Storage and Access Structures to Support a Semantic Data Model

Arvola Chan
Sy Danberg
Stephen Fox
Wen-Te K. Lin
Anil Nori
Daniel Ries

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Abstract

This paper describes the design of storage and access structures for a high performance Ada* compatible database management system. This system supports the database application programming language ADAPLEX [Smith81, Smith82], which is the result of embedding the database sublanguage DAPLEX [Shipman81] in the general purpose language Ada [DoD80]. A prominent feature of the underlying data model is its support for generalization hierarchies [Smith77] which are intended to simplify the mapping from conceptual entities to database objects. An in-depth discussion of the rationale behind our choice of storage and access structures to support semantics intrinsic to the data model and to permit physical database organization tuning is provided in this paper.

1. INTRODUCTION

We are presently engaged in the development of a distributed database management system that is compatible with the programming language Ada [DoD80]. This system supports the general purpose database application programming language ADAPLEX [Smith82], which is the result of embedding the database sublanguage DAPLEX [Shipman81] in Ada. This DBMS is intended to go beyond systems like INGRES and System R, which are based on the older relational technology, in terms of modelling capabilities and ease of use. Two versions of the DBMS are being developed. A centralized DBMS, called the Local Database Manager (LDM), is designed for high performance and for use as a stand-alone system. A distributed DBMS, called the Distributed Database Manager (DDM), interconnects multiple LDMs in a computer network in order to provide rapid access to data for users who are geographically separated. This paper describes the set of storage and access structures supported in the LDM implementation.

The version of DAPLEX used in the formation of ADAPLEX is a simplification of the language described in [Shipman81]. However, all the key concepts have been retained. The semantics of database structure is defined in terms of entity types and relationships between entity types. Aside from the use of functional notations for expressions that significantly enhance the naturalness and readability of programs, the most prominent language feature that distinguishes ADAPLEX from other database languages is its support for the notion of generalization hierarchies [Smith77]. In this paper, we present our design for a set of storage and access structures that supports semantics intrinsic to the data model and permits the tuning of physical database organization. Section 2 provides a summary of the data model underlying the ADAPLEX language. Section 3 identifies our design objectives and presents an in-depth discussion of the rationale behind our design decisions.

2. DATA MODEL SUMMARY

The basic modelling constructs in ADAPLEX are entities and functions. These are intended to correspond to conceptual objects and their properties. Entities with similar generic properties are grouped together to form entity sets. Functions may be single-valued or set-valued. They may also be total or partial. Each (total) function, when applied to a given entity, returns a specific property of that entity. Each property is represented in terms of either a single value or a set of values. Such values can be drawn from noncomposite, Ada-supported data types and character strings, or they can refer to (composite) entities stored in the database as values.

This research was jointly supported by the Defense Advanced Research Projects Agency of the Department of Defense and the Naval Electronic Systems Command under Contract Number N00039-80-C-0402. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Naval Electronic Systems Command, or the U.S. Government.

*Ada is a trademark of the Department of Defense (Ada joint program office).

Consider a university database modelling students, instructors, departments, and courses. Figure 2.1 is a graphical representation of the logical definition for such a database in ADAPLEX. The big rectangles depict (composite) entity types and the smaller rectangles indicate (noncomposite) Ada data types. The single and double arrows represent respectively single-valued and set-valued functions that map entities from their domain types into their corresponding range types.

One notable difference between the data model underlying ADAPLEX and the relational data model is that referential constraints [Date81], which are extremely general and fundamental in database applications but not easily specifiable in relational contexts, are directly supported in ADAPLEX. In other words, the definition of the range of a function in our model is much more precise than the definition of the domain of a column in the relational model. At the same time, for functions that range over noncomposite values, we are able to exploit Ada's type definition facilities and avoid the need to introduce a separate domain definition facility [McLeod76], as has been proposed for a relational environment.

