CMU-ITC-91-103

# An Introduction to Object-Oriented Databases and Database Systems

Michael L. Horowitz
(mh11+@andrew.cmu.edu)

*August 19, 1991*


*Information Technology Center*
*Carnegie Mellon University*
*Pittsburgh, PA 15213*

Acknowledgments to International Business Machines, Inc.

## Abstract

Recent developments in editing applications, especially in the areas of CAD/CAM and multimedia, have provoked interest in integrating the data abstraction capabilities of object-oriented languages with the persistence and concurrency control of database systems. Database systems assume the task of determining the file storage format for the application. In addition, such systems provide support for concurrency control, atomicity of multiple updates, recoverability, authorization, versioning, and search (i.e. associative access).

Sophisticated editing applications, however, require better data modeling capabilities than those normally provided by existing database systems (i.e. those presenting a relational or network data model). Thus, an *impedance mismatch* exists between the way databases view application data and how the application wishes to manipulate that data. A database system that supports an object-oriented data model would eliminate this impedance mismatch and furnish the desired modeling capabilities: object identity, direct access, data abstraction extensibility, inheritance, polymorphism, genericity, encapsulation, embedded semantics, and data type extensibility.

Integrating object-oriented concepts and normal database concepts also presents the opportunity to explore new features that would help application builders: object composition, property propagation, cyclic queries, indexing extensibility, relationship support, database self-containment, and schema evolution.

This paper presents a summary of current database research into new data models based on object-oriented concepts. The concepts themselves are defined and then the different systems are described.

## Acknowledgments

Thanks to many people at the ITC for their helpful comments: in particular, Michael McInerny, David Anderson, John Howard, and Andrew Palay.

## Table of Contents

# 1 Introduction

Databases fulfill several roles in the process of building computer applications. Like a file system, databases provide the means to store data between invocations of an application (i.e. *persistence*). Database systems, however, provide additional services not supported by most, if not all, file systems. For instance, a database system typically provides facilities to coordinate cooperative work on the same data (i.e. *transactions*, *authorization*, and *distribution*) and assurances concerning the integrity of the data in the presence of various kinds of failures (i.e. *versioning* and *stability*). In addition, databases allow applications to manage large amounts of data, providing *buffering* services and *searching* capabilities (i.e. *associative access*). Finally, databases present a uniform *data model* independent of any specific application, presumably easing the burden of application design.

Several data models have been proposed and explored, including hierarchical, network, and relational. Currently, many commercial systems support the relational data model. A relational database consists of a set of named *relations*, each of which is a set of *tuples*. Each tuple, in turn, is an *aggregation* of tagged values (i.e. a collection of *attribute-value* pairs; the attributes are common to all tuples in a relation and are defined by the relation's *schema*). Each tuple represents an entity or part of an entity in an application's data space. A reference to another entity in the space is specified by some subset of the target entity's attribute-value pairs that uniquely identifies the target within a specified database relation (i.e. *value-based* reference).

This paper presents a summary of current research into new data models based on object-oriented concepts. The remainder of this section explores the motivations for such research and the reasons we feel that database systems supporting an object-oriented paradigm are appropriate for our research in the Alexandria project. The following section introduces a generic object-oriented data model and discusses how such models affect database issues. Section 3 enumerates specific research efforts into object-oriented databases and describes which design decisions were taken by each on the various issues. A glossary and an index are included as appendices.

It is assumed the reader understands something about databases in general and the relational data model in particular. Interested readers are directed to *Principles of Database Systems* by Jeffrey Ullman [Ullman 82].

## 1.1 Motivation

Relational database systems have proved their worth in the domain of business applications, particularly those dealing with accounting. The relational data model, however, is not suitable for all application domains. New applications involving complex data modeling (i.e. that do not map well to tables) now require the services normally associated with database systems: persistence, transactions, authorization, distribution, versioning, data stability, buffering, and associative access.

To illustrate, let's examine a CAD/CAM application for a company that manufactures airplanes. The application supports both the specification and design of all parts required to build an airplane. Modeling physical objects does not reduce easily to tabular, or relational, form. In particular, an airplane requires many duplicate parts, each of which would require a unique tag to be stored as a distinct entity in a relational database. Furthermore, the relations representing sets of different parts that are mostly similar would require separate, independent schemas. Finally, the application programmer almost definitely would prefer to manipulate part designs as complex abstractions at a level higher than that provided in the relational model.

Our example application, however, requires database services. An airplane design team typically consists of several people, all of whom will desire access to the current state of the design. In today's workplace, it is likely that these designers will be using workstations distributed over a network. In addition, some people should not be allowed total access to certain aspects of the design (e.g. documenters do not need update access). Finally, a completed design can involve hundreds of thousands of parts and direct access to each part becomes impractical; thus, associative access is essential. For instance, a designer may wish to know how many times a given part has been used before deciding to change its specification.

Object-oriented databases, then, are an attempt to solve the problems mentioned (as well as others) and still maintain the advantages of database systems. Object-oriented databases treat each entity as a distinct object. An assembly composed of several parts, therefore, can refer directly to its components instead of explicitly associating some unique identifier with each component in some relation. In addition, application programmers can manipulate database entities at any desired level of abstraction by extending the set of types recognized by the database system. This is an important point - it means that the programmer need not be concerned with transforming an application's persistent data into a form manipulable by the underlying storage subsystem [Cockshott 84]. In many systems, a programmer can also incorporate totally new, variable-sized data types (e.g. multimedia objects). Finally, object-oriented databases allow embedded semantics by associating procedural information with objects [Smith 87].

Woelk, Kim, and Luther [Woelk 86] summarize the features they feel object-oriented databases should provide for multimedia document management applications:

- *aggregation* support, including modeling **is-part-of**[1] relationships and maintaining knowledge concerning the ordering of subparts;

[1] Bold phrases indicate a kind of relationship.

- *generalization* support for the is-a relationship between types or classes;
- support for *default* values for attributes;
- support for *embedded semantics* by which object properties may be computed instead of stored;
- support for *polymorphism* so that an attribute may represent any of several types that are only weakly related (e.g. the body of a document may be text, a drawing, an image, a composition of these, etc.);
- general *entity-relationship* support (i.e. n-ary relationships with knowledge of which roles are key);
- support for *schema evolution*, in which the types of existing database entities are modified;
- control over object *versioning* and *configurations* of version sets;
- support for *concurrent access*;
- support for *multimedia* data types;
- support for *sharing* subcomponents among separate database objects (e.g. the same picture shared by two documents);
- *associative access*, as opposed to direct access via reachability from a root object as in pure hypertext systems; and
- support for standard database-like *recovery* in the presence of failures.

These features and the others mentioned earlier will be discussed in more detail in later sections.

The next section presents another class of applications that could take advantage of the features provided by object-oriented database systems.

## 1.2 Alexandria

The domain of information management in an era of increasingly easy access to on-line data clearly requires the features provided by database systems: persistence, distribution, access control, associative access, etc. Hypertext systems, such as Intermedia [Smith 87], comprise an initial exploration into the issues concerning information structuring. Pure hypertext technology, however, cannot deal with the quantities of on-line information that will become available, even if a database is used as the underlying storage subsystem (as in Intermedia). More work is needed on the joint problems of access and management to have a meaningful impact on the way information is used.

The primary focus of the Alexandria research project at the Information Technology Center (ITC) of Carnegie Mellon University (CMU) is to investigate what tools computer users need to manage large amounts of on-line information over long periods of time. The goals of the project include [Palay 90]:

1) *Performance* - The system must provide simple, fast access to large amounts of information from multiple, diverse sources.

2) *Flexible access* - The system must support a spectrum of access techniques from browsing (as in a hypertext system) to search (as in a database system).

3) *Structuring* - The system must help the user productively manage information. In particular, it is not sufficient just to provide access to information; the user should be able to impose personalized structure that helps organize the information for later access or, more importantly, for furthering the user's work. In addition, since such structure can become unwieldy, the user should be able to browse and search structure itself.

4) *Data type extensibility* - The system must accommodate a variety of digital media. Although it is not expected that the system will be initially able to recognize features from raster, graphic, audio, or video data, the design of the system should not prevent the management and access of such information.

5) *Structure evolution* - The system should also help the user maintain and evolve the structure imposed on an information space. A user's view does not remain static; often as more information becomes available, the user will want to change the form of his structure, not just the content.

6) *Collaboration* - The system should enable cooperative work within a community of users to encourage the exchange of ideas. Specifically, the system should make it easy for one user to view an information space using the structure built by another. The system, however, must also ensure the privacy of unpublished data.

7) *Maintaining currency* - The system should handle changing information. Users often must keep up with sources that augment or replace previous information (e.g. news wires, electronic bulletin boards).

8) *Integration* - Finally, the system should be integrated fully into the user's computing environment. That is, all applications should be able to take advantage of the system's capabilities and the user should be able to integrate data from any application into their personal information structure.

Almost all of these requirements have some implications regarding the features we desire in the underlying database support. As we shall see in the next section, object-oriented databases typically provide most of these desired features.

The performance goal indicates the need for database support because of the amount of data involved. The desire both for flexible access and individualized structuring require the ability to refer to information entities directly. Pure, hypertext-like browsing is just jumping from one place to another in the information space. Also, shared information should remain in context so that a user can take advantage of any additional structure on that information.

Flexible access and maintaining currency suggest the need to be able to embed computational semantics in the information space. Following a bibliographic reference, for instance, should look like a direct link to the user but may require a search at the database level. Similarly, determining what has changed that is of interest (as defined by the user) in a changing information source necessitates search and test capabilities.

The desire to allow individualized structuring mandates the ability to extend and define complex abstractions within the system. Also, in order to browse and search structure, it must be possible to examine abstraction definitions. Data type extensibility indicates that extension should also apply to the types of values handled by the storage subsystem. Structure evolution goes even further and stipulates that the underlying system must allow types to change in the presence of existing information.

Support for collaboration implies several needs. First, locking and transaction support would help provide the coordination required by cooperative work. Authorization and logging support can help protect privacy, assign accountability, and keep people with diverse roles from interfering with each other. The ability to share subentities among distinct database objects would ease communication. Finally, publishing structure requires that such descriptions be self-contained.

Finally, the need for integration affects the overall architecture of the system. It is not clear whether the database component will be involved in satisfying this goal.

The next section introduces the concepts and features explored by current research efforts into object-oriented databases. Although the goals of the Alexandria project will not be addressed specifically below, the reader should try to correlate the features that distinguish object-oriented databases from relational databases with the goals presented above.

# 2 Object-Oriented Databases

As mentioned above, the development of object-oriented databases represents an attempt to integrate the complex data modeling and software engineering principles of recent programming language designs with the persistence, coordination, and protection characteristics supported by database technology. Of course, the goal is to achieve all of the benefits of both.

