Object-Oriented Databases: Definition and Research Directions

WON KIM

(Invited Paper)

Abstract—During the past several years, an object-oriented approach to programming and designing complex software systems has received tremendous attention in the programming languages, knowledge representation, and database disciplines. Although the object-oriented approach has become popular, there has been much confusion and controversy about what *object-oriented* means. In this paper, we will attempt to contribute towards a foundation for object-oriented databases on the basis of the research results obtained from the ORION series of object-oriented database systems. First, we will define and justify what an object-oriented database should be, on the basis of a small set of central object-oriented concepts. Then, we will shed some light on a number of common misconceptions about object-oriented databases. Next, we will outline the results of relevant recent research in object-oriented databases, and provide directions for future research in object-oriented databases.

Index Terms—Class-composition hierarchy, integration of a programming language and database, class hierarchy, object-oriented data model, object-oriented databases, object-oriented database architecture.

I. Introduction

During the past several years, application of object-oriented concepts has become an important topic of research in a number of disciplines in computer science, such as databases, programming languages, knowledge representation, and even computer architecture. Indeed, the underlying object-oriented concepts are the common thread linking these disciplines, and as such they may be the key to building one type of intelligent high-performance programming system of the foreseeable future.

There is, however, a high degree of confusion about what object-oriented means in general, and what object-oriented database is in particular. The fact that there is no consensus about *precisely* what objects are is unfortunate, in view of the fact that the foundational object-oriented concepts have evolved in three different disciplines: first in programming languages, then in artificial intelligence, and then in databases. Simula-67 [15] is generally regarded as the first object-oriented programming language. Since Simula-67, researchers in programming languages have taken two different paths to promote object-oriented programming, as surveyed in [72]. One was the development of new object-oriented languages, most notably Smalltalk [34], [35], and such languages as Traits [26], [27], Eiffel [71], Trellis/Owl [83], among others. Another was the extension of conventional languages: Flavors [77], [47], Object LISP [61], OakLisp [57], LOOPS [18], and Common LOOPS [91], [19] as extensions of LISP; Objective C [25] and C++ [95] as extensions of C; and Clascal [84] as an extension of Pascal; and so on. After Minsky's introduction of frames [73] as a knowledge representation scheme, researchers in artificial intelligence have developed such frame-based knowledge representation languages as KEE from Intellicorp, ART from Inference, and so on. In the database area, research into semantic data models has led to object-oriented concepts similar to those embedded in programming and knowledge representation languages. The class concept captures the instance-of relationship between an object and the class to which it belongs; the concept of a subclass specializing its superclass captures the generalization (IS-A) relationship [89], and the composition of an object in terms of attributes captures the aggregation relationship [89]. Some of the better-known semantic data models include E/R [22], SDM [36], and DAPLEX [87].

In this paper, we will attempt to contribute towards a foundation for object-oriented databases on the basis of the research results obtained from the ORION series of object-oriented database systems [8], [51], [52] prototyped in the Advanced Computing Technology (ACT) Program at MCC. In Section II, we will first define and justify what an object-oriented database should be, on the basis of a small set of central object-oriented concepts. Then, in Section III we will shed some light on a number of common misconceptions about object-oriented databases. In Section IV, we will outline the results of relevant recent research in object-oriented databases. In Section V, we provide directions for future research in object-oriented databases.

II. Object-Oriented Databases

A data model is a logical organization of the real-world objects (entities), constraints on them, and relationships among objects. A database language is a concrete syntax for a data model. A database system implements a data model. Just as a relational database system is a database system which implements the relational model of data, an object-oriented database system is a database system

Manuscript received October 13, 1989; revised April 5, 1990.
The author is with Unidata, Inc., Austin, TX 78750
IEEE Log Number 9036990.

1041-4347/90/0900-0327$01.00 © 1990 IEEE

which directly supports an object-oriented data model. In the absence of a single standard object-oriented data model, we propose that an object-oriented data model is one which includes at the minimum the core object-oriented concepts we will discuss and justify in this section. Of course, as any database system does, an object-oriented database system must provide persistent storage for the objects and their descriptors (schema). The system must provide the user with an interface for schema definition and modification (a data definition language), and for creating and accessing objects.

This basic system may be extended in several dimensions, just as several types of features were added to the relational database systems during the past decade. First, a query language may be defined within the confines of the core object-oriented data model. (A query language should be a part of the basic system.) Second, one may add integrity features to it, such as transaction management and triggering. Third, one may add performance-related features, such as secondary indexing and clustering. Fourth, it may be extended with concurrency control and authorization for a multiuser environment. Fifth, the core data model may be augmented with additional semantic data modeling concepts to simplify application modeling, most notably versions of objects, and composite objects (assembly-part hierarchy).

In this section, we will propose and describe a core object-oriented data model. The model is based on a set of fundamental object-oriented concepts common to most object-oriented programming and knowledge-representation languages; it has been particularly influenced by Smalltalk, Flavors, and CLOS (Common LISP Object System). Additional modeling concepts, such as versions and composite objects, are important to support specific classes of applications. However, they are supported in only a few object-oriented database systems, and as such we will not regard them as core concepts. We will attempt to shed some insight into the merits of the core object-oriented concepts from a database perspective as a basis for establishing a research agenda to be discussed in Sections IV and V. To make the discussions about the data model concrete, we will also provide a syntax for a representative subset of the data model.

A. Core Object-Oriented Data Model

Object and Object Identifier: In object-oriented systems and languages, any real-world entity is uniformly modeled as an object. Furthermore, an object is associated with a unique identifier.

Attributes and Methods: Every object has a state and a behavior. The state of an object is the set of values for the attributes of the object, and the behavior of an object is the set of methods (program code) which operate on the state of the object. The value of an attribute of an object is also an object in its own right. Furthermore, an attribute of an object may take on a single value or a set of values. A set is not an object, although each element in a set is an object (there is no clear-cut agreement on this in the literature). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing (or function calls).

Class: A class is specified as a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class. The relationship between an object and its class is the familiar instance-of relationship. A class is similar to an abstract data type. A class may also be primitive. A primitive class is one which has associated instances, but which has no attributes, such as integer, string, and Boolean.

The value of an attribute of an object, since it is necessarily an object, also belongs to some class. This class is called the domain of the attribute of the object.

Class Hierarchy and Inheritance: Object-oriented systems allow the user to derive a new class from an existing class; the new class, called a subclass of the existing class, inherits all the attributes and methods of the existing class, called the superclass of the new class. The user may also specify additional attributes and methods for the subclass. A class may have any number of subclasses. Some systems allow a class to have only one superclass, while others allow a class to have any number of superclasses. In the former, a class inherits attributes and methods from only one class; this is called *single inheritance*. In the latter, a class inherits attributes and methods from more than one superclass; this is called *multiple inheritance*. In a system which supports single inheritance, the classes form a hierarchy, called a class hierarchy. If a system supports multiple inheritance, the classes form a rooted directed graph, sometimes called a class lattice.

B. Perspectives of the Core Concepts

In this subsection, we will elaborate on each of the basic object-oriented concepts discussed earlier. The primary purpose of this section is to explain the merits of these concepts, and to begin to bring out their connections to and impacts on databases.

Object and Object Identifier: The uniform treatment of any real-world entity as an object simplifies the user's view of the real world. The object identifier is used to pinpoint an object to retrieve. The identifier of an object is not reused even when the object with which it was associated is deleted from the system. The object identifier has been introduced in object-oriented systems for at least two major reasons. First, the state of an object consists of values for the attributes of the object, and the values are themselves objects, possibly with their own states. Thus, a natural representation for the state of an object is a set of identifiers of the objects which are the values of the attributes of the object. For performance reasons, if the domain of an attribute is a primitive class, the values of the attribute are directly represented; that is, instances of a primitive class have no identifiers associated with them. For example, the domain of the Weight attribute of the class Vehicle in Fig. 1 is the primitive class integer; and the value of the Weight attribute of an instance of the

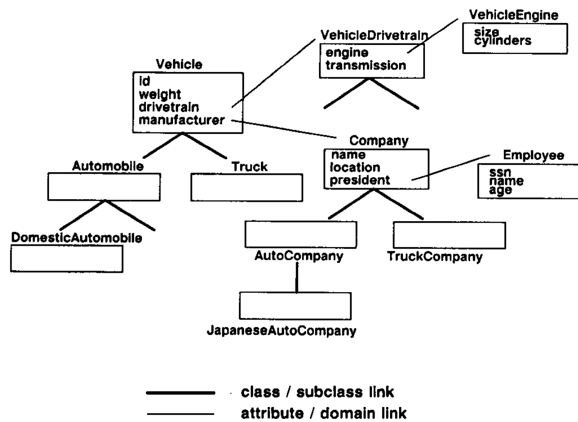

Fig. 1. Class hierarchy and class-composition hierarchy.

class Vehicle may be the integer 7500. In contrast, the domain of the Manufacturer attribute of the class Vehicle is the class Company; the value of the Manufacturer attribute of a Vehicle instance may then be the object identifier of an instance of the class Company.