In relational systems, a real-world entity that plays several roles in an application environment is typically represented by tuples in a number of relations. In the example university database, we might have an instructor named John Doe and a student also named John Doe, who are in fact the same person in real life. In this case, we might want to impose the constraint that the age of John Doe as an instructor should agree with the age of John Doe as a student. This constraint can be more simply expressed in ADAPLEX by declaring a new entity type called person, indicating that student and instructor are subtypes of person, and that age is a function applicable to person. The function inheritance semantics of ADAPLEX automatically guarantees the consistency of age information on student and age information on instructor since age is a function inherited from the supertype person. At the same time, inherited functions can be applied directly to an entity in ADAPLEX data manipulation constructs, without the need for tedious explicit joining operations. Figure 2.2 is a graphical representation of the revised database definition. The double-edged arrows represent is-a relationships (e.g., each student is-a person). A person entity has properties common to both student and instructor entities, specifically name and age. Each student entity not only possesses properties specific to student (i.e., enrollments and advisor), but also inherits the properties of name and age by virtue of being a person. Similarly, each instructor entity has properties specific to instructor (i.e., dept and rank), in addition to the properties name and age inherited from being a person. The actual ADAPLEX syntax used in the definition of this database is shown in Figure 2.3. Notice that the degree of overlap between the extents of two entity types is explicitly constrained. Such overlaps can be total or partial. The overlapping of the person, student, and instructor entity sets in the above example is illustrated graphically in Figure 2.4. The outer circle represents the set of person entities. The two inner circles represent the subset of person entities that are also student entities and instructor entities, respectively. The intersection of these two inner circles represents the

