Automatic representation selection for associative data structures

by PAUL ROVNER

University of Rochester
Rochester, New York

INTRODUCTION

Motivation

One of the persistent hopes for rapid progress in Computer Science is that standard solutions to software design problems will be developed, analyzed, and used. In particular, the hope for high-level languages has always been that enough low-level detail could be left to "the system" to allow a mere human to develop solutions for enormously complex problems. An ideal high-level language would allow a programmer to concentrate on the conceptual development of a solution for his problem, rather than on the design of a representation for his solution once he can express it.

Although existing high-level languages insulate a user from many details, they still require him to do low-level design for many problems, especially for complex ones. Such cases arise either because the language provides only low-level storage structures (records, arrays, pointers, lists) with which the programmer must encode his algorithm, or because it provides a general-purpose data structure for which an arbitrary (and usually sub-optimal) representation design choice has been made a priori (LISP, SAIL). Any single representation choice for a general data structure is very badly matched to some problems; the user of a general data structure must often spend much time designing a way to avoid unacceptable inefficiencies. Furthermore, good storage structure design requires much clerical work, for which a person whose goal is to solve a problem will usually have little time or enthusiasm. An automatic system, on the other hand, is well suited to the job of systematically comparing the cost and applicability of the many representations that are available for program data. In addition, an automatic system is less likely than a person to overlook possible representation candidates.

The problem

This paper explores one aspect of the representation selection problem: storage structure selection for associative data structures. By "associative data structure," we mean a collection of associations between abstract items. We express associations as ordered lists ("tuples") of items. Thus, an association between n given items is (conceptually) an n element list of them, called an "n-tuple". The role of each item in an association is identified by its position in the n-tuple.

Before going into specific detail about representation selection, we present a simple example which will be used to illustrate technical details as they arise. The example data base is a collection of relations among family members, their genders, and their places of residence. For the most part, we will be concerned with three-part associations (triples) such as:

Sexof·Abby=Female
Parentof·Eric=Roberta

or abstractly A·O=V. The A,O,V notation comes from considering the association as:

Attributeof·Object=Value

We view a collection of associations in the computer as a software "associative memory." That is, one could retrieve any (3-part) association by specifying any one or two of its positions. Figure 1 describes these possibilities.

An associative language will contain operations like:

Make Parentof·Eric=Paul

and

Erase Homeof·ANY=Valhalla

which add or delete associations or sets of associations from the collection. There will also be operations to retrieve and make use of associative information like:

Print (Sexof·Peggy)

and

Foreach PERSON such that Homeof·PERSON = Rochester do
Print (PERSON, "lives in Rochester").
Table 1

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A·O=V</td>
<td>Homeof·Paul=Valhalla</td>
<td>The association itself, if present</td>
</tr>
<tr>
<td>A·X=V</td>
<td>Parentof·Eric=Parent</td>
<td>Parents of Eric</td>
</tr>
<tr>
<td>X·O=V</td>
<td>Parentof·CHILD=Paul</td>
<td>Children of Paul</td>
</tr>
<tr>
<td>X·O=V</td>
<td>RELATION·Eric=Rochester</td>
<td>Attributes linking Eric to Rochester</td>
</tr>
<tr>
<td>A·X=V</td>
<td>RELATION·CHILD=Parent</td>
<td>Child-Parent pairs present</td>
</tr>
<tr>
<td>X·Z=V</td>
<td>RELATION·PERSON=Male</td>
<td>All pairs of attributes and objects yielding Male</td>
</tr>
<tr>
<td>X·O=Z</td>
<td>RELATION·Abby=VALUE</td>
<td>All attributes of Abby and their values</td>
</tr>
</tbody>
</table>

Figure 1

In the latter case, PERSON is a variable that is repeatedly "bound" to the middle (Object) position of associations matching the pattern.

We are interested in how to implement a programming system that can do such operations efficiently. The difficulty, of course, is that each program is different and one cannot expect a single implementation to be best for all programs. What is required is an "intelligent compiler" which selects a different collection of storage structures for each program. The first such effort was concerned primarily with sets and lists, which are simpler to deal with than associations. This paper explores the automatic selection problem for associations, and describes an experimental system (called the "selection system") which embodies our ideas. The discussion centers around associative data structures which fit into some "main" memory. (The problem for structures in several different kinds of memory is more complex.)