Second, object-oriented systems and languages, and therefore object-oriented concepts, have been developed largely independently of any consideration of very large databases; that is, they have assumed that all objects reside in a large virtual memory. This means that object identifiers have been used as the sole means of specifying desired objects; the notion of a query for selecting an arbitrary set of objects that satisfy an arbitrary combination of search predicates has been an alien concept to the designers of object-oriented languages.

The fact that an object consists logically of object identifiers and that object identifiers are the only means of specifying objects to access has naturally led to the navigational model of computation in most of the existing object-oriented applications. This of course does not imply that object-oriented systems cannot be augmented with nonnavigational (declarative) manipulation of objects. In fact, an increasing number of object-oriented database systems support or plan to support queries, for example, ORION [10], [54], O2 [101], GemStone [20], and IRIS [13]. In view of the complex nested structure of an object (which we will discuss shortly), strictly nonnavigational manipulation of objects will not replace the navigational access. However, future object-oriented systems are likely to use nonnavigational manipulation of objects to complement the traditional navigational access to objects.

Attributes and Methods: We use the term *attribute* to mean an instance variable; an attribute also corresponds to a column of a relation in relational databases. This is contrasted with the use of the term attribute in data abstraction in which objects with the same abstract interface are grouped and the interface does not include attributes.

The domain of an attribute may be any class: user-defined or primitive. This represents a significant difference from the normalized relational model in which the domain of an attribute is restricted to a primitive class. The fact that the domain of an attribute may be an arbitrary class gives rise to the nested structure of the definition of a class. That is, a class consists of a set of attributes; the domains of some or all of the attributes may be classes with their own sets of attributes, and so on. Then the definition of a class results in a directed graph of classes rooted at that class. If the graph for the definition of a class is restricted to a strict hierarchy, the class takes on the characteristics of a nested relation (the precise differences between the nested definition of a class and a nested relation will not be given here).

We hasten to note that the above hierarchy of classes arises from the aggregation relationship between a class and its attributes and from the fact that the domain of an attribute may be an arbitrary class with its own set of attributes. This rooted directed graph, which we will call a *class-composition hierarchy*, is orthogonal to the concept of a class hierarchy. A class hierarchy captures the generalization relationship between one class and a set of classes specialized from it. A class-composition hierarchy usually has nothing to do with inheritance of attributes and methods. Fig. 1 shows an example schema. The class Vehicle is the root of a class-composition hierarchy which includes the classes VehicleDrivetrain, VehicleEngine, Company, and Employee. The class Vehicle is also the root of a class hierarchy involving the classes Automobile, DomesticAutomobile, and Truck. The class Company is in turn the root of a class hierarchy with subclasses AutoCompany, JapaneseAutoCompany, and TruckCompany. It is also the root of a class-composition hierarchy involving the class Employee.

Unlike the class hierarchy, the links in a class-composition hierarchy may form cycles. For example, if the class Company has an additional attribute, Manufactures, whose domain is the class Vehicle, there will be a link from the class Company to the class Vehicle.

Class: The concept of a class is perhaps the most important link between object-oriented systems and databases. First, it captures an important semantic data modeling concept, namely, the instance-of relationship. Second, it is the basis on which a query may be formulated. In relational databases, a query is issued against a relation or a set of relations; similarly, in object-oriented databases, a query may be issued against a class or a set of classes. (GemStone takes a different view, and supports queries issued against a collection of instances of a class [65], [20].) Without the notion of a class to aggregate together related objects, it is very difficult to conceptualize (and evaluate) a query. Third, the concept of a class can enhance the integrity of object-oriented systems by introducing type checking; the specification of a class as the domain of an attribute makes it possible for the system to restrict the values that the attribute may take on to those objects that belong to that class. Fourth, when many objects share the same set of attributes and methods, in the absence of the class concept, the names and any integrity-related specifications of the attributes must

be replicated in every object. This can not only cause a large waste of storage space, but also makes dynamic changes to the database schema completely impractical.

If we enforce the principle that an object is an instance of a class, and a class is also an object, then we need the notion of a metaclass as the class of a class [34]. Of course, a metaclass is also an object, and it is an instance of a yet higher level metaclass (the infinite recursion must be broken by an arbitrary root metaclass). Most object-oriented database systems do not support the strict notion of metaclasses. For example, in ORION, a system-defined class called CLASS is both the class of all other classes (i.e., the class CLASS is the metaclass of all other classes) and the root of the class hierarchy (i.e., it is the superclass of all other classes).

Class Hierarchy and Inheritance: A class hierarchy captures the generalization relationship between a class and its direct and indirect subclasses. The concept of a class hierarchy and inheritance of attributes and methods along the class hierarchy is what distinguishes object-oriented programming from programming with abstract data types [96]. The fact that a subclass may be specialized from an existing class can simplify the specification of the subclass, since the creator of a subclass can simply reuse the specification of the existing class. Of course, the specification of a class includes both the attributes (and any integrity constraints on them) and the methods which can operate on the class and the objects that belong to the class.

The concept of inheritance does present some problems, including conflicts in the names of attributes and methods [91], [9], and violation of encapsulation [90]. There are two types of name conflict: between a class and its superclass, and between superclasses of a class. In systems which support single inheritance, only the first type of conflict occurs. If the name of an attribute or a method conflicts between a class and a superclass, the name used in the class takes precedence; that is, the attribute or method of the superclass is not inherited. In the case of a conflict between attributes or methods of superclasses, a solution often used is to select one superclass from which to inherit the attributes or methods on the basis of a precedence ordering. Most systems require the superclass-precedence ordering to be specified in each class; some systems determine the precedence ordering at run time.

A second problem with inheritance is the violation of the encapsulation principle. [90] observes that, if one may directly access the instance variables of a class from a subclass, such operations as the renaming or dropping of an instance variable may invalidate the methods defined in the subclass which reference the instance variable. The solution proposed is to restrict access to the instance variables of a class through methods defined for them.

There has been some disagreement as to whether multiple inheritance is really necessary. Although multiple inheritance complicates the name-conflict problem, multiple inheritance is necessary. The CLOS standard and C++ already include it, and even the recent version of Smalltalk supports a form of it. Single inheritance often causes duplication of information and forces upon the users a less intuitive model of the database [9].

C. Object-Oriented Database Interface

The designers of object-oriented database systems have adopted one of two distinct approaches to the design of the user interfaces to their systems. One is the traditional database approach of defining a database language to be embedded in host programming languages. The problem with this approach is that, as has been the case with relational systems, the application programmers have to learn and use two different languages. Furthermore, the application programmers have to negotiate the differences in the data models and data structures allowed in the two languages.

Another approach is to extend object-oriented programming languages with database-related constructs. This approach makes it possible for the application programmers to learn only new constructs of the same language, rather than an entirely new language. This approach is more desirable, unless, for example, applications written in more than one language must share a common database. Systems taking this approach include ORION (extending Common LISP) [8], [51], ZEITGEIST (extending Flavors) [98] being prototyped as Texas Instruments, GemStone from Servio Logic [65], [20], and AllTalk (both extending SmallTalk) [70] at Eastman Kodak. The database interface adopted in STATICE [102] from Symbolics is based on the DAPLEX functional data model (as is also the case with IRIS [31], [32], and the JASMIN system [68] at Fujitsu); however, the STATICE interface has been integrated into the functional programming language LISP.

In this section, we will provide a concrete syntax for some of the basic database operations. The syntax can be easily extended to augment an object-oriented data model with additional semantic modeling concepts, such as composite objects and versions. We will use the message-passing syntax of Smalltalk and Flavors here; however, the function syntax of CLOS and C++ can just as easily be used. The purpose of this brief discussion is twofold. One is to make the abstract discussion of the core data model concrete. Another is to bring out the fact that an object-oriented data model can be cast into a data language which is comparable in simplicity to relational data languages.

Message Passing: All operations on an object are performed by using the message interface of the object which is implemented with a method. A message can be sent to an object (a receiver) by using the following syntax.

(Selector Receiver [Arg1 Arg2 Arg3 · · ·]).

Selector is the name of the message, and Receiver is the object to which the message is to be sent. The name of the message is identical to the name of the corresponding method. The optional arguments, Arg1, Arg2, etc., are objects or can be evaluated to objects. Since a mes-

sage returns an object, an argument may itself be a message. Similarly, the Receiver of a message may also be the result of some other message.

Class Definition: A new class may be defined with the following message.

(**make-class** Classname [**:superclasses** ListofSuperclasses]
[**:attributes** ListofAttributes]
[**:methods** ListofMethodSpecs]).

