May 1981 Report. No. STAN-CS-81-851

Binding in information Processing

by

Gio Wiederhold

Research sponsored in part oy

Defense Advanced Research Projects Agency

Department of Computer Science

Stanford University
Stanford, CA 94305

BINDING IN INFORMATION PROCESSING

Gio Wiederhold

Computer Science Department
Stanford University, Stanford, California

The concept of binding, as used in programming systems, is analyzed and defined in a number of contexts. The attributes of variables to be bound and the phases of binding are enumerated.

The definition is then broadened to cover general issues in information systems. Its applicability is demonstrated in a wide range of system design and implementation issues. A number of Database Management Systems are categorized according to the terrns defined. A first-order quantitative model is developed and compared with current practice. The concepts and the model are considered helpful when used as a tool for the global design phase of large information systems.

content:

1. INTRODUCTION
2. THE EFFECT OF BINDING
   2.1 Efficiency versus flexibility
   2.2 Binding and programming languages
   2.3 Attributes to be bound
3. BINDING IN PROGRAMMING
   3.1 Binding by source language preprocessing .
   3.2 Binding by 'defining constants
   3.3 Binding in loading and linking
   3.4 Binding of parameters
   3.6 A summary example
4. A BROADER DEFINITION OF BINDING
5. BINDING IN DATABASES
   5.1 Binding during data'selection and collection
   5.2 Binding into tuples
   5.3 Binding into conceptual structures
   5.4 Binding into physical structures
6. BINDING FOR ACCESS EFFICIENCY
   6.1 Binding through record arrangements
   6.2 Binding through summarization
   6.3 Auxiliary binding choices:
   6.4 Indexes and their use
   6.5 Direct access
   6.4 Binding while deriving data
7. EXAMPLES OF THE RANGE OF BINDING CHOICES
   7.1 Extremes in data representation
   7.2 Binding in knowledge-based systems
8. IMPLEMENTATION WITHIN DATABASE MANAGEMENT SYSTEMS
   8.1 Binding choices provided by Database Management Systems
   8.2 Schemas
   8.3 Related effects
   8.4 The physical complexity of binding
9. A MODEL OF BINDING
   9.1 A matrix representation
   9.2 A performance analysis
10. CONCLUSION

ACKNOWLEDGEMENTS

REFERENCES

1. INTRODUCTION

The term *binding* is being used with increasing frequency in the computer science literature, in particular when alternatives in system design are being described. Use of the concept can bring both economy and breadth to discussions about information processing. Although it appears from its use that the concept of binding is powerful and deserves some analysis, many computer science reference texts do not attempt to define the term. This contradiction has prompted the generation of this paper. We will first review the traditional uses of the concept of binding, apply it to programming, and then broaden the definition to cover many issues in system design. The utility of the concept will be demonstrated through a wide range of data system topics.

In descriptions of programming systems the term binding has been used to describe the replacement of symbolic references by absolute references. For instance Elson states: "By the term binding we mean execution of the mapping algorithm for deducing the righthand member of a pair from the left" [Elson[77]]. Graham defined binding in a less syntactic sense as the "resolution of variability" [Graham[75]]. In language system design a major decision is the establishment of the point in time when such binding should take place. In one of the earliest computer science texts, Wegner, when discussing program translation, used the notion of binding time to distinguish compiling from interpretation of source code [Wegner[68]]. Earlier Strachey, without using the term binding, described a macro-generator where the symbols are bound iteratively at compile-time by sequential expansions of macros in the source text [Strachey[66]]. ***"Bound Variables"*** is a chapter heading in Dijkstra[62], when presenting effects of the matching of arguments to parameters in a translator for ALGOL-60, but not otherwise used within the text; neither does the term occur in the **ALGOL-60** report itself. The concept in this context is of course identical to the use of variables in Mathematical Logic [cf. Church[56]].

Binding takes place in a compiler when the symbolic names that denote variables are associated with specific memory cells and with computer instructions suitable for the size and type attributes specified for the variable. During execution only the value of the variable is determined. In a pure interpreter binding is delayed until the time that the instructions are executed, since the instruction sequence which generates data may also determine the appropriate size, representation, and storage allocation for the new data value[Smith[70]]. For instance, the interpreter for **APL** programs determines, during execution, from a symbol table, whether the data operands in the statements have the structure of a scalar, an array, or a matrix. The **APL** interpreter then carries out the appropriate sequence of machine instructions, and, when done, stores an appropriate result structure into free storage and updates the symbol table. This interpreter hence binds all attributes of the result to the variable name at the same time that the data values are generated. There are, of course, many intermediate techniques, from incremental compilers which permit

redeclaration of some attributes [Breitbard[68]], to interpreters which preprocess the source text in order to gain execution speed [Nori[73], Bush[79]]. In the latter case the ability to adapt the code to current results is largely lost, portability and compactness of implementation remain.

2. THE EFFECT OF BINDING

2.1 Efficiency versus flexibility

The examples shown in the introduction indicate that late binding of programs to data increases flexibility and adaptability, but also that the deferral of binding decisions tends to increase the time required for program execution since a greater variety of resolution options have to bc considered by the program code. In pure interpreters, every time a statement is executed, the type and size of every operand has to be determined[Smith[70]]. This typically increases execution times by a factor of **20** to 200 [Hellerman[75]]. While hardware which interprets type descriptors [Feustel[73], Flynn[78]] can reduce this factor a great deal, late binding remains significantly more costly in terms of machine cycles required for a given computation.

Defenders of interpretive techniques will correctly point out that where sufficient processing cycles are available, or where the flexibility that is obtained is essential, no economic benefits will be gained by early binding. It is in fact easy to construct examples where interpretation, in a real sense, performs better than compiled code. Many execution failures, for instance those due to data sequences exceeding array limits, waste the compilation effort, and are inefficient.

2.2 Binding and programming languages

The design of a programming language is affected by the selection of early versus late binding. DECLARE statements are needed for early binding, so that the symbols become related to the attributes of the variables. When a language intended for compiling is interpreted, the interpreter cannot take advantage of the flexibility afforded by late binding. If, for instance, the size of a result exceeds the declared and allocated space, an error has to be indicated, although a space reallocation might have been feasible. On the other hand, a language which is designed with interpretation in mind may ignore scope and restrict the size of the names and the syntax of programming statements in order to regain some of the efficiency lost due to interpretation.

In Table 1 we classify some languages into this spectrum. The ranking is based on criteria which are not objectively comparable, though we will justify some of them in the next section. Another problem with this ranking is that implementation decisions made by a compiler writer can, for a particular product, shift entries in this list up or down.

```
Programming languages oriented towards compiling
  |   COBOL    : static, predeclared structures and types
  |   FORTRAN  : subroutines can operate on arrays of undeclared size
  ↓   PASCAL   : dynamic allocation in recursion, rigid parameter matching
      PL/1     : various storage allocation schemes, parameter conversion
      BASIC    : typing implied by variable names
      MUMPS    : the type of a value can be determined at execution time
  ↑   APL      : data stuctures and types determined in execution
  |   SNOBOL   : ability'to execute strings generated during execution
  |   LISP     : programs and data are equally manipulatable
Programming languages oriented towards interpretation
```

Table 1. Categorization of programming languages

2.3 Attributes to be bound

There are several attributes that are bound to a symbol during the translation process and these attributes may be bound at different times. The principal attributes are:

environment : the scope where the symbol obtains its validity and its interpretation. Symbols of identical spelling, but from different environments are distinct and describe distinct data.

type : defines the data representation of an element,. An element will be a member of a predefined set or of a set defined in an preceding type definition.

size : elements of many types can exist in long or short forms.

structure : for aggregations of multiple elements a structure definition will establish the access algorithm to the individual elements.

cardinality : array, matrix, and string structures require specification of the number of individually accessible elements.

location : the address or placement, rule of the element or structure has to be established to permit referencing by hardware instructions.

Table 2. Attributes of variables to be bound

The dependency among these attributes is obvious, so that the earlier attributes can be considered to imply a weaker binding than tne final ones. Binding of symbols to the environment is accomplished early in most languages, so that no symbol disambiguation is needed at execution time. The remaining attributes bind the data to a distinguishable symbol.

If the language does not bind all data attributes formally, then flexibilty appears to increase, but, more demands are placed on the programmer. All attributes are bound eventually to permit execution and the generated and stored representation constrains subsequent usage. The knowledge about attributes that are not explicitly bound by the language and declarations, is either in the programmer's head, in documentation, or in auxiliary tables created by the programmer. If it is in neither place, i.e. not recorded and forgotten, then it might be deduced from inspection of the code sequences which create the data.

For instance, since LISP [McCarthy65] is typeless it is always the programmers responsibility to correctly interpret the data structures that have been created. The absence of type binding is not restricted to interpreters. For instance, some system programming languages, as for instance **BLISS,** do not bind type information to symbols [Wulf71]. We hence find it difficult to rank **BLISS** in Table **1.** A compiler for such a language will not report type conflicts, errors will occur however if program instructions violate the data representation.

Violations of structure or cardinality constraints can be made in many languages by carelessness or deviousness. The location, on the other hand, is always bound to the symbol and expertise is needed to defeat the efforts of the compiler writers. Languages with weak rules of environment or scope, as BASIC or **COBOL,** have minimal limits on the validity of symbols. The languages listed in the middle of the table are strong in scope. Many of the the interpreter-based languages have limited scope expression to lower interpretive cost; a notable exception is **LISP.** In LISP the designers' motives were of course not ease of interpretation, but access to facilities not envisaged in the design of the hardware. It is interesting to note that the interactive environment, made possible by interpretation, has contributed greatly to the popularity of LISP [Sandewall78].

3. BINDING IN PROGRAMMING SYSTEMS

Binding does not only occur in compiling or interpretation of code. As programs proceed from the programmer's hands to the point of actual execution on computing 'machinery they go thorough several phases, and each of these phases involves binding. We will describe five of the binding phases which be invoked.

3.1 Binding by source language preprocessing

Language preprocessors, as exemplified by PL/1 or **BLISS** macros, provide an initial level of binding to compilers and assemblers. Such macros are frequently used to define the computing environment and will adapt the symbolic source code so that system software is made suitable for a specific hardware configuration. The transformed text, seen by the compiler or assembler itself, appears specific to the hardware and the generated code can avoid testing at run-time for configuration parameters such as the memory size and the existence of input/output devices. **A** preprocessor for PASCAL programs, for instance, was used within a software project -of ours to adapt system programs to run on either one of two distinct computer 'types [Lansky80].

3.2 Binding by defining constants

Declaration of a symbolic name with the attribute constant, as seen in PASCAL [Wirth72], provides one of the binding advantages of macros: the compiler knows that the values associated with the constant symbols will remain bound to a **fixed** value during execution so that simpler referencing mechanisms can be used. If the constant is small, then in most computer architectures a single cycle Load-immediate instruction can replace a two cycle Load-from-memory.

3.3 Binding in loading and linking

Most compilers support the joining of separately complied program segments (external procedures) prior to execution. This facility requires some selective deferral of binding decisions. References in compiled code to other external procedures or global variables are left by the compiler in symbolic form to be resolved by a linking loader. In systems which support dynamic binding the linking and loading process itself may be deferred to the time of first reference [Daley[68], Wiederhold[73]]. Dynamic binding permits a compiled procedure to generate or adapt another procedure which then can be invoked subsequently, within the same execution cycle.

3.5 Binding of parameters

The transmission of arguments into subroutines or functions provides further choices for binding. The available methods are well described in compiler textbooks [for example Aho&Ullman[77]], and we will only put the techniques into the binding concept. All binding issues: type, size, structure, cardinality, and location, that were discussed previously, occur in this setting.

When the semantics of a language require that arguments and pararneters are to match fully in terms of type, size, structure, and cardinality as, for instance, in PASCAL, then the binding is rigid and interpretation at runtime is minimized. In more complex languages at least the cardinality of arrays and strings can remain unbound until the time of the CALL. In PL/1, for instance, the notation (*) indicates that the dimension bounds of parameter arrays are to be inherited frorn the bounds of the argument.