So far, we have discussed the facilities provided by databases in general. The sections below describe the additional features provided by object-oriented databases. It is assumed that the reader is familiar with the concepts that characterize object-oriented programming languages. Good presentations of these concepts can be found in both *Smalltalk-80: The Language and its Implementation* by Adele Goldberg and David Robson [Goldberg 83] and *Object-Oriented Software Construction* by Bertrand Meyer [Meyer 88]. Appendix I provides a short introduction for those readers unfamiliar with the terminology.

For applications requiring database support, objects constitute a natural unit for locking, authorization, storage clustering, versioning, and buffering. The object-oriented model also presents other opportunities for improved application-building support in database systems. The following sections describe various database features and how object-oriented concepts interact with those features:

- section 2.1 presents standard database issues;
- section 2.2 defines object-oriented data models and related issues;
- section 2.3 discusses the interaction between database features and programming language constructs to support those features;
- section 2.4 concentrates on the issues specific to querying databases;
- section 2.5 presents the issues relating to database evolution; and
- section 2.6 discusses the lower-level issues concerning the storage management and distribution of objects.

Incidentally, a good introduction to object-oriented databases can be found in the chapter titled "Fundamentals of Object-Oriented Databases" in *Readings in Object-Oriented Database Systems* edited by Stanley Zdonik and David Maier [Zdonik 90]. This presentation involves a few more concepts and definitions, but less motivation. In general, any comparisons of object-oriented systems with previous database technology will be with relational systems because of their pervasiveness.

## 2.1 General Issues

Object-oriented databases must provide the same support for managing concurrency, authorization, distribution, data persistence and stability, versioning, and associative access as any other kind of database system. Most of these features are affected significantly by the data model, but some have only a small connection. This section presents the concepts associated with locking, transactions, triggers, distribution, and versioning. Object-oriented concepts, in general, have little impact on these characteristics.

### 2.1.1 Concurrency Control

When many clients wish simultaneous access to the same persistent data and some clients wish to perform updates, a mechanism must exist to ensure *data consistency*. Otherwise, readers could receive an inconsistent view of the data as some parts might be changed before other parts. Similarly, two writers may base their updates on the same version and then the writer who commits his changes second will cause the changes of the first to become lost.

Any mechanism provided to help application developers ensure consistency is called a *concurrency control* mechanism. Two kinds of concurrency controls exist: optimistic and pessimistic. *Optimistic concurrency* operates on the belief that most transactions either involve reading only or else get aborted. It works by allowing any client to request a copy of the data. When the client wishes to update the database version, a check is performed to see whether the current database version is the same as the one copied. If so, the update proceeds. If not, the update fails. The object-oriented model appears to have little effect on this mechanism; some systems do employ it (e.g. GemStone [Maier 86] and ts1 [Caplinger 87]).

*Pessimistic concurrency*, on the other hand, assumes that conflicts at the time of update either occur too often or waste too much work (in computing the updated data). The basic pessimistic mechanism is *locking*. When a client wishes to update a database entity (or set of entities), the client requests a lock on the data and then no other client may receive a copy of the same data. Locks are issued according the nature of the client's access to the data and the current lock on the data, if any. Thus, it is important to consider the *granularity* of locking; that is, how much data may be locked at a time. Fine-grain locking (i.e. at the object level) allows the most concurrent access but requires the most resources for lock management.

Objects clearly comprise a good unit for locking. Ways of reducing the inefficiencies associated with such fine lock granularity are discussed in section 2.2.8 below. Most object-oriented systems utilize locking (e.g. Coral3 [Merrow 87] and ORION [Kim 89b]).

### 2.1.2 Transactions

In the presence of persistence, locking may not be sufficient to guarantee data consistency. Complex interactions may involve changes to several data objects and consistency may dictate that either all or no changes occur (i.e. *atomicity*). The facility databases provide to achieve atomicity is called *transaction support* and each atomic set

of changes a database interaction wishes to make is termed a *transaction*. The combination of transactions and locking implies that updates to the database are *serializable* (i.e. as if no concurrency were involved).

A transaction, therefore, must accumulate all changes before actually *committing* them to the database.[2] The transaction support is responsible for invoking the appropriate concurrency control and ensuring that the commit is atomic. Typically, database systems use *shadowing* (as in POMS [Cockshott 84]) or *write-ahead logs* to generate the new data versions and *two-phase commit* to ensure that each update completes atomically (as in EXODUS [Carey 86]).

By using transactions and *logging* [Carey 86], databases systems can also provide mechanisms for *recovery* in the presence of failures. That is, if a failure occurs during a transaction commit, the database system can detect whether or not the commit completed, and, if not, can finish the job based on the information in the log. This facility provides data *stability*. Failures can be classified into three categories [Zdonik 90]: *process failures* occur when the application terminates abnormally in the middle of a transaction; *media failures* occur when the storage medium fails (e.g. head crash or bus failure); and *system failures* occur when the database system fails, either because of a bug in its program, the machine it runs on crashes, or the network connection to it dies.

Support for transactions may constrain each execution of an application to be an entire transaction (as in the E [Richardson 89b] and $CO_2$ [Bancilhon 88] programming languages) or may provide language constructs to allow multiple transactions and multiple *save-points*[3] within each transaction (as in embedded OSQL in Iris [Fishman 87]). Clearly, the latter capability allows greater flexibility for application development.

Not all application domains require transaction support. The assumption in Coral3 [Merrow 87], for instance, is that the objects manipulated are large enough that no complex interactions are necessary. That is, all "transactions" occur to single objects in their applications. In this case, concurrency control need consist of locks only.

### 2.1.3 Triggers and Notifiers

Monitoring the contents of a database can be prohibitive because of the amount of information involved. Many database systems, therefore, provide mechanisms that perform user-level actions automatically when specified events occur (e.g. an entity changes value). By supporting these mechanisms internally, database systems can achieve efficient implementations.

Automatic actions that perform some update on the database are usually called *triggers*. Actions that just notify the user when the specified event occurs are sometimes called *notifiers*, or *alerters*. The events that cause triggers to execute can either be specific database operations (e.g. set a value or delete an object) or the transition of an arbitrary

---

[2] Note this also allows one to *abort* a transaction, ensuring that no changes occur.

[3] *Save-points* act like nested transactions, in that all changes back to some specified point may be aborted without aborting the entire transaction.

predicate from false to true (as in VBase [Andrews 87] and POSTGRES [Stonebraker 86a]).

The actions performed by triggers are often used to maintain application invariants. For example, triggers can ensure that component objects are deleted when the containing object is deleted (see section 2.2.8 on Composition) or that inverse relationships are maintained (e.g. the division of an employee should always equal the division that hires that employee; see section 2.2.9 on Relationship Support). The system itself can also use triggers to maintain indexes [Zdonik 88].

The problem with triggers is that updates can propagate throughout the database if the application developer is not careful. As a result, a small transaction can end up monopolizing a large portion of the database. In addition, the potential for deadlock[4] between transactions increases. Notifiers, on the other hand, do not exhibit this problem because they are read-only. They can prove very useful in domains where new information is added to a database continually by alerting users only if the new information is of interest.

### 2.1.4 Distribution

*Distribution* presents several opportunities for database architectures. In particular, database services may be spread among several servers. One advantage of separating applications from database support is that true concurrency can be realized. Thus, even if an application involves significant amounts of computation (as in CAD), it need not affect overall database performance adversely.

Distribution occurs in two ways. First, database features can be divided among several processes. Doing so enables concurrency and permits database access over both local-area and wide-area networks. Second, servers providing database functionality can be replicated, which affords protection and increases concurrency as well.

The systems surveyed for this paper divide database services into at most three groupings of functionality. More levels of servers (with smaller sets of functionality) could be used, but at increasingly higher communication costs without any clear gain in concurrency or flexibility. Typically, services are divided into those dealing with:

(1) storage issues (including:
- secondary storage management,
- variable-sized data support,
- buffering,
- locking,
- transactions or other concurrency control,
- logging and recovery,
- versions);

[4] When two or more processes require resources held by the other(s) in order to progress. See the glossary.

(2) data model issues (including:
- object support,
- embedded semantics,
- shared object semantics,
- authorization,
- session control,
- indexing,
- querying and query optimization,
- configurations); and

(3) client support (including:
- embedded semantics execution,
- data packing/unpacking,
- data model presentation).

Systems that adhere to this model include ENCORE [Hornick 87] and Iris [Fishman 87]. Of course, some systems partition responsibilities slightly differently; GemStone, for example, bundles authorization and indexing in with the storage support at the lowest level [Maier 86].[5]

Each of the three parts may be instantiated any number of times. For instance, client support is always instantiated once per client. Data model support can be replicated as many times as necessary to achieve the greatest concurrency. Multiple data model servers can also reduce the contention that can be caused by complex queries. ENCORE and $O_2$, for instance, maintain one data model component for each client [Hornick 87, Deux 90]. Iris, on the other hand, uses just one for all clients [Fishman 87]. Apparently, GemStone allows any number of data model servers [Maier 86]. No one has explored the possibility of using multiple data model servers to impose different data models on the same data.

Since storage servers manage secondary storage, they reflect the actual location of the "database". Allowing more than one storage server can represent either division or replication of the stored data. The first choice effectively indicates support for simultaneous access to different databases. Most systems surveyed do not support access to multiple databases. Iris does allow connecting to different databases during the same application execution but apparently does not support simultaneous access [Fishman 87]. Allowing for simultaneous access is essential for applications that wish to provide users with transparent access to the union of the information in more than one database. In fact, this capability is required by the Alexandria project, since we wish to merge personalized structure with the global structure held in the central information repository. Even though it is not a true database system, the telesophy system tsl is designed to support such applications; thus, it provides simultaneous access to multiple "information unit servers" [Schatz 89].

---

[5] In fact, the GemStone system has different names for each component of the architecture: `Stone` refers to the storage server, `Gem` to the object server, and `Agent` to the client support.

Multiple storage servers may also increase access performance by replicating the data at each server. Replication in the presence of updates, however, creates new problems for maintaining data consistency. As yet, no object-oriented database system has attempted this strategy.

Note that the storage server need not consist of a single process. In the Coral3 system, the distributed file system with locks acts as the central storage server; no one processor has sole responsibility for concurrency control [Merrow 87].

### 2.1.5 Versions and Configurations

In design applications, maintaining the history of changes is almost as important as maintaining the current state of the database. Database systems support history by saving previous *versions* of entities as well as each entity's current state. Users, then, can request the current, modifiable *transient version* or a past, immutable *working version* of an entity [Banerjee 87a].