The following discussion illustrates the behavior of the system using the following simple program as an example:

BEGIN
Attributes Parentof, Sexof, Homeof
Item Constant Male, Female, Peggy, Abby, Rochester,...
Item Variable CHILD, PARENT
Make Parentof·Abby=Paul
Make Parentof·Abby=Roberta
Make Sexof·Abby=Female
Make Homeof·Abby=Rochester
Make Homeof·Thor=Valhalla

(1) Foreach CHILD, PARENT such that
    Sexof·CHILD=Male
    and Parentof·CHILD=Parent
do
    Print ("Sonof" PARENT "is" CHILD)
(2) Foreach CHILD, PARENT such that
    Parentof·CHILD=Parent
    and Homeof·PARENT=Rochester
do
    Print (CHILD "writes home to Rochester for money")

END

Figure 2

First let us consider the even simpler program with the part in the box omitted. The program builds a collection of associations involving the three attributes. The "Foreach" part computes (parent,son) relationships in two steps. First it chooses an association of the form

Sexof·O=Male

and binds CHILD to the value of "O" in the association. It then uses that value to match an association of the form

Parentof·CHILD=V

and assigns the matching V to Parent, and prints. This is repeated until there are no more matching associations. We will see how the system chooses storage structures for this simple program.

A MODEL OF ASSOCIATIONS AND HOW THEY ARE USED

How should a program (in an associative language) be described for the purpose of designing a storage structure for its associative data structures? The first thing to note is that a program usually deals with more than one class of association. A class of associations is a group which will be represented the same way internally. In the example, there are two classes: one based on the Parentof relation and one on the Sexof relation. Since there are no searches on the Homeof relation (with the boxed statement omitted), the selection system would not include those associations at all. For each class of associations, the system will characterize the operations performed on that class and the possible values for unbound variables. In our simple case, each class has several MAKE operations and one SEARCH.

The differences among classes of associations are often reflected in the ways that the program accesses and modifies them. That is, a program often uses different parameter patterns for associative operations on different classes of associations. For example, the simple program includes a search for male children, i.e.,

\[
\text{given: Sexof} \cdot \text{variable to be bound} = \text{given: Male}
\]

but not for the sex of a given child, i.e.,

\[
\text{given: Sexof} \cdot \text{given: Eric} = \text{variable to be bound}
\]
For the "Parentof" association, on the other hand, there are two SEARCH parameter patterns (with the statement in the box included):

\[(\text{given: Parentof}) \cdot (\text{given: a child}) = (\text{variable})\]

and

\[(\text{given}) \cdot (\text{variable}) = (\text{variable})\]

In another case, a program may require access to attributes that relate two items, for example spatial relations between two objects in a 3-D scene: above, below, behind, or in-front-of.

Other reflections of the differences in associations are to be found in properties of the associations themselves. For example, the sex of a person has only one value, but a parent may have several children.

Often, the conceptual differences in the associations with which a program deals provide information that is useful for making low-level representation decisions. For example, a representation that immediately comes to mind for the Sexof attribute is a pre-defined (at compile-time) field of every PERSON record, to hold a code (one bit is enough) which specifies Male or Female.

If the program used only searches for the sex of a given child, such a representation would be just fine. But the program does not use the Sexof relation this way, but instead asks that people who are Male be enumerated. The above representation would require that all PERSON records (both male and female) be scanned for ones that are marked "Male". For this purpose, a list of male people might be a better representation. This example points out that classes of associations should be characterized both in terms of the operations of the program and the properties of the data.

Now that we have an intuitive understanding of what "class of associations" means, we will define it more precisely, and sketch how the automatic selection system determines the different classes of associations that a given program uses.

The process begins with a flow analysis phase, in which the system determines the possible run-time values of item variables at their points of use in the program, and the possible associations that could exist at run-time. The first approximation to the collection of classes of associations is one class for each MAKE statement, representing those associations that the MAKE statement would create (at run-time). For our simple example, each such class would contain one triple.

The next phase causes two classes to merge if any single SEARCH or ERASE operation could access associations from both classes. The basic idea is that each SEARCH or ERASE operation divides the store of triples into two subsets: those triples which could match the parameter pattern of the operation (subset A) and those triples which could not (subset B). If the A subsets for two operations intersect, the algorithm merges the subsets, and associates both operations with the result. This continues until there are no more mergers to perform. The result is a model of the (disjoint) sets of triples (called "classes of triples") that will exist when the program is run. Each class has an associated set of operations which access only the triples of the class.

The use of this algorithm reflects an implicit assumption about computation cost: a representation that would require access to more than one storage structure to service a single associative retrieval operation is assumed a priori to be too expensive. For our simple example (with the statement in the box included), the algorithm would yield three classes of associations: one for each of the three attributes.