The cost of supporting flexibility through automatic mechanisms within a CALL varies a great deal for the live data attributes. Setting the values of cardinality or dimension bounds into the code of the called procedure is not difficult. If the number of dimensions changes, giving access to the cross-section of a matrix argument, which is also permitted in PL/1, then more run-time overhead is incurred. Other changes of structure are, to my knowledge, never formally supported. A change of size or type is also costly, unless the compiler has access to sufficient information to generate conversion instructions into the code. Changes of type and size are often limited to parameters without structure and of cardinality one. When a language forces the compiler to cope with arbitrary cross-sections or unknown types, then indirection will be used, so that at least partial interpretation takes place at execution time.

The ALGOL-60 language distinguishes two types of variable bindings explicitly, CALL by VALUE and CALL by NAME. A VALUE parameter is evaluated and hence bound at the execution time of the CALL and cannot be affected by subsequent changes of the environment. The use of a NAME parameter postpones binding of the parameter to the argument. The argument expression is to be evaluated only at the time of usage within the called procedure, although the environment for the evaluation of the binding is that of the caller. The value of the parameter will hence be affected by value changes made to the calling environment. Implementations of ALGOL allow the called procedure to gain access to that environment by executing mini procedures for each NAME argument which were compiled with the CALL state-

ment. These mini procedures evaluate or set the argument and in this manner bind it to the calling environment in terms of type and structure. While some suprising constructs are possible, for instance *Jensens'* device which permits a general evaluation of arbitrary functions given as arguments [Jensen[61]], modern programming techniques frown on these techniques which are dependent on side effects and aliases.

ALGOL-68 carefully restricts parameter conversion by passing all arguments by VALUE. Any conversions take place prior to argument passing, so that no repetetive interpretation takes place. The use of a reference as a parameter type gives the called procedure access to the calling environment, so that large structures do not have to be copied. The referenced structures have to match according to strong *coercion* rules. To permit the called procedure to operate on array structures transmitted by reference, prirnitive functions are available that provide the dimensions, i.e. the cardinality of the argument and its row components.

Many languages, for instance FORTRAN-66, do not specify type, structure or cardinality-binding through a CALL, and others have compilers that do not enforce it. The referencing mechanism only promises that the location of the argument is transmitted. The ability to mismatch type, size, and cardinality in FORTRAN is used and misused freely.

The location attribute of arguments is always unbound, and in fact defines the concept of a procedure versus a macro expansion. Resolution of location, and storage allocation and setting for value parameters is the initial task for the called procedure. If elements or structures are transmitted via reference then the values of the argument are often not bound to the program code until the time of reference. Inconsistencies in implementation of binding can lead to errors if arguments are accessed by more than one method within the called program. This problem of *aliasing* is well known and one of the major problems in the verification of complex programs.

The environment for the evalution of the argument is an issue that can lead to subtle, but important differences in language implementations[Organick[78]]. We distinguish deep binding, where the argument carries its environment along, and shallow binding, where it does not. The problem addressed by deep binding is that complex arguments, as functions or data structures, can be created within procedures. When the procedure is completed and terminates, definitions relevant to created structures have to be retained.

With shallow binding the environment for the symbol is the most recent instance encountered. Since most recursive language compilers are implemented using a stack concept, where every invocation of a procedure deposits a set of local definitions, called a display, on the stack, this environment is the one of the procedures which is associated with the most recent display containing the symbol. When a procedure terminates, the corresponding display is removed from the stack.

Deep binding is most explicitly managed in LISP. When a function is defined in LISP the LAMBDA list names explicitly all the parameters that are to be bound when the function is invoked [Church[41], Scott[72]]. Other variables remain free, and are to be interpreted according to the most recent binding when encountered. In order to manage deep binding correctly in LISP, the equivalent of a display is not

unstacked when a procedure terminates. New procedure invocations grow separate stack branches, the data structure used is a tree of displays. The LISP garbage collection mechanism will remove stack branches that are no longer referenced [Allen78].

3.6 A summary example

We have found a wide range of binding choices in current programming language systems, but also that the choices are presented in quite inconsistent forms. For a PL/1 environment we can cite for instance five levels:

1 compile time macro facilities,

2 static variables and constants, assigned fixed attributes by the compiler,

3 external variables resolved by the loader,

4 controlled and automatic storage of dynamically allocated program variables managed by the operating system, and

5 type and size adjustment at execution time for parameterized variable structures using nearly interpretive routines which are included as run-time support code for the compiler.

We have demonstrated with these examples that binding is a concept which has much breadth, and that it is convenient to discuss issues in programmming systems design and their tradeoffs in terms of binding decisions. We will return repeatedly to the paradigm of binding as shown in these programming examples as we broaden the definition of binding.

4. A BROADER DEFINITION OF BINDING

In our work on database design [Wiederhold77] we found that binding was a recurring theme when issues of flexibility versus efficiency had to be discussed. A general definition suitable to the broader view is:

Binding is
the Commitment of Knowledge to Structure.

Anytime that knowledge about data usage or constraints on data values is utilized to translate application objectives into into a database structure and into associated programs the extent of binding is increased. With every binding step some flexibility is lost, but the opportunity for automated processing is increased.

The range of binding choices is particularly striking when we view entire systems from their inception to the time they are able to produce results. At every point in this process the designer uses some knowledge to impose structure, and each structuring step reduces the number of implementation alternatives that remain. Since it takes a long time to bring information systems into operation, these binding decisions are made over a span of several years, so that a global insight, using the common concept of binding, will be important. The motivation for an improved conception of binding will be restated at the conclusion of this paper.

5. BINDING IN DATABASES

We will now, in the central part of this paper, illustrate binding opportunities at various points in the design process for systems which process large quantities of data. Databases are an essential part of such systems, but we are not only concerned about the data, but also about the meaning represented by these data. To identify the breadth of concerns in this and the following sections we consider that we deal with binding decisions applied to information *systems*. The objective of collecting and processing large quantities of data must be information. This section will consider binding issues within the data itself, and Section 6 will deal with other binding options provided by file and database systems technology.

5.1 Binding during Data Selection and Collection

The real-world entities to be described within an information system have typically many attributes. Some of the attributes which define an entity are obvious to a designer, others may be difficult to perceive. Of the known attributes, certain are expected to be important and others are not seen to have much import in the conception of a system application. Some useful attributes may be ignored since they are assumed to be too costly to collect.

It is obvious that only recognized aspects of the entities can be collected into the database, so that the initial explicit binding decision is the selection of some of these aspects as attributes. When, for instance, the productivity of a factory is measured in dollars, we fail to measure its benefits in terms of satisfaction of its employeees and its effect on the environment. The productivity in terms of the weight of its output may also not be regarded as useful. The selection of attributes restricts the class of potential analyses of the enterprise.

The observed values of these attributes are to be measured and encoded into data representations and the measure and encoding chosen also binds the design. When we describe the disease of a patient using a diagnostic term, say "Arthritis", we resort to a shorthand notation for a complex constellation of symptoms, history, and physical findings. This type of binding might be reduced by permitting much latitude in the choice of entries and their representations. Unfortunately, even keeping all data in the apparently natural form of free text does not avoid binding completely. The terminology used in written text is inseparably bound to the writer and his environment. Words which express standards of size – i.e. "a large computer" – or measures of goodness – "a trivial error" – or – "a minor ailment" – vary rapidly over time, and what one person considers an unimportant aspect of an entity may well be another's primary concern [Komaroff79]. At the same time, if encoding is avoided, the cost of encoding of the text is moved to the time of data analysis, and incurred every time the data is to be used.

5.2 Binding into Tuples.

The next stage in the binding process occurs when we assign the data to entities and to attributes of entities. The minimal structured data entry is a binary relation of two values, forming a triplet as shown in Figure 1, where the entity `JOHN` and a value of a date are bound by the relationship `birthdate.`

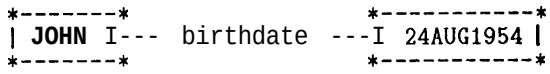


Figure 1 Binary relation

It is sensible to bind these two values together, neither value is apt to change independently, and the interpretation of the relationship is fixed. A binary relationship of this type is in fact a minimal information-carrying structure. If any of the three components is omitted no 'specific information remains. Languages used in Artificial Intelligence provide facilities to describe and manipulate these atomic relationships. The data structures are constructed as needed from atomic elements provided in list processing languages as LISP. An example of the use of such tuples as basic data structures for programs occurs in the LEAP system [Feldman[69]] as well as in its derivatives and relatives. LEAP labels the elements of the triplet as object, attribute, *value,* and provides functions to build and interrogate data structures composed of complex binary relationships.

LEAP treats the three parts in a symmetric manner, but database theory considers values as 24AUG1954 to be functionally dependent on the object, here JOHN, and birthdate to be the attribute function upon the object. The introduction of the notion of functional dependency leads to an asymmetry in binding. The effect of this asymmetry is noticed in updating, as shown in Figure 2. To promote JOHN to SUPERVISOR requires a change of the value field only, changes of object or attribute have further implications.

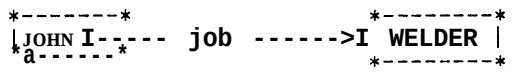


Figure 2 Functional dependency

These binary relationships are soon bound into more complex structures; a development of our example, forming a hierarchy, is shown in Figure 3.

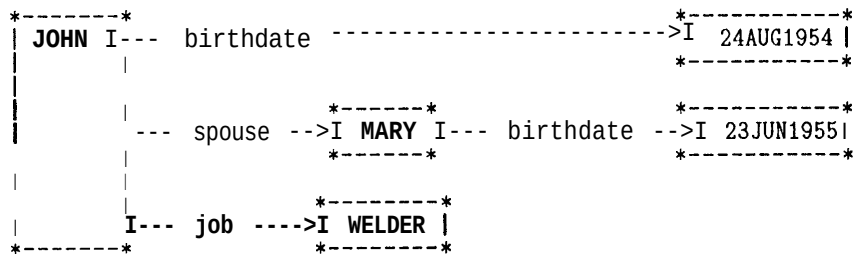


Figure 3 Binary relations forming a hierarchy

When no constraints are placed on the binding structure, the interconnections will form complex nets [Abrial[74]]. In Figure 4 a relationship lives-at is added which transforms the structure into a network. Examples of extensive representations of knowledge in this form are found in artificial intelligence applications using the term *semantic nets* [Quillian[68], Duda[78]].

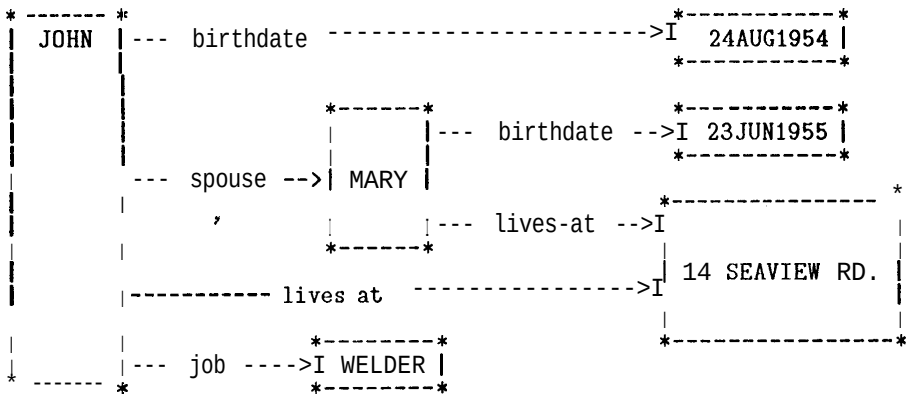


Figure 4 A network of binary relations

In order to make knowledge available for mathematical and statistical processing, a simpler structure is more appropriate. It is common to arrange the data into records, a structure which is modeled well by n-tuples in a relational algebra. Figure 5 shows an n-tuple which describes an Employee.

name	birthdate	spouse	spouse-bd	address	job
JOHN	24AUG1954	MARY	23JUN1955	14 SEAVIEW RD.	WELDER

Figure 5 An n-tuple

By composing data into tuples a strong relationship among data items placed in the same tuple is implied. In the pure relational model the diversity of relationships among the data items within a tuple is ignored, the columns may in fact be freely permuted [Codd[70]]. The use of a single strength of binding within the tuple leads to a loss of source information; but manipulability of the data is greatly enhanced.