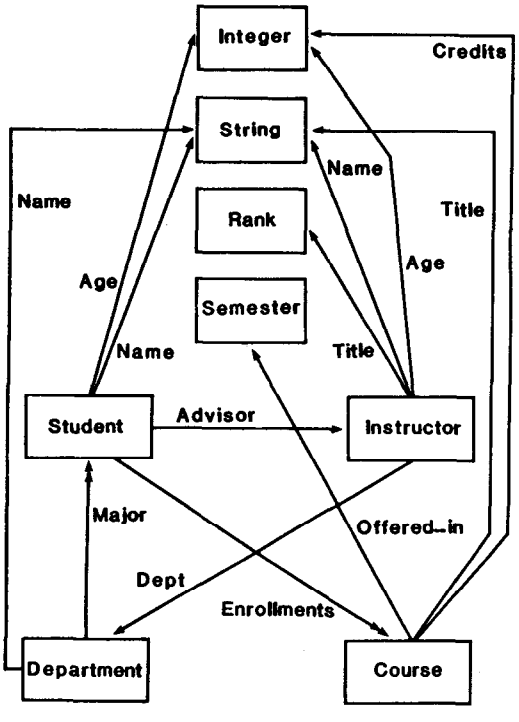

Figure 2.1 An ADPAPELX Database

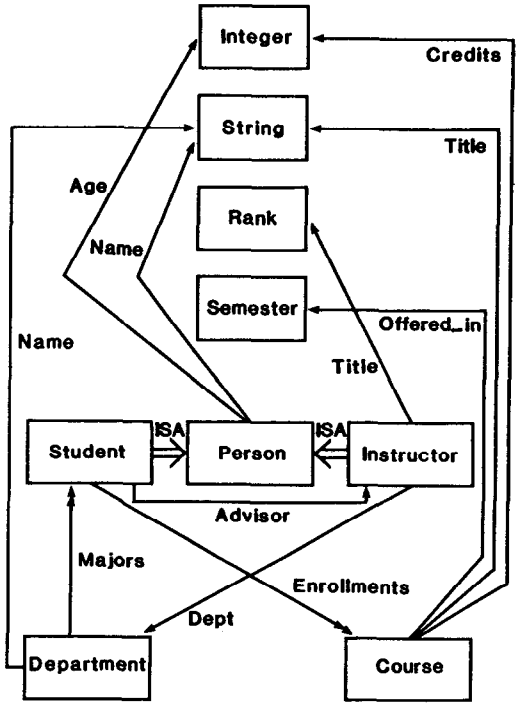

Figure 2.2 An ADPAPELX Database with Type Overlap

```
database UNIVERSITY is

type RANK is
  (ASST_PROF, ASSOC_PROF, FULL_PROF);

type SEMESTER is
  (F, W, S);

type DEPARTMENT;

type COURSE;

type PERSON is entity
  NAME: STRING(1..30);
  AGE: INTEGER;
end entity;

type INSTRUCTOR is entity
  TITLE: RANK;
  DEPT: DEPARTMENT;
end entity;

type STUDENT is entity
  ADVISOR: INSTRUCTOR partial;
  ENROLLMENTS: set of COURSE;
end entity;

type COURSE is entity
  TITLE: STRING(1..30);
  OFFERED_IN: SEMESTER;
  CREDITS: INTEGER range 1..4;
end entity;

type DEPARTMENT is entity
  NAME: STRING(1..30);
  MAJORS: set of STUDENT;
end entity;

unique NAME within PERSON;
unique NAME within DEPARTMENT;
unique TITLE within COURSE;
contain INSTRUCTOR in PERSON;
contain STUDENT in PERSON;
share INSTRUCTOR with STUDENT;

end UNIVERSITY;
```

Figure 2.3 Definition of an Example Database

subset of person entities that are both student entities and instructor entities.

Aside from general integrity constraints that may be explicitly declared as part of the database definition and that are enforced at the end of each database transaction, there are a number of invariant properties implied by the data model. These latter are in some sense treated as being more fundamental. Their validity is enforced at the end of each user-specified database interaction, rather than at the grosser transaction level. These fundamental constraints include:

- Referential/range constraint. The range of an entity-valued function may be another entity type in the database. When an entity of the latter type is deleted, it is necessary to ensure that there are no dangling references. For scalar and string functions, Ada provides the facilities for constraining the range of possible values in the underlying value set. For example, the range of integers, the precision of real numbers, and the enumeration of values in a discrete type can all be defined.

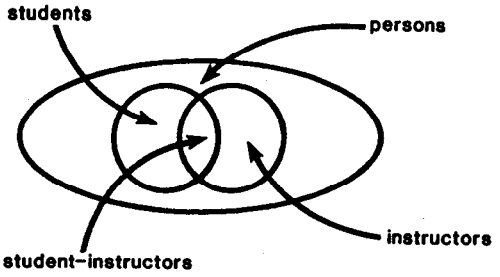

Figure 2.4 Example of Entity Set Overlap

- Extent overlap constraint. An entity can be included into the extent of an entity type only if overlaps among the extents of all of the types to which it currently belongs, and the extent of the type to which it is to be included, are permissible. At the same time, excluding an entity from the extent of a specified type will also exclude it from the extent of all subtypes whose extents are completely contained in the extent of the type in question.

- Totality constraint. A total function must be defined for all elements in its domain at all times; when a new entity is created, all of its values for various total functions must be known.

- Uniqueness constraint. One or more groups of single-valued functions within an entity type may optionally be declared to be unique. That is, each group of functions will yield distinct combinations of values when applied to distinct entities of the underlying type. This type of constraint is enforced automatically on insertions and updates.

This concludes our overview of the ADAPLEX data model. The interested readers are referred to [Smith81, Smith82] for more details on the syntax and semantics of the ADAPLEX language.

3. STORAGE AND ACCESS STRUCTURE DESIGN

Our choice for the set of data structures and implementation options to incorporate in the LDM has been motivated primarily by three considerations:

- Support for high-level ADAPLEX modelling constructs. Our data model provides several functional capabilities not supported by models used in contemporary systems. In particular, we need to devise new structures to efficiently represent information concerning entities that belong to multiple overlapping entity types.

- Maintenance of fundamental semantic integrity constraints. The underlying data model implies several fundamental constraints that must be enforced on database updates. Because of the universal nature of these constraints, it is desirable to design special structures to facilitate their enforcement.

- Performance tuning. Since different structures and implementation options are best suited for different patterns of use, efficiency can be attained only through organization tuning. We seek to achieve good performance by providing the database designer with a good range of implementation alternatives that he can choose to match against the requirements of his applications.(1)

We are assuming an environment where the bulk of the database is stored on conventional block-oriented storage devices. In this context, two fundamental design issues are: the appropriate clustering of information often used together to maximize the locality of reference, and the efficient support for frequently traversed associative access paths to minimize the amount of sequential searching required. More specifically, we are concerned with:

- Grouping of information concerning entities into logical records. Logical records of the same type are assumed to store the same set of fields.