Classname is the name of the new class. Each of the keyword arguments following the Classname is optional. The ListofSuperclasses associated with the **:superclasses** keyword is a list of the superclasses of the new class (this simple construct captures the class-hierarchy concept). The ListofAttributes associated with the **:attribute** keyword is a list of attribute specifications (this is the generalization of the familiar syntax for the definition of relations in relational databases). An attribute specification is a list consisting of an attribute name and keywords with associated values, as follows:

(AttributeName [**:domain** DomainSpec]
[**:inherit-from** Superclass]).

A DomainSpec specifies the domain of an attribute. The keyword **:inherit-from** is used to control inheritance. If the keyword is not provided, the attribute is a new attribute for the class being defined. If the keyword is provided, the Superclass specified is the name of the superclass from which the attribute will be inherited; if the Superclass is not given, the attribute is inherited from the first superclass in the ListofSuperclasses.

The ListofMethodSpecs associated with the **:methods** keyword is a list of pairs (MethodName Superclass). The MethodName is the name of a method to be inherited from the Superclass. If Superclass is not specified, the method is a new method for the class being defined.

Object Manipulation: The syntax for object manipulation, that is for the creation, query, delete, and update, is similar to that for relational database manipulation, except that the query expression in a query is significantly different from that for relational databases. An instance can be created by sending a **make** message to the class to which the instance will belong.

(**make** Classname :Attribute1 value1
...
:AttributeN valueN).

To select instances of a class that satisfy a given query expression, we may use a **select** message.

(**select** Class QueryExpression)

where QueryExpression is a Boolean expression of predicates.

A set of objects (possibly an empty set) containing qualifying instances of the class is returned.

To delete all instances of a class that satisfy a given query expression, a **delete** message may be used.

(**delete** Class QueryExpression).

To delete a specific object, a **delete-object** message is used.

(**delete-object** Object)

where Object is the object identifier.

Similarly, a **change** message may be used to replace the value of an attribute of all instances of a class that satisfy a given Boolean expression.

(**change** Class
[QueryExpression] AttributeName NewValue).

III. Overlaps with Other Areas and Standardization

There have been two reasons for the confusion about object-oriented databases. One is largely the absence of an object-oriented standard in programming languages and artificial intelligence. Another is that some of the fundamental object-oriented concepts are aspects of several other areas of database research. In this section, we will attempt to shed some light on the two reasons.

A. Overlaps with Other Areas

Design Databases: At about the time relational database systems first became commercially available, researchers in databases and computer-aided design (CAD) recognized important limitations of the data model and transaction model supported in relational (and prerelational) database systems, giving impetus to research into design databases (also called engineering databases or CAD databases) [39], [40], [2]. The relational model is not rich enough to admit the nested construction of complex designs or to properly capture the semantics of versions and representations (views) of a design. To allow the modeling of the nested construction of a complex design, a number of researchers proposed modifications to the relational model. These proposals include complex objects [37], [63], [49] or nested relations (which formalizes the notion of complex objects) [67], [43], [1], [28], [99], [30], [42]. To capture the semantics of versions and representations of a design, a number of researchers have proposed models of design objects [11], [46]. Although they may be structurally similar to complex objects, versions or representations of a design object have very different semantics; for example, the versions of a design object are related through the version-of relationships and specific policies may be imposed on them with respect to their creation, update, and deletion. A complex object simply represents the nested composition of a complex artifact; the components of a complex object are related through the consists-of relationships.

Sometimes complex objects or design objects have been mistaken for objects from object-oriented programming languages. The computer-aided design and engineering

(CAD/CAE) communities have recognized the need for databases and object-oriented concepts for very different purposes. CAD/CAE systems need database support largely to offload the chore of managing complex interrelationships among many design components (version management and configuration management). The object-oriented approach facilitates the development and maintenance of the CAD/CAE system software, and it is a rather natural paradigm for the end users (designers and engineers) to interact with the system. What these systems need is database support for objects which carry the object semantics found in object-oriented programming languages. As we have already discussed, complex objects represent at best one aspect of an object-oriented data model, and the basic object-oriented concepts are not sufficient for modeling design objects.

Hierarchical and Network Databases: There are at least two types of similarities between object-oriented and hierarchical and network databases. One important similarity is the nested structure of objects in object-oriented databases, and the nested structure of records in hierarchical databases. Although both databases admit objects (records) which refer to other objects (records) for the values of their attributes, there is an important difference. The nested object schema in object-oriented databases contains cycles [10]; although hierarchical databases can admit cycles, they require artificial record types to be introduced in the schema.

Another similarity is between the object identifiers in object-oriented databases and the record pointers in hierarchical databases. However, an object identifier is a logical pointer and is never reused, and as such it may be used for enforcing referential integrity. A record pointer is a physical pointer and is reused.

However, there are major differences that contrast object-oriented databases from hierarchical and network databases. Object-oriented databases support such concepts as a class hierarchy, inheritance, and methods; hierarchical and network databases obviously do not include these concepts.

Extensible Databases: Extensible databases share one thing with object-oriented databases, namely, extensibility. The goal of research into extensible database systems is to find approaches for building a database system such that the system may be easily extended to accommodate new functionality [60], [21], or for building a database system by assembling components from a library of database-system building blocks [12]. The notion of inheritance is what makes systems implemented in an object-oriented style (i.e., which make use of the data encapsulation and inheritance principles) rather extensible. If a database system is implemented in an object-oriented style or in an object-oriented programming language, then it tends to make it easier to add new database functionality (i.e., more extensible) than if it is implemented in conventional programming style. However, extensibility of a database system is merely a characteristic of the architecture of a database system, rather than a requirement for an object-oriented database system. Even if an object-oriented database system is not implemented in an object-oriented style, it will still be easy for the user to add new classes and data types to the system. To our knowledge, no object-oriented database systems operational today have been implemented entirely in object-oriented programming languages. Some systems have implemented some of their components in an object-oriented style or object-oriented programming language. For example, the multimedia data management subsystem of ORION [104] has been implemented by making direct use of the message passing and class hierarchy (and inheritance) concepts to make it easy to add new types of multimedia data and devices to the system.

Semantic Databases: Semantic data models, such as the Entity Relationship Model, the DAPLEX functional model, and the Semantic Data Model, attempt to capture explicitly a rich set of semantic relationships among real-world entities. The generalization/specialization relationship between a superclass and its subclass, the aggregation relationship between a class and its attributes, and the instance-of relationship between an instance and its class (and the superclass of the class) are all included in semantic data models.

In terms of modeling power, we view a core object-oriented data model largely as a subset of a semantic data model; of course, semantic data models lack methods. For reasons of performance and ease of use, the core object-oriented model needs to be extended to include additional semantic modeling concepts for specific classes of applications. The most relevant of such concepts include versions and composite objects (assembly-part hierarchy). These modeling concepts allow the user to deal with a collection of related objects as a single unit. For example, a composite (complex) object is a collection of objects related by the part-of relationship, and it may be used as a unit of access in a query and a unit of integrity.

Relational Databases: To the best of our knowledge, nobody mistakes a relational database for an object-oriented database. There are clear differences. An object-oriented data model has the notions of a class hierarchy, a class-composition hierarchy (for nested objects), and methods—none of which is a part of the normalized relational model. There are efforts to extend the relational model or language with these fundamental object-oriented concepts. POSTGRES [94], [82] is the best example of these efforts.

There have been a number of criticisms, some valid and some invalid, of object-oriented databases, mostly from proponents of relational databases. We will now address these points. The navigational model of computation used in object-oriented applications apparently has given rise to the criticism that object-oriented databases represent a throwback to the days of hierarchical and network databases. This criticism focuses on only one aspect of object-oriented systems. One has to keep in mind that navigational access is used primarily to selectively traverse a set of complex nested objects, and that there is more to an

object-oriented data model than just the nesting of an object. Furthermore, there are important applications from computer-aided design and artificial intelligence which absolutely must navigate through a large database. For example, navigation and object identifiers make it possible to traverse a tree structure often used in such applications without resorting to value-based retrieval of objects (records) pointed to by a given object (record) [66].

It is true that object-oriented data models proposed thus far are not based on an elegant mathematical theory [56], as is the case for the relational model. However, as we will show in the next sections, object-oriented databases can be an exciting area of research even in the absence of a mathematical theory, although it will no doubt be more satisfying if it is based on an elegant mathematical theory. Furthermore, a number of researchers are currently attempting to define a query model and an "object-oriented algebra" (corresponding to the relational algebra).

Object-oriented databases are sometimes criticized for the apparent complexity of inheritance [56]. This is not a particularly useful criticism, if we recognize that one major objective of database systems is to directly support the data modeling requirements of their intended applications. In particular, one major objective of object-oriented database systems is to directly support object-oriented concepts, which are the data modeling requirements of object-oriented applications.