The example shown in Fig. 5 demonstrates also an undesirable binding decision: since the birthdate of the spouse is dependent on the current value of the spouse we have here a structure which will contain inconsistent values when the spouse field is updated. If furthermore MARY is also listed in an Employee tuple then both spouses' birthdates will appear redundantly and this may lead to more update problems. The problem shown here can be solved by systematically avoiding secondary dependencies within tuples, so that the binding structure is simplified. This process is known as normalization of the data model, in the example this leads to the creation of two sets of tuples as shown in Figure 6; the structure is now bound only according to primary functional dependencies [Codd[72]].

name	spouse	address	job
JOHN	MARY	14 SEAVIEW RD.	WELDER

name	birthdate
JOHN	24AUG1954
MARY	23JUN1955

Figure 6 Relationships arranged to avoid secondary dependencies

The database will have to accomodate many tuples of each type. Similar tuples are placed into relations, and we will name the two relations of Fig. 6 Employee and Personal-data. How satisfactory is this arrangement of the data into these relations?

We consider the case that two spouses may be employees. The structure above will include similar tuples for both in the Employee relation, and the fact that JOHN and MARY are husband and wife, and their address, would appear redundantly. Changes to these data values will require multiple updates. A solution to this problem is given by an alternate structure. We place Household data into a separate set of tuples, as shown in Figure 7. Now non-employed spouses will also be listed in the first set of tuples; MARY might be managing her Household or a company division.

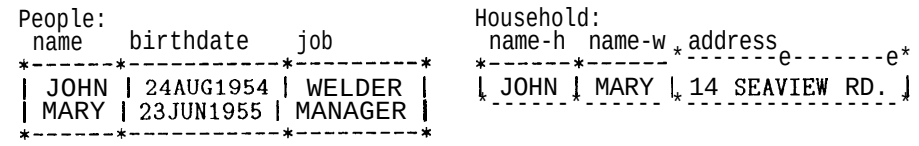


Figure 7 Relationships arranged to avoid redundancies

Tuples for employees often require many attributes beyond those kept for people in general, so that the solution used in in Fig. 7 for the attribute job is actually quite awkward. The concept of a subset relation of a more general relation provides the answer [Smith77]. Figure 8 presents the data from this example for the case that MARY is not an employee. No redundancies, other than a repeated name as identification, are created if MARY becomes employed.

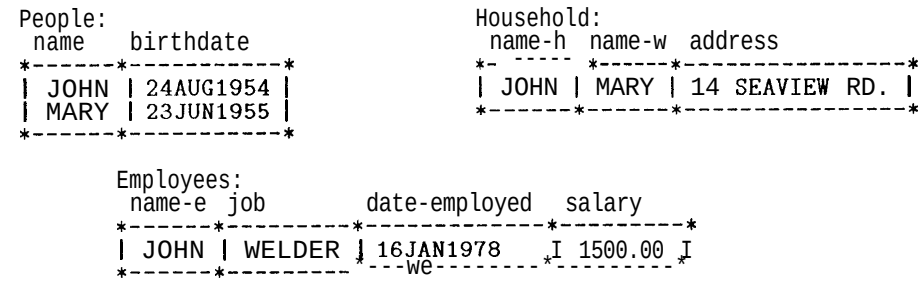


Figure 8 A database with a subset relation

The rearrangement of data into multiple sets of tuples avoids most of the problems of representation of a complex structure by permitting repeated use of simple data structures, but note that two extremely different binding arrangements are now in use. There is a quite strong binding implied among the elements placed within one tuple, and a weak binding between tuples in different relations. The weak binding is exhibited by matching values in attribute columns that cover identical domains. The difference in binding between name and birthdate, job, or spouse is not based on semantic differences, but is due to structural requirements.

In Fig. 8 the values of the names provide the linkage information for relationships between tuples. In literal implementations of the relational model the weak, value-based binding between tuples is only used at the time when information is to be retrieved from the database. A considerable computational effort may be

required when relationships that are weakly bound, have to be explored [Gotlieb[75]]. For example, a search to find instances of older spouses, will require for each result a scan through both files that implement these relations.

The original semantic data structure can, in general, be modelled by more than one valid normalized set of relations. Without knowledge of the original structure several alternative semantic interpretations can be made from the relations, as was shown in Bernstein&Goodman*'.

5.3 Binding into conceptual structures

While the data are organized into distinct normalized relations the knowledge about the interrelationships of these data will remain crucial to the operation of the database[Chen[76]]. The terrns generalization and aggregation have been used to describe the processes which designers use to organize the relational schemas into a model of the overall database [Smith[77]]. In our examples People could be considered a generalization of Employees, and a Household can be viewed an aggregation of People and Children. Such relationships impose constraints on the databases which implement these concepts. We will not want to enter into our database Employees or Households which have names not found within People.

Existing file structures and databases do recognize such intermediate forms of binding. For instance references between People and Household tuples may be described and explicitly maintained, or Children may be linked into a hierarchical structure under a relation Households [Wiederhold[77]].

Another relationship type that has to be recognized is an association among entities and subsets. We define a structural model where relationships among entities are represented using connections of three types, which confer well defined maintenance semantics on the database being modelled[ElMasri[80]]. The three connection types used are *reference,* ownership, and identity.

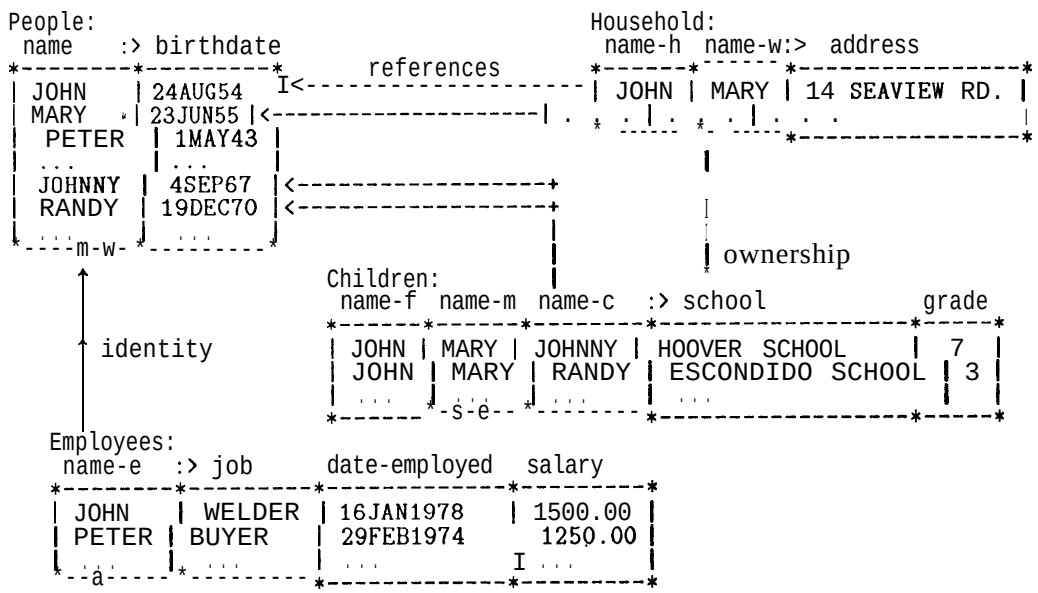


Figure 9 Tuples with reference, ownership, and identity connections

Figure 9 shows the example of Fig. 8 with references and a subset, linked by an identity connection. There is also a further relation Children with tuples that are owned tuples of the Household relation. We use the term nest to describe such a set of tuples that are dependent on or owned by another tuple in another relation. **A** reference from the Children's name-c field classifies Children also as People with birthdates. All related tuples contain copies of the relating attributes to'define the actual instances of the relating connections, i.e. which Household owns JOHNNY , or who is the WELDER.

The three types of connections have been formally defined [Wiederhold[80]], but we note here only that their semantics establish increasing degrees of binding between tuples. A reference connection describes a n : 1, n = 0.. $.n_{max}$, relationship, and requires only that the referenced tuple exist prior to insertion of a reference. An ownership connection defines a 1 : m, m = 0.. $.m_{max}$, dependency, so that furthermore owned tuples have to be deleted if the owner is deleted. An identity connection also forces deletion of the subset tuple, but restricts the relationship to 1 : s, $s = 0, 1$. The formal maintenance of these constraints also requires the identification of key fields, they are shown above to the left of a :> symbol.

The maintenance of connections increases the degree of binding of the database. Updates of the relations in this database model become constrained, but the knowledge represented in this model increases the fidelity of the model to the real world. If the constraints implied by these bindings can be maintained whenever the database is updated, then many common database errors will be avoided. Knowledge of the database structure can also provide substantial benefits during the processing of queries. We can, for instance, be assured that all Children can be located via the Household relation or that each Employee has a corresponding People tuple with a birthdate field.

5.4 Binding into physical structures

In order to implement the semantically relevant bindings discussed above, some structures have to be added to the simple tables of data which implement pure relations. The concept of a tuple in the models is mapped, directly or with some transformations, into records, and records of one or more types are assembled into files. The data from several relations may hence appear in one file, and occasionally data from one relation may appear in more than one file. We introduce the complexity of this mapping to permit a clean separation of the logical issues, which dominate the model design, from the performance issues, which dominate the physical implementation.

These file and record structures will bind the database, increasing the update cost, but reducing the cost of information retrieval. There are many choices for physical implementation of each of the connection types, so that the balance of update cost to retrieval benefit can be adjusted. In practice the choice is limited to the actual alternatives available in a given database system implementation, and by the ability of programmers to deal with the alternatives in a rational way. We will now discuss implementation alternatives in general, and in the next section present the choices available in DBMS's.

A model, as shown in Fig. 9, may be implemented with various degrees of binding. Table 3 lists a number of physical binding choices in increasing degree of binding. The initial mapping of tuples to records is direct.

1 No binding exists in the database structure, all constraints are the programmer's responsibility.

2 Constraints are given as assertion statements to the DBMS, to be automatically checked on database update [Stonebraker[75]].

3 Connected tuples are physically clustered, by placing them in the same block whenever possible[Astrahan[76]].

4 Connections are implemented through pointer structures. Fields in the records contain references to other records in a form that can be rapidly interpreted [Bachman[72]].

5 Connected tuples are merged into records, often creating redundant entries or null fields in the database [McGee[77]].

Table 3 : Implementations of tuple relationships.

The problems encountered in the implementation of bindings in databases is the cost of poor locality, the difficulty of referencing remote records, which may not have a fixed address, the high cost of searching the database files when references are symbolic, and the wide range in cardinality of reference and ownership connections.

The bindings which are only provided by programmer's knowledge or by assertions do not provide direct retrieval benefits. Intelligent processing of queries can take the assertions into account and permit rephrasing of queries to improve access efficiency or to provide more cooperative answers [King[80], Kaplan[79]].

Pointers are the primary means of binding of connections. A symbolic value, as shown in the earlier figures, requires a scan through the destination file for resolution. With auxiliary structures in the destination file, for instance indexes as discussed in Sect. 6.1, this approach can be reasonably fast. Two further levels of binding by pointers are listed in Table 4.

Symbolic A key value is stored in the source tuple. A scan or an index is used to locate the destination tuple.

Indirect An index to a table is stored. The table entry provides the physical address of the destination tuple.

Direct The physical address of the destination tuple, or its computational equivalent, is stored in the source tuple.

Table 4 : Implementations of pointers.

Records of a file which is referenced symbolically can be moved freely, a distinct advantage since records can be rearranged to improve locality or space utilization. If the pointer field contains a record identification index, then moving the record requires a change in the table. The third choice, an address pointer, permits immediate access to the destination record, but binds the destination record to a specific physical location. A direct or indirect pointer can replace a symbolic reference or provide a redundant, alternate representation of the reference.

Simple pointers are one-directional. They are adequate for 1 : 1 and n : 1 relationships, but to implement a 1 : n relationship space for n pointers would be required within the source record, and n is in general not known when the record is first, written. The obvious solution is to place the n records into a linked list, so that each source and destination record contains one pointer. The ordering of the list provides another option for cost-of-access minimization, letting the most frequently accessed destination record come first.

In most cases a need exists for traversal of the connection in both directions, but the binding requirements may well differ. Placing pointers according to each requirement is a solution, but if a 1 : n linked list exists then linking the last destination element back to the source, creating a ring, is a simple and common technique to support the n : 1 path. The cost of traversing rings that have many records is high however, and leads to a desire for clustering them in the same or nearby blocks.

