A STORAGE MANAGER FOR THE HYPERNODE MODEL

E. Tuv¹, A. Poulovassilis² and M. Levene³

¹,³ Department of Computer Science,
University College London,
Gower St., London WC1E 6BT
e-mail: {e.tuv, m.levene}@uk.ac.cs.ucl

² Department of Computing,
King's College London,
Strand, London WC2R 2LS
e-mail: a.poulovassilis@uk.ac.kcl.cc.oak

Abstract

We describe the implementation of a Storage Manager (SM) for the Hypernode model, a new data model whose aim is to integrate object-oriented and deductive databases. The single data structure of this model is the hypernode, a directed graph whose nodes may themselves be directed graphs. The components of the SM manipulate these graphs in a persistent store. The main effort of the first prototype of the SM has been to develop a modular and extensible system which can be used as a reliable and stable core for future versions. In particular, the SM caters for object-identity and referential sharing between hypernodes, large and dynamic hypernodes, clustering strategies on secondary storage, and retrieval operations which utilise indexing techniques. The main contribution of the SM is the single graph data structure which permeates throughout all the levels of the implementation; in this way efficiency can be achieved within all the components of the SM as a result of optimising this data structure, and also interfacing between the components of the SM is simple and uniform.

1 Introduction

Recent database research has focussed on deductive and object-oriented databases [ULL88, GAR89]. These are largely complementary: the former supports both stored and derived relations and the latter supports data abstraction mechanisms such as classification, identification, encapsulation and inheritance. Hence, recent database research has aimed at integrating the two paradigms. This integration has generally taken the route of extending logic-based deductive database languages with features such as object identity, sets, functions, methods and inheritance [ULL91]. Taking a different approach, we have developed a graph-based model called the Hypernode Model [LEV90, POU90]
to support such an integration. In contrast to other graph-based data models [CON90, GYS90, LEV91, TOM89], we use nested, possibly recursively defined, directed graphs termed hypernodes.

A hypernode is a pair, (N,E), of nodes and directed edges such that the nodes of N are either primitive values or themselves hypernodes. Hypernodes have unique value-independent labels which serve as object identifiers. We illustrate a hypernode in Figure 1. It represents a couple, C, consisting of two people, PER1 and PER2, whose children are nested within further hypernodes. In Figure 2 we show the children of person PER2, which would become visible if we “exposed” the hypernodes labelled PER3 and PER4.

The labels C and PER1-PER4 in these figures are superscripted with the tags COUPLE and PERSON, respectively. These tags indicate the types of their associated hypernodes and are omitted whenever they are understood from context. Types give us a means of defining database schemas and of enforcing constraints on the structure and content of hypernodes (see [POU90] for a detailed description of types). In fact, types are just hypernodes and so can be queried and updated using the same formalism as for data. We also note the use of the node none^PERSON in Figure 2 - it is null value of type PERSON denoting “does not exist”.

The hypernode model comes equipped with a computationally powerful declarative language called Hyperlog. Thus, the model and language share features with both
deductive and object-oriented databases. In common with other deductive database languages, Hyperlog is rule-based and supports derivation rules and database updates. In common with object-oriented databases the hypernode model supports complex objects and the data abstraction concepts of classification (via types), identification (via unique labels) and encapsulation of data (via the nesting of graphs). In [LEV91] we showed how structural inheritance is also supported naturally by nested graph structures.

Several advantages to database management accrue from our use of nested graphs. At the physical level the implementation of a single persistent data structure (the graph) allows special-purpose storage and indexing techniques to be developed and optimised. At the conceptual level hypernodes provide a formally-defined yet simple data structuring tool, capable of representing complex database schemas and objects. Finally, at the external level, graph-based formalisms generally enhance the usability of complex systems [HARE88]. Furthermore, the nesting of hypernodes provides greater flexibility in the design and browsing of densely connected database graphs. Thus, the hypernode model is promising for application domains such as CAD, CASE and Hypertext [NIE89].

