Database Pointers in Navigational and Object–Oriented Database Management Systems: A Comparison

Mark L. Gillenson
University of Memphis

Raymond D. Frost
Central Connecticut State University

Michael G. Kilpatrick
University of Miami

As information systems and, more specifically, database management systems, attempt to model particular application environments, they must be able to account for and keep track of how the entities in the environments relate to each other. In the first or navigational generation of DBMS, relationships were maintained by pointer chains that connected the records representing the related entities. In the second or relational generation of DBMS, the tuples, representing related entities were not connected by pointers, but could be “joined” at query time based on common values of particular fields. In the third or object–oriented generation of DBMS, there are two major structural approaches. One is a pointer–based approach while the other, is designed to add advanced, object–oriented features to the relational model. Recently, perhaps inevitably, interest in the object–oriented pointer–based approach has led to questions of whether it is, in some sense, a return to navigational DBMS. While object–oriented database clearly has major features that go far beyond the capabilities of first generation DBMSs, this article will show that a comparative analysis of pointer usage in navigational DBMS and in object–oriented DBMS can yield interesting results, plus a better understanding of the object–oriented DBMS paradigm.

Objects in the world relate to each other in an endless variety of ways. As information systems (IS) and, more specifically, database management systems (DBMS), attempt to model particular application environments, they must be able to account for and keep track of how the entities in the environments relate to each other. Traditionally, this has meant providing support for the familiar one-to-one, one-to-many, and many-to-many relationships between things or, more formally, entities, in the application environment.

In the first or navigational generation of DBMS, characterized by the hierarchical IMS system and the CODASYL network systems, relationships were maintained by pointer chains that connected the records representing the related entities. The advantage in performance that the pointers provided was offset by disadvantages in design complexity and inflexibility. These disadvantages, among other factors, led to the second generation of DBMS, known as relational DBMS (RDBMS). In RDBMS, the records, or tuples, representing related entities are not connected by pointers, but can be “joined” at query time based on common values of particular fields.

The newest generation of DBMS, object–oriented DBMS (OODBMS), includes several advanced features (Huges, 1991; Hurson, et al., 1993; Navathe, 1992). Among these: Modules of code, known as methods, can be associated with the data representing particular objects (entities) or classes of like–structured objects and stored right in the database with the object data. The methods and objects can be encapsulated so that only the stored methods, triggered by messages sent to them, can manipulate the object data. The data can be structured in inheritance hierarchies so that attributes and methods common to different “sub–classes” of objects can be inherited from “super–classes.” Composite
objects can be created from component objects.

While there is general agreement, at least at a high level, on the advanced features which, if present, allow a DBMS to be called an OODBMS, there is little, if any, agreement at the more detailed structural level. This is a significant departure from the earlier generations of DBMS. In the first generation, there were very few hierarchical DBMS and, by virtue of its huge market penetration, IMS was its own standard. The various CODASYL network DBMS all subscribed to the basic CODASYL specifications. The relative straightforwardness of the relational structure concept and the universal acceptance of the SQL query language make the various RDBMS quite similar at the structural level, although they vary in their query optimizers, distribution capabilities, and so on. Among the various OODBMS under development or commercially available, there are two major structural approaches. One, which will be referred to here as “native” OODBMS, is a pointer–based approach. The other, known as “extended–relational” OODBMS, is designed to add advanced, OODBMS features to the relational model.

Recently, perhaps inevitably, the pointer–based approach of the native OODBMS has led to questions of whether native OODBMS (which henceforth will be referred to simply as OODBMS) is, in some sense, at some level, a return to navigational DBMS (Osborn, 1989). Indeed, Date (1990) states, “A department object that contains a set of employee objects is very similar to an IMS hierarchy in which employee “segments” (this is the IMS term) are subordinate to department “segments.” While OODBMSs clearly have major features that go far beyond the capabilities of first generation DBMSs (Atkinson, et al., 1989), this article will show that a comparative analysis of pointer usage in navigational DBMS and in OODBMS can yield interesting results, plus a better understanding of the OODBMS paradigm. This article begins with a brief review of navigational DBMS pointer structures, since some readers may be a bit rusty on the topic and others, who grew up with relational database, may have never received a firm grounding in them. Then it discusses pointer structures in OODBMS. For reasons that will be explained later, the treatment of OODBMS pointer structures will be done in a product–neutral, rather than a product specific way. Finally, there will be a comparison of the two.

Navigational DBMS Pointer Structures

Basic Pointer Structure Concepts

There are a limited number of ways of associating related records in a computer. If there is a one–to–one relationship between the two entities that the records represent then the address of each record can be placed in the other record as a pointer to it. Consider the more general case of a multiple association, specifically in a one–to–many relationship. Suppose that a company has many branch offices and each of its employees is assigned to a single one of the offices. One way of associating the related records is to store the employee records of the employees who are assigned to a particular office immediately after the office record on the disk. Unfortunately, with this group of office and employee records sandwiched in with other such groups, there may be no room to insert a new employee record, causing considerable trouble. A second way is to store the address of the appropriate office record in each employee record and to have a list of employee record addresses in the office record. The problem here is that, in general, there is no way to know how many employees are in each office and therefore no way to know how much space to leave in each office record for employee pointers (assuming fixed–length fields).

A third approach is to have a single pointer position in each office record, which is used to point to one of its employee records. A pointer in that employee record, in turn, points to another employee record associated with the office and so on through all of the employee records for the office, establishing a pointer chain through the employee records for the office. Starting with the office record, this “child and twin” pointer arrangement can be traversed to find all of the employees in the office. This arrangement has several advantages. Each office record and each employee record always has exactly one pointer position for the office to employee association, regardless of how many employees are in an office. Furthermore, inserting a new employee record for an office does not require physically shifting records, but rather updating the pointer values. To find the office that a particular employee works in from the employee’s record, there are several possibilities. If the employee record was reached through the pointer chain from the office record, the traversing software might have kept track of the office record it began with. Each employee record could have a pointer in it that points directly to its associated office record. The pointer chain from the office record through the employee records could be circular, returning the office record after the last employee, thus permitting a traversal to the office record from any of its employee records. A backwards pointer chain could be established, starting at the last employee of an office and ending at the office record, also permitting a traversal from any employee record to its associated office record.

IMS Pointer Structures

Suppose a company wants to keep a database of the office equipment that it owns, the manufacturers from which the equipment was acquired, and the manufacturers’ employees, such as salespeople, accounting people, etc., with whom it has contact. Figure 1 shows an IMS hierarchical structure for this database. The Manufacturer record (or “segment,” in IMS terms) is the root. Each branch in the hierarchy represents a