Pointers may be added to the attributes specified in the tuple or may be used to replace symbolic fields. If, for instance, in Fig. 9 pointers are used in the Children tuples instead of parents' names a significant space saving accrues. Certain queries will require now more tuple accesses. An example is retrieval of the parents' names of Children at HOOVER SCHOOL. Careless replacement of values by pointers can make some queries almost impossible, although expected queries may be processed faster. Space savings, especially for tuples low in an ownership hierarchy, can be spectacular, but should not justify creating logical query processing problems. A taste of the space saving can be given by showing a tuple that contain grades of courses taken by students of schools of schooltypes of schooldistricts of a certain state. If each school also maintains personal data on Students then the leading four fields can be replaced by a ring of pointers from JOHNNY SMITH's record, as shown in Figure 10.

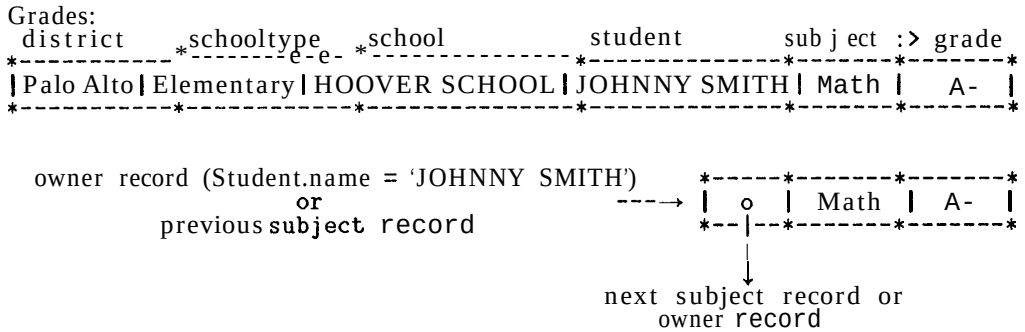


Figure 10 : Tuple and record at a low level in a hierarchy

Rings of pointers are also used to implement networks, that is bindirigs of databases where one record can have more than one owner. A typical example is modelled in Figure 11. Given, for instance, three relations as shown, of which the last one is an association of the other two, we see that two symmetric relationships exist in the model. These relationships are modelled by two ownership connections and are easily implemented by two ring structures, one persons-having-given-skill(l) and the other skills-of -a-given-person(a).

```
Person :
  name :> person-description

Skill :
  skill-type :> skill-description

Skills-of-persons :
  skill-type, name :> experience-level
```

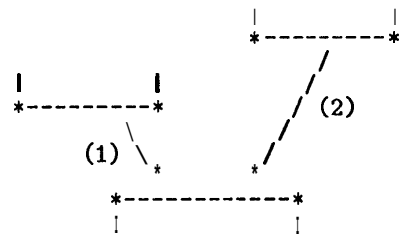


Figure 11 Association

The next degree of binding is accomplished by physical clustering. Here tuples with important connections are placed in close proximity, so that the retrieval of a block or segment from a storage device has a high probability of also retrieving the related tuples. In order to obtain an effect from this technique the file system has to recognize when a block containing a record is already within a buffer in memory, so that a disk access can be avoided. An identity connection is often implemented with physical clustering, so that the tuples of the People and Employees relations 'shown in Fig. 9 will appear in the following sequence:

People: JOHN . . . . Employee: JOHN . . . . People: MARY . . . .
People: PETER . . . . Employee: PETER . . . . . .
People: JOHNNY . . . . People: RANDY . . . . . .

This clustering technique is often extended to ownership connections in hierarchical databases, by placing the tree into a preorder physical sequence [Knuth[73]]. One tree instance can become very large and span many blocks. The traditional master-detail file is also an example of such a physical arrangment. If there are multiple ownership connections then the designer has to decide which of the connections is to benefit from physical clustering, since only one physical sequence is feasible on our storage devices. Once this design decision is made the performance of a symmetric model becomes very asymmetric. There are instances where access times to subsets of network elements are seconds in one direction and hours in the other.

Replication of owned tuples can solve this access problem, but at a very high maintenance cost. Such structures occurred in traditional data processing files, where for instance billing details were kept both with the customer file and with the shipping file. The problems of keeping the logically very strongly bound data consistent was a major motivation for moving to database technology. The traditional binding techniques remain valuable if they can be employed without creating logical problems.

The strongest physical binding between tuples is achieved when tuples are actually merged into single records. Not only will the data be clustered in storage, but a single operating system action will fetch the related tuples. An identity connection is a prime candidate for implementation by merging tuples. If the People and Employees tuples from our example are merged then the 1 : 1 constraint is easily enforced and exploited. The resulting records will have several null-fields for non-employed people.

Tuples related by reference connections may also be merged, but now data elements may have to be be replicated. For example, in the data structure of Fig. 7, replication the addresses for People can eliminate the Household tuples. Such replication increases performance whenever the address is needed while the birthdate or the job fields are referenced.

The transformation from tuples to records should be noted by the database system so that updates will be routed to all replicated data elements. In the above example a Household address change will affect two records. The tradeoff is similar to the one seen in replication with clustering and such replication is done only if update is known to be infrequent.

When databases are distributed over multiple computers, the need to keep replicate data to improve access speed is especially strong. Significant access delays occur in these systems, since transfer of blocks over communication lines is even slower than block retrieval from disk storage. Replication of tuples is hence common in distributed systems, and techniques to insure consistency are of great interest. Again, making a carefully reasoned choice about the degree of actual consistency maintenance among these tuples can effect system cost greatly. The requirements for queries to a distributed database have been classified into no, *weak,* or strong consistency [Gray[76]]. A further classification can specify the currency status of the database expected by the issuer of a query: no currency, up-to-date to a given time (t-bound), time-consistent at a given time (t-vintage,), or latest [Garcia[80]]. The processing demands of algorithms which satisfy the more stringent demands of consistency among replicated data increase sharply. While the term binding is not used in these analyses a strong relationship is obvious.

6. BINDING FOR ACCESS EFFICIENCY

In the examples shown in Section 5 some intrinsic semantic relationships of the stored entities were exploited for implementation of binding needs. To these natural connections certain artificial relationships can be added in order to gain access speed. Linkages based on artificial relationships arc always redundant and can hence be created and deleted without affecting the information content of the database. We will first present arrangements of the database itself that improve access, and later present auxiliary structures for that purpose. The rearrangements again employ clustering.

6.1 Sequential clustering to improve access performance

A common clustering arrangement of tuples into a file places records together according to some predictable order. In a sequential file, for instance, the tuples have

been mapped to fixed length records, and the records placed in sequence according to a key attribute. We find now a two-dimensional structure, as shown in Figure 12.

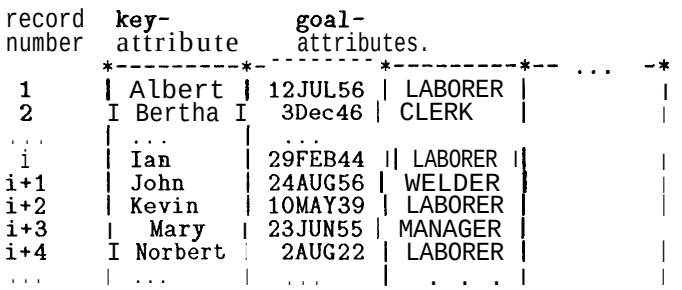


Figure 12 A sequential file

The elements appearing within one record are strongly bound to each other, implementing the tuple binding concepts from a relational model. The sequential ordering of the records in the file does not represent a meaningful relationship in terms of the data semantics. The records are maintained in an alphabetic key sequence in order to increase the access speed for subsequent usage of of the file. A prime purpose of the sequentiality of files is the performance of rapid merges with other data that are sorted according to the same key attribute.

The sequential structure is also effective for retrieval of individual records if the search argument matches the key attribute, since a binary search can be used. The retrieval efficiency gained by this redundant binding costs dearly when the file is to be updated. Insertion of new records is so costly that sequential files are typically updated only periodically, in batch mode; the price is paid in terms of not having the file up-to-date.

The two-dimensional organization displayed in sequential files can be transposed when access efficiency is desired for access by attribute. Now one record will contain all the names, the next record all the birthdates, etc.

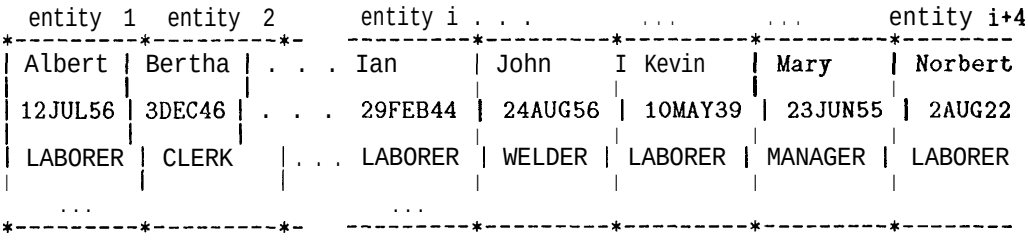


Figure 13 A transposed file

Now the structure of the record itself exists only for access purposes, and does not represent any semantically meaningful relationship. This structure is efficient when data analysis is directed towards patterns of data values, rather than towards retrieval of data describing individual entities [Wiederhold[75]]. An example is the selection of stocks having a certain price to earnings ratio. In a transposed file only two records would bc accessed and scanned: the record containing all values of price and the record containing all the values of earnings. In such situations, where **data**

are used mainly for statistical inference, transposition, providing structural binding by attribute type, gives very fast access.

Transposed files, however, are even more costly to keep up-to-date than sequential files. The insertion of a new entity forces a replacement of every record with a larger record. A transposed file is also intuitively‘ not attractive. The loss of a semantically meaningful structure to gain improved performance is not obvious and not relevant when discussing retrieval of single data elements, say the profit on sales of company **C.** However, the value of this individual data item means little if the average profit of all businesses in the industry is not known. The computation of the average using a file which is not transposed or already summarized is a costly process requiring retrieval of many records.

6.2 Binding through summarization

The process of recalculating averages can be avoided if the data are structured so that values which are common to a subset of records or to a nest, are kept in one master record. Hierarchical data organizations have a structure which makes it convenient to keep summary data available.

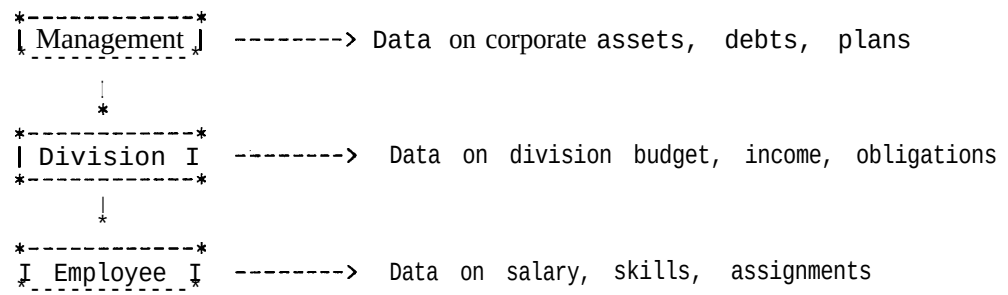


Figure 14 Hierarchical data structure

At the lower levels there are many instances of detailed data items. At the higher levels detail, as for instance salaries, is summarized into new concepts as budgets and salary obligations. Candidate computations for summarization are:

COUNT, SUM, AVERAGE and VARIANCE, RANGE (Minimum and Maximum)

as well as various percentiles, and counts of selected special conditions.

The values produced by summarization are stored with descriptive names in the database. Exarnplcs of such values arc the number-of -employees, sales-totals, average-salary, etc. Summarizations are typically maintained in files by procedures which operate on lower level data. The summarizations are, of course, redundant relative to the the lower level data, and require additional computation in order to remain up-to-date. These redundant elements are strongly bound to each other, the knowledge about their relationship has to be represented symbolically or procedurally to assure long term consistency of the database. We now find that procedures have to be bound to the data or at least to their description in the database schema.

The CODASYL definition and the IMS system [McGee77] allow the specification of database update procedures which are automatically executed when designated data-elements are changed or accessed. The complexities of these systems become great, but are tolerated where the pattern of access demands for retrieval is such that such summarization binding pays off. Not having the summarization procedures bound to the database forces reliance an organization to rely on scheduling of summarization programs to be executed between updates and queries.