We are currently implementing a prototype DBMS for the hypernode model which we intend to evaluate via a hypertext application. The three-level architecture of our DBMS is illustrated in Figure 3 (cf. System R [AST76]).

At the lowest level, the Storage Manager stores hypernodes and types as labelled graphs, $G = (N,E)$, and the Index Manager supports efficiently two operations:

(i) **Value-to-hypernode.** Given a primitive node, $n$, return the set of labels $\{G_1, \ldots, G_r\}$ such that for each graph $G_i = (N_i, E_i), n \in N_i$, and

(ii) **Label-to-Hypernode.** Given a label, $G$, return the set of labels $\{G_1, \ldots, G_r\}$ such that for each graph $G_i = (N_i, E_i), G \in N_i$. 

Figure 3. The Hypernode Database System Architecture.
These operations are invoked by the Hyperlog evaluator during its execution of Hyperlog programs. An example of a simple query hypernode is illustrated in Figure 4. This query returns the set of all PERSONS who satisfy the following two conditions: (i) they all have the surname *Smith* and (ii) they all have the child referenced by the label *PER3*. This query makes use of both the above operations. It first retrieves all labels which satisfy the first condition using operation (i), then it retrieves all labels which satisfy the second condition using operation (ii). The intersection of these two sets of labels and the set of PERSONS is the required result.

The main aim of this first prototype DBMS is to demonstrate the feasibility of the hypernode model and Hyperlog in a single-user environment. Transaction management, concurrency control and recovery will be incorporated during a second phase, and at the first instance we envisage that conventional techniques will be employed [GRA78]. For simplicity, in the rest of the paper a transaction will be considered as a single execution of a Hyperlog program.

In parallel to the research and development of data models for complex objects, such as object-oriented data models and the nested relational data model [SCH90], a large research effort has been invested into the development and research of storage management systems to support these models. A short survey of these storage management systems can be found in [KHO90]. These storage managers provide support for object identity, referential sharing, and large and complex objects, which are also requirements of the hypernode model. Our storage manager is unique in that the single graph data structure permeates through all levels of the system. Furthermore, this implementation strategy yields a more efficient storage manager than would have resulted from implementing our hypernode data structure on top of an existing kernel. We have chosen not to use an existing system since we require a flexible modular system that allows us to modify the policy of any given component without affecting the rest of the system.

In this paper we are specifically concerned with the design and implementation of the, Storage Manager (SM) and, to a lesser extent, Index Manager (IM) components of the above architecture. The SM is a set of modules carrying out the manipulation of graphs in a persistent store. It also provides support for different placement strategies [VAL86] for graphs. The SM is built on top of a kernel system which provides the low-level storage management of variable-length untyped streams of bytes. This kernel ensures portability, allowing the SM to be installed in different environments. Our current SM is being developed under UNIX\(^1\), where the kernel has to overcome the well-known problem

\(^1\)Unix is a trademark of AT&T Bell Laboratories
of the UNIX file system whereby large files may be physically fragmented on the disk.

The outline of the rest of the paper is as follows. In Section 2 we discuss the requirements for the SM, and the particular problems they pose, with respect to the hypernode model and its intended application domains. In Section 3 we describe our implementation of the SM: we discuss the motivation behind its design, give an overview of the overall architecture, and describe the functionality and implementation of each of the modules. In Section 4 we discuss indexing and clustering issues. Finally, in Section 5 we give our concluding remarks and discuss ongoing and further research issues.

2 Requirements for the Storage Manager

The hypernode model presents particular requirements which must be met by the Storage Manager. In this section we enumerate these requirements and the problems they pose. In the sections that follow we demonstrate how these problems are solved in our implementation.

Object Identity and Referential Sharing. A hypernode can be referenced by any number of hypernodes via its unique label. Thus, when a hypernode is deleted, there may remain other hypernodes which still reference it. This may cause problems with referential dangling if the referenced location is subsequently overwritten with other data at the storage level, or if a reference to deleted data exists. For example, in Figures 1 and 2, while updating PER1 one can delete the hypernode labelled PER3 and in addition remove its label. In this case a reference to PER3 remains in PER2 and causes referential dangling.