- Placement of logical records into physical files. Each file is a linear address space that is mapped into physical blocks of storage devices. Logical records of the same type may optionally be divided into groups, each of which may then be stored in different files, possibly using different placement strategies. We will refer to each of these groups of logical records as a storage record type. Different storage record types that originate from the same generalization hierarchy may also be stored in the same file to achieve the desired clustering of information.

- Support for efficient associative access to stored records. The primary organization or placement strategy for the stored records in a file will determine the primary access path to these records. In addition, auxiliary access structures can be maintained in order to provide direct access based on secondary key fields that are not used to determine record placement.

3.1 Representing Entities and Entity Functions

The basic modelling concepts in ADAPLEX are those of entities and entity functions. To represent entity functions (in particular, entity-valued functions), it is important that entities be uniquely identifiable. However, the data model does not require that each entity be uniquely identifiable externally. That is, for entities of a given type, there does not necessarily exist a function (or a combination of functions) that yields a distinct value (or a distinct combination of values) when applied to each of the entities. Therefore, for internal unique identification purposes, an entity identifier is assigned to each entity upon creation. This entity identifier then serves to stand for the entity in the representation of functions.(2)

The set of functions that are applicable to an entity depends on the entity type(s) to which it belongs. Three different categories of information about an entity need to be stored:

- Values for applicable functions. This corresponds to values for attributes relevant to the entities in question and is typical of information accessed by applications in current database systems.

- Typing information. Given an entity, it is often necessary to determine the set of entity types (among a set of overlapping types) to which it belongs. Such a capability is essential for determining whether a function can legally be applied to the entity on hand. For example, in looping through entities of the type person, it is legal to apply the enrollments function to an entity only if that entity is also included in the type student.

- Additional control information. The deletion semantics of ADAPLEX requires that upon excluding an entity from an entity type, that entity must no longer be referenced by other entities (i.e., it is no longer in the active range of entity-valued functions). An efficient way to check for the satisfaction of such constraints is through the maintenance of reference counts that indicate the number of times each entity is referred to by entity functions, one for each entity type to which it belongs.(3)

Below, we describe our representation schemes for the above categories of function. We will first describe the mapping of entity functions into logical records and then introduce the notion of an entity directory as a receptacle for the remaining typing and reference count information.

3.1.1 Mapping Function Values Into Logical Records

As mentioned earlier, an important performance consideration is the clustering of information often needed together. In terms of the representation of functions, there are a number of obvious clustering alternatives:

---

(1) Our desire for tunability must, however, be balanced against the complexity and size of required software. Besides, in the absence of powerful design aids, we must ensure that the design freedom we provide to designers can be exploited effectively.

(2) Of course, given an entity identifier, it should be possible to obtain efficiently all information known about the corresponding entity.

(3) An exclusion operation is legal only if the corresponding reference count is zero.

- The no grouping approach. Each entity function is stored as a binary relation (i.e., a two-attribute file).
- The complete grouping approach. The values for all functions that are applicable to an entity (independent of entity types within a generalization hierarchy) are stored in the same record.
- The semantic grouping approach. The values for all (noninherited) functions applicable to an entity from the viewpoint of a particular entity type are stored in the same record.
- The arbitrary grouping approach. The values for functions applicable to an entity are stored in an arbitrary number of records to suit the usage pattern.

Our decision here is to use a combination of the semantic grouping approach and the no grouping approach. As a default, we will use the semantic grouping approach and store values for all noninherited applicable functions from the same entity-type viewpoint in the same record. In cases where we have arbitrarily long (repeating/varying length) fields that might complicate storage allocation, we provide for the option of storing such fields as individual secondary records. Our rationale for such a choice is that while the no grouping approach results in an overly fragmented database,(4) the complete grouping approach has the opposite effect.(5) As we shall see later, when coupled with the horizontal partitioning and clustering options, our approach is flexible enough to permit the grouping together of all information known about all entities of a given type, while being completely isolated from other irrelevant information.(6) By clustering all of the record types that store information on a set of entities from different viewpoints, an organization that approximates the complete grouping approach can also be obtained as a special case. Finally, we disallow arbitrary grouping of functions because we fear that this may result in too enormous a physical design space, one which a human database designer may not be able to utilize effectively. Besides, a significant increase in software complexity may also result.