Summarization in a database system can be performed whenever a lower level data element is updated, or it can be deferred to the time of query execution. In the former cases the procedures are bound to the updated element, and executed whenever the base date are changed. Frequent or batched updates will be costly. A cost-of-living salary increase will cause many recomputations of the average-salary. When computation is performed with the query execution the response time will suffer, and multiple accesses to a summarized element will cause excessive recomputation. The latter case would occur if for instance the ratio of individual sales to sales-totals is to be reported. Recomputation costs can become extreme if several levels are involved.

We have here again an example of alternatives in binding. Neither extreme is desirable. Generation of summarizations at data-entry time creates a delay at that time which may be unaffordable, and computation at retrieval time can make the response delay intolerable. In either case the high cost of consistency maintenance concurrent with user transactions forces many systems to abandon automatic maintenance of consistency of stored summarizations: the higher level data are only updated periodically. We have suggested an automatic procedure based on time-stamping and identification of currency needs of the query, in order to make such binding dynamic and efficient, but we know of no implementation to date[Wiederhold77].

6.3 Auxiliary binding choices

The binding methods presented in Section 6.1 organize the the entire database structure to achieve some specific access goal. A clustering of the data into sequential or transposed files represents a single and far-reaching binding choice. When the objectives are more diffuse it is better to attach auxiliary structures, for instance indexes or pointer lists to the database. The variety of techniques is again great. The additiunal structures are derived from the data in the database, and hence require additional maintenance at update time. Since the structures are derivable -it is possible to defer updates of auxiliary structures to a time of low system usage. Queries executed during the deferral period will either give incomplete results or incur the costs of looking both at the regular database and at a file of recent, incompleted update transactions, and correlating the data found in the two paths to avoid duplicates. We will present two approaches which use immediate updating, first and briefly the use of indexes, and then the use of pointerlists, addressed by a hashing algorithm, where the choices can be presented in a well-ordered fashion.

6.4 Indexes and their use

The most common auxiliary access structures are indexes. Index entries replicate

values from selected attribute fields of the relation, and combine them with a reference to the database record. The entries are sorted and placed into a table or tree. The tables of the older indexed-sequential file types are not updated when changes to the files are made, but the files themselves are maintained so that insertions can be found. A periodic reorganization creates a new index [Wiederhold77]. The more recent alternative keeps indexes in tree form, which are updated as the database is changed [Bayer70]. An important binding decision is the selection of attributes for indexing. If all attributes of a databases were indexed the database would probably at least double in size, and the insertion of a single record would require changes to the indexes of all the record's attributes. The problem of index assignment in the design and during operation of a database has been successfully studied [Hammer76, Schkolnick76,Whang81] so that we will not cover it here. When queries to the database involve multiple attributes simultaneously, the problem of index assignment becomes more complex, since the design decision depends on the strategy used by the query processor.

6.5 Direct access

As indicated above, we will use direct access as an example of alternatives of auxiliary binding. Direct access or hashing provides rapid access to files. We will demonstrate the range of binding choices available for implementation of direct access by sampling methods for multi-attribute retrieval.

A basic direct access file consists of an addressable file storage segment, a *key*-to-address transformation (kat) algorithm which gives the candidate address of a record with that key in the segment, and a means to cope with collisions due to the non-uniqueness of kat transformations which use hashing techniques. Direct access, if collisions can be minimized, provides very fast access for storage and retrieval of records whose key is known. In these examples we consider the more complex case of multi-attribute access, so that any partition of attributes can be used for the retrieval.

A typical approach to provide access for more than one attribute is to interpose a pointer list for each attribute. This list is accessed via a kat for the search key attribute and provides the actual address in the data file. The pointer list is itself a file with only two fields, attribute value and actual data address as shown in Figure 15. The value field can be used to resolve collisions, but may be omitted if few collisions are expected. In that case accesses to the main data file are needed to assure the absence of a collision.

The pointer lists arc again redundant relative to the database itself and every access to the database is mediated by them. This indirection causes some loss of speed, but generality has been gained. For each attribute that may be used for retrieval a pointer list can be created, and furthermore, since all access is indirect, the constraint imposed by direct hashing on the location of the records in the data file itself has been removed. This simplifies dealing with growing files [Fagin78]. This binding to the data file has been replaced by bindings to the auxiliary structures. Updates of the data impose matching updates of the access structure. Any change to a data value of an attribute that has been defined to have a pointer list will

require an update of an entry in this list, generally involving deletion of the old entry and insertion of the new entry.

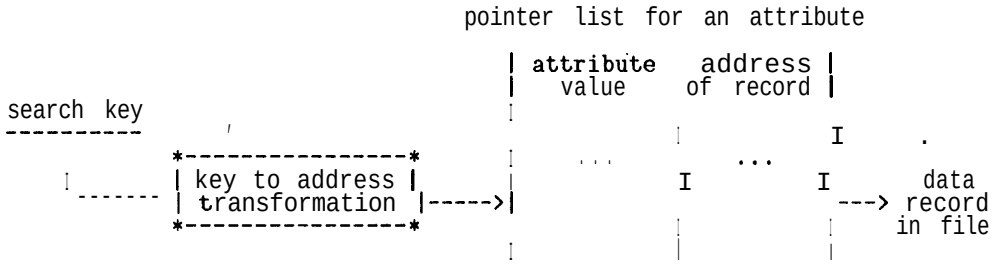
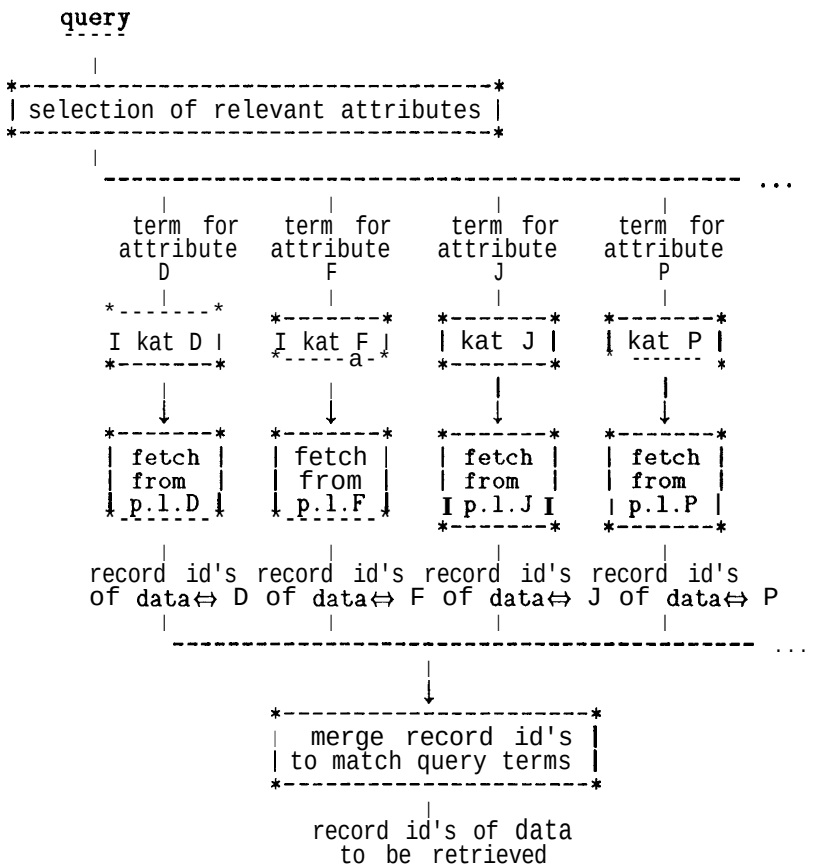


Figure 15 Direct access to a file via a pointer list

Multi-attribute queries require that several search keys, chosen from the set of candidate search attributes, are matched with item values in a file record. The match for each search attribute value can correspond to many records in the data file. We consider now three alternatives for multi-attribute retrieval. The alternatives shown use distinct degrees of binding.



Fieure 16 Multi-attribute direct access

The first alternative uses the pointer lists that were created for. retrieval by one attribute. When a query is processed the the appropriate attributes are determined, the search keys are transformed, each of the pointer lists is accessed, and lists of candidate pointers into the data file are obtained. These lists are merged as specified in the query, and the remaining addresses are used to fetch the goal records from the file. This process is illustrated in Figure **16.** The retrieval is quite rapid, although multiple access lists have to be accessed and merged. The access speed for retrieval is now a function of the number of search attributes, and the size of the address lists extracted for each search value.

The second alternative requires additional access lists, bound to specific multi-attribute queries. If, for instance queries involving all four attributes D, F, J, and P were frequent, an additinal access structure could be created by combining these four attribute values, and filling in a pointer list according to this combination. To combine the attribute values the strings of the values would be catenated (||) and the result submitted to the kat. This list would be bound a priori to any updates of these attributes, and assure fast retrieval for such four-attribute queries. An extreme case of such binding occurs if pointer lists to the file were constructed using all the possible search attribute combinations. There would be $n^2 - 1$ pointer lists for tuples having n attributes. In order to simplify access management a single file could be used for all the pointerlists and insertion of a record would create $n^2 - 1$ entries [Wong[71]]. Figure **17** gives a simple illustration of a new employee record with three potential search keys. There will be 7 entries in the combined pointer list to correspond to the seven query combinations. The combination where no search attribute value is specified, { any || any || any }, would of course retrieve the entire file.

```
          salary       skill    department

John: { 20.000  II  Welder  || Fdundry }→ kat → pointer list entry 1
      { 20.000  ||    Welder  |   any     }→ kat → pointer list entry 2
      { 20.000  II     any    | Foundry }→ kat → pointer list entry 3
      {  20.000  ||    any    |   any     }→ kat → pointer list entry 4
      {   any    ||   Welder  | Foundry }→ kat → pointer list entry 5
      {   any    ||  Welder   |   any     }→ kat → pointer list entry 6
      {   any    ||    any    | Foundry }→ kat → pointer list entry 7
```

Figure 17 Hashed entries for all partial matc.. queries to one entity

This alternative takes only a single auxiliary file access to locate a record which matches any of the potential queries. The access file will be large, much greater than the data file, and costly to maintain, whereas the space for the access lists in the first alternative is approximately equal to the data file size.

A third alternative is less extreme, and demonstrates an intermediate binding method [Rivest[76],Burkhard[76]]. Here the attributes are first processed by the kat and then the combinations are generated by catenation of the addresse: . The combined address determines the placement ‘of the record as shown in Figure 18. The address segments will be short. For up to 500 employees a nine-bit address is sufficient, so that three attributes only have to contribute three bits each.

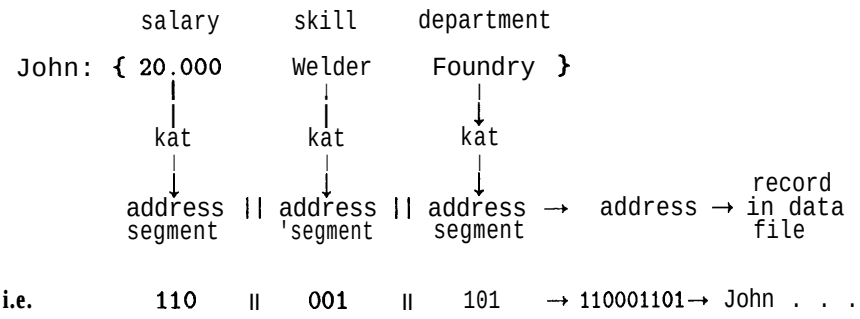


Figure 18 Address generation by segments

A partial match query creates address segments for the specified attributes, and then searches the file at all record addresses which satisfy the known address segments. If, for instance, only the salary of 20.000 and the Foundry is specified, then all candidate records will appear at addresses of the form 110 xxx 101. Records at the eight addresses (110 000 101, 110 001 101, . . . , 110 111 101} will then be retrieved, together with any linked data records due to collisions.

7. EXAMPLES OF THE RANGE OF BINDING CHOICES

In order to illustrate that the range of binding is wider than perhaps obvious from the earlier discussions, we will review some examples of extremes.

7.1 Extremes in data representation

There have been proponents of complete avoidance of binding when entering data into a database. Instances of this view have occurred in medical environments, where medical data were kept in unstructured textual form. The expectation was that, by the time the database was large enough to warrant processing for statistical or deductive inference of disease phenomena, the techniques to process natural language text would be well enough developed to permit the the appropriate analysis. It is unfortunately doubtful whether the meaning of a sentence remains the same over time. Both medicine and language can be expected to change between entry and use. Such delayed binding would hence cause a loss of classification knowledge that could have been captured when the data were entered.