Large or Complex Hypernodes. Hypernodes may be large or complex as a result of a number of characteristics, any combination of which is possible:

(i) A large node set: for example a hypernode representing a program which consists of a large number of modules, or a document with a large number of paragraphs.

(ii) Large primitive nodes: for example a primitive node which is a large item of text in a hypertext database, or a bitmap representing a detailed image.

(iii) A dense edge set: the number of edges between the nodes of a hypernode can be \(O(n^2)\), where \(n\) is the number of nodes, for example in the transitive closure of an accessibility graph in a hypertext database (created to increase browsing efficiency), or in the design of a VLSI chip with many inter-connections between its components.

Dynamic Hypernodes. Hypernodes may be highly dynamic i.e. nodes and edges may be added and deleted arbitrarily and frequently, for example in a hypertext system where a user's view of the database, which is represented as a graph, can be changed dynamically. Thus the storage techniques employed for hypernodes should support such updates efficiently.
Clustering. Conventional storage strategies like vertical and horizontal partitioning are not sufficient for the hypernode model since hypernodes do not relate to each other solely via static properties such as their type or node values but also via their linkage (e.g. the pattern of referencing between hypernodes) cf. [VAL86]. For example, a set of hypernodes which is accessed by a specified root hypernode should be clustered together with this root hypernode. Examples of some options for clustering can be found in [HOR87]. Moreover the dynamic nature of a hypernode database, and especially the dynamic linkage between hypernodes, implies that dynamic clustering schemes are required, as in CACTIS [HUD89] for example, and also reclustering schemes.

Retrieval. In conventional index strategies for relational databases [AST76] keys are defined and maintained over the scope of a single relation. On the other hand, the hypernode model and its intended application domains require retrieval of any node in the scope of the whole database, for example as in ANDA [DES88] for the nested relational model.

Extensibility. The SM implementation must be extensible in the sense that new types of hypernodes can be added without making any code changes. Furthermore, although the current prototype is intended for a single-user single-site environment, it should serve as the core for a future multi-user, distributed environment, and for new policies such as new buffering and clustering policies.

3 Implementation of the Storage Manager

In the implementation of the SM we attempted to address the requirements outlined in Section 2 above. Here we describe the architecture of the SM and the implementation of each one of its components. In 3.1 we give the overall architecture of the SM. In 3.2 we focus on the support which is provided by the kernel of the SM. In 3.3 we give a description of the physical representation of hypernodes, both in secondary storage and in memory. The characteristics and uses of labels of hypernodes are described in 3.4. Finally, the buffering of hypernodes in memory is discussed in 3.5.

3.1 Overall Architecture

The primary goal in the development of the prototype SM was to build a stable core system which could subsequently be extended, modified and tested in almost all of its functionalities and modules. For that reason we decided to build a modular and extensible system. Like EXODUS [CAR86] and DASDBS [SCH90] we adopted the approach of building this system over a Kernel that provides a low-level implementation of a persistent store for objects, which are just uninterpreted streams of bytes. The architecture of the SM is illustrated in Figure 5.

Each file in our DBMS is termed an object store and each object store is identified by a unique identifier, the object store id. The logical address of any object is the composition of the object store id with an offset within that object store. The functionality of an object store is thus similar to the files of the EXODUS system [CAR86] and the bags
of CMS [ATK83]. However, our object stores differ from these data structures in their internal organisation.

The Kernel module provides secondary storage management, and attempts to cluster the objects it manages in contiguous locations in secondary storage. In order to allow extensibility, the Kernel has minimal semantics embedded into it. Thus, it has no knowledge of hypernodes, types, and labels, and recognises only untyped streams of bytes. The Kernel frees the higher levels of the SM from interacting with the operating system and the hardware media. The Kernel also overcomes some of the problems inherent in file systems which are provided by conventional operating systems. In UNIX, for example, a large file may be fragmented into several pieces located in different areas on disk. In contrast, the Kernel's file system allocates contiguous pages on the secondary storage medium for each object store.