In summary, to store the values of functions applicable to entities, there will be one primary logical record type corresponding to each entity type. Typically, each primary logical record includes one field for the identifier of the entity being represented, and a number of repeating or nonrepeating fields for each set-valued or single-valued function (as applied to the entity in question and not specified for separate representation). In addition, there may be zero or more secondary logical records for separately represented functions. Only the primary logical records may be considered for further horizontal partitioning and clustering. Each type of secondary logical record will be stored as a separate two-attribute file(7) that will permit efficient associative access based on entity identifiers. In case an entity belongs to multiple entity types, there will be one primary logical record for each entity type to which it belongs.

3.1.2 Entity Directory

To keep the remaining information concerning entities, an entity directory is maintained for each generalization hierarchy. The information stored in the entity directory is essentially redundant and can be obtained through sequential searching of logical records that represent entities. The purpose of the entity directory, however, is to centralize all information known about entities in order to permit efficient access. In the entity directory, there will be one entry for each entity that belongs to at least one of the types in the underlying generalization hierarchy. In addition to the typing information and the reference count information, the directory entry for each entity will also contain physical pointers to the primary storage records that store values for applicable functions, one for each entity type to which it belongs. Thus, given an entity identifier, all stored information concerning the entity can be located either directly in the entity directory itself or indirectly through it.

Occasionally, an entity may belong to an arbitrary number of types in a generalization hierarchy. Thus, an entry in an entity directory may have to store a varying number of pointers. We use a varying length record representation for the entries to reduce storage overhead. The organization of the entries also must support efficient associative access based on entity identifiers. Furthermore, to permit the inclusion of new entities and to reuse the space occupied by entries for defunct entities, it is important that a dynamic file organization be used. For this reason, we choose to organize the entity directory using linear hashing [Larson80, Litwin80]. Each associative retrieval of an entry based on entity identifier can typically be made in one page access, regardless of growth or shrinkage of the directory.

3.2 Horizontal Partitioning of Primary Logical Records

In order to achieve better inter and intra entity type information clustering, we support

---

(4) It is frequently true that values for multiple functions applied to the same entity are often needed together.

(5) The end result is that unnecessary data transfers often have to be made.

(6) An entity type that is lower in a generalization hierarchy conceptually inherits all the functions applicable to its ancestors in the hierarchy. Rather than duplicating such information, we allow the use of clustering to appropriately juxtapose the related information.

(7) That is, the entity identifier will be included as one of the attributes.

the options of mapping one primary logical record type into several disjoint storage record types, and also the option of clustering multiple storage record types originating from the same generalization hierarchy in the same file.

Consider the following generalization hierarchy involving the entity types persons, students, and instructors.(8) Assume that students and instructors do not overlap (i.e., a person cannot be both a student and an instructor). An alternative to storing all the person records in the same file is to divide the person records into disjoint groups, and to store the groups of records in different files. If we view all of the logical records of a given type as a table, then the grouping may be viewed as partitioning this table horizontally. Instead of horizontal partitioning based on arbitrary criteria, we require that the partitioning be based on properties of overlapping type membership only. In the above generalization hierarchy, we can divide person records into records for:

- person who is a student
- person who is an instructor
- person who is neither a student nor an instructor

Alternatively, to suit a different usage pattern, we can divide the person records into records for:

- person who is an instructor
- person who is not an instructor

Now consider a generalization hierarchy where student and instructor do overlap. Here we may want to divide person records into:

- person who is a student but is not an instructor
- person who is an instructor but is not a student
- person who is both a student and an instructor.
- person who is neither a student nor an instructor.

In essence, the blocks of a horizontal partitioning scheme are defined by a number of nonoverlapping block definition predicates. Each block definition predicate may consist of a conjunction of type inclusion/noninclusion conditions involving types that overlap with the type in question.(9) In addition to the blocks defined by each of these predicates, a complementary block is also induced by the complement of their disjunction when this complement is satisfiable. That is, records that do not satisfy any of the block definition predicates will be stored in the complementary block.