B. Standardization

The programming language, knowledge representation, and database disciplines, and even any one of them by itself, presently do not agree on a single standard for object-oriented concepts. However, we expect that only a limited number of object-oriented data models will be widely accepted in the near future, and efforts for a standard object-oriented data model are currently underway. Although it is true that there is no consensus about *precisely* what object-oriented means, if one examines existing object-oriented programming languages, knowledge representation languages, and semantic data models, one can identify a small set of fundamental concepts which are common to many of them. The trends today indicate that soon each discipline will adopt a small number of de facto standards, based largely on the commercial success of some of the programming languages, knowledge representation languages, and database systems. Recently, CLOS has been proposed as a standard object-oriented extension to Common LISP. Furthermore, Objective C [25] and C++ [95] have become popular; and it appears that the popularity of C++ will significantly increase, now that AT&T has announced its plans to support it, and efficient compilers are becoming available, even in the public domain [100]. The knowledge representation languages may be standardized around such commercially successful products as KEE. A number of object-oriented database systems have become commercially available recently, including Ontos from Ontologic, GemStone from ServioLogic, and ORION II from Itasca Systems.

IV. Relevant Research: Some Results and Suggestions

In this section and the next, we will outline an agenda of relevant research in object-oriented databases. In this section, we will summarize some of the more interesting results obtained thus far, also pointing out some of the problems that have not yet been satisfactorily addressed by the current results. The interested reader should refer to the literature for detailed discussions of the issues and solutions we summarize in this section.

We can group relevant research into two categories. One is the integration of an object-oriented programming language with a database system, in effect, support for persistent programming. Another is the database system architecture; this is the focus of discussion in this section.

A. Integration of a Programming Language and a Database System

The combined notions of a class, attributes, and a class hierarchy mean that the semantic data modeling concepts instance-of, aggregation, and generalization are inherent in the object-oriented paradigm. This means that the gap between applications implemented in an object-oriented programming language (or merely in an object-oriented style) and an object-oriented database is much narrower than that between object-oriented applications and non-object-oriented database. In particular, one of the problems with implementing object-oriented applications on top of a relational database system is that a relational system does not directly support a class hierarchy and the nested definition of a class, and as such the application programmers must map these constructs to relations. The gap is also much narrower than that which exists between conventional programming languages and conventional database systems, for example, between a PL/1 program and IBM's SQL/DS relational database system [38]. Since object-oriented programmers or designers already model their application artifacts in the object-oriented paradigm, mapping them to objects supported in an object-oriented database system should not introduce a large measure of what has been labeled as "impedance mismatch" [24]. In fact, the removal of the impedance mismatch between a programming language and a database is the goal of research into persistent programming.

Many object-oriented database systems have integrated an object-oriented programming language and a database system, that is, to extend a programming language with database support. GemStone [65] and AllTalk [70] extended Smalltalk with database support, while ORION [51] and Statice [102] provide database support for object-oriented extensions to Common LISP. VBase [4] provides database support for to a proprietary object-oriented programming language called COPS and TDL. The key technique used to support persistent programming in these systems is the management of workspace, or in-memory

object buffers. The LOOM (large object-oriented memory) system [44] developed at Xerox introduced this technique. A directory of in-memory objects is maintained, and a descriptor is associated with each in-memory object. An object is referenced by any number of other objects through its descriptor. The descriptor of an object has the memory pointer to the object, and other control information to enhance the system performance.

B. Database System Architecture

The implementers of currently operational object-oriented database systems have no doubt had to make a large number of decisions in architecting and implementing their systems, including storage structures for objects, concurrency control and recovery, query processing, schema definition and modification, and even authorization. ORION, GemStone, and IRIS provide richer sets of features than other systems. Many of the techniques used in implementing conventional database systems, especially query optimization and processing, concurrency control and recovery, and disk storage layout, are largely applicable to building object-oriented database systems. Since these techniques have been extensively documented in the literature, we will not discuss them. However, object-oriented concepts often require either entirely new techniques or significant extensions to the conventional techniques for architecting a database system. In this section, we will focus only on those results which have revealed the impacts of the core object-oriented concepts on the architecture of a database system. This section is organized around the core object-oriented concepts discussed in Section II.

1) Object Identifier: In object-oriented systems, each object has a system-generated unique identifier. A few different approaches are possible in constructing object identifiers. In one approach, used in ORION, an object identifier consists of a ⟨class identifier, instance identifier⟩ pair, where the class identifier is the identifier of the class to which the object belongs, and the instance identifier is the identifier of the instance either within the class or within the entire database. The complete specification of the attributes and methods for all instances of a class is maintained in a class object. When a message is sent to an object, the system can extract the class identifier of the object from the object identifier specified, and look up the class object to determine if the message is valid, and then may fetch the object and dispatch the corresponding method. Class objects may be cached to optimize system performance.

In another approach, used in Smalltalk, an object identifier consists only of an instance identifier. This approach still requires the class identifier of an object to be maintained; however, it is stored in a separate system-defined attribute of an object. In systems using this approach, message processing may be somewhat inefficient. When a message is sent to an object, the system must first fetch the object, examine the class identifier in the object, and then look up the methods stored in the class object. This implies that invalid messages cause needless fetching of objects. Furthermore, this approach renders run-time type checking expensive, since the types (domains) of the objects referenced in an object can only be determined by actually fetching the objects and examining the class identifiers stored in them.

The fact that an object must maintain the class identifier, either as part of the object identifier or as a separate system-defined attribute, may restrict the mobility of objects from one class to another class. If an object migrates from one class to another, its class identifier must be updated; that is, the object must be updated. Object migration is considerably more difficult in systems in which an object identifier contains the class identifier. An object may reference any number of other objects, and an object may be referenced by any number of other objects. This means that, if the class identifier of an object is changed, all objects which contain references to the object will wind up with invalid references. It is of course very expensive to be able to identify such objects and update their references to the new identifier.

2) Class Hierarchy and Class-Composition Hierarchy: An object-oriented data model, as it overlaps substantially with semantic data models, is richer than the relational model. In particular, the generalization and aggregation relationships inherent in object-oriented data models have required a reexamination of a number of architectural concepts developed for relational database systems, including schema evolution, queries, secondary indexing, authorization, concurrency control, and storage structures.

a) Schema Evolution: Relational database systems allow a new relation to be created, an existing relation to be dropped, and a new column to be added to an existing relation [38]. Usually, all the system has to do is simply to record the existence or absence of the new relation and column in the system (the schema relations). Other existing relations and columns are not impacted by the schema changes. This simple situation does not hold for object-oriented databases.

The database schema for an object-oriented database has two dimensions. One dimension is the class hierarchy which captures the generalization relationship between a class and its subclasses. Another dimension is the class-composition hierarchy which represents the aggregation relationship between a class and its attributes and the domains of the attributes. In other words, every class in an object-oriented database simultaneously belongs somewhere in the class hierarchy and somewhere in the class-composition hierarchy. The semantics of the class hierarchy is what complicates schema changes. For example, when a class is dropped, all its subclasses will lose the attributes and methods they had inherited from the class, and therefore instances of the subclasses will lose the values of the attributes. Furthermore, when a new class is added and the class will inherit attributes and methods

from any existing classes specified as its superclasses, and will also provide attributes and methods for its subclasses to inherit from it. The class hierarchy also gives rise to a number of meaningful schema changes beyond those possible under the relational model. For example, making an existing class a new superclass of another existing class is a meaningful operation.

Research into schema evolution for an object-oriented data model has been conducted in the context of ORION, GemStone, and the ObServer object server [88] at Brown University. The ORION [9] and ObServer philosophy is to mask the effects of schema changes on instance objects, whereas the GemStone approach [80] is to immediately reflect the schema changes on instance objects.

One important shortcoming of the current framework for schema evolution is that it assumes a single schema. This means that all schema changes one user makes will impact all other users' view of the database; for example, once any user deletes an attribute from a class, or changes the superclass/subclass relationship between a pair of classes, all other users will see the changes. This problem is by no means unique to object-oriented databases; the same problem has existed in relational databases. The traditional solution to this problem has been to limit the privilege to make schema changes to database administrators or to support views over stored relations. A good solution would be to support versions or views of schema, so that different users may view the database through different versions or views of the schema. The idea was first proposed in the context of the ObServer project [88], and a comprehensive model of versions of schema has been proposed for ORION [50]. This model still needs to be validated through implementation. Another shortcoming in the current framework for schema evolution is that it really does not address the problem of generalizing existing classes, that is, creating a new superclass for existing classes and factoring out attributes and methods common to the classes into the superclass. The approach suggested in [85] is a useful starting point for addressing this problem.

b) Query: Because of the nested structure of the definition of a class, one may use the nested-relational model as the starting point for defining a query model for object-oriented databases. In this regard, the current research into the theory of nested relations is likely to yield results relevant to object-oriented databases. However, the theory of nested relations is inadequate as a model of queries for object-oriented databases. There are a few important reasons for this. First, the definition of a class may form a directed cyclic graph, as we discussed earlier, while the nested-relational model deals with a strict hierarchy of relations. Second, the current theory of nested relations has not taken into account some of the object-oriented concepts.