On top of the Kernel is the Buffer Manager module. This module maintains objects in memory in their in-memory format and is responsible for transferring hypernodes between secondary storage and memory on demand. In order to locate the physical address of a hypernode and to retrieve the hypernode from the Kernel, the Buffer Manager uses the services of the Label Manager to map the label to the physical address of the hypernode. It then uses the Hypernode Manager to translate the hypernode from its secondary storage format into its in-memory format. We discuss these components and the interaction between them in more detail in sections 3.2 to 3.5 below.

3.2 The Kernel

The Kernel module comprises two sub-modules, the File System and the Object Store Manager. The interface to the Kernel is via messages that are sent to the latter which reads, writes and retrieves objects, regarding them as uninterpreted streams of bytes. The Object Store Manager allocates and deallocates objects within object stores and controls space utilisation at the object store level, while the File System allocates and deallocates object store pages on request from the Object Store Manager. The Object Store Manager also handles requests from the Buffer Manager (see 3.5) and dispatches
them to the appropriate object store.

In the context of the Object Store Manager, an object is of one type only: an uninterpreted stream of bytes. The Object Store Manager supports objects of any size but hides details regarding their storage from its clients, providing them with only one interface which is independent of the size of its objects. Thus, from the point of view of the clients, objects are stored contiguously on the secondary storage medium, even if at times this is not necessarily so. Unlike the Object Store Manager, the higher levels of the SM do have knowledge of the internal structure of objects and can operate on this internal structure. This fact is one of the key ingredients to the support of extensibility that is provided by the kernel.

Although the Object Store Manager has no knowledge of the semantics of the objects it maintains, it can accept hints from its clients regarding the placement of objects. For example, a client can provide a preferred address (location) for an object, in order to allow physical clustering of logically related objects cf. CMS [ATK83] and EXODUS [CAR86]. The client can also specify an expected maximum size for the object, in order to pre-empt movement of the object as it grows. Objects can be moved within and between object stores when necessary. The File System ensures that the pages of an object store are in fact stored contiguously in secondary storage.

3.3 The Hypernode physical data structure and its interface

In this sub-section we describe the physical representation of a single hypernode in secondary storage and in the buffering area. In our design of this physical data structure we were motivated by a number of principles:

(i) Clients must interact with hypernodes via a pre-defined interface of primitive operations (see 3.3.2). In addition, this interface must be uniform for all hypernodes irrespective of their size and complexity.

(ii) Several of the hypernode operations query or retrieve only a portion of a hypernode (for example, get node, is a node in a hypernode), so the physical data structure should allow access to these components only and should not retrieve into memory the hypernode as a whole. This is crucial for hypernodes with a large number of nodes or with large sized primitive nodes.

(iii) The cost of retrieval and update operations on sub-parts of large hypernodes should be as independent as possible of the overall size of the hypernode. For example, a modification to a hypernode with a small number of large-sized of primitive nodes should be as efficient as a modification to a hypernode with the same number of small-sized primitive nodes.

(iv) The actual data of a hypernode should be maintained contiguously on disk, so that the retrieval of a hypernode requires a minimal number of disk accesses. Thus, we have tried to keep the data structure as linear (i.e. in a contiguous array of characters) as possible and to avoid pointer chasing on the secondary storage. A large hypernode may have to be divided into a set of linear spaces which are clustered together by the kernel.
(v) It has been shown, for example in [SCH90], that copying and translation of complex objects between their in-memory format and their secondary storage format can be a bottleneck in a storage system, so seamlessness between the on-disk and in-memory format of hypernodes is required. In particular, we have designed a data structure which requires only a minimal translation between disk and memory without degrading the performance of either the on-disk or in-memory operations on it. We have described the on-disk strategy in point (iv) above. In contrast, the in-memory storage and manipulation of hypernodes is made more efficient by utilising pointer chasing via dynamic data structures such as linked lists.