(8) That is, each student is also a person and each instructor is also a person.

(9) The use of disjunction of subtype membership properties to define blocks is in effect supported since we allow the placement of two or more blocks from the same horizontal partitioning scheme in the same file.

3.3 Placement of Storage Records

The primary organization of a file determines how records are to be positioned within the file. In general, the placement criteria may be based on:

- The entity identifier of the record
- One or more other fields stored in the record
- The positioning of related records

Typical file organizations may be dichotomized as static versus dynamic. In a static organization, records do not move once they have been inserted in the file. When the original (primary) space assigned to the file runs out, overflow space (typically additional pages chained onto the original pages) is used to accommodate the subsequently inserted records.(10) Contrarily, in a dynamic organization, the amount of primary space assigned to a file grows or shrinks dynamically in response to insertions and deletions. Records are moved as a result of page splitting and merging operations (used to maintain a certain loading factor) and to guarantee a certain level of associative access efficiency and uniformity.

For an infrequently updated file, a static organization typically is faster than a dynamic organization. However, the amount of overflow in a statically organized file is liable to become excessive and unbalanced, requiring costly periodic reorganization of the whole file. This will result in the file's inaccessibility while reorganization is in progress. In a dynamically organized file, reorganization is performed incrementally and continuously, so that performance and accessibility tends to be more uniform. The drawback with having to move records around in response to insertions and deletions is that pointers to these records cannot readily be maintained. On the other hand, the storage of such pointers is often necessary in auxiliary access structures in order to provide additional access paths to the records. As we shall see in subsequent discussions, it is possible to replace physical pointers to records with logical pointers consisting of entity identifiers in order to minimize the impact of record relocation. However, this will require indirection through the entity directory for each access.

It is our belief that there will be situations where a static organization is more desirable than a dynamic one, and vice versa. However, in an attempt to limit the size and complexity of the system, we have decided to support dynamic organizations only in the initial implementation. Our rationale is that stability is often more critical than performance, and that the need to initiate reorganization is too much of a burden on users in many applications [Stonebraker80]. As will be discussed in Section 3.6,

(10) The distinction between primary and overflow is that access to a record in the overflow space can be made only by first accessing other records in the primary space. Thus, it is more expensive to access a record in the overflow space.

we have an optimization scheme for approximating the performance characteristics of static file organizations through the storage of hybrid pointers (combination of logical and physical pointers).

From an alternate viewpoint, we can distinguish between organizations based on address calculation (randomization) and those that use tree-structured directories. Typically, a tree-structured organization provides the capability of accessing records in key order, which is not feasible in randomized organizations. On the other hand, a randomized organization is usually more efficient for accessing individual records. To accommodate a range of applications, we have decided to support both randomized and tree-structured organizations. Thus, the dynamic organizations we support initially will include B*-tree [Bayer72, Comer79] and linear hashing [Larson80, Litwin80].

It should be noted that the choice of primary organization is allowed for only in the case of primary storage records. Secondary storage records will always be organized using linear hashing since the predominant access mode will be keyed on individual entity identifiers.

3.4 Clustering of Storage Record Types

In addition to positioning criteria based purely on record contents, we also support the placement of records dependent on the position of related records. For example, we may want to store a student storage record next to a person storage record when they represent the same underlying entity. In particular, we may combine clustering with horizontal partitioning to achieve better juxtapositioning of information within the same generalization hierarchy. For example, we may map person logical records into storage records for person who is also an instructor, and storage records for person who is not an instructor, and then cluster the instructor storage records with the first group of person storage records. In this way, all the information concerning instructor entities will be readily accessible together.

In the above example, the clustering is based on a one-to-one relationship, namely, records representing the <u>same</u> entity are to be stored close to each other. In this case, we require the related records to be stored adjacently on the <u>same</u> page, so that a single page access will suffice for their simultaneous access.(11) We will call such clustering <u>contiguous</u>. As a special case, if we cluster both student records and instructor records with the corresponding person records, we effectively have a scheme very similar to one that is obtained by storing values for all applicable functions in the same record.(12) In general, we allow multiple storage records representing the same underlying entity to be clustered together. We also allow multiple storage record types that originate from the the same logical record type to provide the functions for determining record placement.