The query model developed for ORION [10], despite its limitations, is one of the few which are based on an explicit consideration of the power and constraints of object-oriented concepts. The model restricts the target of a query to a single class or a class hierarchy rooted at that class. This is an important restriction, since this excludes relational-join-like queries. (We will return to this point later.) However, the model explicitly takes into consideration some of the important consequences of object-oriented concepts. First, it allows the user to use the directed graph model of the definition of the target class for specifying a query; predicates may be applied to any attributes of any classes on the graph. This is similar to the nested-relational extensions of the relational selection operation. Second, a query may be directed against a single class or a class hierarchy rooted at the class. This is important, since a class hierarchy captures the IS-A relationship between a class and all its subclasses, and as such instances of a class may be regarded as belonging to the class and all classes on the superclass chain starting from the class. In fact, the domain of an attribute of a class is the specified class and all direct and indirect subclasses of the class.

As we discussed earlier, object-oriented systems model every real-world entity as an object with a unique identifier, the object belongs to a class, and a class has a position somewhere in the class hierarchy. Designers of object-oriented query languages must never forget these fundamental principles. These simple principles impose some difficult constraints on the query model for object-oriented databases. In relational databases, the result of a query is itself a relation. This means that the result of a single-class query involving a projection of some of the attributes is simply a relation with a subset of the attributes of the original relation. Furthermore, the result of joining two relations is a relation with a union of the attributes of the original relations. The situation is not so simple for object-oriented databases. (We are suggesting that there are issues to consider, not that the problem is impossible to solve.) One difficulty is that, because of the nested definition of classes, the join and set operations have to be defined over nested classes. This is one of the points that has not been taken into account in some of the proposed object-oriented query languages. The current theory of the nested-relational model does not provide a pragmatic definition of the join operation. One interesting challenge is to define a query model for object-oriented databases which will admit operations equivalent to relational joins and set operators and which will honor all fundamental principles of the object-oriented systems. [54] is one of the first attempts to define such a query model.

Despite the differences in the data models, object-oriented queries may be evaluated in a manner similar to relational queries [86]. This observation is not really surprising. As discussed earlier, the object-oriented database equivalent of the relational selection operation is simply the retrieval of the instances of the target class, and the retrieval of these instances requires retrieval of the instances of other classes they recursively reference as values of their attributes. The retrieval of an instance of a

class $C2$ which is the value of an attribute A of an instance of a class $C1$ is the familiar join operation between the classes $C1$ and $C2$, in which the join attributes are the attribute A of the class $C1$ and the system-defined "object-identifier attribute" of the class $C2$. Just as n relations may be joined in $n!$ permutations of the relations, n classes on the class-composition hierarchy of the target class may be "joined" in $n!$ permutations, although of course some permutations yield Cartesian product results and may be omitted from consideration.

c) Indexing:

Class-hierarchy indexing: The generalization relationship in the class hierarchy has an interesting consequence on secondary indexing. In relational systems, when a query is specified against a relation, the relation is obviously the only relation to be searched. In object-oriented systems, there are two meaningful interpretations for the target of a query issued against a class. One is obviously the class. The other is the class hierarchy rooted at the class, that is, the class and all its direct and indirect subclasses, since each subclass "IS-A" user-specified class.

This interpretation may also be extended to the domain of an attribute. When the user specifies a class D as the domain of an attribute of a class C, the attribute may take on as its values objects from the class D and any direct or indirect subclass of D.

In relational database systems, one secondary index is maintained for a combination of attributes of one relation. Since a class inherits attributes from its superclasses, all direct and indirect subclasses of a class will share the same attributes. Therefore, in object-oriented databases, it may make sense to maintain one secondary index for a combination of attributes for all classes rooted at a user-specified class, instead of maintaining one index per class for the class hierarchy. This idea is similar to the proposal in [74] and [75] for maintaining a secondary index for an attribute common to more than one relation.

The technique of indexing attributes for a class hierarchy is called *class-hierarchy indexing*, while the conventional approach of indexing attributes per class may be called *single-class indexing*. On the basis of an analysis of the storage space and performance tradeoffs, ORION supports class-hierarchy indexing [53].

Nested-Attribute Indexing: We will use the term nested-attribute indexing to refer to indexing on a class-composition hierarchy. In a nested-attribute index on a class, the attribute indexed is an indirect, nested attribute of the class. In other words, the attribute indexed is not an attribute of the indexed class. For example, the domain of the Manufacturer attribute of the class Vehicle is the class Company; and the class Company has the Location attribute. Then Location is a nested-attribute of the class Vehicle, since the class Company is the domain of an attribute of the class Vehicle. In a nested-attribute index on a class, the index record associates the value of the attribute with a list of object identifiers of the instances of the class. For example, in the nested-attribute index on the Location nested-attribute of the class Vehicle, an index record associates a distinct key value of the Location attribute, say "Detroit," with a list of object identifiers of Vehicle whose Manufacturer is an instance of the Company class whose Location is the key value (i.e., vehicle manufactured in Detroit). Nested-attribute indexing makes it possible to evaluate a type of complex query by traversing a single index. The type of query for which nested-attribute indexing is ideally suited is one which contains a predicate on a deeply nested attribute of the indexed class.

The properties of nested-attribute indexing have not been studied in enough detail. However, it appears that it will be rather complex and expensive to update a nested-attribute index. A nested-attribute index on a class need potentially be updated, upon insert, delete, and update of instances of any class in the sequence of classes between the class to which the indexed attribute actually belongs and the class for which the index is maintained. [64] proposes an indexing technique called *identity indexing*; it consists of a set of indexes on the object-identifier attribute of the classes on a class-composition hierarchy. This technique incurs low update overhead, but is slower than nested-attribute indexing for retrieval.

d) Authorization: Object-oriented concepts also require rethinking of the authorization model implemented in relational database systems. The current model of authorization has been designed for the relational model of data, and as such, the units of authorization are the database, a relation, and a column of a relation. The generalization and aggregation relationships and inheritance on the class hierarchy make it difficult to apply the current model of authorization to object-oriented databases. ORION makes a first attempt to define an authorization model for object-oriented and semantic databases [81].

We will consider some of the questions related to authorization for object-oriented databases. First, the right to create a subclass of an existing class must be defined as a new type of authorization. This is different from an authorization to simply create a class, since creating a subclass means inheriting attributes and methods defined in one or more existing classes.

Second, a new unit of authorization may need to be added, namely a nested object. Since an object is in general a potentially large collection of objects related through the aggregation relationship, it may make sense to add an object as a unit of authorization. The case for treating an object as a unit of authorization becomes stronger, when the basic object-oriented data model is extended with versions. A versionable object consists of a number of related versions, and an authorization on a versionable object implies the same authorization on each of the related versions [81].

There is one aspect of inheritance which does not actually complicate the authorization model; however, since it may appear to do so, we will discuss it briefly. The issue is whether to treat an authorization on a class as a property of the class, so that subclasses of the class will inherit it. This is undesirable, since it implies that a user

with an authorization on a class will have the same authorization on all subclasses of the class created by different users. This leads in particular to the view that the creator of a class automatically has a full authorization on all subclasses and instances of the subclasses created by other users. It is more reasonable to require the other users to explicitly grant the creator of the superclass appropriate authorizations on the subclasses.

e) Concurrency Control: The class hierarchy introduces a new dimension to the problems of concurrency control in database systems. In object-oriented systems, a class inherits attributes and methods from its superclasses, and if attributes or methods are deleted from or added to a class, they will also be deleted from or added to all its subclasses. This means that while a transaction accesses instances of a class, another transaction should not be able to modify the definition of any of the superclasses of the class.

Furthermore, a query directed against a class in general requires evaluation against not only that class, but also all its subclasses. Also, the domain of an attribute of the target class is in general also a class hierarchy rooted at the domain class. This means that while a transaction is evaluating a query, a set of class subhierarchies must not be modified by a conflicting transaction.

ORION supports an extension of the traditional granularity locking protocol to address the concurrency control problems which the class hierarchy introduces in object-oriented database systems. Two locking protocols are considered [33]. One is based on explicit locking of all subclasses of a class to be accessed. For example, if the definition of a class is to be modified, the class and all its subclasses are locked in exclusive mode. Furthermore, if a class hierarchy rooted at a particular class is to be accessed for query evaluation, every class in the class hierarchy is locked in share mode. The other protocol is based on implicit locking of the subclasses of a class to be accessed. If the definition of a class is to be updated, only that class is locked in write mode (different from and more powerful than the exclusive mode), and all its subclasses are implicitly locked in write mode. However, this also requires intention write locks on all classes on a superclass chain of the class whose definition is being updated. It is not known at this time which of the two protocols is more appropriate, or whether there may be other, possibly better, protocols.

f) Storage Structure:

Class hierarchy: Relational database systems represent the database schema in the form of a set of relations, including a relation for all other relations in the database, a relation for all columns of each relation, and so on. It is more difficult to represent and maintain the schema of an object-oriented database, since the schema is no longer a simple collection of largely independent relations, but a collection of classes which are interrelated to one another through the generalization and aggregation relationships.