A LIBRARY OF REPRESENTATIONS

The automatic selection system knows how to deal with roughly twenty different storage structure representations, but these are all based on a few fundamental concepts. One representation of a set of triples is simply to store triples in a vector of entries three items wide and to search all entries for the answers to the questions in Figure 1. There are a variety of record techniques in which a fixed location in a block of storage is associated with an attribute, e.g., one could have records for people with a Sexof field, etc. Another common technique is to form a linked list of attribute-value pairs for a given object (a "property-list"). Other useful ideas include inverted lists (a link through all tuples which contain a given item), hash-coding, and using one-bit fields to represent the presence or absence of a property.

The selection system has a built-in library of representations for associations. The library is organized in two parts: "associative retrieval techniques" for servicing individual associative operations, and "associative representations" for common combinations of operations. The following discussion introduces the retrieval techniques and storage structures that provide access to triples, and describes the ways in which the storage structures can be varied and combined to synthesize "representations." This two-level organization is useful for proposing candidate representations: The analysis of which retrieval techniques are applicable to each associative operation on a class of triples need not be re-done for each (composite) representation that is considered. For example, once hash-coding is eliminated as a possible retrieval technique for an operation, no representations which would use hash-coding for the operation will be tried. Such methods for avoiding duplication of analysis effort are crucial for controlling combinatorial explosion in the selection process.

We have chosen four basic retrieval techniques upon which to base the example representations in the library: field selection records, property lists, inverted files, and hash tables. The diagrams in Figure 3 illustrate simple storage structures which implement these retrieval techniques.

The library consists of twenty-three prototype representations which are based on these simple storage structures. We sketch these below: for a complete discussion, see Reference 6.

For the purpose of the discussion below, we will assume...
1. FIELD SELECTION RECORD

OBJECT:

ATTRIBUTE:

VALUES

2. PROPERTY LIST SEARCH

OBJECT:

ATTRIBUTE:

VALUES

VALUES

3. INVERTED FILE SEARCH

ATTRIBUTE: $(A_1)$

VALUES

VALUES

4. HASH TABLE LOOK-UP

$f(A,O,V)$:

VALUES

VALUES

Figure 3
that each item has a unique (integer) "internal name." We will discuss using this integer as a hash-code operand, or as a key to various useful information about the item. Specifically, we will assume that the internal name of an item is a pointer to a block of contiguous storage cells, and refer to such blocks as "item descriptors." A typical item descriptor contains space for bit vectors and pointers to various lists that the selection system might allocate for triples that contain the item.

**Field selection records**

There are three variations, corresponding to the cases where the value field of the triple is binary (one bit), single-valued (e.g., the Homeof attribute), and multiple-valued.

**Complex inverted list**

This representation provides one sequential block of storage cells in memory for each triple (called "triple block"), with one field for each item. Each block can be threaded on either one, two, or three lists: one list for each item in the triple. The head of each such list is contained in an item descriptor.

**P-List**

A P-list is a list of (attribute-value) pairs. The head of each list is contained in the item descriptor for each object. There may be more than one value for each (attribute,object) pair.

**Two argument hash-table**

Two items (internal values of the attribute and object) are hash-coded (using a bucket hash) to locate a cell in a hash-table; the cell contains the attribute, the object, and a list of values. The cell can be threaded in either one or two lists: one list for each hash operand. The header for each such list resides in the item descriptor for the item which is the hash-operand. This variation is effectively a combination of a hash-coding retrieval technique and either one or two P-list retrieval techniques.

Another variation of the hash-table representation is to thread each value in an inverted list (of triples which have that value). Each of these representations combines a hash-coding retrieval technique with an inverted file retrieval technique.

A third variation is to use entries in the hash table to point to triple blocks, which can be threaded in a complex inverted list.

**Three argument hash-table**

The three item internal names are used as hash-coding operands. Each cell in the hash table represents one triple either by storing the three items, or by pointing to a triple block (which can be threaded as above). In addition, each cell can be threaded in one, two, or three inverted lists.

The above representations allow great flexibility in combining property-list, inverted list, and hash-coding retrieval techniques. This flexibility is most useful for dealing with ERASE operations, which must delete all paths to a triple in a complex representation when the triple is erased. In particular, certain of the above representations support back-pointers in their internal lists. The selection system makes use of this information. A more thorough discussion of the details is beyond the scope of this paper; the reader is referred to Reference 6.

Each retrieval technique in the library is associated with an "applicability" rule. This is an heuristic which is used to determine whether the access structure should be used to realize a given ERASE or SEARCH operation. Applicability rules usually reflect "common sense" (to a storage structure designer) rules, and are employed to limit combinatorial explosion in the selection process. These are useful constraints that represent pre-canned cost analysis decisions, like "blind search should be avoided whenever possible". Applicability rules are also used to detect relatively common special cases for which an unusual representation is particularly well suited.