Since different database entities do not change at the same rate, the concept of a *configuration* represents the collection of consistent versions of related entities. For example, a configuration might contain all part designs of an airplane on a certain date.

In object-oriented data models, objects may refer to other objects. Thus, to achieve consistency in the presence of versioning, it should be possible to tag each object reference as *generic* (i.e. referring to the most current transient version) or *specific* (i.e. referring to a specific working version) [Kim 88]. Currently, such tags (as in ORION [Kim 88]) constitute the sole support for configurations among the systems surveyed; that is, a configuration can be implemented as an object with references to specific versions of all pertinent entities. Explicit support for configurations would clearly be more expedient for an application programmer.

Systems may require that individual objects or all instances of a given class be declared as versionable by the application programmer (as in ORION [Kim 88] or Iris [Wilkinson 90]). On the other hand, systems may provide versioning capability at the lowest level, allowing any instance to be versioned at any time (as in EXODUS [Carey 86]).

## 2.2 Data Model Issues

The primary advantages object-oriented databases have over other kinds of databases involve the complexity and extensibility of their data models and the additional features those models can support. Object-oriented data models allow programmers to design application entities at abstraction levels appropriate to their problem domains. Relational, hierarchical, and network data models generally do not provide any extensibility of their type systems.

Object-oriented models also incorporate mechanisms for inheritance, polymorphism, and type parameterization that augment the flexibility available to application developers. Furthermore, viewing database entities as objects enables the integration of facilities that support object identity, composition, property propagation, data sharing, embedded semantics, and authorization. Each section below describes one of these mechanisms in detail and discusses its benefits for the application programmer.

Before starting, it would be instructive to contrast relational and object-oriented databases. A relational database consists of a set of named relations, each of which consists of a set of tuples and a schema that describes the form of each tuple. A tuple consists of a set of attribute-value pairs. The attributes and the domains of their respective values are specified by the relation's schema. Thus, only tuple values need be stored.

An object-oriented database typically consists of a set of named objects (as in EXTRA [Carey 88]).[6] Each object embodies an aggregation of data much like a tuple, except that the set of attribute-value pairs need not be fixed (i.e. an object can represent a set of other objects). The database consists of all objects reachable from the "root" objects through their attributes. Thus, objects act as both tuples and relations, and object classes act as relation schemas.

### 2.2.1 Object Identity

One advantage all object-oriented data models share with network data models is *object identity*; that is, the ability to refer to any persistent object directly [Laffra 90]. The direct connectivity provided by object identity, though, is different from that in a network model because of the inherent typing of the destination object, which can be used to validate the semantic correctness of the modeled data [Duhl 88].

Object identity eliminates the need to assign unique identifiers explicitly to separate instances of the same airplane part. Similarly, entities that coincidentally have the same attribute values (e.g. two employees named John Smith) retain their individuality.

Object identity can potentially save time and storage as well. In relational databases, a tuple in one relation must refer to another tuple in (possibly) another relation by specifying its key in that relation (i.e. *value-based identity*). Retrieving the tuple requires knowing which relation contains it and a search of that relation. Furthermore, storing the

---

[6] Sometimes, as in the Persistent Object Management System (POMS), the system restricts the set of root objects to contain only one element [Cockshott 84].

key may involve multiple attribute values. In an object-oriented system, the application programmer need not be concerned with the details of referring to another entity (e.g. deciding which attributes constitute the key and building the query necessary to retrieve the tuple).

Note that the time savings, however, may not be realized in some systems. Although an object reference is direct conceptually, it may be implemented using a relational-like storage subsystem, which requires at least a hash table search to retrieve the referenced object. The Iris database system is built in this fashion [Fishman 87].

There are many concepts related to object identity. For example, one object is *reachable* from another if there exists a path from the second to the first in the network defined by object references.

*Strong identity* implies that an object continues to exist as long as there is any reference to it from any other object in the database. Thus, an object cannot be destroyed explicitly in a system that supports strong identity (e.g. GemStone [Maier 86]). Strong identity can lead to *logical pinning* in some applications, where inaccessible objects continue to reside in the database [Stein 89]. This can happen, for example, when a cyclic structure is no longer reachable from a database "root" object.

Systems that allow explicit destruction are said to provide *weak identity*. In such systems, the problems associated with *dangling references* arise, where a referenced object may no longer exist in the database [Stein 89]. The ORION [Kim 88] and EXODUS [Richardson 87] systems provide weak identity. Note that dangling references are also a problem in relational systems.

Object identity, however, is not always good. Relational systems derive a lot of flexibility by using value-based identity and joins. In particular, representing arbitrary many-to-many relationships becomes much easier. Object-oriented models must provide additional mechanism in order to support general relationships [Rumbaugh 87]. Object-oriented models, however, do "provide a framework for unifying value-based and identity-based access" [Zdonik 90].

### 2.2.2 Data Models

In general, most object-oriented systems (e.g. VBase [Andrews 87], ORION [Banerjee 87a], EXTRA [Carey 88], ENCORE [Hornick 87], $O_2$ [Lecluse 88], GemStone [Maier 86], GEM [Zaniolo 83]) present a data model consisting of the following kinds of objects:

- *atomic values* (e.g. integer, string, boolean, and floating point values);
- *tuples*, or *aggregations* of named attribute-value pairs; and
- *sets* of values,

where the values in tuples and sets may be any object in the data model. Thus, an attribute in a tuple may refer to another tuple or to a set [Osborn 88]. Note that this model, since it allows mutual references between objects, is more powerful than the pure, nested tuple-set model [Lecluse 88].

In object-oriented models, a class describes the form of each object. If we consider the set of all instances of a class in a given database, the basic model described above satisfies the well-definedness properties proposed for aggregations by Smith and Smith [Smith 77a]. To be precise, because of object identity, each object in a "relation" (i.e. set of instances) has a unique key and, in the presence of strong identity, every object referenced by another exists in the database.

Some systems extend the basic model with additional capabilities:

- OPAL provides relation and tree constructors as well as sets [Maier 86];
- EXTRA provides variable-sized arrays (i.e. sequences) as well as sets [Carey 88];
- VBase allows optional attributes [Andrews 87];
- GEM allows attributes to have no value [Zaniolo 83] (others do as well).

Although these capabilities are not strictly necessary, they can simplify the application programmer's task if available.

Some of the systems surveyed do not provide the basic, object-oriented model. The tsl system, for instance, provides only a flat value space [Caplinger 87]. Its model is not as powerful; it cannot represent sets of objects directly, for instance.

In Iris, objects consist solely of the operations that query their behavior. Instead of tuples and sets, Iris supports functions that represent relationships between objects [Fishman 87].[7] Thus, a unary function that maps objects to values acts as a tuple attribute. In addition, such functions may be multiple-valued, so sets are not needed directly either.

### 2.2.3 Inheritance

Smith and Smith describe an orthogonal extension to their aggregation model that supports the concept of generalization. Just as a class captures the common properties of a set of objects, *generalization* captures the common properties of a set of classes [Smith 77b]. Object-oriented systems support generalization through inheritance.

Wegner distinguishes four kinds of inheritance [Wegner 89]: (1) *behavior compatibility*, in which inherited attributes always have the same semantics (i.e. as an algebra with interpretations); (2) *signature compatibility*, in which attributes may be extended horizontally by adding new attributes or vertically by constraining existing attributes (i.e. as a syntactic algebra); (3) *name compatibility*, in which only implementation is shared; and (4) *cancellation*, in which only some implementation is shared (i.e. some attributes may be eliminated by the inheritor). The four kinds of inheritance are progressively more permissive. Each, however, carries progressively less semantics when used. Most systems strive to provide a form of behavior compatibility.

Behavior compatibility includes subset subtyping (e.g. positive integers are contained in all integers), isomorphic embedding (e.g. all integers may be floating point values),[8] and

---

[7] Generally, these functions are stored as tables in the underlying relational storage subsystem.

[8] Such embeddings are called *isomorphic* because there exists a one-to-one correspondence between the elements of one domain and a subset of the other.

**is-a** hierarchies (e.g. a student is a person). In signature compatibility, horizontal extensions are behaviorally compatible, but vertical extensions may not be (because the added restrictions involve disallowed values during updates). Vertical extensions can satisfy read-only compatibility where values are only examined and not changed. Name compatibility essentially involves just the concept of overriding.

Systems may or may not allow multiple inheritance. Those systems that provide only single inheritance include:[9]

- OPAL in GemStone [Maier 86]
- Type Definition Language (TDL) in VBase [Andrews 87]

The following systems provide multiple inheritance:

- EXTRA in EXODUS [Carey 88]
- ORION [Banerjee 87a]
- ENCORE [Zdonik 86]
- Iris [Fishman 87]

Apparently, the current implementation of the $O_2$ data model of the Altair project now supports multiple inheritance [Deux 90] although an earlier report indicated that users could specify only single inheritance [Bancilhon 88][10] Instead of inheritance, GEM supports union types; they argue that such incremental changes to the relational data model is more graceful and compatible with existing approaches [Zaniolo 83].

Although multiple inheritance makes complex modeling easier, it introduces some complications. In particular, *conflict resolution* is required when two attributes with identical names are inherited from two different superclasses. One method is to use the specification order by which the superclasses were inherited (e.g. inherit the attribute from the earlier superclass) [Banerjee 87b]. Another is to force the application programmer to specify explicitly the superclass from which to inherit the attribute (as in $O_2$ [Deux 90]). Both of these methods essentially cancel the effect of the hidden attribute. A better method, perhaps, is to force the programmer to rename conflicts so that all attributes remain available. During renaming, the programmer can also specify that conflicting attributes should be treated as identical instead of distinct.

### 2.2.4 Polymorphism

Much of the flexibility in the object-oriented data model derives from polymorphism. *Polymorphism* is the ability to manipulate many types at once in an application. Object-oriented databases provide polymorphism in two ways. First, an attribute may take on any value that is type compatible with its declared domain. In object-oriented systems, a value is *type compatible* with a domain if its class is a descendant of the domain class in the **is-a** inheritance lattice [Meyer 88]. In other systems, less natural mechanisms must be used to achieve polymorphism (such as union types in GEM [Zaniolo 83]).

---

[9] These systems may have removed this restriction since the date of their last technical report.

[10] The formal data model specified by the Altair project always supported multiple inheritance [Lecluse 88].

The second way databases provide polymorphism is during message invocations. Recall that inheriting classes may override methods. Thus, the class of the message receiver may be any that contains the message in its behavior. Minimally, this includes all descendants of the receiver's declared class (if static type checking is performed).