(vi) Instances of hypernodes of the same type can share common information and this should not be replicated in each instance of that type but instead should be stored at the type level. Thus, with each type in our system we associate a template hypernode which consists of the nodes shared by all instances of that type (cf. static class members in C++ [STR86]).

In the SM we support only one data structure, namely the hypernode, which is sufficient to emulate other data structures such as lists, tuples and sets. Other storage managers have to support several different data structures: for example FAD [BAN87] supports tuples and sets and \( O_2 \) [VEL89] supports tuples, sets and lists. Moreover, most of the existing systems that do utilise one data structure only, such as AIM-P [KUS87] and DASDBS [SCH90] (i.e. the nested relation), are limited to hierarchical structures and do not support a general directed graph.

3.3.1 The Hypernode physical data structure

Figure 6 illustrates the general hypernode data structure. As is shown in this figure, the data structure consists of three main parts: the header part, the nodes part and the edges part. The main idea behind the design of the hypernode data structure is our desire to keep the layout of the data in a contiguous array of characters. In this structure, pointers are either offsets into that array or an address of another object in the same object store. When one of the parts of the hypernode becomes too large (e.g. it outgrows a page size) then it is separated from the original array and is maintained as a separate object in the same object store. It is written by the kernel to a location which is as near as possible to its original location. In this case the pointer to that part is modified to the address of the separated object. By separating large parts of the hypernode from its other parts we ensure that the length of any single subpart will not affect the overall behaviour of operations on the hypernode. For example, a small hypernode with one large node is maintained in two separate areas; the large node resides in the first one, while the remaining parts of the hypernode reside in the second.

**Header part.** The header part is of fixed length and contains general information about the hypernode such as its number of nodes, the number of nodes in the template hypernode, and pointers to the different parts of the hypernode.

**Nodes part.** This consists of two sub-parts, a list of node headers and an associated data part. Each node header contains information pertaining to one node in the hypern-
node, such as the type of the node (a primitive type or a label), a flag to indicate whether that node has been deleted, and either the value of the node (for integers and labels), or a pointer to the node value i.e. either the offset in the data area part or the address of a separated object.

Each node within a hypernode has an associated locally unique identifier termed the \textit{Node Id}. This is the entry number of the node in the node header list. The value associated with the Node Id may be changed during the lifetime of a hypernode as a result of the deletion of some other nodes and the compression process which reclaims unused data in the data area part and in the node header list. Node Ids are hidden from end-users who can only access nodes by their values. However, for optimisation reasons, Node Ids are available to several modules of the SM so that nodes can be manipulated without knowledge of their actual values. Node Ids are non-zero integers (zero is reserved for error handling). A negative Node Id is a reference to the shared node list within the template hypernode. For example, Figures 7a, 7b and 8 illustrate the template hypernode associated with the PERSON type and an instance of that type, PER2. In these figures the node types S and L, correspond to stream of bytes and label respectively.

The second sub-part of the nodes part is an array of characters containing the values of primitive nodes. From time to time this array may have to be compressed, for example after many deletions which have occurred to a particular hypernode.

\textit{Edges part}. For each node in the hypernode and in its associated template hypernode we maintain an adjacency list. A list of pointers, one corresponding to each node, point to these adjacency lists which are stored as a contiguous array of Node Ids. The

Figure 6: Hypernode data structure
same idea of offsets being converted into references to separate objects is used for storing this contiguous array as it grows.

Edges between a template hypernode and its instances are not common to all that type's instances. Thus, each hypernode also maintains all the edges between its internal nodes and its template nodes (thus, in Figure 6 the number of adjacency list pointers is $m+n$).