Besides contiguous clustering based on one-to-one relationships, it is possible to perform clustering based on one-to-many relationships. For example, if there is a one-to-many relationship between department and employee (department is a single-valued function applicable to employee entities), we may require each employee record to be stored close to the corresponding department record. In this case, it may not always be possible to store all of the employee records related to a particular department record on the same page. Rather, it may be more reasonable to require that they be stored only in the same general vicinity (a small fraction of the file space). We will call this type of clustering <u>noncontiguous</u>. One practical way to implement noncontiguous clustering is in conjunction with a static file organization. Instead of requiring that all related records be found on the same page, related records are localized only on pages assigned to the same bucket. In this case, all related records can be located by a sequential scan of the entire bucket. As in contiguous clustering, multiple types of records may be clustered. For example, we may want to store Employee records close to the related Department records, and to store Dependent records close to corresponding Employee records. However, in view of our decision not to support static organization initially, we must also postpone support for noncontiguous clustering.

3.5 Auxiliary Access Structures

In addition to primary access paths provided by record placement strategies, often it is desirable to support associative access based on additional criteria. As in conventional systems, we permit the maintenance of simple and combined indices on logical records of a given type. Conceptually, an index provides a mapping from an indexed key value (or combination of values) to a set of pointers to the storage records that contain the indexed value (or combination of values). (As will be seen in the next section,

---

(11) In fact, we will construct a hybrid record to combine the information from the original records representing the same underlying entity. In general, a (hybrid or nonhybrid) record may consist of a fixed length portion followed by a varying length portion. We require only that the fixed length portion of the combined record not span page boundaries.

(12) A single record header is used to describe a group of records that represent the same underlying entity that is being clustered together. This header will also replicate the typing information in order to eliminate access to the entity directory when it is necessary to obtain information about an entity from the viewpoints of several overlapping entity types, and this information is already clustered in the same hybrid record. Pointers from the entity directory point to the combined record instead of to the individual records.

we will use only logical and hybrid pointers to point to dynamically organized records.) As for the organization for the index file(13) the options of using either a B*-tree organization or a linear hashing organization may both be useful. (There is no advantage for using a static organization for the index file since records in this file are not pointed to by records in other files.) A linear hashing organization provides more efficient access based on an equality search. Typically, a single access is all that is needed to locate a particular index entry. A B*-tree organization, on the other hand, requires one access for each level of the tree, while providing a fuller range of functional capabilities: the ability to access index entries in key order makes it useful in the resolution of range queries. In addition, it is also possible to use such an index to retrieve all records in key order. Both types of organizations are allowed for in the LDM implementation.

Another relevant organizational issue is how a pointer list should be represented. While most contemporary systems use a sorted array representation, there are also some which automatically convert an array representation to a bit-map representation when a list gets long. The advantage of the latter scheme is that it results in a much more compact representation on which bitwise operations can be performed in order to implement set operations on pointer lists. However, it may be more difficult to intersect pointer lists that use different representation. For the sake of software simplicity, we restrict our initial implementation to the array representation only.

3.6 Hybrid Pointers

Pointers in data structures are essential for supporting associative access. These pointers may be of a logical nature, or they may be physically oriented. A logical pointer has the advantage of providing a higher level of data independence. However, once a logical pointer is obtained, an extra level of searching must be performed to acquire an actual physical pointer to the desired information. In our context, the entity identifier serves as a logical pointer, with the entity directory providing the indirection. When a storage record that stores information concerning an entity has to be relocated, only the corresponding entry in the entity directory needs to be updated; all other records that store the entity identifier of the affected entity need not be modified. For indices that point to dynamically organized records, it would be appropriate to store logical pointers to simplify pointer maintenance.

As a physical pointer, we use the page number together with the direct or indirect(14) offset of the record within the page. Physical pointers have the advantage of directness. However, the price we pay is pointer maintenance when records are relocated. Since we support only dynamic organizations, we do not permit the use of physical pointers in isolation in secondary indices or in the representation of functions.