The class hierarchy and inheritance entail a performance problem. If the class hierarchy is maintained in the system such that each class represents only the attributes defined for it, processing a message will require a search up the class hierarchy to identify the superclass from which the class of the target object inherited the attribute or method in question. This has led the designers of a number of object-oriented systems to flatten the class hierarchy, that is, to keep in each class information about all the attributes and methods applicable to the class, both defined for the class and inherited from its superclass chain [9], [105].

Clustering: To expedite the retrieval of related data, database systems often take hints from the users (or database administrators) to store related data physically close together. In relational database systems, tuples of a relation may be stored in the same segment of disk pages on the basis of the values in a column (or a combination of columns) of the relation. Furthermore, in systems, such as SQL/DS, tuples of different relations may be stored in the same segment to facilitate joining of the relations.

In object-oriented databases, as in relational databases, it is useful to cluster objects belonging to the same class in one contiguous segment. Furthermore, since a nested object consists of objects belonging to a number of different classes, it also makes sense to be able to store objects of different classes in the same segment, as in SQL/DS. However, not all constituents of a nested object are equally likely to be of interest to any given application; for example, it may be useful to store physical components of a vehicle object in the same segment, but not the company object for the manufacturer attribute of the vehicle object. In other words, it may make sense to cluster some, but not all, constituent objects of a nested object. This means unfortunately that the users will need to specify a subgraph of the nested schema graph of a class for clustering. Furthermore, as in relational systems, it is expensive to dynamically cluster and decluster in one segment a set of classes in an object-oriented database, and clustering of nested objects will of course cause objects of any single class to be dispersed over a larger number of pages than if the class is the only class to be stored in one segment.

V. Directions for Future Research

Object-oriented databases are still a fertile ground for research. We now offer some thoughts on future research directions, in addition to the suggestions in Section IV.

Formalization: First of all, there is a clear need to formalize and/or at least standardize object-oriented concepts if a true foundation for object-oriented databases is to be laid. The notions of inheritance and queries (and query processing) are topics for future theoretical research. There has been an interesting debate about the merits of inheritance and delegation [92] as a mechanism for information sharing. In view of the fact that the importance of an object-oriented approach is founded on the reusability and extensibility it offers, it is obvious that more research should be directed to the notions of inheritance and/or delegation. Furthermore, the current at-

tempt to extend relational query languages into object-oriented query languages has resulted in languages which are not compatible with some of the object-oriented concepts. The ultimate solution may be an amalgamation of the proposals that account for the class hierarchy and the proposals for query languages for the nested relational model. An ultimate language may have to be powerful enough to support operations equivalent to relational joins and set operations.

Database Design Tools: The richness of an object-oriented data model is a mixed blessing. On one hand, it makes it easier for the users to model their applications. On the other hand, the complexity of the object-oriented schema significantly complicates the problems of logical database design and physical database design. Thus, the need for friendly and efficient design aids for the logical design and physical design of object-oriented databases is significantly stronger than that for relational databases. We have already mentioned the problems of clustering object-oriented databases, and the need for class-hierarchy indexes to expedite the evaluation of object-oriented queries. The current research into storage structures for nonfirst normal form relational databases [30] is certainly relevant for the physical design of object-oriented databases. The framework for the evolution of an object-oriented database which the ORION research established [9] is an important first step for the logical design of object-oriented databases.

Optimization of High-Frequency Operations: There are a number of very frequent operations in any object-oriented system. Applications send messages to objects, by presenting the system with the logical identifiers of the objects. The system must be very efficient in determining and dispatching the corresponding methods. Furthermore, the system must determine the physical location of the objects very fast; the objects may or may not be in memory. This in turn means that, if the objects are on disk, the mapping of the logical identifiers of the objects to their physical addresses must be done very fast. It is obviously important to identify very frequent operations in object-oriented database systems and optimize their performance; although microcoding is an option for some of them, it is not desirable since it will make the system less portable.

Language-Independent Kernel: An important question that arises, in the absence of a common definition for object-oriented concepts, is then whether a different database system must be built for each different object-oriented data model, or if a number of applications based on "similar" data models may use a common database system by embedding any necessary mapping in the applications to account for model incompatibilities among the applications.

It would be highly undesirable for "too many" object-oriented database systems to proliferate, each supporting a different data model. The approach to building an object-oriented database system that we would like to encourage is to define a core data model, directly support the core data model in a database system, and force the application programmers to bridge the differences between the core data model and the application's model of objects which may differ from the core model. Of course, even a standard core object-oriented data model has not yet been established. However, this is the general approach taken in many of the object-oriented database systems which have been built. This approach, although not aesthetically appealing, is no different from that espoused in the current database technology. The relational data model, for example, does not directly capture some of the semantic data modeling concepts, such as generalization. The users of a relational database system are forced to map their application model to the more primitive relational model. In this regard, insofar as a number of important semantic data modeling concepts are inherent in an object-oriented data model, an object-oriented database system narrows the gap between the application model and the database-supported data model. We expect that the mapping from an arbitrary (but "reasonable") variation of the core object-oriented data model to the core model may not be overly burdensome to the application programmers. One case in point is the fact that the ORION system does not directly support the notion of metaclasses. An application of ORION, the PROTEUS expert system shell, however, required metaclasses. As a result, the PROTEUS system had to map the metaclasses to ORION classes; the mapping, although unpleasant, was not very complex [5].

One interesting observation made in [7] is that a relatively small change in an object-oriented data model may require substantial changes to the architecture of a database system. The concept of a metaclass is a case in point. Although it is not very difficult to do a kludge mapping of metaclasses to the ORION data model, we have already determined that to directly implement metaclasses in ORION will indeed require fairly significant changes to a number of different components of ORION. There may be a few other significant variations of and embellishments to the core object-oriented model which will impact the architecture of a database system which directly supports a core object-oriented data model.

One interesting approach to building object-oriented database systems is to define a storage-level subsystem, and use it as the kernel to support a large number of different object-oriented data models. A different higher layer may be built on top of the common kernel to directly support a specific object-oriented data model. This will be an interesting and potentially fruitful area of research. Such efforts as DASDBS [79] at the University of Darmstadt are aimed at defining a kernel for different higher level data models. The O2 project at Altair, France [7], HP's IRIS project, and the ObServer project are also efforts to provide a common storage for a number of different object-oriented language front-ends.

Distributed Object Management: Most of the current object-oriented database systems, such as VBase, Statice, IRIS, GemStone, one version of ORION, and O2, have

adopted a client/server architecture. A client/server system is one in which a centralized object server manages the entire persistent database on behalf of a number of client machines, and each user is provided with a private workspace on a client machine in which copies of persistent objects are cached.

A client/server system is a restricted distributed database system. A fully distributed object-oriented database system is one in which the objects are distributed across different sites on a network, and the physical distribution of objects is completely transparent to the users of the system. A distributed version of ORION is currently operational, and the AVANCE system [3], [16] prototyped at the University of Stockholm is designed to be a distributed object-oriented database system; there are some distributed Smalltalk systems [29], [14], [69] and object-oriented storage systems [103].

It required major efforts to extend uniprocessor relational systems System R [17] and INGRES [93] to distributed systems R* [59], [62], [76] and INGRES*, respectively. We expect that less effort will be necessary to extend the functionality of a uniprocessor object-oriented database system to a distributed system. The reason is that much of the technology developed for distributed relational database systems [58] can be directly applied to building distributed object-oriented database systems. However, architectural techniques necessary for building a uniprocessor object-oriented database system must be extended and merged with distributed relational database technology; this will still require significant efforts and ingenuity, and will certainly be a fertile ground for interesting research.

Semantic Modeling: There are two types of research areas which are important, but in some sense somewhat tangential to object-oriented database research. One is augmenting the data modeling power of the core object-oriented concepts with semantic data modeling concepts. To be sure, a core object-oriented data model captures a number of key semantic data modeling concepts. However, it by no means captures all important concepts, most notable versions, composite objects, and inverse relationships. VBase [4] and IRIS [32] support the inverse relationship, and ORION has extended a core object-oriented data model with versions [23] and composite objects [49]. There are additional semantic modeling concepts which may be important for some major applications. Just as the object-oriented concepts in a core object-oriented data model necessitated extensions and changes to the architecture of a database system, we expect that some of the additional semantic modeling concepts will have further impacts on the architecture of an object-oriented database system, including query evaluation, storage structures, and concurrency control.