For example, let us consider the basic record-style representation of a triple. A typical choice for the sample universe described above would be to have a record for each person with fields for his Sex, Home, and Parents. Since there may be two parents, the Parentof field might point to a set of parents, yielding something like the following structure for Eric:

<table>
<thead>
<tr>
<th>Parentof</th>
<th>-</th>
<th>Male</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sexof</td>
<td>-</td>
<td>Male</td>
</tr>
<tr>
<td>Homeof</td>
<td>Rochester</td>
<td></td>
</tr>
</tbody>
</table>

There are additional complications, such as how to represent the set of parents. More to the point, this representation will not support an associative search such as:

```
Parentof-CHILD=Paul
```

without exhaustive search. The applicability rule for the record-style access structure would eliminate it as a candidate for the Sexof association because of the need for exhaustive search. Figure 4 contains a partial list of the computation cost tradeoff assumptions that underlie the heuristic applicability rules.

It is the nature of heuristic search that it is always possible to concoct situations in which a path that would turn out to yield an optimal solution is left untried. As for any heuristic method, the use of applicability rules is meant to achieve a
1. It is cost effective to maintain ordered lists of VALUE items in instances of the following access structures:
   - Field Selection Records
   - Property lists
   - Two Operand Hash Tables

2. It is not worthwhile to consider an associative retrieval algorithm which requires that all items that could appear in a given operand position for a given class of triples be enumerated and tested.

3. The bookkeeping overhead for dealing with storage blocks for Field Selection Records whose lengths might vary at run-time is not cost effective.

4. It is cost effective to maintain an ordering relation on the elements of a property list access structure.

5. It is not worthwhile to consider an associative retrieval algorithm which requires the examination of all triples in a property list access structure for a given item.

6. It is cost effective to maintain an ordering relation on the elements of an inverted list access structure.

7. It is not worthwhile to consider an associative retrieval algorithm which requires the examination of all triples in an inverted list access structure for a given item.

8. It is cost effective to use the necessary storage space and execution time to maintain explicit "back pointers" in an inverted list access structure if it would otherwise be necessary to search the list from its beginning to find the list element which precedes a given one.

9. A hash coding scheme which uses chaining for collisions is as good as any other hash coding scheme.

10. It is not worthwhile to consider an associative retrieval algorithm which requires preliminary searching to locate the buckets in a hash table which might contain answers.

The convention for denoting variations of the inverted file technique is that the first letter in a permutation indicates the position by which the list is accessed, and the second indicates the position by which the list is ordered. The two options here are "the list is accessed by A (i.e., ATTRIBUTE), and ordered by O (i.e., OBJECT)," and "the list is accessed by O, and ordered by A." For our simple example, these correspond to keeping a single list of all (child,parent) pairs, ordered by the internal name of the child, and keeping an unordered list for each child of his parents.

After each operation in a class is associated with all techniques that can be used to realize it, the second phase of proposal occurs: finding candidate storage structures that provide techniques for all operations of a class. In our example (with the statement in the box included), the Parentof class has two search operations, with the following forms:

\{(given)\cdot(given)=\langle\text{variable}\rangle\}

and

\{(given)\cdot\langle\text{variable}\rangle=\langle\text{variable}\rangle\}

A slightly more complex case is a class with several operations where all operations have the same set of applicable techniques, and for each of these techniques the same set of permutations. This will occur, for example, if all operations of the class have the same form (i.e., the same pattern of \{(given)\cdot\langle\text{variable}\rangle\}, or ANY parameter descriptions). In such a case we treat the set of operations as a single operation, and the task is the same as described above. All operations of the class will use the same associative retrieval technique.

In the general case, there are several forms of SEARCH operations which retrieve associations, and the sets of applicable techniques are not the same for all operations. Even in such a case, the intersection of these sets may not be empty. That is, there may be at least one technique which is applicable to more than one operation. The general prob-
lem is to find ways to decompose a set of operations into subsets each of which can be serviced by one associative retrieval technique.

Each representation in the library is cross referenced by the retrieval techniques that it provides. The algorithm for proposing representations for a decomposition is sketched below:

Each representation which provides the required number of techniques is considered. Each applicable technique of each subset of operations in the decomposition is examined. If it is a technique provided by one of the representations, then the applicability rule for each such representation is evaluated. The evaluation yields a (perhaps empty) set of "candidate representations" to be proposed.

Each such "candidate representation" is a data structure which instantiates one of the representations in the library. That is, it specifies which retrieval techniques are to service which (subsets of) operations, and how (i.e., the operand permutation). In other words, an applicability rule for a representation specifies acceptable matches between the techniques that are provided by the representation and the techniques that are applicable to the given operations.

To avoid duplication of effort, the selection system treats similar candidate representations as a unit whenever possible. Specifically, candidate representations which differ only in their operand permutations are usually treated as a unit. For instance, the inverted file representation for the example above has two permutations which would be treated as a unit until final cost analysis is done.