Determining the actual method to be executed is called *method resolution. Dynamic method resolution* determines the method at run-time by using the class of the actual receiver.[11] Almost all systems use this technique. *Static method resolution*, or *overloading*, determines the method before execution according to the declared class of the receiver. Iris uses overload resolution unless explicitly requested otherwise [Fishman 87].

### 2.2.5 Genericity

Additional flexibility is achieved when systems allow classes to have *type parameters*. In particular, collection classes (e.g. set, array, stack, tree, graph, list, and queue) need not be written for every component class used in an application. For example, one class can handle both "set-of-employee" and "set-of-vehicle". This capability is called *genericity* [Meyer 88].

Systems such as GemStone [Maier 86] provide genericity without explicit type parameters because they do not require type declarations. On the other hand, such systems cannot guarantee that all elements of a set instance are type compatible. In our example, one might end up with a set of employees and vehicles intermixed.

Systems that do use type parameters to provide genericity include the C Object Processor (COP) language for VBase [Andrews 87] and the E language of the EXODUS system [Richardson 89b]. Systems that allow and check type parameters help the programmer maintain the semantic correctness of the application's persistent data.

### 2.2.6 Extensibility

Object-oriented database systems provide extensibility in several ways. The most important involves the ability to extend the set of types recognized by the database. *Data abstraction extensibility* enables the application programmer to construct complex persistent objects [Laffra 90]. All object-oriented systems provide this kind of extensibility.

Generally, application objects do not map easily onto the basic types provided by a relational database system. Thus, *data type extensibility* would allow applications to derive completely new interpretations of stored data (e.g. multimedia objects [Woelk 86]). Some databases provide full type extensibility (e.g. EXODUS [Carey 86]), some limit extensibility to fixed length data (e.g. in an extension to INGRES [Stonebraker 88]), and some do not provide this kind of extensibility at all.

---

[11] A system may support dynamic method resolution even if it statically checks the type compatibility of message invocations. In such cases, static checking restricts the set of methods that might be executed to those defined in a subtree of the class hierarchy and enables more efficient determinations at run-time, such as the use of dispatch vectors (as in C++ [Stroustrup 86]).

Once new object types are allowed in a database, the database system should support efficient access and query optimization for instances of those types. In particular, *indexing extensibility* enables the integration of non-standard indexes, such as multidimensional access methods, into the system [Stonebraker 88, Zdonik 88]. Indexing extensibility may also be achieved by providing indexing composition operators that can understand structured data [Bertino 89]. Some systems are already experimenting with how an application programmer can specify the performance characteristics of indexes for query optimization (e.g. Iris [Derrett 89], EXCESS [Carey 88], ORION [Kim 89a]).

In most other data models, the only operators available to the application developer are those dealing with the model (e.g. get and set attribute for the relational model). Once the system can recognize other object types, it becomes possible and desirable to allow *procedural extensibility* or *embedded semantics* [Maier 86, Carey 88].

By storing methods in the database, several benefits are realized. First, queries may execute more efficiently since complex operators can be compiled (e.g. find all rectangles whose height is twice their width). Second, if viewing is just another action, queries do not need any special treatment: they can be represented procedurally [Zhu 89]. Third, one can abstract away access mechanisms; clients need not know whether a value is being retrieved by look-up or by computation (as in Iris [Fishman 87]). Finally, embedding semantics within a database helps make it *self-contained*, which expedites the design and implementation of generic applications (i.e. applications that operate over disparate sets of data; VBase [Andrews 87] and E [Richardson 89a] support this capability).

One problem with allowing embedded semantics is that it is difficult to protect the database from inept or malicious behavior. Depending upon its architecture, a method with a bug can cause an entire database system to crash. Similarly, embedded methods can compromise a database's integrity by storing incorrect data [Stonebraker 88]. An architecture like that used in ENCORE [Hornick 87], in which a separate process executes object semantics, can help protect against the first problem. Versioning and authorization can help with the second.

Note that providing extensibility for one feature (e.g. data type extensibility) has an effect on other aspects of the system (e.g. storage, locking, logging, etc. of variable-sized data). I feel that such interrelationships should lead to a complete re-design of the model presented to application developers, but some clearly disagree [Zaniolo 83, Stonebraker 88]. A complete re-design would allow application input at the proper places regarding query optimization, storage clustering, attribute composition, etc. On the other hand, drastic changes are hard to assimilate.

### 2.2.7 Integrity Constraints

*Integrity constraints* in a database system restrict the applicability of certain operations so that the validity of the data model or the semantic consistency of the stored information is maintained. In a relational database, for example, one cannot insert two distinct tuples with identical key values in the same relation. Similarly, object-oriented

systems that support the concept of key uniqueness in collections must ensure that two distinct objects in a collection do not have the same key values. Such support exists in the Iris database system [Fishman 87].

Several other integrity constraints arise in systems supporting object-oriented data models [Banerjee 87b]. First, object-oriented systems restrict class inheritance so that the superclass-subclass graph forms either a strict hierarchy (for single inheritance) or a lattice (for multiple inheritance). Thus, it is not possible to construct a class that inherits properties from itself.

Next, a class must constitute a *name space* for all attributes, including instance variables and methods. This means that one cannot assign the same name (i.e. identifier) to two different attributes. Although the name space must include the messages acceptable to the class and all superclasses, the instance variables of ancestor classes should not be included if strict encapsulation is enforced (i.e. only methods defined in the class declaring the instance variable may reference the variable's name; e.g. see the E programming language [Richardson 89a]). Of course, if encapsulation is not enforced, all inherited instance variable names must participate in the name space (as in Iris [Fishman 87]).

When multiple inheritance is permitted, some form of *conflict resolution* (see section 2.2.3 on Inheritance) must exist so that each inherited attribute has a unique source. Otherwise, ambiguity would result just as when two attributes of the same name are declared in one class. Also, other than the cancellation that occurs during conflict resolution, all of the systems surveyed require that a class inherit all attributes of a superclass (i.e. no explicit cancellation is allowed).

Systems that associate types with attributes enforce *type compatibility* (see section 2.2.4 on Polymorphism). That is, the system checks statically or dynamically that the class of any value assigned to an attribute is a descendant of the attribute's domain class in the is-a inheritance lattice. The $O_2$ system, for instance, statically checks all attribute assignments [Bancilhon 88].

Some systems go further and allow the application developer to specify additional constraints on the values that may be assigned to an attribute. Typically, these constraints limit the range of acceptable values (e.g. days of the month must be between 1 and 31, inclusive). More complex constraints involve predicates that legal values must satisfy; systems usually provide triggers to handle these cases (see section 2.1.3 on Triggers and Notifiers). Generally, triggers are also the only mechanism available to help ensure the maintenance of class invariants (i.e. predicates that must hold true for all class instances before and after each method invocation).

Finally, a system provides *referential integrity* if it guarantees that every object has a unique identifier and that if an object is referenced in a database (i.e. its identifier is *present*), then it resides in that database [Stein 89]. Clearly, databases that support *strong identity* (see section 2.2.1 on Object Identity) provide referential integrity. Another mechanism to achieve referential integrity would be to eliminate all references to deleted

objects.

If a database system provides other capabilities, additional integrity constraints might be required; see the sections below on Composition and Relationship Support.

Up to now, this paper has discussed only those features that are inherent in an object-oriented data model. Subsequent sections present capabilities that object-oriented systems can provide to ease the semantic modeling task of the application developer.

### 2.2.8 Composition

One way to help model the semantics of an application domain is to support *object composition* and *property propagation*.[12] Composition captures the semantics associated with the **is-part-of** relationship between objects [Kim 87]. Property propagation provides flexibility in determining attribute values (e.g. the color of a car should determine the color of its fenders) and allows finer control over such generic operations as delete, print, copy, equal, and save [Rumbaugh 88].

Consider again the task of modeling airplane designs. In a given design, each landing gear assembly consists of several parts, including a wheel, an axle, struts, and so forth. When operating on the assembly as a whole, all of these parts should participate as well. Object composition achieves this effect by associating a special composition property with the instance variables of the "owner" object; in this case, the landing gear assembly.

In its strongest sense, object composition implies that the part cannot exist without its owner nor be shared with another owner. That is, the object referenced by a composite instance variable must be destroyed when the owning object is destroyed and may only be created as part of the creation process of the owner. Similarly, since the referenced object "is part of" the whole, it cannot be "part of" another composite object [Kim 87].

These characteristics lead to new integrity constraints (see the previous section). In particular, no assignment may occur to a composite attribute outside of any constructor for the composite object. Second, an instance variable may not be changed from non-composite to composite, unless one can guarantee that existing objects of the class being modified do not already refer to another object's "part". Finally, when a new version of a composite object is created, any attribute tagged as referring to a specific version must be assigned either a copy of the referenced "part" or a null value [Kim 87].

Composition can improve database efficiency by allowing the application developer to increase the granularity of locking (see section 2.1.1 on Concurrency Control) and to specify object clustering for the storage management subsystem (see section 2.6.3 on Clustering) [Kim 87]. Thus, one can simultaneously lock all objects that are transitively "part of" a single, root composite object. (Note that locking becomes more complicated because a lock request on an object cannot be granted without checking that

---

[12] Rumbaugh actually uses the term "attribute" propagation, but I feel that there would then be confusion with class attributes (i.e. methods and instance variables).

the object is not "part of" a locked composite object.) Clustering improves performance by grouping objects together on secondary storage that are likely to have similar access patterns. Intuitively, the set of objects that are "part of" a composite object should exhibit such behavior [Hornick 87].

Composition as described above can sometimes be too restrictive. In such cases, the application developer probably would not use the facility. Kim, Bertino, and Garza describe how to make composition more flexible during their research using the ORION system [Kim 89b]. In particular, they separate the concepts of composition, exclusivity, and dependence.

If an instance variable is tagged as composite, they allow the reference to be either exclusive or shared. An *exclusive reference* means that an object "part" may have only one owner, whereas a *shared reference* allows multiple owners. Similarly, a reference may be either dependent or independent. A *dependent reference* indicates that the existence of the referenced object depends on its owner; i.e. it must be created and destroyed when its owner is. An *independent reference* may be assigned to at any time, not just within constructors. Thus, a "part" may have a life of its own [Kim 89b].

Again, integrity constraints must be changed because of the added functionality. In particular, deletion of a composite object containing dependent references will delete the referenced objects only if they are exclusive or it is the last container. Also, for example, to change a non-composite attribute into a shared composite attribute, one must ensure that there are no exclusive composite references of any sort within the database to objects that are already referenced by that attribute. Kim, et. al. list all of the new constraints in their paper [Kim 89b].