3.3.2 Operations on hypernodes

The operations on hypernodes fall into four categories:

1. *Update operations.*

(i) *Add a node to a hypernode.* This operation accepts a node value as input and appends it to the node header list and also to the data area if required. If the new node is a large primitive node then it is separated from the data area of the hypernode and a reference to it is appended to the node header list. The return value of the operation is a Node Id. If the input node value exists in the hypernode then the input is ignored and the return value is its Node Id. If the operation fails then the Node Id value returned is zero.
(ii) **Add an edge to a hypernode.** The edge can be supplied as a pair of node values or Node Ids. The edge is inserted into the appropriate location in the list of edges. The two nodes must exist in the hypernode otherwise an error code is returned by the operation.

(iii) **Remove a linked node.** The input node may be a value or a Node Id. If the node is found it is tagged as deleted and its space may be reclaimed later. The edges to the node are also checked and removed. If the node is not found an error code is returned by the operation.

(iv) **Remove an isolated node.** The input to this operation must be an isolated node. If the node has edges to it or from it, an error code is returned by the operation.

(v) **Remove an edge.** The input to this operation is an edge. If the edge is found it is removed from the list of edges.

2. **Traversal operations.**

(i) **Get all references from a node.** This operation is used to get a set of Node Ids where each one of them is referenced by a specified node. If the set is empty then the return value is NULL.

(ii) **Get all references to a node.** This operation is similar to the above operation except that it looks for the nodes that reference a specified node.
3. Membership operations.

(i) Is a node in a hypernode. This operation is used to check whether a specified node is contained in a specified hypernode.

(ii) Is an edge in a hypernode. This operation checks whether a specified edge is contained in a specified hypernode.

4. Retrieval operations.

(i) Get node. This operation is used to obtain a node value given a Node Id. If the node value is not a label or a fixed length primitive, such as an integer, its length must be obtained first. In this case a pointer to the address of its value is returned by the operation.

3.4 Labels and Label Tables

One of the most important concepts of the hypernode model is the fact that each hypernode is equipped with a system-wide unique identifier, its label. Thus labels provide hypernodes with object identity [KHO86]. The label is a built-in value associated with a hypernode at its time of creation and is the mechanism whereby hypernodes can reference each other. In general, one label can be referenced by more than one hypernode. This referential sharing raises the problem of referential dangling which occurs whenever a reference to a hypernode exists but the hypernode itself has been deleted and its label has possibly been reused for the creation of a new hypernode.

There are several ways of implementing this unique identity of hypernodes (see [KHO86]). In our system we have adopted an approach similar to that of ORION [KIM88] whereby each label consists of two components, the type of the hypernode and an instance identifier. We maintain labels in one or more tables termed label tables. One label table is maintained for each hypernode type defined in the database. In addition, a system-reserved label table is used as a metalevel table for all types. Each entry in a label table contains information pertaining to one label. This information is used to control access to, and manipulation of, the hypernode associated with the label. Interaction with the label tables is controlled by the Label Tables Manager.

The primary usage of the label tables is to provide a mapping from labels to the physical addresses of hypernodes. A further important function of the label tables is the creation of new labels. A new label is requested automatically when a new hypernode is created, and the table of the appropriate type supplies this new label. Each label table maintains information on used labels and employs a very simple hashing scheme to accelerate access to specific entries.

The provision for a separate label table for each type, as opposed to one global table, is useful for several reasons: scanning hypernodes of the same type is made simple, changes to the type structure of hypernodes are easier to maintain (for example, the deletion of a type), access to a hypernode's label entry is direct, and space utilisation of each table is maximal (i.e. unused space in the table is minimal).

There are two operations supported by the SM which can cause referential dangling: one is the delete operation of a hypernode and the other is the wipe operation of a
hypernode. When a hypernode is deleted its internal space is reclaimed and its label is mapped to a special null address. In contrast, the wipe operation is more intricate since it removes both the hypernode and its label, which can subsequently be re-used. In the implementation of wipe, references to the hypernode must be located and modified to reference a special null label. Each label table has an entry where the instance part of the label is this null label and the address is the null address; in Figure 2, nonePERSON is an example of the null label of type PERSON. All the references to a specified hypernode can be located in one disk access since the Index Manager (see Section 4) maintains such an index for all the labels in the database.