The system cannot always treat representations which differ only in their operand permutations as a unit. Aside from a few such special cases, however, differences in operand permutations are ignored until cost analysis is done, thus saving both space and time in the early phases of automatic selection.

Proposing redundant representations

The discussion above describes a method for proposing candidate representations for a class of triples where each triple in the class is stored in memory only once. In such a representation, each triple will be represented by either one triple record (with perhaps several threads), or one VALUE cell (perhaps in a list of them), or one field of an item record.

The selection system also considers representations in which triples are stored more than once. The advantage of keeping multiple copies is that each copy can serve a special purpose for which its storage structure can be custom designed. In a case where there is no single representation which will service all the operations on the class efficiently, there may be a way to decompose the set of operations into subsets for which efficient special purpose representations can be found. It is more likely that an efficient storage structure can be found for a small set of associative operations than for a large set. Depending on the cost tradeoff between storage space and execution time, such a decom-

position might be "cheaper" than a monolithic representation, even though more storage space is required. For example, a membership relation ("Memberof\(\cdot\)\(\text{Set}=\text{Element}\)") between sets and their elements might be represented both as a list for each set of its elements and as a list for each element of the sets to which it belongs. Such a representation could provide speedy access both for queries about the elements of a given set:

\[
\text{Memberof}(\text{given})=(\text{variable to bind})
\]

and for queries about the sets to which a given element belongs:

\[
\text{Memberof}(\text{variable to bind})=(\text{given})
\]

To propose redundant representations, the selection system uses the method for proposing non-redundant representations as a subroutine. The problem is to discover decompositions for which efficient redundant representations can be found. The method is based on enumerating the ways of partitioning the set of ERASE and SEARCH operations on a class of triples. Roughly, we look for ways to decompose the set of operations on a class into a collection of subclasses, each of which can be analyzed for applicable storage structure representations (using the methods described above) as if it contained the only operations on the class. The idea is to capture in the notion of "subclass" a way to identify groups of operations for which efficient storage structures exist. The result of the analysis of a given class of triples is a collection of candidate redundant representations, each having one "efficient" (see below) storage structure for each subclass in one partition. Thus, for a particular partition (i.e., set of n subclasses), each proposed representation requires n copies of each triple: one copy in each of n storage structures. Each MAKE operation for the class of triples must create its triple in each of the n storage structures, and each ERASE operation must remove triples which match its operand pattern from all n storage structures (problems of critical races and deadlock do not arise).

In our example of a redundant representation for a set membership relation, each association would be stored twice: once as an entry on the list of elements in the given set, and once as an entry on the list of sets which contain the given element.

In any representation, each execution of a SEARCH operation must find all triples which satisfy the operand pattern of the SEARCH. We assume that each SEARCH operation is associated with exactly one subclass. That is, it will find all its answers from one storage structure. Thus, each subclass must service all MAKE operations for the class. For consistency, each storage structure will contain all the triples of the class (one could imagine clever ways to relax this requirement, but we do not consider them here).

Unless the rules for acceptable redundant representations are more restrictive than the rules for non-redundant representations, the number of candidate representations will be very large, and the cost of the overall analysis will be huge. There is always a way of using our library and
applicability rules to design a storage structure to service a
given set of associative operations. If everything else fails,
a complex inverted list representation could be used, with
a separate thread for each operation. Thus, every possible
subclass would be acceptable, and each partition of the class
of triples would be plausible. If several representations were
applicable to each subclass, things get worse. If we consider
the fact that the number of applicable redundant representa-
tions for a partition is the product of the numbers of
representations for its subclasses, we can begin to see that
unless the applicability rules are tightened for subclasses,
there could be very many candidate redundant representa-
tions indeed.

To limit the combinatorial explosion of candidate repre-
sentations that use multiple copies of the store of triples, we
have chosen a specific notion of efficiency and an heuristic
rule to decide when to propose such representations. In
general, we consider a storage structure to be inefficient for
an operation if significant searching is required to service
the operation. Many of the cost tradeoff assumptions that
are used by the selection system are based on this idea, and
they provide (crude) thresholds for early detection of rep-
resentations that are likely to be eliminated eventually in the
detailed cost analysis. The most expensive kind of searching
is “unrestricted” searching; it is this sort that the applica-

bility rules disallow. For example, we do not even consider
a record technique for a SEARCH operation which has only
the ATTRIBUTE given; a search through all possible items
which could be the OBJECT of a triple would be required
to perform the SEARCH. It is convenient to distinguish four
types of search:

1. unrestricted
2. loosely restricted
3. tightly restricted
4. fully restricted

An example of “unrestricted” search is given above. By
“loosely restricted” search, we mean (for example) search
for elements which have a given attribute in a list of attribu-
ture-value) pairs. By “tightly restricted” search, we mean
(for example) search for a given item in a set of them (e.g.,
is Paul a Parent of Eric?). By “fully restricted” search, we
mean access to answers via computation rather than search,
e.g., selection of a given field of a given record, or hash-
table look-up.

