and do                    (c(R1) probes)
                  distribute(t1', R2, A1, A2);   (c(R1) - d(A1,R1) times)
            end;

procedure distribute(t1, R2, A1, A2);
      begin                                      (c(R1) times)
      for each t2 of R2
        with t2[A2] = t1[A1]
          and not flag
        set flag and do                          (r + c(R1) probes)
            output (t1[A1] | t2[A2]);
      parallel clear flag
        in all t2 of R2
        with t2[A2] = t1[A1];                    (c(R1) probes)
      end;
```

Algorithm 2. Internal Join

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.

Algorithm 2 computes the equi-join of of relations R_1 and R_2 over the compound attributes A_1 and A_2, respectively.

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 this algorithm sets the tuples of the two relations 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 the following section, linear complexity is preserved in the external algorithm for equi-join as well. It is also worth noting 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.

Lest these results be misinterpreted, it should be emphasized the *worst case* behavior of our join 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.

7. External evaluation

In this section, we will describe the algorithms for evaluating the project and join operators in the case where the argument relations exceed the capacity of PAM. Both algorithms involve the *partitioning* of the argument relation or relations into *key-disjoint buckets*. Typically, one such bucket (which, in the case of the join operator, will in general include tuples from both argument relations) is transferred into PAM during each successive revolution of SAM, and an internal projection or join is performed. In each case, partitioning is accomplished by associatively examining the values of some compound attribute of the argument relations(s)—in the first case, the compound projected attribute, and in the second, the compound join attribute.

In the case of projection, the buckets are non-intersecting subsets of tuples chosen from the single argument relation. A set of buckets will be called *key-disjoint* under projection if no bucket contains any tuple whose compound projected attribute is the same as that of some tuple belonging to a different bucket. In most cases, the partitioning and transfer procedure described below will tend to produce buckets no larger than the capacity of PAM; the external evaluation algorithm is, however, designed to accomodate the case of *PAM overflow*, in which a bucket exceeds the capacity of PAM, without significant degradation of overall performance.

In the absence of PAM overflow, 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. It is assumed that the relative speeds of PAM and SAM are such that each bucket may be transferred in the course of a single SAM revolution. It is instructive to observe that, except in the case of PAM overflow, each tuple of the argument relation is processed only once in primary storage, in contrast with the best currently known general techniques for the external projection, which are based on external sorting. The algorithm for external evaluation is complicated somewhat by the case of PAM overflow; as we shall see, however, the aggregate effect of such overflows on the efficiency of the external evaluation algorithms can be shown to be statistically negligible under most practical circumstances.

To illustrate the notion of key-disjoint partitioning and transfer, 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:

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

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 projected attribute value of either 1 or 7 into PAM during a single cycle. 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 *all* such tuples are in fact collected on the same cycle (disregarding for the moment the case of bucket overflow.)

Let us now consider the mechanism by which the argument relation is partitioned for transfer from SAM to PAM in the course of external projection. Note that if the values of the compound projected attribute were uniformly distributed, the argument relation could simply be partitioned into buckets representing contiguous, equal-sized ranges of values, each just large enough to fill PAM with

argument tuples. Our partitioning algorithm accomodates the problem of non-uniform distribution of the compound projected attribute values by assigning tuples to PAM-sized buckets using a hashing function computed associativcly by the per-track hardware. In the discussion which follows, we assume that all keys are mapped onto a range $[0, H_{max}]$. In the first step of the algorithm for projective 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.

Let us now consider the case of PAM overflow, which occurs when a single disjoint bucket exceeds the capacity of PAM. The simplest (and by far the most common) case is that of a bucket 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 the internal evaluation in question is performed. 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 evaluation 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

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

In the worst case (corresponding to the situation where *all* projected attribute values fall within a given segment, and must thus be assigned to the same bucket), the partitioning and transfer process has a complexity of $O(n^2)$ (albeit with small

constants). Assuming that the distribution of hash values is reasonably close to uniform over the range $[1, H_{max}]$, however, it will be shown that PAM overflow makes only a linear contribution to the average case complexity of the partioning and transfer algorithm on our proposed machine.

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.

To this end, it is convenient to view the collection of a particular bucket full of tuples 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

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

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

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

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 argument relations.

Note that this approximation differs from the one most commonly employed in analyzing hash coding behavior in database management applications (see Wiederhold [1977], for example). In the more common analysis 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 for our application 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

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

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

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 external join is analogous to that for external projection. In the case of the join operator, each bucket is in general composed of tuples from *both* of the two argument relations whose hashed compound join attribute values fall within the range corresponding to that bucket. A set of join buckets is called key-disjoint if no bucket contains any tuple whose compound join attribute is the same as that of some tuple belonging to a different bucket. In the case of join partitioning, the number of *hash intervals*, h, is set equal to

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

As in the case of projection, external evaluation of the join operator involves the transfer of buckets into PAM for internal evaluation, and in contrast to sorting-based methods, requires that each argument tuple be processed only once in PAM, except in the (infrequent) event of PAM overflow.

The procedure for *recovery* from PAM overflows, on the other hand, 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

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

Proof of the linear contribution of overflow recovery to the external join algorithm is similar to that presented above for projection.

In practice, the time required for evaluation of both join and project 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.

8. Summary

The architecture and algorithms which we have described are designed to support the efficient execution of relational algebraic operations in the context of applications in which the argument relations may be quite large. In particular, the time required for the projection, and equi-join operators 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. This represents an $O(\log n)$ improvement over the best presently known general methods on a von Neumann machine. In the case where the argument relations are both large, the proposed architecture should also permit a significant improvement (by a factor roughly proportional to the capacity of PAM) over the performance of previously implemented or proposed database machine architectures based on associative secondary storage devices.

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 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

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

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*, 1971.

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

DeWitt, David J., "DIRECT—A multiprocessor organization for supporting relational database management systems", *IEEE Trans. Computers*, v. c-28, no. 6, June, 1979.

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

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

Moulder, Richard, "An implementation of a data management system on an associative processor", *Proc. AFIPS National Computer Conference*, 1973.

Ozkarahan, Esen A., Schuster, Stewart A., and Sevcik, K. C., "Performance evaluation of a relational associative processor", *ACM Transactions on Database Systems*, v. 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, Univ. Toronto, Sept. 1974.

Rudolph, J. A., "A production implementation of an associative array processor: STARAN", *Proc. AFIPS Fall Joint Computer Conference*, v. 41, pt. 1, AFIPS Press, Montvale, NJ, pp. 229–241, 1972.

Shaw, David Elliot, "A hierarchical associative architecture for the parallel evaluation of relational algebraic database primitives", to appear as Computer Science Department Report, Stanford University, 1979.

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

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

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