In the current implementation of the DBMS hypernodes persist between transactions as result of explicit requests to the SM. Not all hypernodes are required to persist between transactions so, since the outcome of not reusing labels of non-persistent hypernodes would be overly large label tables, the wipe command is used when removing transient hypernodes.

The disadvantage of our implementation of labels is the indirection mechanism used to access any hypernode: we need two accesses, one to obtain the address from the label and another to access the hypernode itself. However, in practice, buffering of hypernodes in memory reduces the actual number of disk accesses (we discuss our buffering mechanism in more detail in 3.5). The advantage of our implementation is that whenever a hypernode is accessed its type is also available, and so any specific information associated with that type can be retrieved automatically. An example of such information is the template hypernode which acts as a common structure for all instances of that type and so is not stored within instances (see Figures 7b and 8 where the nodes of the PERSON template hypernode are referenced by PER2 and are not physically stored in PER2).

Finally, since our system is typed we are concerned with migration of hypernodes from one type to another. This operation can be implemented at the same cost as the wipe operation since all the references to the hypernode have to be found and modified.

3.5 The Buffer Manager

DBMSs that use the buffering services of the operating system suffer from several disadvantages [STO81]: the operating system has no knowledge of the implementation it serves so the buffer management it provides is not always suited to the requirements of the DBMS; the page-based buffer management used by virtual memory systems is inefficient in the environment of a complex object DBMS since the size of an object is not necessarily bounded by the page size; a further problem with page-based buffering is that objects in their in-memory format are likely to be different from objects in their on-disk format and translation between the two formats is cpu-intensive[SCH90].

Our solution to the above problems is similar to that of ORION[KIM90] and LOOM[KAE81]. Our buffer is divided into three areas: the first area is used by the page pool in order to fetch pages from the disk, the second area is used for buffering hypernodes, and the third area is an access table controlling access to individual hypernodes. Each entry of the access table is associated with one hypernode of which some portion is present in the buffering area. The number of entries in the access table may be very large and so the Buffer Manager uses hashing to accelerate access to entries.
The information stored in each entry of the access table is used to control access to hypernodes and comprises the address of the hypernode in secondary storage (if it is a new hypernode then this value will be the null address), a dirty flag which indicates whether the hypernode has been modified (in which case it has to be updated in the persistent store), the address of the hypernode in the buffer, and obviously the label of the hypernode in order to identify the correct entry in the label table lookup process. In the current version of our DBMS we do not cater for concurrency control and security, but it would be quite natural to add such information to the access table since the smallest unit of security and concurrency control will be a single hypernode and hypernodes are always accessed through this table.

When Hyperlog requests a hypernode it passes its label to the Buffer Manager which tries to locate the label in the buffering area. If found, the hypernode can be retrieved; otherwise a hypernode fault (cf. object fault [KIM88]) occurs and the hypernode is retrieved from the disk by the Buffer Manager which uses, for this purpose, the label tables and the Kernel. If there is not enough memory in the buffer area then the Buffer Manager employs a replacement policy to obtain enough room (currently we use a LRU mechanism). Furthermore, hypernodes are written to disk by an explicit command. Hypernodes that are written to disk may still remain in the buffer in case they are required by other transactions.

4 Clustering and Indexing

In this section we address two further physical-level aspects of the hypernode DBMS which are outside the scope of the storage manager, namely clustering and indexing.

4.1 Clustering

The smallest unit of clustering is the hypernode itself since it represents a conceptual clustering of attributes and related hypernodes (via their labels). However, different placement policies can be obtained as in ObServer [HOR87] and in addition a hypernode may be stored separately from its referenced hypernodes.