The cost tradeoff assumptions used in proposing non-redu-
dant representations lead to applicability rules which
eliminate representations that require unrestricted search.
Loosely restricted search is allowed for non-redundant can-
didate representations, but not for redundant ones. In other
words,

We assume redundant representations which require
loosely restricted search to be inefficient enough to be
eliminated a priori as candidates for final selection. Intu-

itively, it is only worth the price of multiple representa-
tions if searching can be tightly (or fully) restricted.

As redundant representations are synthesized for a class
of triples, they are tested for loosely restricted search re-
quirements. If such search is required, the representation is
not proposed as a candidate.

Now we consider the problem of proposing candidate
redundant representations. The crude way to do it is to
generate all possible ways of partitioning the set of SEARCH
and ERASE operations of the class of triples. For each
partition, each subclass would then be analyzed to determine
applicable non-redundant representations. The cross pro-
duct of the sets of representations for the subclasses of a
partition would be the applicable redundant representations.
The partition with one subclass (i.e., non-redundant) would
fall out.

The inherent symmetry of the problem leads to combi-
natorial duplication of effort; early detection of situations
that have already been analyzed is important.

The analysis deals with subclasses, which are sets of
SEARCH, ERASE, and MAKE operations, and partitions,
which are (initially disjoint) sets of subclasses. Each asso-
ociative operation corresponds to an expression in the source
program, and is represented by the corresponding expres-
sion node in the syntax tree of the program.

The selection system uses a central data structure to keep
track of the subclasses and partitions which it analyzes. This
data structure (called the “set dictionary”) is similar in
motivation to the “discrimination net” of QLISP: it asso-
ciates a unique descriptor with each set that is entered.

The set dictionary is used by the selection system to avoid
duplicating:

1. The computation of applicable non-redundant repre-
sentations for each unique subclass (i.e., set of oper-
ations).
2. The computation of redundant representations for each
unique partition.

Whenever a new subclass or partition is analyzed, it is
entered into a set dictionary and the result of the analysis
is recorded with the entry. A sketch of the method for
proposing candidate redundant representations is presented
below:

A recursive function in the selection system generates
the partitions of a set of operations, using another recur-
sive function to generate the subsets (subclasses) of the
given set. Each subset that is generated is looked up in
the set dictionary. If not found, it is assigned a (new)
descriptor, entered, and tested to determine whether ap-
licable representations exist (this test considers cost
tradeoff assumptions for redundant representations. The
new descriptor is associated with the result of the test. If
it is found in the set dictionary, then it has already been
tested. If the result of the test indicates no applicable
representations exist, then the next subset is generated
and the process continues. Thus, no subset is considered
further unless it has applicable representations; i.e., once
a subclass is analyzed and found to fail, no partition which
contains the subclass will be considered. Furthermore,
because of the structure of the library, if a subclass fails, any subclass that contains it will fail. In other words, if there is no efficient representation for a given set of operations, there will not be one for any larger set that contains the n operations. Thus, supersets of failed classes can be eliminated automatically.

A method similar to the one described above for subclasses is used for avoiding duplication of analysis effort for partitions. A partition is a set of subclasses. Each such set has a unique entry in the set dictionary. Thus, there are two kinds of entries in the set dictionary: sets of operations (subclasses) and sets of subclasses (partitions). After a partition is analyzed the first time, its descriptor is marked, and subsequent attempts to analyze it are short-circuited.

The cost of avoiding duplication of analysis effort is the set dictionary look-up time; the large relative cost of analyzing a subclass makes this overhead worthwhile. Another advantage of using the set dictionary is that storage space is conserved: only one copy of each unique set is kept, and only one copy of the analysis results is kept.

HOW TO SELECT A GOOD REPRESENTATION FROM THE CANDIDATES

In this section we show how to estimate the cost of each candidate representation and choose the cheapest one. We consider two components of computation cost: execution times (in micro-seconds) and storage space (in 36-bit words). For a given program, we are interested both in the execution time required to service its associative operations, and the storage space required for its associative data structures. We use three kinds of parameters to characterize each class of triples in a given program:

1. the average sizes of substructures of the class of triples (e.g., For a given A and O, how many V's will there be on average?).
2. probabilities of occurrence of searches that fail and changes that are redundant.
3. relative frequencies of statement executions.