Although composite attributes may be used to propagate information between an "owner" and its "parts", the ability to propagate different properties independently requires additional mechanism [Rumbaugh 88]. Generic operations (e.g. equal, delete, copy, print, display, save) each need control over propagation. Most systems provide only three kinds of generic operations: name, shallow, and deep. For instance, name equality just checks whether two object references are identical, shallow equality checks whether the instance variables of two objects are identical, and deep equality recursively checks all references. *Property propagation* allows the application developer to indicate the instance variables that should participate in the recursive step. Thus, displaying a landing gear assembly might not show part numbers, but creating a duplicate should copy the part numbers as well.

In addition to the ORION project, the EXTRA data model of the EXODUS project allows shared and exclusive dependent composition (but apparently not independent) [Carey 88] and the ENCORE project provides composition for locking, clustering, and versioning [Hornick 87]. No system surveyed provides as much flexibility for property propagation as advocated by Rumbaugh; Rumbaugh has, however, implemented a programming language that includes features for property propagation [Rumbaugh 88].

### 2.2.9 Relationship Support

Composition augments the semantics of object interrelationships. However, object references are still inherently unidirectional. In using direct references between entities, the basic object-oriented data model inhibits the maintenance of data independence and the expression of multi-directional relationships, which improves semantic modeling [Chen 76]. The ability to express arbitrary relationships between objects would eliminate these problems and complement the advantages of the object-oriented paradigm [Rumbaugh 87].

For example, consider a landing gear assembly in our hypothetical airplane design application. If the designer wishes to maintain the set of struts for the assembly, he declares an instance variable in the assembly object to hold the set of struts. If the designer requires that each strut know which assembly it is a part of, then he can declare an instance variable in the strut object to refer to the containing assembly object. Without relationship support, however, the designer must ensure that all struts contained in an assembly's strut set refer back to that assembly. This invariant must be maintained explicitly for all operations (e.g. insertion and deletion) that affect this relationship.

A system that provides *entity-relationship support*, then, can maintain this extremely common invariant automatically. In addition to enabling symmetry, new integrity constraints can guarantee the uniqueness of either or both participants in one-to-many, many-to-one, and one-to-one relationships [Chen 76]. Furthermore, object relationships become explicit instead of being hidden throughout object implementations [Rumbaugh 87]. Finally, the use of relationships enhances the *data independence* of a database (i.e. the degree by which the data is independent of any one application).

In his programming language DSM, Rumbaugh has demonstrated how to integrate relationship support and object-oriented modeling [Rumbaugh 87]. Some database systems do provide relationship support through automatic updating of inverse relationships and key values (as in VBase [Andrews 87] and Iris [Fishman 87]). The GEM database language implements the entity-relationship model directly [Zaniolo 83]. Note that relationship support and triggers (see section 2.1.3 on Triggers and Notifiers) may be used to implement composite attributes (see the previous section on Composition).

### 2.2.10 Access to Meta-information

*Meta-information* in a database consists of the definitions that describe the information contained in the database. Thus, relation schemas and indexes constitute the meta-information for relational databases. In object-oriented databases, indexes, the dictionary of root objects, and the class definitions, including the properties of each instance variable (e.g. value domain, composite-ness, key-ness, relationship to other objects, and default value), comprise the meta-information of interest.

Access to meta-information is important for two reasons. First, generic applications can be built that manage databases and their information by examining the class definitions. Second, applications (like Alexandria) that allow users to manipulate and edit information structure should be able to model such structure directly onto class

definitions. Both the Iris [Fishman 87] and VBase [Andrews 87] database systems provide built-in functions to provide access to class and index definitions.

### 2.2.11 Data Sharing

Databases control the sharing of data by multiple users with transaction mechanisms. This section, however, addresses the issues concerning the sharing of data by other data.

*Data sharing* occurs when two or more objects refer to another, different object. In an application that deals with multimedia documents, this can happen when two documents contain the same picture. In CAD applications, data sharing takes place when two designs include a part that is defined in a common library. In both cases, data sharing is a means of reducing storage requirements and communicating updates [Woelk 86].

Object-oriented data models implement sharing naturally through direct object references. The difficult issues concern how updates to shared data are propagated while maintaining maximum concurrency. Using composition (see section 2.2.8 on Composition), it is possible to control the version of a shared object reference and to lock against concurrent updates. The problem with locking, however, is that transactions that operate on other objects that share data with a locked composite object cannot progress, even if they do not depend on the value of the shared data. One solution is to create a new locking mode that locks a composite object except for specified shared references which may be already locked by other transactions. Another is a form of optimistic concurrency. Allow the lock on a composite object to succeed even if a shared reference is already locked, but cause the transaction to abort if the shared object's value changes when the other transaction commits. No system surveyed has addressed these issues other than through composition.

### 2.2.12 Authorization

Maintaining privacy and preventing unsanctioned updates in a database constitute significant concerns. Database systems provide *authorization*, or *access control*, mechanisms to achieve these goals. An authorization is effectively a relationship among users, operations, and database entities. In a relational database, the entities are relations, relation columns, tuples, and schemas. In an object-oriented database, entities may be classes, objects, instance variables, or methods. As an example (for either case), the authorization relationship may indicate that a particular user has permission to read employee salaries but not to update them. On the other hand, because of the ability to embed semantics, only object-oriented database systems can restrict a user's access to executing a method that returns the total of all salaries without allowing that user to read any individual employee's salary.

This model for authorization is very flexible but also very expensive in terms of storage. Thus, it is essential that the *explicit authorizations* kept in the database be augmented with rules for determining *implicit authorizations*. There are two well-known paradigms for arranging authorizations. *Capability lists* associate permitted operations on classes of objects for each user while *access lists* associate authorized user categories by operation for each object. Neither paradigm alone is particularly flexible.

The object-oriented data model provides an opportunity to design an authorization mechanism that is both efficient and flexible [Rabitti 88]. A crucial observation is that each component of an authorization can be organized into a lattice of categories. Individual users can be members of groups and groups can be members of higher-order classifications, or "roles", and so on. Similarly, individual operations can be grouped into sets that may share characteristics (e.g. append and write). Finally, objects and classes naturally fall into a lattice through composition and inheritance.

As a result of this observation, a basic rule for determining implicit authorizations would be that an authorization exists for a given user, operation, and object if there is an explicit authorization for any user group, operation category, and object class and the user is a member of that group, the operation is in that category, and the object inherits from that class.

A second important observation is that sometimes it is simpler to grant sweeping authorizations and list exceptions than to try to list only those authorizations that are valid. Thus, one can augment the set of explicit *positive authorizations* with *negative authorizations* [Rabitti 88]. The rule for implicit authorizations, then, must also be augmented to determine when negative authorizations override positive authorizations, and vice versa. Certainly, the closest authorization along any path in the lattice overrides those farther along that path, but problems arise when differing authorizations apply from different paths. Possibilities include assigning priorities to users, operations, or objects; choosing the authorization closest in the lattice; or choosing negative authorizations over positive ones (to be safe).

Composite objects present another opportunity for implicit authorization [Kim 89b]. Authorizations involving a composite class or object can imply the same authorizations for all component objects. Conflicting authorizations for a component object, in this case, can be resolved in favor of the authorization applicable to the composite object dereferenced.

The overall effect of these observations is to reduce the number of explicit authorizations that must be stored in the database without adversely affecting flexibility. Other rules may be found that can further reduce the storage requirements for authorization. Also, note that the rules for determining authorization can apply to arbitrary property propagation as well.

## 2.3 Language Issues

Database developers program applications in whatever language the database system supports. Frequently, systems provide languages that fit the database's data model (e.g. SQL and SEQUEL for relational databases). These languages are not computationally complete, in general, so some database systems provide translators that allow a programmer to *embed* database language statements within general-purpose programming language programs (e.g. C or FORTRAN).

Database languages may be based on any of several *computational paradigms*. Languages using *data model paradigms* include SQL, SEQUEL, and object-oriented variants such as POSTQUEL [Stonebraker 86a], OSQL (used in Iris) [Fishman 87], and an unnamed language used in ENCORE [Zdonik 86]. Data descriptions may be written in special, separate *declarative* languages that do not allow arbitrary computation (or queries), such as the Type Definition Language (TDL) used in the VBase system [Andrews 87]. Finally, languages that do allow general computation may be based on any paradigm that is Turing-equivalent. For example, Zhu proposes that *rule-based* systems be used, in which computation is specified by pattern-action pairs (i.e. whenever a rule's pattern is matched, the corresponding action is performed on the data that matched the pattern) [Zhu 89]. Although many possibilities exist, however, most object-oriented systems use the *imperative paradigm*, in which database actions occur as normal statements within a program as in E (used in EXODUS) [Richardson 89b] or $CO_2$ (used in $O_2$ by the Altair project) [Bancilhon 88].

Traditionally, programming languages and database systems provide separate (but complementary) facilities. For example, languages have not dealt with semantic composition, persistence, or versioning, while databases have not dealt with structure traversal and computation [Kim 88]. In providing a framework for supporting both sets of features, object-oriented databases also provide an opportunity for achieving a unified language interface to those features. The following sections discuss the issues affecting such a unification.

### 2.3.1 Persistence

The first issue from a programming language perspective regarding the integration of database facilities concerns the handling of persistent entities, or values. Questions to resolve include how orthogonal are persistent values, what determines that a given value is persistent, and how to specify the database containing the persistent values of interest.

Full *orthogonality* is achieved when any value that is manipulated may be persistent [Laffra 90]. A system that provides essentially no orthogonality is Coral3, since persistent objects must be copied into transient space to be operated on and then returned to their special, persistent "holders" [Merrow 87]. Languages that allow embedding (e.g. embedded OSQL in C [Fishman 87]) may or may not achieve orthogonality. Persistent values must be transferred first between embedded language and host language variables. Once they are in host language variables, it is conceivable that persistent values may then be manipulated by the host language. The cleanest way to achieve orthogonality, of course, is to integrate persistence into a single paradigm programming language, as was done in the EXODUS project with the E programming language

[Richardson 89b].

Languages may determine whether or not a given value is persistent in several ways. Easiest would be to assume that all objects manipulated during the execution of an application are persistent (as apparently is the case in OPAL in the GemStone system [Maier 86]). A second approach is to identify one or more root objects and assume that all objects reachable from these roots are persistent (as in POMS [Cockshott 84]). Another approach is to specify special types for persistent values (as in E [Richardson 89b] and $CO_2$ [Bancilhon 88]).[13]

It has been claimed that using persistent type declarations yields the greatest execution efficiency while still providing orthogonality [Richardson 87]. Unfortunately, for applications that manipulate both persistent and transient data, this can lead to two identical sets of types. A more natural approach[14] would be to declare whether or not an object is persistent when it is created. Appropriate compiler technology exists so that an application loses no execution efficiency unless the full flexibility of a feature is used [Horowitz 88]. In this case, the appropriate way to enable efficiency is to declare whether a specific variable may hold persistent values.