The other extreme, early binding to gain maximal efficiency for processing, has been the basis for the early major database management systems. Many data base systems used in business as **IMS** (IBM), **IDS** (Honeywell), the early proposals of the Database Task Group **(CODASYL),** and the distributed systems being developed at CCA [Bernstein&Shipman^{80}] show the effects of such a philosophy. All possible paths that a query may take are pre-defined using various access mechanisms. If queries occur which need to follow some path which is yet undefined, then the database has to be reorganized or the relevant subset of the database has to be extracted and processed outside of the database management system. The reorganization of a bound database can take considerable time. An official from the Equitable Life Assurance Cornpany has stated that the reorganization of their database takes

70 hours of computer processing and hence presents a major interruption for the enterprise.

7.2 Binding in knowledge-based systems

Extreme cases of binding can also be found in systems which use the techniques of artificial intelligence on data. Quite complex network structures have been used in knowledge-based systems, such as are seen in applications involving natural language understanding [Hendrix[77]]. Implementations generally use linked lists of nodes or frames. Each node is labelled with a term and defines the term through connections with other nodes. A program can relate one concept to another by following paths of connections through the network. The connections may be labelled in order to denote appropriateness or according to strength so that choices may to be made by the processing algorithms. If the knowledge is incorporated into the network by direct insertion of nodes and connections then the structure of the resulting semantic net is very rigidly bound. Such a knowledge base is carefully crafted and rarely dynamically updated. These semantic nets have rarely exceeded a few hundred entries. In order to deal with excessive complexity due to size the net may be partitioned so that the bindings are organized in a hierarchical manner.

An alternative method for the representation of collected knowledge is found in rule-based systems. Here the relationships are expressed in symbolic form [Davis[77]]. Queries about terms in the knowledge base may require inferencing through many rules. A rule has the form:

```
IF { term-predicate rel-op state } THEN { term-result := value };
```

and any rule where the predicate is satisfied can be executed. The result of execution changes the state of the system, enabling and disabling other rules. An example of such a rule in a knowledge base about personnel management is

```
IF age = young AND salary-increase-rate > average
                                    THEN training-payoff := high ;
```

This rule may be invoked in the process of determining a promotion; initial data given may have been a `birthdate` and a salary history. Searches through the rule base to match predicate conditions of the state variables take place until the desired closures are found. In an unbound rule base the symbolic searches will require much processing effort.

The processing effort can be greatly reduced by execution of a binding phase, invoked after the knowledge is collected, but prior to its use for queries or the making of inferences. A backchaining process will scan the entire knowledge base and implement connections by setting up pointers between related nodes [van Melle[80]]. The system is now tightly bound and can be processed like a traditional semantic net. During processing new knowledge may create new potential connections, these may be left in symbolic form or may also be bound, depending on the specific system design.

Since knowledge bases work on abstractions, rather than on data, the size of the stored bases is much smaller. Because of their smallness, performance issues

in knowledge bases have not received as much attention as databases have. As knowledge-based systems increase in size, in number and in range of applications, binding issues will gain interest.

8. IMPLEMENTATION WITHIN DATABASE MANAGEMENT SYSTEMS

Traditional Database Management Systems (DBMS) are designed with the expectation that they will spend much of their time servicing known patterns of usage. Binding to accomodate these patterns is hence important.

We have described about half a dozen basic implementation structures which support differing strengths of binding. In this Section we will look at the implementation practice and draw some conclusions from these observations.

8.1 Binding Choices provided by Database Management Systems

The choice of structures, made available by the designers of the various DBMS's, gives a clue to the usage assumptions made during DBMS design. The user, having selected a DBMS, is then constrained to select among a few alternatives. In the simplest, often called user-oriented systems, there may in fact be only one choice.

Most routine production work is carried out today with designer-oriented systems. Here, before the database is initially loaded, a designer specifies the bindings to be used. The design has to satisfy both the constraints due to logical correctness requirements and provide adequate performance for all users. The choices may be difficult.

Most current production oriented systems support single (TOTAL, IMAGE) or multi-level hierarchies (System 2000, BASIS), a family of systems supports networks of multiple hierarchies based on multiple attributes (CODASYL oriented systems), and one major system, IMS, provides nearly arbitrary interlinking hierarchies using procedural linkage definitions. The table below lists some well-known systems in order of decreasing binding. We use the term record to describe a set of data elements which is manipulated as a single physical unit and made available as such to the user.

System	Technique	Effect
IBM IMS (HISAM)	Data elements and their nested descendants are packed physically adjacent, using a pre-order sequence. Additional hierarchies can be defined and use pointers.	Rapid sequential retrieval of records with nested segments, slow or limited on-line update. Users of secondary hierarchies obtain logically identical, but lower performing functions.
Cullinane IDMS (CODASYL)	Data elements and pointers to related records are placed into records. Chains are used to link nested records to each other and their owners.	Data elements in nests require an extra access step, but new subrecords can be inserted with a similar effort.

INTEL-MRI System 2000	Data elements at the same level are placed physically adjacent, nested records are placed into files with auxiliary index structures for retrieval.	Data elements in nests require an extra access step, new subrecords can be independently inserted to simplify update.
Software AC ADABAS	Data elements are placed into records which have a permanent identification. One level of nested data can exist within the record. Pointers are kept in separate pointerlists and can relate records from several relations. The lists can be maintained throughout or generated during retrieval.	Rapid processing of potentially complex multi-attribute queries that return single or a few records is done by locating the pointers and merging lists of pointers for selection of data elements.
CII SOCRATE	Relationships are described in a schema, data elements are found through hashing of symbolic names and values into a virtual memory space.	Complex structures can be implemented. Retrieval time is is proportional to the number of single elements retrieved.
RSI Oracle	Records are placed in hierarchical files, assuring physical clustering according to one access path. Queries will take advantage of available clusters without user direction.	Full generality of relational operation is provided. Retrieval times depend generally on the size of the relations, retrieval of records according to a hierarchy is rapid.
IBM System R	Records are placed in areas and physical clustering is encouraged, but not enforced. Retrieval functions optimize by rearranging query terms and selecting between index use and sort-merge procedures.	Retrieval times depend on the number of records accessed and the effectiveness of clustering. The user language contains statements to advise the system about clustering. Update of the database is rapid.

Table 4. Binding strategies in Database Management Systems

In the final two systems in the table, examples of implementations of the relational model, the bindings above the record level are used only for access purposes, and these can be varied to serve the requirements of performance, without affecting the users' interfaces.

In relational systems, which impose few update'constraints, binding is weak. In order to avoid excessive recomputation of relationships during a period where a certain set of relationships is explored, it is possible to create temporary working sets of tuples in a workspace, which are not updated when the underlying database changes. The workspace does not have to obey the rigorous rules which assure non-redundancy and structural independence [Todd[75]]. The use of temporary bound copies of data based on unbound collections of data is a very effective compromise when one wishes to have a stable collection of data during analysis. It assures consistency and integrity of the answers obtained, and allows effective summarization of past data. The process steps being executed to create the workspace can be catalogued so that, when a significant number of updates have been made to the source database, a new copy of the workspace can be created.

8 . 2 Schemas

In order to specify binding alternatives to a DBMS we give statements to a schema processor. A schema language addresses of course many other issues as well, and binding choices are only given in implicit form. We will indicate here briefly aspects of data description that reflect binding decisions as discussed above. Directives, given in the schema, which lead to physical binding include:

data value constraints
assignment of attributes to records
descriptions of hierarchical relationships
ordering specifications
references and triggering options for database procedures

Schema languages are hence the primary tool for binding specification, although the descriptive power of most of them is oriented towards implementation details. The knowledge about the relationships, described in the schema, is used to build the database.

In most implementations this knowledge becomes embodied within the database and the procedures that are generate to operate on the data. Relationships are represented by sequential placement of related records (clustering) where possible and via pointers for other linkages between data items. These linkages are then exploited during database retrieval. They reduce flexibility during update, and increase the effort expended during update. Which relationships should be bound is an implementation decision which affects the performance in a critical way and is difficult to change once the database has been loaded. A change in the description, the schema, can imply major database changes.' Other implementations keep the schema available for interpretation and place no or little linkage information within the database.

The alternatives in schema implementation are again directly comparable with the alternative binding decisions faced in program translation. When knowledge

about data relationships is derived by matching symbolic field values then an interpretation has to take place in order to locate and process the data. When the knowledge about relationships is stored within the data in the form of addresses or pointers, then the retrieval can proceed at hardware access speed. In databases where access to the data itself is relatively slow, the cost of interpretation of database manipulation commands, using a simple symbolic schema, small enough to reside in primary memory, will be quite bearable. Insufficient binding has a high cost when the search for matching symbolic values has to scan large data spaces, which are not rapidly accessible. We do assume in general that, even when the data are large, the data descriptions will be much smaller. The schemas deal in fact with abstractions, and can be viewed as knowledge bases about the database. It is hence the structure of the data themselves and not the structure of the data description which requires most attention to performance in system design. Schema languages which recognize this tradeoff have the potential to simplify database administration. The **TOD** schema language for instance allows the specification of auxiliary access structures for each attribute from the set

```
{ INDEX, RANGE, TRANSPOSED, PRIVATE }
```

and the system interprets the schema and automatically takes advantaee of these structures when they are available [Wiederhold[75]].

8.3 Related effects

The choice of binding time also has effects on the design of the mechanisms which assure the integrity of the database and the protection of the privacy of its contents. The redundancy required to provide such security provides an additional binding between data elements. The definition of access paths which delimit access privileges in order to protect privacy implies binding knowledge about data and access paths to the definition of the users. Late binding means that a sophisticated procedural interface is required to assure that privacy rules cannot be subverted. On the other hand, extreme reliance on pre-specified access paths has its own dangers. If an intruder finds an unplanned access path combination, i.e. a loophole, then violations of security may remain undetected for a long period of time.

8.4 The complexity of binding

The physical complexity of network databases with many bound relationships is frightening. Verification within the actual database of consistency, completeness, closure, and lack of circularity becomes nearly impossible. The question is now, "When does it make sense to bind relationships in a physical manner?" We have given many examples where binding improved query performance, but it is obvious that in order to have the benefit of bound linkages when responding to any possible question, the number of linkages which would have to be established among n data elements is on the order of n^2. If only direct linkages are represented, then the number will be much less than this limit, but the number of possible relationships among the n items will still greatly exceed the number of items itself.

There are however a number of issues which further reduce the number of linkages to be considered. First, we have a limitation due to imperfect foresight.

Not all of the possible relationships will be recognized at design time, and hence not be candidates for early binding. The available design methodologies for conceptual models give no guarantee of completeness. Knowledge-based systems will attempt completeness among a relatively small set of items, but do provide for additions during processing. **A** second point is that we cannot expect benefits from all bindings. In practice binding in a database is not determined by the existence of a relationship, but is only implemented when the designer foresees a processing use for the relationship. The number of implemented linkages is further reduced by having conceptual levels of binding, as indicated throughout: binding into tuples, between tuples, and between relations in the schema.

From current practice we can assess the degree of structural complexity caused by binding which is tolerable. Network structures in databases pose a typical management problem. We find that large systems have seen limited binding. Measurements of such systems have indicated on the order of twenty fields per record, with fairly wide variances. Values ranged from two to nearly a thousand. The total number of record types, or relations, varied from ten to more than a hundred, with an average of about forty. Connections between relations are described in **CODASYL** implementations by the schema through definitions of sets of linkages. Bachman[74] has reported that networks implemented using the **IDS** database system contain records that are on the average linked to two superior relations. We can consider that the key fields participate in all relationships, and the dependent fields generally in only one. Ordering specifications would cause selected fields to have a further binding implemented, and procedural restrictions, which are not easily measured, can implement reference connections in a CODASYL-based system for some fields. When these considerations are taken together it appears that the total number of implemented connections for dependent data items is still less than two, and the average over all items is not much more.

We see that, in practice, the degree of implemention of structural binding is quite limited. If we assume that experienced database designers achieve reasonably optimal databases, we can conclude that at the measured levels of binding an optirnum is reached. At that point the cost of complexity should begin to exceed the benefits of binding. We do recognize that the measurements are suspect, since the designers freedom to choose binding alternatives is severely limited in existing systems. While system limitations might reduce bindings, it may well be that in order to compensate for performance loss, alternate and perhaps additional auxiliary bindings may be chosen. The fact that any DBMS limits the choices is, of course, also due to the problems of complexity of its maintenance, which is just another aspect of the binding issues we want to address. A sirnple model will be used in the next section to demonstrate the likelihood of an optimum in degree of binding.