The computations for estimating the space and time costs of a given candidate representation for a given class of triples are based on evaluating formulas which are expressed in terms of these parameters. Each representation in the library is supplied with two functions: one for estimating storage space cost, and one for estimating execution time cost. The arguments to a space function specify a set of associative operations and for each operation the associative retrieval technique that will be used to realize it if the given representation is chosen. This information is used by the space function to identify the parameters for estimating storage space cost.

A function for estimating time cost is a little more complicated. The arguments are the same, but the time function must sum the estimated contributions to total execution time of each of the operations. For each operation, this estimate is the product of two terms: the number of times the operation was executed in the sample execution and the estimated cost of one execution of the operation for the candidate representation. The estimated time cost of one execution of the operation is determined by a formula that is specific to the use of the indicated associative retrieval technique in the indicated representation. Each representation in the library is provided with formulas for each of its associative retrieval techniques. There is one such formula for each associative operation to which a technique applies. Thus, for each operation, the arguments to the time function are used to select a formula and to identify necessary parameters.

For our example, the space cost formula for a record representation for Parentof and the time cost formula for a search operation of the form

$$A \cdot O = X$$

are shown in Figure 5. The space formula represents the requirement that there be a field of each record (0.5 cells) for each set-valued attribute (e.g., Parentof), and that space for the set of values is needed. In our example, there is only one attribute (Parentof), hence $NA=1$. $NO$ (number of objects) represents the number of people. $NVALS(A,O)$ represents the average number of parents per person.

The flow chart for the search algorithm is also shown in Figure 5. The time cost formula represents the cost of the algorithm. $C1$, $C2$, and $C3$ are constant terms; to a good
approximation they depend only on the machine and on basic design decisions and can be determined a priori. \( P_1 \) represents the probability that the search succeeds. The call on the \texttt{SGENCOST} function represents the execution time required to generate the elements of a set of a specified size \((\text{NVALS}(A,O))\). The values of \( P_1 \), \( N_O \), and \( \text{NVALS}(A,O) \) are determined from answers to questions that the system asks. The choice of which questions to ask is based on which cost formulas are being used in a given run of the selection system.

Parameters that depend only on the representations in the library (e.g., \( C_l \), \( C_2 \), \( C_3 \)) are determined once when the library is defined, and built into the library. Their values are always available. Parameters that depend on the program are determined from information supplied by the programmer in response to questions that the selection system asks. For example, the selection system asks three questions about the use of the \texttt{Sexof} relationship in our simple example. They are:

1. How many males are there likely to be? Sample answer: 5.
2. How often will the \texttt{Sexof} search fail to match? Sample answer: Never.
3. How often will a redundant \texttt{MAKE} occur? Sample answer: Never.

Not all parameters are needed: only ones required to evaluate formulas that are relevant to the representation candidates for a class of triples. Moreover, some parameters can be computed from others. For example, if the answers to

"For a given \( A \), how many \( O \)'s (AVG) will have at least one \( V \)?"

and

"For a given \( A \) and \( O \), what is the average number of \( V \)'s?"

are known, then the selection system can deduce the average number of triples, since it can determine (by asking a question, if necessary) the average number of items that can appear in a triple of the class in a given position. Thus, any two of the above three answers determine the third. The system makes use of such dependencies to derive needed parameters from available ones if it can. If it can't, it will ask a question.

We should say a word here about our assumptions regarding parameters. In any system that does automatic representation selection, a model of the expected behavior of the program is necessary. To make the analysis tractable, we made many intuitive decisions about the appropriate level of detail for cost formulas and their parameters. Two major assumptions about program behavior characterize our model and underlie our work on cost analysis:

1. It is in general impossible to tell for an arbitrary program precisely how it will behave when it is executed.

Instead, we depend on the relative frequencies of statement executions in one example execution, and on estimates given by the programmer about program behavior and data structure size. We assume that the programmer has good enough intuitions about his program to provide a good general characterization. After all, he would have to use such intuitions if he were to design a representation without the help of an automatic system.

2. To really do an optimal selection, we would need to use distributions of the values of the relevant parameters both over time within a run and between runs. We assume here that use of the average values of parameters (i.e., scalars) over the entire execution of the program provides enough information for good decisions about representation choice.

In other words, we are using a crude estimate of computation cost. This estimate should be useful for gross distinctions, but not fine ones. A human storage structure designer works much the same way: he often uses rough estimates of relative frequencies of operations and of relative sizes of data structure components to guide the choice of a storage structure.