Finally, a language achieves *database independence* when the application can specify the sources of persistent values. Unfortunately, most systems assume that all persistent values reside in a single database (as in E [Richardson 89b]), and, sometimes, the actual database used can even depend on the environment in force when the application was compiled. Clearly, object semantics must be stored after compilation and made available during execution in order to manipulate objects. However, the same application should be able to maintain separate databases for different users and share access to object semantics [Bancilhon 88].[15] As noted in section 2.1.4 on Distribution, the Iris database does allow an application to connect to different databases during the same execution, but apparently forbids simultaneous access [Fishman 87]. Simultaneous access is necessary for those applications, like Alexandria, that wish to present the information from different sources transparently.

### 2.3.2 Impedance Mismatch

The concept that complements orthogonality is *transparency*, which evaluates how well a programming language can hide the distinction between persistent and transient values [Laffra 90]. For example, one measure is whether a program can pass a persistent object to a routine that doesn't know whether or not the parameter is persistent.

Transparency and orthogonality address the larger issue of impedance mismatch in database programming languages. *Impedance mismatch* reflects the degree to which an application programmer must handle persistent values differently from transient values.

---

[13] Note that, for this approach, the compiler must ensure that no persistent pointer is assigned to a non-persistent pointer [Richardson 89b].

[14] This approach would also be more transparent; see the next section.

[15] Local databases can still be made self-contained (see section 2.2.6 on Extensibility) by copying object semantics into them.

A language providing orthogonality and transparency effectively eliminates impedance mismatch.

Clearly, different models of computation for persistent and transient values causes impedance mismatch (e.g. the declarative TDL vs. the imperative COP in VBase [Andrews 87] or embedded OSQL in Iris [Fishman 87]). Similarly, lack of orthogonality or transparency causes problems.

There are only three times when a programmer must know whether data is persistent or transient: when inserting an object into a database (e.g. during object creation), when removing an object from a database (e.g. during object destruction), and when efficiency is required. Computers deal most efficiently with transient values. Thus, a language should provide features that allow the programmer to specify when only transient values will be used.[16] Other than at these three times, a database programming language should provide complete orthogonality and transparency.

### 2.3.3 Software Engineering Issues

Because of their experimental nature, database programming language designs present an opportunity to try out new features that support good software engineering. Since a discussion of software engineering in general is outside the scope of this paper, this section discusses only those features that have been tried by the systems surveyed.

In object-oriented systems, classes constitute appropriate units for abstraction, reusability, and decomposition. The language mechanism that allows programmers to hide implementation details and define small interfaces is called *encapsulation*. Almost all systems provide encapsulation through classes; one notable exception is Iris, which has no class or encapsulation mechanism [Fishman 87]. Zdonik argues that encapsulation can interfere with certain database operations, such as query optimization [Zdonik 88], but it is well-known that the language compiler should cross encapsulation boundaries to find needed information. Classes and encapsulation also provide natural boundaries for *name spaces*, which delineate the scope of name-to-object relationships. Systems with imperative languages, such as COP in VBase [Andrews 87], can also take advantage of the block structuring inherent in the host language.

Although object-oriented type compatibility rules allow *static typing*, that is, checking the legality of assignments and message sends at translation time, to be very flexible, greatest flexibility is still achieved by *dynamic typing*, or checking during execution. Static typing is used in $CO_2$ [Bancilhon 88], COP [Andrews 87], and E [Richardson 87], whereas dynamic typing is used in OPAL [Maier 86], ORION [Kim 88] and Coral3 [Merrow 87]. OSQL (in Iris) goes even further than dynamic typing by allowing objects to acquire types dynamically, which is particularly useful for extending existing databases for new applications [Fishman 87].

---

[16] See, for example, the discussion in the previous section concerning the use of special database types (as in E [Richardson 89b]) vs. variable declarations.

Other features explored include type generators (as in COP [Andrews 87] and E [Richardson 89b]; see the discussion in section 2.2.5 on Genericity), structured exceptions (as in COP [Andrews 87]), and iterators (as in E [Richardson 89b]). Iterators allow programs to fetch component values one at a time from compositions like sets, lists, queues, and trees. In particular, iterators are especially useful because they enable arbitrary searching.

### 2.3.4 Host Languages

A *host language* is the language into which a database language is embedded or on which an integrated database language design is based. Embedded languages can use almost any host language as long as some mechanism exists that transforms database values into values the host language can manipulate. Iris, for instance, comes with preprocessors that allow programmers to embed OSQL statements in C or FORTRAN programs [Fishman 87].

Similarly, databases that present themselves as a function library can also be used in almost any host language. Abstractions (i.e. data types) must be defined for the data model concepts that may be manipulated (e.g. relation, tuple, database). An equivalent mechanism for transforming values must still be provided, however [Donahue 86]. The GemStone system, for instance, is accessible to programs written in C and Pascal [Purdy 87, Kernighan 78, Jensen 74].

No system surveyed designed an entirely new, integrated language for both database and arbitrary computation. Several have extended the designs of existing languages:

- the ORION system [Kim 88] extends Common LISP [Steele 84];
- GemStone's language OPAL [Maier 86] and the Coral3 system [Merrow 87] both extend Smalltalk [Goldberg 83];
- the COP language in the VBase project [Andrews 87] and the $CO_2$ language in the Altair project [Bancilhon 88] both extend C [Kernighan 78];
- PS-Algol in the POMS system [Cockshott 84] extends Algol-68 [Van Wijngaarden 75];
- the E language in the EXODUS project [Richardson 87] extends C++ [Stroustrup 86].

To a large degree, the features of the host languages affect the characteristics of their respective systems relative to the issues presented in the previous section. For instance, the ORION system provides dynamic typing because Common LISP does. On the other hand, the EXODUS and VBase projects did integrate interesting non-database features into their language designs (see the last paragraph of section 2.3.3 on Software Engineering Issues).

## 2.4 Query Issues

Relational databases provide *associative access* mechanisms that allow applications to find and manipulate entities based on their values instead of their reachability from other entities. Object-oriented database systems can provide both. This section examines the characteristics of search for object-oriented data models.

An application specifies a search of a portion of a database by issuing a query. A *query* is a description in some language of the nature of the objects to be retrieved and the domain over which to search. Query languages for object-oriented systems must allow more complex formulations than relational query languages. For example, applications dealing with hierarchically defined documents, such as those in SGML format, require the ability to search based on hierarchical structure [Macleod 89]. Section 2.4.1 below describes the capabilities that have been explored for object-oriented query languages.

Database systems generally support two means for speeding query execution. Indexes are search structures imposed on a collection of persistent entities in order to minimize the number of relatively slow secondary storage accesses and achieve sublinear complexity for value-based comparisons. Systems also attempt to optimize queries by transforming them into equivalent forms that execute faster. Sections 2.4.2 and 2.4.3 below discuss the characteristics of indexes and query optimization respectively in object-oriented databases.

### 2.4.1 Query Language

As noted above, databases must provide a language in which to specify queries. Most of the issues discussed in section 2.3 on Language Issues apply, most notably the computational paradigm used and the level of impedance mismatch. Many object-oriented systems have extended existing relational query languages (e.g. OSQL in Iris extends SQL [Fishman 87] and EXCESS in EXODUS extends QUEL [Carey 88]), while others expect searches to be specified in their extended imperative language (e.g. Common LISP in ORION [Kim 88], $CO_2$ in $O_2$ [Bancilhon 88], and OPAL in GemStone [Maier 86]).

The specification of a query must express the domain of objects to be searched and characterize the nature of the objects to be retrieved. In some systems, searches can only occur over entire classes, whereas other systems allow searches over arbitrary collections (e.g. ENCORE [Zdonik 88], $O_2$ [Bancilhon 88], EXCESS [Carey 88], ORION [Banerjee 87a], GemStone [Maier 86]). Because of inheritance, the system should also allow one to specify whether instances of subclasses should be included when searching over a given class's instances [Banerjee 88]. None of the systems surveyed actually makes this distinction.

Characterizing the objects to be retrieved involves defining a *selection*, or *filter predicate* (i.e. boolean expression) that each object must satisfy. The standard operations that object-oriented systems provide for composing a predicate include [Osborn 88]:

- constants (for direct comparison),
- comparison operators (such as "less than"),
- set operations (minimally union, intersection, and set difference),
- combination (which acts like Cartesian product), and
- partitioning (which acts like projection).[17]

An extension that appears in almost every object-oriented system is the ability to specify paths in a selection predicate. A *path*, or *nested expression*, is a sequence of attribute tags which computes a target object reachable from a given, identified object [Banerjee 88]. For example, one might request all employees whose company's president is older than fifty as follows:

```
select E
for each Employee E
where E.Division.Company.President.Age > 50
```

Thus, the path consists of the sequence of attributes <Division, Company, President, Age>. Each attribute in a path effectively acts like an *identity join*, ensuring equal object identities. The initial use of the dot notation constitutes a *functional join* [Zaniolo 83]. Most systems also allow traditional *value-based joins*, in which the value of one attribute is compared against the value of another attribute [Carey 88]. For instance, imagine if the expression E.Age replaced the constant 50 in the example above.

The standard operators described form an algebra on which there exists a set of equivalence transformations. These transformations may be used to improve the performance of query execution [Osborn 88]. Some systems even transform an object-based query into an equivalent relational query, which may then be optimized; the Iris system, in fact, must do this since its underlying storage system is a relational database [Fishman 87]. These issues are discussed in more detail below.

One of the more interesting advances introduced by object-oriented databases is the ability to incorporate user-defined operators into query predicates. Arbitrary operators may execute faster than equivalent predicates expressed using standard operators since they can be compiled. On the other hand, introducing non-standard operators can represent a security risk [Wilkinson 90] and complicate query optimization; again, see below.

Some researchers have examined how to deal with *cyclic queries*. Cyclic queries fall into two categories: (1) those whose predicates relate an object to itself and (2) those that iteratively operate on objects retrieved until no new objects result, which is useful for computing transitive closures. An example of the first kind occurs when requesting all employees managed by their spouse:

---

[17] Projection in an object-oriented system is a little strange since no user-defined class would correspond to the result. Some systems do not provide projection, and some believe that it is antithetical to the data model [Banerjee 88]. Other systems synthesize a semantics-free class for each projection. It is not clear whether the loss of projection would have an adverse impact on application design.

```
select E
for each Employee E
where E.Manager = E.Spouse
```

An example of the second kind occurs when requesting all direct and indirect managers of an employee named Joe:

```
select into Managers(M)
for each Employee E, Employee M
where (E in Managers and E.Manager = M)
   or (E.Name = "Joe" and E.Manager = M)
```

Both kinds generate a directed, cyclic graph as a query representation instead of an acyclic graph. Cyclic queries complicate the generation of access plans (i.e. which expressions in the query graph to execute when). In particular, cyclic queries greatly expand the number of possible access plans to be searched. Thus, one would like to show boundedness and termination, especially for iterative queries. Kim, Kim, and Dale describe several heuristics for computing efficient access plans for cyclic queries. They suggest a combination of forward and reverse traversals of the query graph (according to edge direction) and basing local decisions on the estimated cost of evaluating the expression corresponding to each node [Kim 89a]. They also seem to claim that using indexes instead of enumerating test cases is not helpful since some objects retrieved by an index search may not satisfy a cyclic predicate. I disagree, since a linear search after a logarithmic retrieval should often be faster than just a linear search.