9. A PERFORMANCE MODEL OF BINDING

We can use the complexity concepts that have been discussed in the section above to construct a simple cost model of binding. This model will not be precise enough to provide performance estimates for database design at an engineering level, but will provide some quantitative insight into the effect of binding decisions.

9.1 A matrix representation

We consider a database D with a set of $\{e_1, \ldots e_i, \ldots e_n\}$ entries, where each of the entries e_i participates, and derives its meaning from its participation in up to p_i relationships. We permit at most one relationship between any pair {e,, e_j}. For the entire D the sum of the number of relationships p,, i = 1.. .n, is then bound by n^2, although the total will be typically much less. Instances of these relationships could be placed into a matrix Q of size n $\times$ n, which would be quite sparsely filled.

We have to collect relationship information into a processible form to make it useful for an implementation. Using the structural database model the relationships are defined in a conceptual schema by having attributes assigned to relations and by defining connections among relations. The connections specify the relating attributes. An instance of a connection between records of D is represented by explicit entries in files which correspond one-to-one with relations. We assume that all relationships are reducible to binary relationships, and hence we ignore the complexity of multi-attribute connections.

We have to augment the conceptual model with the capability to collect relationships created for access efficiency, for instance an ordering constraint placed onto a set of tuples according to sorne key. Such a constraint would be described by entries in Q that bind pairs of ruling part entries. These entries can be added to the set of potential and the set of recognized relationships for each e,.

Out of these relationships a certain number will be selected by the database designer for actual implemented binding.

As we discussed above, we cannot expect to recognize all potential relationships, ‘so we will define s_i to be the number of relationships of e_i that are recognized and noted in the conceptual and implementation schemas, and we will also define b_i to be the number which are actually bound. From the discussion in Sect. 5.1 we also recall that each element in a database has to be associated with at least one other element.

There is hence an ordering of the number of binding choices for an element e_i

$$1 \leq b_i \leq s_i \leq p_i < n$$

with typically $p_i \ll$ n. Unfortunately, the true value of p_i is never known, so that we will continue to carry n as an absolute upper bound.

We considered conceptually that binding among all pairs of elements in E is defined by entries in a matrix Q, with elements q_{ij}. The states taken by any element q_{ij} are one of the set $\{bound, recognized, potential, null\}$. Being $bound$ implies *recognized,* and *recognized* implies *potential.* We note that the matrix is not symmetric about the diagonal since there are binding techniques, such as hashed access, which cannot be exploited in a symmetric fashion. The number of entries in q_{ij}, j = 1. . .n, which are not null, is equal to p_i.

9.2 A performance analysis

Benefits are obtained from binding when the database is used for retrieval, and additional costs are incurred for binding at update time. Each bound linkage is equivalent in this first order cost model and brings equal benefits at equal costs. These

simplifications appear drastic, but these assumptions are for instance approximated by a CODASYL approach without clustering and without directly accessed entry points. We will furthermore deal with an environment where n, the size of the database, remains constant, but relationships are updated as entities change.

Updates in this model incur costs U proportional to the extent of binding of the elements, so that for element e_i

$$U_i = ub_i, \qquad 1 \leq b_i \leq p_i < n$$

where u is the unit update cost.

Typically an update is preceded by a retrieval operation in order to locate the element which is to be changed. This cost is not included in U, but is seen as part of the retrieval effort.

The relationships which are bound during updating are used at retrieval time. Since there are b_i entries associated with every element e,, the decision as to which connection to follow, partitions the remaining search space into *n/b,* subspaces. In an optimally structured database each successive decision will further partition the previously selected subspace. In order to find a destination element in D the number of decisions is hence $log_{b_i} n$, where i identifies the nodes on the retrieval path. We assume now that there is indeed a bound path to be followed from every node that has been reached. This is typically true for routine processing in databases where performance was considered in the design phase, and those are the databases measured earlier.

The cost of a decision at e,, regarding which path of the b_i alternatives to follow, is also a function of *b,*. Since the choices are typically few, and of different types we will assume that the cost of making the choice is proportional to *b,*. The retrieval cost R is then

$$R_i = rb_i log_{b_i} n, \qquad 1 \leq b_i \leq p_i < n$$

with r as the unit retrieval cost.

In most databases, retrieval transactions occur with a greater frequency than updates. A ratio f of 10 is not unusual for a dynamic database, and for static databases the number can be much higher. At the same time the cost of an update operation on an element is typically much higher than extracting a value. An optimistic ratio for u/r is 2, but a ratio u/r of 10 is not out of range. In order to combine the effect of U and R we make the following assumptions:

1 an update is always preceded by a retrieval.

2 two cases of retrieval activity, respectively 9 of 10 and 99 of 100 transactions.

3 two cases of update cost ratios u/r, 2 and 10.

Figure 19 shows the behavior of U and R for a database of n = 100 000 entries for values of the average binding B of b_i ranging from 1.. .6. The values of the total cost $T_f = U + f*R$ were computed for both retrieval to update activity ratios and for both update cost ratios.

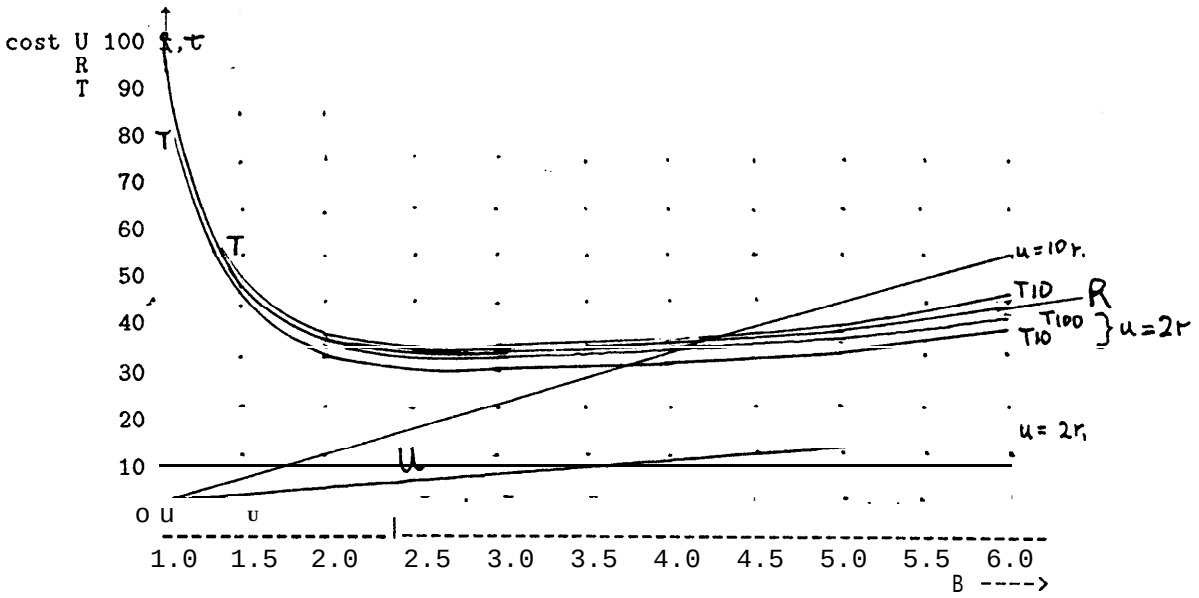


Figure 19 Tradeoff between binding cost and benefits

The aggregates are plotted in Fig. 19 as T10 and T100, scaled to a single transaction cost basis. T10 is plotted for update cost ratios u/r, of 2 and 10, for T100 the difference is negligible. We notice first that the optima for the combinations considered are achieved at values of B ranging from 2.5 to 2.75. This range is suprisingly narrow. We see furthermore that an reasonable overall performance, that is within 20% of the optimum for each case, is obtained within a range corresponding to $1.5 < B < 5$, and that further bindings do not have a high cost penalty.

These observations support a view that the desigriers of current applications have a reasonable intuition about binding levels. We do need to recall again **that** we assumed perfect useability of bindings for retrieval, and this is only achieved in practice when processing known transaction patterns. A greater variety of query types would tend to increase the number of bindings so that the useful bindings can remain at the optimal level.

We assumed in the evaluations above that the decision cost at a node was proportional to the number of available choices. This is a a reasonable assumption in network structures. Several of the access structures that are being introduced to a greater *extent into databases make use of physical clustering, such as indexes, transpositions, or hierarchies defined by the users. In such structures, once the physical access has been made, a series of decisions can be evaluated at little incremental cost. To assess these conditions we looked also at a model which did not assume homogeneity of binding. To capture the fact that access to physically adjacent data is an order of magnitude more rapid than randorn access, the cost of making decisions at a node was set to be proportional to $1 + (B - 1)/100$, so that now

$$R' = r(1 + (B - 1)/100)log_B n$$

For the same conditions, with u/r = 10 the optimum point for 90% retrieval activity (f = 10) occurs now at B = 4.5 and for 99% retrieval (f = 100) transactions at B = 14. A tentative conclusion for such databases is that a greater extent of binding may be appropriate for such systems. Unfortunately there seem to be few measurements of current practice in these systems. R'database using the transposed system cited in Sect. 6.1 shows a value of B = 12 with a value of f estimated at *20.*

The model presented can be similarly exploited to assess effects of database size, update to retrieval ratio, other ratios of update to retrieval transaction costs. While this model is certainly simplistic, it allows a qualitative assessment of binding choices, and satisfies the objective of summarizing some important design choices in a broad, conceptual manner.

10. CONCLUSION: BINDING AS A SYSTEM DESIGN CONCEPT

Via this circuitous route, we have returned to the issue which was used to illustrate binding in information systems initially. This issue was: "To what extent should relationships between data, that are recognized early, be specified so that they are implemented at data entry time?" Insertion of binding information into the database with each update implies the use of coding, pointers, and physical adjacency. The alternative keeps the data, to the greatest feasible extent, in the form and order that it is obtained. Relationships are established by search when the query demands. An important intermediate stage is seen in relational database systems where the data is formalized and structured into relations containing coded data in identical formatted tuples, and any inter-relation binding is deferred to query execution time. The optimal organization of data into tuples is not obvious, and the rationale for any particular compromise is a mix of theoretical and implementation considerations.

Looking at information and database systems from a standpoint of binding time provides a gross picture and obscures many of the detailed considerations required to actually implement databases. 'Binding does provide a point of view which cuts across many structural levels which have to be considered: data representation, data structuring, file design, and information structuring; and when used as a guideline in the design of systems can assist in providing a consistent design philosophy which is difficult to achieve by other means. At each of the structural levels, the same basic relationships hold: efficiency is obtained by aggregation and by linking, flexibility is obtained by using atomic concepts and avoidance of binding.

We can observe failures in system design where the lack of a central direction which considers the effects of two concepts has led to unbalanced systems: we find systems where the programming and data representation is extremely flexible and the data structure is inflexible. In these systems the extensive knowledge captured in the databases can only be extracted with the help of experts and at great costs. We see similarly systems where flexibility in structuring cannot be exploited because of inflexibility in data representation concepts or information system concepts. Now the data are accessible, but nothing of interest can be found.

Another system design attitude can be described by the sequence: build it, measure it, tune it. The flexibility of modern software encourages this approach.

However, measurement as design tool for information systems has to be used with care. If lack of binding causes a high cost of retrieval, then retrieval usage will be discouraged, so that updates will predominate. System measurements taken at that point will indicate that the system is optimally designed, since a high update to retrieval ratio favors minimal binding.

We hence advocate the concept of binding as the primary tool for system design and evaluation. Issues to be resolved in the design of information systems are of two types. It is important to assure that decisions at one level do not bind the system to an extent that flexibility designed into other levels cannot be utilized. The other issue is that the flexibility, associated with lack of binding, can be costly, and should be invested at the most profitable stage.

The design of systems to generate information involves a wide spectrum of expertise and takes place over along timespan. The breadth that the concept of binding brings to system design can give individuals of differing background a focus in a joint system design effort.