Additional Database Features: A second area, again somewhat tangential to object-oriented database research, is enhancing the functions of object-oriented database systems in support of the application environments. As we mentioned earlier, the impetus to research into object-oriented databases has come from the increasing use of the object-oriented approach in the design and implementation of AI, CAD, and OIS systems. These systems aim to facilitate collaborative and interactive access to an integrated database. The way in which the end users use thcse systems or interact among themselves requires fundamental changes in the notion of database integrity and transactions. The conventional model of transactions is simply unacceptable in engineering and design systems, where the duration of a transaction is long, lasting hours and days. The conventional model shields each transaction from the effects of all other concurrently running transactions. As such, when locking is used to control concurrency, once a transaction holds a lock on an entity, all other transactions requiring conflicting access to the entity are blocked; if optimistic concurrency control is used, a transaction which has been allowed to proceed to its commit point must be undone, if it had violated integrity. Furthermore, if a transaction is to be aborted, all updates of the transaction must be undone. It is these aspects of the conventional transaction model which make it unacceptable for long-duration transactions. There have been a number of proposals for modeling long-duration transactions [48], [45], [6], [55], and a proposal supported in GemStone to combine the pessimistic locking protocol with an optimistic concurrency-control protocol. However, we feel that more research is needed before a truly satisfactory model can be found which will remove the undesirable aspects of the conventional model of transaction and yet will ensure some notion of database integrity.

VI. Summary

In this paper, we provided and discussed a working definition of object-oriented databases on the basis of a small number of fundamental modeling and programming concepts common to object-oriented programming languages, knowledge-representation languages, and application systems. Next, we discussed a number of common misconceptions about object-oriented databases. Then, on the basis of this working definition, we outlined noteworthy results of relevant research in object-oriented databases, pointing out further research to strengthen the results, and then we provided directions for longer term future research in object-oriented databases.

Acknowledgment

The members of the ORION project worked with dedication and distinction to make ORION a reality. The original members of the ORION project, who brought up the single workstation-based ORION prototype (the first ORION prototype) from scratch, included N. Ballou, J. Banerjee, H.-T. Chou, J. Garza, and D. Woelk. Furthermore, F. Rabitti and E. Bertino (both of CNR, Italy), as visiting scientists, made significant research contributions.

References

[1] S. Abiteboul and N. Bidoit, "Non first normal form relations to represent hierarchically organized data," in *Proc. ACM SIGACT-SIGMOD Symp. Principles Database Syst.*, 1984, pp. 191-200.

[2] ACM, *Proc. Databases for Eng. Appl., Database Week*, May 1983.

[3] M. Ahlsen *et al.*, "An architecture for object management in OIS," *ACM Trans. Office Inform. Syst.*, vol. 2, no. 3, July 1984.

[4] T. Andrews and C. Andrews, "Combining language and database advances in an object-oriented development environment," in *Proc. 2nd Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Orlando, FL, Oct. 1987, pp. 430-440.

[5] N. Ballou *et al.*, "Coupling an expert system shell with an object-oriented database system," *J. Object-Oriented Programming*, vol. 1, no. 2, pp. 12-21, June/July 1988.

[6] F. Bancilhon, W. Kim, and H. Korth, "A model of CAD transactions," in *Proc. Int. Conf. Very Large Data Bases*, Stockholm, Sweden, Aug. 1985.

[7] F. Bancilhon, "Object-oriented database systems," in *Proc. ACM SIGACT-SIGMOD Symp. Principles Database Syst.*, Austin, TX, Mar. 1988.

[8] J. Banerjee *et al.*, "Data model issues for object-oriented applications," *ACM Trans. Office Inform. Syst.*, Jan. 1987.

[9] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth, "Semantics and implementation of schema evolution in object-oriented databases," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1987.

[10] J. Banerjee, W. Kim, and K. C. Kim, "Queries in object-oriented databases," in *Proc. 4th Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 1988.

[11] D. Batory and W. Kim, "Modeling concepts for VLSI CAD objects," *ACM Trans. Database Syst.*, vol. 10, no. 3, Sept. 1985.

[12] D. Batory *et al.*, "GENESIS: An extensible database management system," *IEEE Trans. Software Eng.*, to be published.

[13] D. Beech, "A foundation for evolution from relational to object databases," in *Proc. Conf. Extending Data Base Technol.*, Venice, Italy, Apr. 1988.

[14] J. Bennet, "The design and implementation of distributed Smalltalk," in *Proc. 2nd Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Orlando, FL, Oct. 1987, pp. 318-330.

[15] G. Birtwistle *et al.*, *Simula Begin*. Berlin, Germany: Studentliteratur and Auerbach, 1973.

[16] A. Bjornerstedt and C. Hulten, "Version control in an object-oriented architecture," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[17] M. Blasgen *et al.*, "System R: An architectural update," *IBM Syst. J.*, vol. 20, no. 1, pp. 41-62, Jan. 1981.

[18] D. G. Bobrow and M. Stefik, *The LOOPS Manual*, Xerox PARC, Palo Alto, CA, 1983.

[19] D. G. Bobrow *et al.*, "CommonLoops: Merging Common Lisp and object-oriented programming," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Portland, OR, Oct. 1986.

[20] R. Bretl *et al.*, "The GemStone data management system," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[21] M. Carey, D. DeWitt, J. E. Richardson, and E. J. Shekita, "Object and file management in the EXODUS extensible database system," in *Proc. 12th Int. Conf. Very Large Data Bases*, Aug. 1986, pp. 91-100.

[22] P. Chen, "The entity-relationship model: Toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9-36, Jan. 1976.

[23] H. T. Chou and W. Kim, "Versions and change notification in an object-oriented database system," in *Proc. Design Automat. Conf.*, June 1988.

[24] G. Copeland and D. Maier, "Making Smalltalk a database system," in *Proc. ACM SIGMOD Int. Conf. Management Data*, June 1984, pp. 316-325.

[25] B. Cox, "Message/object programming: An evolutionary change in programming technology," *IEEE Software*, Jan. 1984.

[26] G. A. Curry *et al.*, "Traits: An approach to multiple-inheritance subclassing," in *Proc. SIGOA Conf. Office Automat. Syst.*, Apr. 1982.

[27] G. A. Curry and R. M. Ayers, "Experience with Traits in the Xerox Star workstation," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 519-527, Sept. 1984.

[28] P. Dadam *et al.*, "A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies," in *Proc. ACM SIGMOD Int. Conf. Management Data*, Washington, DC, 1986, pp. 356-366.

[29] D. Decouchant, "Design of a distributed object manager for the Smalltalk-80 system," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Portland, OR, Oct. 1986, pp. 444-452.

[30] A. Deshpande and D. Van Gucht, "An implementation for nested relational databases," in *Proc. Int. Conf. Very Large Data Bases*, 1988.

[31] D. Fishman *et al.*, "IRIS: An object-oriented database management system," *ACM Trans. Office Inform. Syst.*, vol. 5, no. 1, pp. 48-69, Jan. 1987.

[32] D. Fishman *et al.*, "Overview of the IRIS DBMS," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[33] J. F. Garza and W. Kim, "Transaction management in an object-oriented database system," in *Proc. ACM SIGMOD Int. Conf. Management Data*, June 1988.

[34] A. Goldberg, "Introducing the Smalltalk-80 system," *Byte*, vol. 6, no. 8, pp. 14-26, Aug. 1981.

[35] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.

[36] M. Hammer and D. McLeod, "Database description with SDM: A semantic data model," *ACM Trans. Database Syst.*, vol. 6, no. 3, Sept. 1981.

[37] R. Haskin and R. Lorie, "On extending the functions of a relational database system," in *Proc. ACM SIGMOD Int. Conf. Management Data*, June 1982, pp. 207-212.

[38] SQL/Data System: Concepts and Facilities, GH24-5013-0, File S370-50, IBM Corp., Jan. 1981.

[39] IEEE Computer Society, *Database Engineering*, special issue on Engineering Design Databases, R. Katz, Ed., June 1982.

[40] IEEE Computer Society, *Database Engineering*, special issue on Engineering Design Databases, R. Katz, Ed., June 1984.

[41] IEEE Computer Society, *Database Engineering*, special issue on Object-Oriented Databases, F. Lochovsky, Ed., Dec. 1985.

[42] IEEE Computer Society, *Database Engineering*, special issue on Non-First Normal Form Relational Databases, Z. M. Ozsoyoglu, Ed., Sept. 1988.

[43] G. Jaeschke and H. Schek, "Remarks on the algebra on non first normal form relations," in *Proc. ACM SIGACT-SIGMOD Symp. Principles Database Syst.*, 1982, pp. 124-138.

[44] T. Kaehler, "Virtual memory for an object-oriented language," *Byte*, pp. 378-387, Aug. 1981.

[45] R. Katz and S. Weiss, "Design transaction management," in *Proc. 21st Design Automat. Conf.*, Albuquerque, NM, June 1984.