As an optimization, however, we support the option of combining a physical pointer with a logical pointer to form a hybrid pointer. For example, when representing a function from course to student, it may be useful to store both the student entity identifier, and the physical pointer to the student record. The rationale is that often when the student function is applied to the course entity, one is interested only in the student aspects of the target entity. Similarly, in the index on age for the student record type, we can store both the entity identifiers and the physical pointers to the student records. In general, we can follow the physical portion of a hybrid pointer to find the pointed-to record, and then compare the entity identifier stored there against the logical pointer portion of the hybrid pointer on hand. If the two entity identifiers do not match, we know that the pointed-to record has been relocated. In this case, the corresponding entry in the entity directory should be examined to determine the new address of the relocated record, and the hybrid pointer should be updated. The advantage of this scheme is that records can be relocated without regard to the pointers that point to them. Only the entity directory needs to be updated. The hybrid pointers are revalidated when they are next used. Thus in a high update situation, a record may be relocated many times before pointers pointing to it need be updated.

4. SUMMARY

We have presented a set of storage and access structures for supporting a semantic data model. The prominent features of this data model, which are intended to capture more application semantics than constructs found in conventional data models, include the notions of generalization hierarchies and referential constraints. Our design allows for the flexible tuning of database organizations to match application requirements. The design space encompasses such options as horizontal and vertical partitioning of information within an entity type, as well as the clustering of information across entity types within the same generalization hierarchy. Dynamic file organizations are used for the storing of data records, and the concept of hybrid pointers is introduced for the

(13) Each record in this file consists of an indexed key value and an associated pointer list.

(14) In the case of varying length records, it is often desirable to be able to relocate a record within a page without having to update all pointers that point to the relocated record. The indirection of the physical pointer can solve the problem by means of indexing information stored at the bottom of the same page.

purpose of pointing to dynamically relocatable records in the representation of inverted lists in secondary indices, and in the representation of interentity relationships. The underlying DBMS that supports the discussed set of storage and access structures is being developed by the Computer Corporation of America. It is scheduled to be completed in 1983.

5. ACKNOWLEDGEMENTS

We are indebted to Professor Philip Bernstein, Professor Nathan Goodman, Dr. Randy Katz, Terry Landers, Frank Manola, Dr. James Rothnie, Dr. Diane Smith, and Dr. John Smith for providing us with invaluable input and feedback during the design of the Local Database Manager to support the ADAPLEX language.

6. REFERENCES

[Bayer72]
Bayer, R., C. McCreight, "Organization and Maintenance of Large Ordered Indexes," Acta Informatica, Vol. 1, No. 3, 1972.

[Comer79]
Comer, D., "The Ubiquitous B-Tree," ACM Computing Surveys, Vol. 11, No. 2, June, 1979.

[Date81]
Date, C. J., "Referential Integrity," VLDB Conference Proceedings, 1981.

[DoD80]
United States Department of Defense, "Reference Manual for the Ada Programming Language," Proposed Standard Document, July 1980.

[Larson80]
Larson, P., "Linear Hashing With Partial Expansions," VLDB Conference Proceedings, 1980.

[Litwin80]
Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," VLDB Conference Proceedings, 1980.

[McLeod76]
McLeod, D. J., "High Level Domain Definition in a Relational Database," Proceedings for ACM SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, 1976.

[Shipman81]
Shipman, D., "The Functional Data Model and the Data Language DAPLEX," ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

[Smith77]
Smith, J. M., D. C. P. Smith, "Database Abstractions: Aggregation and Generalization," ACM Transactions on Database Systems, Vol. 2, No. 2, June, 1977.

[Smith81]
Smith, J. M., S. Fox, T. Landers, "Reference Manual for ADAPLEX," Technical Report CCA-81-02, Computer Corporation of America, January 1981.

[Smith82]
Smith, J. M., S. Fox, T. Landers, "ADAPLEX: The Integration of the DAPLEX Database Language with the Ada Programming Language," Technical Report, Computer Corporation of America, in preparation.

[Stonebraker80]
Stonebraker, M., "Retrospection on a Database System," ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980.