ACKNOWLEDGEMENTS

This work was initiated while receiving a fellowship from the National Library of Medicine and is now supported in part by the Knowledge Base Management Systems Project, under contract N39-80-G-132 from the Defense Advanced Research Projects Agency of the United States Department of Defense. Efforts on Programming Language Systems are supported through subcontract P9083403 from the Lawrence Livermore Laboratory, serving the S-l multiprocessor architecture project for the Advanced Digital Processor Technology Base Development for Navy Applications, ONR order no. N14-78-F23. The views and conclusions contained in this document are those of the author and should not be interpreted as representative of the official policies of the US Government or its agencies.

I wish to thank especially the many colleagues who have been willing to discuss system design and operational issues in the terms presented in this paper, and have provided the questions and feedback neccessary to clarify the principles in my mind.

The camera-ready final manuscript was prepared with the aid of TEX [Knuth79].

REFERENCES

Abrial(74) J.-R. Abrial: "Data Semantics"; Data Base Management, (Klimbie&-Koffeman, cds), North-Holland 1974.

Aho&Ullman(77) A.V. Aho and J.D. Ullman: Principles of Compiler Design ; Addison- Wesley, 1977

Allen(78) J. Allen: Anatomy of LISP; McGraw-Hill, 1978

Astrahan(76) M.A. Astrahan et al: "System R: A Relational Approach to Database Management"; ACM Transactions on Database Systems, vol. 1, no. 2, June 1976, pp. 97-137.

Bachman(72) C.W. Bachman: "The Evolution of Storage Structures"; CACM, vol. 16, no. 11, Nov. 1973, pp. 653-658.

Bachman(74) C.W. Bachman: "Implementation Techniques for Data Structure Sets"; Data *Base* Management *Systems,* (Jardine, ed), North Holland 1974.

Bayer(70) R. Bayer: "Symmetric Binary B-trees – Data Structure and Maintenance Algorithms "; *Acta Informatica,* vol. 1, no. 4, Dec. 1972, pp. 290-306.

Bernstein&Goodman(80) P.A. Bernstein and N. Goodman: "What Does Boyce-Codd Normal Form Do"; *Proc. of the 6th VLDB,* (Yao, ed), IEEE, 1980, pp. 245-259.

Bernstein&Shipman(80) P.A. Bernstein, D.W. Shipman, and J.B. Rothnie: "Concurrency Control in Distributed Databases (SDD-1)"; *ACM* Transactions *on* Database *Systems,* vol. 5, no. 1, March 1980, pp. 18-51.

Breitbard(68) G.Y. Breitbard and G. Wiederhold: "PL/ACME, An Incremental Compiler for a Subset of PL/1"; Information Processing 68, Proc. 1968 IFIP Conf., North Holland 1969, pp. 358-363.

Burkhard(76) W.A. Burkhard: "Hashing and Trie Algorithms for Partial-Match Retrieval"; *ACM* Transactions *on* Database *Systems,* vol. 1, no. 2, 1976, pp. 175-187.

Bush(79) R. Bush: *USAMINT-A* U-Code Assembler *and* Interpreter ; Stanford Univ. Computer System Lab. Technical Note no. 160, 1979.

Chen(76) P.P.S. Chen: "The Entity-Relationship Model: Toward a unified View of Data"; *ACM* Transactions *on* Database *Systems,* vol. 1, no. 1, March 1976, pp. 9-36.

Church(41) A. Church : "The Calculi of Lambda-conversion "; Annals of Mathematics Studies, Princeton Univ. Press, 1941.

Church(56) A. Church: Introduction to Mathematical Logic ; Princeton Univ. Press, 1956.

Codd(70) E.F. Codd: "A Relational Model of Data for Large Shared Data Banks "; *CACM,* vol. 13, no. 6, June 1960, pp. 377-387.

Codd(72) E.F. Codd: "Further Normalization of the Data Base Relational Model "; Data *Base* Systems, (Rustin, ed), Prentice Hall, 1972, pp. 33-64.

Davis(77) R. Davis and J. King: "An Overview of Production Systems"; Machine *Intelligence* 8, (Elcock&Michie, eds), Ellis Horwood, 1977, pp. 300-332.

Daley(68) R.C. Daley and J.B. Dennis: 'Virtual Memory, Processes, and Sharing in MULTICS"; *CACM,* vol. 11, no. 5, May 1968, pp. 306-312.

Dijkstra(62) E.W. Dijkstra: A Primer *of* ALGOL *60* Programming ; Academic Press, 1962 (translation of an earlier manual, written in Dutch, for the ALGOL translator for the Xl computer at the Mathematisch Centrum in Amsterdam, 1961).

Duda(78) R.O. Duda, P.E. Hart, N.J. Nilsson, and G.L. Sutherland: "Semantic Network Representations in Rule-Based Inference Systems"; *Pattern-*Directed Inference Systems, (Waterman&Hayes-Roth), Academic Press, 1978, pp. 203-221.

ElMasri(80) R.A. ElMasri and G. Wiederhold: "Properties of Relationships and Their Representation"; *Proc. of the 1980 NCC,* AFIPS vol. 49, pp. 319-326.

Elson(73) M. Elson: Concepts of Programming Languages ; Science Research Associates, 1973.

Fagin(78) R.Fagin et al : "Extendible Hashing, A Fast Access Method for Dynamic Files"; IBM Research Report *RJ2305,* July 1978.

Feldman(69) J.A. Feldman and P.D. Rovner : "An Algol-Based Associative Language"; CACM, vol. 12, no. 8, Aug. 1969, pp. 439-449.

Feustel(73) E.A. Feustel: "On the Advantages of Tagged Architecture"; IEEE Transactions on Computers, vol. C-22, no. 7, July 1973, pp. 644-656.

Flynn(77) M. J. Flynn : "The Interpretive Interface: Resources and Program Presentation in Computer Organization"; *Proc.* of *the Symp. on High-speed* Computers *and* Algorithm Organization, Univ. of Illinois, 1977, pp. 41-69.

Garcia(80) H. Garcia-Molina and G. Wiederhold: Read-Only Transactions *in* a Distributed Database; Report TR 267, Dept. of El.Eng., Princeton Univ. and STAN-CS-80-797, Stanford, April 1980.

Gotlieb(75) L.R. Gotlieb : "Computing Joins of Relations"; *Proc. of the 1975 SIGMOD conf.*, (King, ed), pp. 55-63.

Graham(75) R.M. Graham: Principles of System Programming ; Wiley, 1975.

Gray(76) J.N. Gray, R.A. Lorie, G.R. Potzulo, and I.L. Traiger: "Granularity of Locks and Degrees of Consistency in a Shared Database"; Modelling in Database *Systems,* North Holland 1976, pp. 365-394.

Hammer(78) M. Hammer and D. McLeod : "The Semantic Data Model: A Modelling Mechanism for Data Base Applications"; *Proc.* of the *1978 ACM SIGMOD,* (Lowenthal&Dale, eds), pp. 26-36.

Hellerman(75) H. Hellerman and T.F. Conroy: Computer *Systems* Performance ; McGraw-Hill, 1975.

Hendrix(77) G.G. Hendrix : "Expanding the Utility of Semantic Networks through Partitioning"; *SRI* Artificial *Intelligence* Group tech. note *105,* June 1977.

Jensen(61) J. Jensen and P. Naur: "Call by name, An Irnplementation of ALGOL 60 Procedures"; *BIT,* vol. 1, no. 1, 1961, p. 38.

Kaplan(79) S.J. Kaplan : "Cooperative Responses from a Portable Natural Language Data Base Query System"; Ph.D. Dissertation, University of Pennsylvania, July 1979.

King(80) J.J. King : "Modelling Concepts for Reasoning About Access to Knowledge"; *Proc.* NBS-ACM Workshop OR Databases and Conceptual *Modelling,* SIGPLAN Notices, vol. 16, no. 1, Jan. 1981, pp. 138-140.

Knuth(73) D.E. Knuth : *The* Art of Computer Programming – *Fundamental* Algorithms, *2nd ed. ;* Addison Wesley, 1973.

Knuth(79) D.E. Knuth: TEX *and* METAFONT ; Digital Press, 1979.

Komaroff(79) A.L. Komaroff: "The Variability and Inaccuracy of Medical Data "; *Proc. of the IEEE,* vol. 67, no. 9, Sept. 1979, pp. 1196-1207.

Lanksy(80) A. Lansky : PASMAC, Macro Processor *for* PASCAL ; CSL Technical report 162, Stanford University, July 1970.

McCarthy(65) J. McCarthy: *The* LISP 1.5 Programming Manual ; MIT Press 1965.

McGee(77) W.C. McGee: "The Information Management System IMS/VS"; *IBM* Systems Journal, vol. 16, no. 2, 1977, pp. 84-168.

Nori(75) K. Nori, U. Ammon, K. Jensen, et al: Pascal Compiler *Implementation* Notes ; ETH Zurich, 1975.

Organick(78) E.I. Organick, A.I. Forsythe, and R.P.Plummer : Programming *Language Structures* ; Academic Press, 1978.

Quillian(68) M.R. Quillian: "Semantic Memory"; Semantic Information *Processing,* (Minsky, ed), MIT Press, **1968,** pp. 227-270.

Rivest(76) R.L. Rivest : "On Self-Organizing Sequential Search Heuristics"; *CACM,* vol. **19,** no. 2, Feb. 1976, pp. 63-67.

Sandewall(78) E. Sandewall: "Programming in an Interactive Environment, the **LISP** Experience"; Computing *Surveys,* vol. 10, no. **1,** March 1978, pp. 35-71.

Schkolnick(75) M. Schkolnick: "The Optimal Selection of Secondary Indices for Files" ; Information Systems, vol. 1, March 1975, pp. 141-146.

Scott(72) D. Scott: "Mathematical Concepts in Programming Languages"; *Proc. of the 1972 SJCC. AFIPS vol. 40,* pp. 225-234.

Smith(70) J. W. Smith: "JOSS-II Design Philosophy "; Annual *Review of* Automatic *Programming,* vol. 6, Pergammon Press **1970,** pp. 183-256.

Smith(77) J.M. Smith and D.C.P. Smith: "Database Abstractions: Aggregation and Generalization "; *ACM* Transactions *on* Database *Systems,* vol. 2, no. 2, June 1977, pp. 105-133.

Stonebraker(75) M. Stonebraker : "Implementation Techniques of Integrity Constraints and Views by Query Modification "; *Pro.. of the 1975 SIGMOD* Conference, (W.F. King ed), ACM, pp. 65-78.

Todd(75) S.J.P. Todd: "Peterlee Relational Test Vehicle PRTV, A Technical Overview" ; *IBM Scientific Center* Report *UKSC75,* Peterlee, England, July 1975, Abstract in *Proc. of the* First *VLDB* (Kerr, ed), **1975,** pp. *554-556.*

vanMelle(80) W. van Mellc: A Domain-Independent *System That Aids in Constructing Knowledge-Based Consultation* Programs ; Ph.D. Dissertation, Stanford University, CS Tech. Rpt. 80-820, **1980.**

Wegner(68) P. Wegner: *Programming Languages, Information Structures, and Machine Organization* ; McGraw-Hill, **1968.**

Whang(81) K. Whang: *Separability – An Approach to Physical Database Design* ; to be published as a Stanford Technical Report, **1981.**

Wiederhold(73) G. Wiederhold: "The Need and a Method to Obliterate Control Languages"; *SIGPLAN Notices,* vol. **8,** no. **9,** Sept. **1973,** pp. **140-141.**

Wiederhold(75) G. Wiederhold, J. Fries, and S. Weyl: "Structured Organization of Clinical Data Bases"; *Proc. 1975 NCC,* AFIPS vol. 44, pp. 479-486.

Wiederhold(77) Gio Wiederhold: *Database Design* ; McGraw-Hill, **1977.**

Wiederhold(79) G. Wiederhold and R. El-Masri: "The Structural Model for Database Design "; *Entity-Relationship Approach to Sys tern Analysis and Design,* (Chen, ed), North-Holland 1980, pp. 237-257.

Wong(71) E. Wong and T.C. Chiang: "Canonical Structure in Attribute Based File Organization"; *CACM,* vol. 14, no. 9, Sept. **1971,** pp. 593-597.

Wulf(71) W.A. Wulf, D.B. Russell and A.N. Habermann: **"BLISS:** A Language for System Programming"; *CACM,* vol. 14, no. 12, Dec. 1971, pp. 780-790.