Hypernodes are clustered together physically on the disk in units of object stores which are provided by the Kernel. In the current version a default clustering mechanism is defined in the database at creation time and can be modified on-line at any stage. Two different policies are supported: one is to store each hypernode in an object store according to its type, and the other one is to store all new hypernodes in a default object store. This default object store can be changed dynamically. Moreover each hypernode may be written to a different object store from the default, and at location as near as possible to a specified location, typically the physical address of another hypernode.

4.2 Indexing

In relational databases primary indexes can be built on a selected key attribute or on a number of such attributes, and additional secondary indexes can be built on request
In contrast, in the hypernode DBMS we need to provide general-purpose secondary index support for the retrieval of hypernodes given a partial specification of their node and edge sets. Two basic operations are supported by the Index Manager which are sufficient to implement such retrieval requests, namely Value-to-hypernode and Label-to-hypernode, discussed in the introduction.

A separate data structure is used to implement each of these operations due to the simple fact that with primitive nodes a pre-defined order is available while there is no such natural ordering over labels. Thus, for operation value-to-hypernode we use a simple prefix B-tree [BAY77] while for operation label-to-hypernode we use the same hashing technique as for the label tables. Both the leaves of the B-tree and the hash value of a label reference to a list of labels which is ordered by type. Template nodes, which we recall are not stored within instance hypernodes, appear only once within these indexes and return labels of template hypernodes. These labels are recognised as such by the higher modules of the architecture and are replaced by instance labels as required.

The default scope of our indexes is the entire hypernode database. However, this default can be overridden in the database schema. Moreover, the indexes are extensible in the sense that further special-purpose access methods can be implemented for specific types in higher levels of the system. For example, we are investigating the implementation of full text retrieval indexes for our hypertext application. Finally, in the current DBMS we support two primitive types, integer and byte stream, where the ordering defined over streams is a native character comparison. Although for new primitive types new index techniques are required, we can use the current index over byte streams for a range of new types such as text in a hypertext system.

5 Concluding remarks

The major effort in the implementation of this first prototype version of the SM was in creating a stable and reliable system which can be used as a core for future research and development. We have shown that the SM caters for all of the requirements we identified in Section 2. In particular, labels provide us with object identity and referential sharing. With respect to complex, dynamic hypernodes our internal representation of a hypernode is efficient for both in-memory and secondary storage manipulation irrespective to its size and complexity. Clustering is supported by segmentation of secondary storage into units of object stores and retrieval is supported over the whole scope of the database. The modularity of the system and the minimal semantics embedded into the kernel are key ingredients for the extensibility of the SM. The SM has been implemented in C++ under UNIX. The object-oriented features of C++ have proven to be useful in supporting the modularity of the SM.

The major contribution of the SM is the single graph data structure which permeates throughout all the levels of the implementation. This is unlike other system managers which cater for several different data structures, such as sets, lists and tuples. There are several advantages to our approach: efficiency can be achieved within all the components of the SM as a result of optimising this data structure, interfacing between the components of the SM is simple and uniform, and a compact interface is provided to Hyperlog.
We are currently improving and extending two facets of the SM:

- We are attempting support many DBMS functions using the hypernode data structure. For example, persistence of hypernodes in the current version is achieved by an explicit command that is sent to the SM manager during a transaction. We are currently implementing a different mechanism whereby persistence is achieved through reachability, as in PS-ALGOL [ATK83a] and FAD [BAN87]. The reachability tree which describes the whole hypernode database and the hypernodes that are derived from a given root node, can itself be maintained as a hypernode. Other examples are clustering which can use a placement tree, also a hypernode, and versioning that can be maintained as a graph data structure.

- Optimisation to buffering sets of hypernodes and prefetching mechanisms can improve the overall performance of the system significantly. We are currently working on implementing different policies for buffering and prefetching in order to support specific applications such as hypertext.

6 Acknowledgements

We are grateful to Carol Small for her useful comments on this paper and to George Loizou for many fruitful discussions. The work we describe here forms part of the Hypernode Project, which is supported by S.E.R.C grant no. GR/G 26662.

References