Because we are not making precise distinctions for estimating cost, it is not necessary to formulate precise characterizations of the associative search techniques and storage structures in the library. It is adequate to outline each algorithm and compose a cost formula for it in terms of the costs of its primitive steps, the parameters that describe average data structure size, and the parameters that describe expected program behavior. After enough experience with automatic selection systems like this one, we will learn where (heuristics, libraries, formulas, parameters, etc.) additional effort should go.

After both space and time costs are estimated, if a representation is at least as expensive in both time and space as one already analyzed, it is eliminated as a candidate. Otherwise, its estimated overall computation cost is computed (see the discussion below), and it is added to an ordered list of candidates that have already been analyzed. The list is ordered by estimated overall computation cost. The final step in the current selection system is to print out detailed descriptions of the top few candidate representations on the list.

The choice for real programs of an appropriate overall measure of computation cost is a good research problem. Attempts at a solution might well lead to the development of new ways of describing programs and their behavior, and to new ideas and standards for the systems in which programs function. Such a study is certainly worthwhile, but although we could make use of the results, it is outside the scope of this work and is left for future research. Instead (following Reference 4), we have provided in the selection system a function which defines the overall computation cost of a candidate representation to be the product of its storage space and execution time requirements. It would be easy to define a new function of space and time to replace...
the one provided, or to experiment with several in the con­
text of the selection system.

Once cost estimates are derived for all candidates, we
rely on the methods of Reference 4 to choose the best
representation for the class of triples in the context of the
entire program. In particular, we refer the reader to Low's
book for a discussion of the following problems:

1. estimating and considering the cost of the non-associ­
avtive operations of the program.
2. defining an “overall” cost function of space and time.
3. considering the interaction between representation
choices for separate data structures whose life-times
overlap.

RESULTS FOR THE SIMPLE EXAMPLE (Figure 2)

Based on its analysis and on the answers to questions, the
system proposes and evaluates several possible storage
structures for the Sexof relation. The best of these is a
record having one field which points to the set of MALES
(this set is represented as a linked list). Based on the cost
formulas, this has an expected size of 5.5 cells and an ex­
pected time of 457 instructions. Another alternative would
be to list each association separately in a table. This has an
expected size of 13.0 and an expected time of 935. The set
representation is better than this (and all others) in both time
and space, and is chosen. This is a very simplified (almost
degenerate) case of the selection system's operation.

In selecting a storage representation for the class based
on the Parentof relation, the system goes through a similar
procedure. In this case, it selects to assign a record for each
person, with a field of the record indicating the set of par­
ts.

Now let us consider the more complex program including
the statement in the box. Since the Homeof relation is used,
there are now three classes of associations. In our tiny
sample program, the only use of Homeof is to test if some­
one lives in Rochester. This, plus the fact that there is a
small fixed number of people, causes the selection system
to choose as the best structure a single word per person
indicating whether he lives in Rochester. This representation
is estimated to require one cell and 300 instructions.

A more interesting effect of adding the boxed statement is to change the choice for representing the class based on
the Parentof relation. Now the program calls for enumer­
ating all Child-Parent pairs as well as finding the PARENT
of a given child. The selection system considers making two
separate data structures or making one data structure which
does both jobs. The best choice is to have a list of all
children with each child indicating the set of his parents.
This is estimated to require 16.5 cells and 1600 instructions.

SUMMARY

This paper explores some of the problems of automatic
representation selection for associative data structures. In
particular, it presents methods for selecting in-core storage
structures for programs which use associative data and op­
erations which are similar to the ones that the SAIL pro­
gramming language provides.

A model of the associative data and operations of such a
program is defined. We describe a library of storage struc­
ture representations for three-part associations (triples) that
includes records, property lists, inverted lists and hash ta­
bles. We show how to use the model in conjunction with
the library to find “efficient” candidate representations for
the associative data of a given program. This choice of
candidate representations is guided by heuristic rules that
embody educated assumptions about computation cost
tradeoffs. In addition to representations that provide a single
method of access to associative data, we consider representa­
tions that provide multiple access paths to the data and
ones that provide multiple copies of the data.

Finally, we discuss methods for estimating the computa­
tion cost of each candidate representation, and for selecting
the “best” one.

ACKNOWLEDGMENTS

The author is indebted to Jim Low, Jerry Feldman, and Tom
Cheatham for their ideas, guidance, and encouragement dur­
ing the course of this work, and to Peggy Meeker, who put
up with my drafts, changes and deadlines cheerfully while
doing a meticulous job of editing and typing the manuscript.

REFERENCES

guage,” CACM, August 1969.
4. Low, J. R., Automatic Coding: Choice of Data Structures, ISR16, Birk­
hauser-Verlag, 1976.
5. Maurer, W. D. and T. G. Lewis, “Hash Table Methods,” ACM Comput­
Science Department, University of Rochester, 1976.
Center Technical Note 81, August 1973.