[46] R. Katz, E. Chang, and R. Bhateja, "Version modeling concepts for computer-aided design databases," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1986, pp. 379-386.

[47] S. Keene and D. Moon, "Flavors: Object-oriented programming on symbolics computers," in *Proc. Common LISP Conf.*, Boston, MA, 1985.

[48] W. Kim, D. McNabb, R. Lorie, and W. Plouffe, "A transaction mechanism for engineering design databases," in *Proc. Int. Conf. Very Large Databases*, Singapore, 1984.

[49] W. Kim *et al.*, "Composite object support in an object-oriented database system," in *Proc. 2nd Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Orlando, FL, Oct. 1987.

[50] W. Kim and H. T. Chou, "Versions of schema for object-oriented databases," in *Proc. Int. Conf. Very Large Data Bases*, Aug.-Sept. 1988.

[51] W. Kim *et al*, "Integrating an object-oriented programming system with a database system," in *Proc. 2nd Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, San Diego, CA, Sept. 1988.

[52] W. Kim *et al.*, "Features of the ORION object-oriented database system," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[53] W. Kim, K. C. Kim, and A. Dale, "Indexing techniques for object-oriented databases," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[54] W. Kim, "A model of queries for object-oriented databases," in *Proc. Int. Conf. Very Large Data Bases*, Amsterdam, The Netherlands, Aug. 1989.

[55] H. Korth, W. Kim, and F. Bancilhon. "On long-duration CAD transactions," *Inform Sci.*, Oct. 1988.

[56] Laguna Beach Report on the Future Directions for Database Re-

search, presented as a panel position paper at the Int. Conf. Very Large Data Bases, Long Beach, CA, Sept. 1988.

[57] K. Lang and B. Pearlmutter, "Oaklisp: An object-oriented scheme with first class types," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Portland, OR, Oct. 1986.

[58] B. Lindsay *et al.*, "Notes on distributed databases," in *Distributed Data Bases*, I. Draffen and F. Poole, Eds. Cambridge, MA: Cambridge University Press, 1980.

[59] B. Lindsay *et al.*, "Computation and communication in R*: A distributed database manager," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 24-38, Feb. 1984.

[60] B. Lindsay, J. McPherson, and H. Pirahesh, "A data management extension architecture," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1987, pp. 220-226.

[61] *ObjectLISP User Manual*, LMI, Cambridge, MA, 1985.

[62] G. Lohman *et al.*, "Query processing in R*," in *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Eds. New York: Springer-Verlag, 1985.

[63] R. Lorie and W. Plouffe, "Complex objects and their use in design transactions," in *Proc. Databases Eng. Appl., Database Week 1983*, ACM, May 1983, pp. 115-121.

[64] D. Maier and J. Stein, "Indexing in an object-oriented DBMS," in *Proc. Int. Workshop Object-Oriented Database Syst.*, Asilomar, CA, Sept. 23-26, 1986.

[65] D. Maier *et al.*, "Development of an object-oriented DBMS," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Portland, OR, Oct. 1986.

[66] D. Maier, "Making database systems fast enough for CAD applications," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[67] A. Makinouchi, "A consideration of normal form of not-necessarily normalized relations in the relational data model," in *Proc. Int. Conf. Very Large Data Bases*, 1977, pp. 447-453.

[68] A. Makinouchi and H. Ishikawa, "The model and architecture of the object-oriented database system JASMIN," working paper, Fujitsu, Ltd., Kawasaki, Japan, 1988.

[69] P. McCullough, "Transparent forwarding: First steps," in *Proc. 2nd Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Orlando, FL, Oct. 1987, pp. 331-341.

[70] F. Mellender, S. Riegel, and A. Straw, "Optimizing Smalltalk message performance," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[71] B. Meyer, "Eiffel: A language for software engineering," Tech. Rep., Dep. Comput. Sci., Univ. of California at Santa Barbara, Nov. 1985.

[72] J. Micallef, "Encapsulation, reusability, and extensibility in object-oriented programming languages," *J. Object-Oriented Programming*, vol. 1, no. 1, Apr./May 1988, pp. 12-36.

[73] M. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision*, P. Winston, Ed. New York: McGraw-Hill, 1975.

[74] M. Missikoff, "A domain-based internal schema for relational database machines," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1982, pp. 215-224.

[75] M. Missikoff and M. Scholl, "Relational queries in domain based DBMS," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1983, pp. 219-227.

[76] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the R* distributed database management system," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378-396, Dec. 1986.

[77] D. Moon, "Object-oriented programming with flavors," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, 1986.

[78] —, "The Common LISP object-oriented programming standard system," in *Object-Oriented Concepts, Applications, and Databases*, W. Kim and F. Lochovsky, Eds. New York: Addison-Wesley, 1989.

[79] H. Paul *et al.*, "Architecture and implementation of Darmstadt Database kernel system," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1987, San Francisco, CA, 1987.

[80] J. Penney and J. Stein, "Class modification in the GemStone object-oriented DBMS," in *Proc. 2nd Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Orlando, FL, Oct. 1987.

[81] F. Rabitti, D. Woelk, and W. Kim, "A model of authorization for object-oriented and semantic databases," in *Proc. Int. Conf. Extending Database Technol.*, Venice, Italy, Mar. 1988.

[82] L. Rowe and M. Stonebraker, "The POSTGRES data model," in *Proc. Int. Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, pp. 83-95.

[83] C. Schaffert *et al.*, "An introduction to Trellis/Owl," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Sept. 1986.

[84] K. Schmucker, *Macintosh Library: Object-Oriented Programming for the Macintosh*. Hasbrouck Heights, NJ: Hayden, 1986.

[85] M. Schrefl and E. Neuhold, "Object class definition by generalization using upward inheritance," in *Proc. Int. Conf. Data Eng.*, Feb. 1988, pp. 4-13.

[86] P. G. Selinger *et al.*, "Access path selection in a relational database management system," in *Proc. ACM SIGMOD Int. Conf. Management Data*, Boston, MA, 1979, pp. 23-34.

[87] D. Shipman, "The functional data model and the data language DAPLEX," *ACM Trans. Database Syst.*, vol. 6, no. 1, Mar. 1981.

[88] A. Skarra, S. Zdonik, and S. Reiss, "An object server for an object-oriented database," in *Proc. Int. Workshop Object-Oriented Database Syst.*, ACM/IEEE, 1986.

[89] J. Smith and D. Smith, "Database abstraction: Aggregation and generalization," *ACM Trans. Database Syst.*, vol. 2, no. 2, pp. 105-133, June 1977.

[90] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Portland, OR, Oct. 1986, pp. 38-45.

[91] M. Stefik and D. G. Bobrow, "Object-oriented programming: Themes and variations," *AI Magazine*, pp. 40-62, Jan. 1986.

[92] L. Stein, H. Liebermann, and D. Ungar, "A shared view of sharing: The treaty of Orlando," in *Object-Oriented Concepts, Databases, and Application*, W. Kim and F. Lochovsky, Eds. Reading, MA: Addison-Wesley, 1989.

[93] M. Stonebraker, "The design and implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 189-222, Sept. 1976.

[94] M. Stonebraker and L. Rowe, "The design of POSTGRES," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1986.

[95] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.

[96] —, "What is object-oriented programming?," *IEEE Software*, pp. 10-20, May 1988.

[97] *FLAV Objects, Message Passing, and Flavors*, Symbolics, Inc., Cambridge, MA, 1984.

[98] S. Thatte, private communication, May 1988.

[99] S. Thomas and P. Fischer, "Nested relational structures," in *Advances in Computing Research III, The Theory of Databases*, P. Kanellakis, Ed. JAI Press, 1986, pp. 269-307.

[100] M. Tiemann, "User's guide to GNU C++," MCC Tech. Rep., ACA-ESP-099-88, Mar. 1988.

[101] F. Velez, G. Bernard, and V. Darnis, "The O2 object manager: An overview," in *Proc. 15th Int. Conf. Very Large Data Bases*, Amsterdam, The Netherlands, Aug. 1989.

[102] D. Weinreb *et al.*, "An object-oriented database system to support an integrated programming environment," *IEEE Database Eng. Bull.*, R. King, Ed., vol. 11, no. 2, pp. 33-43, June 1988.

[103] D. Wiebe, "A distributed repository for immutable persistent objects," in *Proc. 1st Int. Conf. Object-Oriented Programming Syst., Languages, Appl.*, Portland, OR, Oct. 1986, pp. 453-465.

[104] D. Woelk and W. Kim, "Multimedia information management in an object-oriented database system," in *Proc. Int. Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, pp. 319-329.

[105] R. V. Zara and D. R. Henke, "Building a layered database for design automation," in *Proc. 22nd Design Automat Conf.*, 1985, pp. 645-651.

Won Kim, for a photograph and biography, see this issue, p. 269.