### 2.4.2 Indexing

An *index* is a search structure in a database imposed on a collection of persistent entities. An index is used to reduce either the number relatively slow secondary storage accesses (i.e. disk probes) or the number of comparisons needed to find entities with attribute values in a given range. A search structure is based on one or more attributes (e.g. sort alphabetically by employee last name); thus, indexes act as *associative memory* relating attribute values to database entities. More than one index may be associated with an entity set.

In relational databases, an index sorts one or more attributes of a relation. In some object-oriented systems, an index may be applied only to the set of instances of a class. Most systems, however, allow indexes on arbitrary collections of objects (e.g. GemStone [Maier 86] and EXODUS [Richardson 89b]).

Several issues arise concerning indexes in object-oriented systems. First, different search structures are appropriate for different indexes. Several standard indexing techniques are in common use, including single attribute sorts (e.g. B-trees and its variants), radix sorts on multiple attributes, and multi-dimensional sorts (e.g. k-d-b trees for points [Robinson 81] and R-trees for hyper-rectangles [Guttman 84]). Database systems, however, should allow extension by user-defined structures. The ability to incorporate object semantics in object-oriented databases provides a natural mechanism for such flexibility.

Second, as in other data models, indexes require updates when attribute values of member objects change [Laffra 90]. For system-supported index structures, this should not present a problem. User-defined structures, however, require hooks so that the

appropriate invariants can be maintained. Triggers (see section 2.1.3 on Triggers and Notifiers) can provide this functionality [Zdonik 88].

Third, extended indexing operators may take advantage of generically written index methods. Search structures are excellent candidates for parameterization by component type. Thus, the B-tree algorithm, for example, can be written to work for any type having an operator that satisfies total ordering predicates [Stonebraker 88]. Object-oriented data models reap additional benefits because any algorithm written for one class will work for all classes that inherit that class.

Fourth, an index on the collection of class instances can include all instances of its subclasses (i.e. *class hierarchy indexing*) or only those instances of that class (i.e. *single class indexing*). The former yields better storage and query execution efficiency while the latter provides more flexibility for those queries specifying single-class results [Deux 90]. ORION [Kim 90] and $O_2$ [Deux 90] have chosen to provide only class hierarchy indexing.

Finally, the ability to specify nested or cyclic references in queries present opportunities for special indexing structures [Banerjee 88, Kim 89a]. Bertino and Kim identify three new structures for systems that allow path expressions in queries [Bertino 89].[18] A *nested index* associates all possible start values for each known end value of a given path expression. Such an index would help for queries requesting all employees whose company's president is older than fifty (see the first example in the previous section). A *path index* associates for each end value of a given path expression all values that yield that end value when starting anywhere along the path. This appears to be equivalent to a union of the corresponding nested indexes. In our example, a path index on the expression `E.Division.Company.President.Age` would, for each known age, associate all the appropriate subpaths to each age, starting from employees, divisions, companies, or presidents. Last, they define a *multi-index* to be the set of nested indexes that apply to the decomposition of the given path into single links. Thus, a multi-index on our path expression consists of nested indexes on the expressions `E.Division`, `D.Company`, `C.President`, and `P.Age`.

The three indexes have different resource requirements and performance characteristics. Nested and path indexes perform better when retrievals dominate, while multi-indexes perform better when modifications dominate. Their statistical analysis led to the conclusions that nested indexes should be used for paths shorter than three links and that multi-indexes are probably optimal for longer paths [Bertino 89].

### 2.4.3 Query Optimization

As in other data models, it is possible in object-oriented database systems to transform queries into equivalent forms that execute faster [Osborn 88]. *Query optimization* is the process of discovering such equivalent forms. Note that query "optimization" does not really optimize anything; it just generates a semantically identical query that executes faster [Graefe 88]. Clearly, any speed-up achieved must at least recover the work

[18] Note that these are not the only new index structures that are possible for nested or cyclic expressions.

expended in computing the improvement.

Transformations fall into three categories: (1) algebraic equivalences (e.g. projections may be pushed through selections), (2) transformations that make sense only in certain situations (e.g. removing redundant joins), and (3) those that help plan query execution (e.g. noting the existence and relative cost of indexes to affect which expression gets evaluated first) [Derrett 89].

Applying transformations is complicated in the object-oriented model because of the presence of user-defined operators in queries. One would like to treat user-defined and built-in operators identically (as in EXCESS [Carey 88]), but their presence forces interpretation of transformations instead of allowing optimization to be hard-coded into the system.

One mechanism for specifying the applicability of transformations is rule systems [Derrett 89]. Rules take the form of predicate-transformation pairs; when a piece of a query graph satisfies a predicate, the system may apply corresponding transformation on the subgraph. Rules may also include cost functions (e.g. expected number of entities returned or number of disk pages touched) to help determine which applicable rule to execute first [Graefe 88, Stonebraker 88]. In particular, these cost functions can take into account the existence of indexes [Zdonik 88]. Rule systems should possess several properties:

(1) soundness, both for individual rules and the rule set as a whole,[19]
(2) order independence (i.e. it shouldn't matter which of several applicable rules fires first),
(3) self-containment (i.e. a rule is not essential; it can be removed and the set remains sound),
(4) boundedness (i.e. an algorithm must exist that can execute the rules using finite and bounded resources), and
(5) efficacy (i.e. the rule set as a whole should yield more efficient queries).

In Iris, they found that dividing optimization into phases with separate rule sets makes designing rule sets that satisfy these criteria easier [Derrett 89]. Rule-based optimizers are also used in the EXODUS project [Graefe 87].

Graefe and Maier have proposed another approach to handle user-defined operators in queries. Instead of telling the optimizer what transformations are possible, the optimizer asks each user-defined operator to "reveal" an equivalent, more efficient query expression. This approach increases encapsulation and extensibility. The objective of such revelations should be to replace object-at-a-time evaluation with more efficient set-at-a-time searches [Graefe 88].

---

[19] A rule is *sound* if it represents a valid transformation. A rule set is *sound* if all applicable rule sequences yield valid transformations.

## 2.5 Database Evolution

As a result of the long-term nature of databases, an important issue concerns how database systems manage change. Many of the features of database systems deal with changes to the data: concurrency control, transactions, atomicity, data stability, triggers, notifiers, index maintenance, versions, and configurations. This section, however, concentrates on managing changes to the definitions that structure the data (i.e. changes to classes or schemas).

The three major concerns regarding *database evolution* involve what changes may occur, how those changes affect existing database entities, and how existing data in the database is reconciled with those changes. The next three sections consider each of these in turn.

### 2.5.1 Schema Changes

In an object-oriented database, the potential changes to class definitions and the class hierarchy include [Banerjee 87b]:

- adding or removing an instance variable;
- changing the name, domain, or default value of an instance variable;
- changing the composition or propagation properties of an instance variable;
- adding or removing a method;
- changing the name, implementation, or signature of a method;
- adding or removing a superclass;
- changing the conflict resolution order of inherited superclasses;
- adding or removing a class from the class hierarchy; and
- changing the name of a class.

Of course, these changes must be performed within transactions for the same reasons that changes to the data are [Hornick 87]. Some systems provide special transactions for changing class definitions. GemStone, for instance, requires the use of pessimistic locking for class modifications even though optimistic concurrency control is normal [Penney 87]. Others require that all other activity cease while database evolution occurs.

### 2.5.2 Effects of Changes

Integrity constraints reflect the invariants that must be maintained in order to ensure data consistency. Thus, the system must either prohibit certain changes or modify objects in the database to re-establish the invariants. The GemStone system forbids the removal of a class with instances [Penney 87], whereas the ORION system just removes all instances as well [Banerjee 87b].

In fact, the ORION system attempts to provide rules for modifying objects for almost all class definition changes. For example, removing an instance variable from a class causes it to be removed from all instances of that class as long as no other instance variable of the same name is inherited. Furthermore, each instance of an inheriting class must be modified similarly as long as its class does not override the instance variable. Still, some changes cannot be allowed. The user may not introduce a cycle into the class hierarchy [Banerjee 87b].

Class changes also affect the definitions of user-defined operators. In particular, some message invocations may no longer be valid [Skarra 86]. It is difficult to see how rules for automatically modifying embedded semantics can be generated. Certainly, however, notification of the existence of such problems should be the minimum response.

One can also examine the effects class changes have on the results of queries. By enumerating each possible combination of query operators and class modifications, Osborn has shown that one can predict whether a query will result in the same set of objects before and after a class change [Osborn 89]. These predictions can enhance one's confidence that a given change will not affect existing applications adversely.

### 2.5.3 Database Conversion

Once a class change has occurred and object modifications identified, those modifications have to be made. Some systems make the modifications immediately (i.e. *eager conversion*; as in GemStone [Penney 87]) while others perform *lazy conversion* by inserting checks for inconsistent objects in every operation (as in ORION [Banerjee 87b]). The first method often takes a database off-line for significant amounts of time and the second adds a non-trivial cost to each operation.

Some systems attempt to circumvent these problems by avoiding object modifications. The ENCORE system, for instance, treats classes as objects and generates new versions as a result of changes [Hornick 87]. Thus, any object existing in the database before the change is also treated as a previous version; its class is the old version [Skarra 86]. A system can help applications based on the new type access objects of the old type by providing *emulation*; that is, by supporting operations that allow the object to appear to be of the new type.

Another approach is taken by the Iris system. Objects in the Iris system may acquire or lose types dynamically.[20] Thus, if an object no longer matches a changed definition, the user can choose to remove the type from the object instead of modifying the object to match the type. In general, Iris tends to restrict class modifications so that object modifications are not necessary. For example, a class cannot be removed unless it has no instances, nor can new supertype-subtype relationships be established [Fishman 87].

All of the effects of a class modification apply to all databases that use the affected class definition. Clearly, every database must be converted for those systems that allow several databases to share definitions.

---

[20] Duhl calls this capability *dynamic type acquisition* [Duhl 88].

## 2.6 Storage Management

Generally, storage management support for a database is responsible for secondary storage management, variable-sized data support, buffering, caching, concurrency control, versioning, logging and recovery. This section addresses the issues concerning data storage, buffering, caching, clustering, and interoperability raised by object-oriented models.

Although relational back-ends have been used for storage management (as in Iris [Fishman 87]), new designs have also been explored for object-oriented systems. The object-oriented data model leads to several performance constraints and presents several opportunities for efficiency. For instance, the storage of complex objects generated in object-oriented database systems would benefit from support for variable-sized data [Carey 86]. Support for multimedia data also requires the ability to store different sizes potentially for each object [Stonebraker 88].

The property of object identity implies the need for efficient access of direct references. Users will tend to follow such references instead of searching and fast look-ups would obviate the need for some user-imposed indexes [Donahue 86]. On the other hand, direct references reduce the need for searching and enable new, more efficient indexes (see section 2.4.2 on Indexing). Also, the ability to build composite objects can make locking, authorization, versioning, and storage management more efficient.

The next section describes various schemes used to store objects and manage the transformation between storage and object formats. Section 2.6.2 deals with how objects affect buffering and caching. A specific method for enhancing the performance of storage management is discussed in the following section. Finally, the last section discusses issues relating to interoperability.

### 2.6.1 Storage Schemes

The simplest way to store persistent data between executions of an application is to save the program's image. This technique, however, forces persistent data to reside always in the same place, to maintain the same format, and to fit within the process's address space [Cockshott 84]. By treating persistent data differently, database systems avoid these problems and provide flexibility as to the sources of data that applications can manipulate.

Storage management for object-oriented systems must deal with issues concerning *storage reclamation* (e.g. *garbage collection*), object identity, variable-sized data, and transforming between external storage and internal run-time formats.

Reclaiming storage can either be done automatically when objects become inaccessible (as in GemStone [Maier 86]) or explicitly by applications using special operators (as in E [Richardson 89a]). Explicit object deallocation is faster but can lead to dangling references.

Storage reclamation of working versions (see section 2.1.5 on Versions and Configurations) that are still referenced requires some kind of archiving strategy. For example, a system can archive all versions (or configurations) older than a given date. Or, if access times are available, the system could archive only those versions (or configurations) that have not been read for a given amount of time. Such archiving is necessary to reduce normal storage requirements.

Databases support object identity by assigning a unique *object identifier* to each object which is never re-used, even when the associated object becomes deallocated. Different databases, however, incorporate different levels of indirection between an object identifier and the actual storage for its associated object. EXODUS maps each object's identifier directly to its storage, which implies that the object's header cannot be moved and that resizing requires another scheme (see below) [Carey 86]. ENCORE's storage manager ObServer (OBject SERVER), on the other hand, uses two levels of identifiers. The top-level object identifier maps to a set of low-level identifiers which then map to the actual storage chunks that comprise the object [Hornick 87]. When two or more levels are used, the top-level identifier is called a *surrogate*.

Resizing and handling variable-sized data can be handled in several ways. The telesophy system ts1 assumes that objects are resized infrequently and therefore stores each object in one contiguous chunk [Caplinger 87]. ObServer uses a level of indirection to allow each object to refer to its "chunks" [Hornick 87].[21] EXODUS uses an implicit indirection scheme in which each object is represented by a B+ tree (which also supports efficient versioning) [Carey 86].

Finally, since direct references cannot be represented as actual pointers in secondary storage, some transformation must be performed between the storage format and the format supported by the data model. Some systems transform each access of a direct reference (as in ENCORE's ObServer [Hornick 87]) while others convert all references to pointers when each object is buffered (as in ORION [Kim 88]). The first makes buffering easier but slows normal execution. Overall, however, fewer conversions may be needed. The second scheme allows execution to proceed at normal processor speeds, which is important in highly interactive applications. The $O_2$ system compromises by transforming each object once, but only if it actually manipulated [Bancilhon 88].

### 2.6.2 Buffer Management

Typically, a storage manager handles requests from many database clients simultaneously and the total amount of data referenced could easily overflow its address space. Thus, a storage manager must buffer the data actually required at any one time, and treat that buffer as a cache.

Most systems implicitly pin the objects referenced by a transaction so that they will remain in the buffer for the transaction's duration. A commit or abort then unpins the data which allows the system to remove the objects from the cache. The E language

---

[21] Each chunk corresponds to the memory required to store the instance variables of each class inherited by the class of the object, including one for any instance variables added by the object's class.

run-time system, however, just pins the data implicitly. The system requires that the application explicitly unpin the data and indicate whether the data was modified. By making pinning explicit, the EXODUS system can allow an application direct, efficient access to the buffered data. No copying into the application's address space is necessary [Carey 86].

ORION uses a double buffering scheme in which the first buffer contains the file format of an object and the second buffer contains the transformed format. Since referenced objects may have been removed from the cache, the transformed reference actually points to a header which keeps track of whether the object is resident. ORION, however, does not maintain a mapping from object references to headers, so when an object is brought back into the cache, it is given a new header. The old headers, when dereferenced, must then be forwarded. Unreferenced memory (data and headers) are garbage collected [Kim 88].

POMS,[22] on the other hand, maintains a double hash table that maps between persistent references and in-memory addresses, which obviates the need for special headers. It also uses special operating system primitives to trap attempted dereferences of persistent identifiers (i.e. all negative addresses are "protected"). Thus, using the central hash table, the system can maintain the state of its cache without unnecessary forwarding structures and execution can proceed at normal speeds without special checks [Cockshott 84].

Buffering does have a negative impact on the execution of queries within transactions, since all current data does not reside in one place. Some of the data is in the buffers while the rest of the data (perhaps represented by indexes) is in secondary storage. Kim, et. al. conclude that the system can either perform the query twice and merge the results or flush the buffers before executing the query. The latter, of course, can have an adverse impact on storage management support [Kim 90].

Finally, since storage managers cannot know ahead of time the access patterns of applications, greater efficiency could be achieved if the system can accept hints concerning its buffer replacement policy (as in the EXODUS system [Carey 86]).

### 2.6.3 Clustering

Storage management performance can be improved by reducing the number of secondary storage operations required. The primary method for accomplishing this is to group together objects that will be referenced together, effectively reducing the persistent "working set" of an application. Since most systems cannot know a priori which objects will be referenced together, some systems allow applications to give hints for *clustering* objects into *segments* on secondary storage.

Clustering hints can be determined explicitly or implicitly. The VBase system, for instance, allows explicit clustering hints when objects are created [Andrews 87]. A system that provides composite object support can determine clustering implicitly, since

[22] The Persistent Object Management System.

objects that are "part of" another are likely to be accessed when the parent object is [Kim 87]. Similarly, objects in a collection (e.g. all objects of a given class) are also likely to be accessed together, so a system might also cluster such objects implicitly (as in ENCORE [Hornick 87]). Finally, a system could monitor access patterns and attempt to cluster objects in the hope that such patterns are predictive of future behavior (also as in ENCORE [Hornick 87]).

### 2.6.4 Interoperability

The last issue to be dealt with concerns *interoperability*, or the ability to transmit persistent data in a form understandable to clients [Laffra 90]. This is particularly important for database systems that act as servers in a distributed, heterogeneous environment. The cleanest approach is to define pack and unpack operators for each class and each client machine. These operators convert objects between a storage or transfer format understood by all run-time systems (e.g. ASCII text or ANS.1 byte stream) and the specific client format for each object. This approach has been taken in the telesophy system tsl [Caplinger 87].

A related problem arises concerning the storage format and execution of object semantics. As discussed earlier in section 2.1.4 on Distribution, data model support, including the execution of object semantics (i.e. methods), usually occurs in data model servers. In a heterogeneous network environment, this can pose a problem. If the language expressing object semantics is compiled, then the server can run only on machines for which object code is available. Similarly, if the language is interpreted, then the server can run only on machines for which an interpreter has been written. The latter approach is easier to implement, but decreases the execution efficiency of applications. No system surveyed has addressed this issue.

# 3 Research Efforts

Most of the issues discussed in this report were collected from presentations of research efforts into database design. This section summarizes each of these efforts individually, instead of spreading a system's description throughout the discussion of design issues, as above.

## 3.1 POSTGRES

*Affiliation:*

University of California, Berkeley

*Publications:*


[Rowe 87] L. A. Rowe, M. Stonebraker. *The POSTGRES Data Model.*

[Stonebraker 86a] M. Stonebraker, L. A. Rowe. *The Design of POSTGRES.*

[Stonebraker 87a] M. Stonebraker. *The Design of the POSTGRES Storage System.*

[Stonebraker 87b] M. Stonebraker, E. N. Hanson, S. Potamianos. *The POSTGRES Rule Manager.*


*Description:*

POSTGRES involves an effort to extend the relational data model to include embedded semantics, data type extensibility (e.g. multimedia data), cyclic queries, rule-based triggers, and versioning. Embedded semantics is achieved by allowing attributes whose values may be either POSTQUEL statements or external language code fragments (e.g. a C procedure). POSTQUEL is an extension of QUEL, a language based on the relational data model paradigm. Versioning can be achieved either by time of creation, modification, or deletion, or by explicit version.

Although based on the relational data model, the POSTGRES data model does possess certain object-oriented characteristics. The schema definition for a relation may inherit the attributes of one or more other schemas. POSTQUEL also allows the creation of user-defined operators, which may then be used in other POSTQUEL statements, including queries. The model, however, does not support object identity.

Other issues: POSTGRES uses pessimistic concurrency with non-nested transactions. Its architecture allows distribution where a single POSTMASTER is responsible for session control and triggers. The POSTGRES run-time system (one per client) is responsible for secondary storage management, caching, logging, recovery, authorization, query execution, execution of embedded semantics, and object support. There is no explicit support for configurations, polymorphism, genericity, user-defined indexes, or access to meta-information. Composition and relationship support do not apply in this data model.

## 3.2 EXODUS

*Affiliation:*

University of Wisconsin-Madison

*Publications:*


[Carey 86] M. J. Carey, D. J. DeWitt, J. E. Richardson, E. J. Shekita. *Object and File Management in the EXODUS Extensible Database System.*

[Carey 88] M. J. Carey, D. J. DeWitt, S. L. Vandenberg. *A Data Model and Query Language for EXODUS.*

[Graefe 87] G. Graefe, D. J. DeWitt. *The EXODUS Optimizer Generator.*

[Richardson 87] J. E. Richardson, M. J. Carey. *Programming Constructs for Database System Implementation in EXODUS.*

[Richardson 89a] J. E. Richardson, M. J. Carey. *Persistence in the E Language: Issues and Implementation.*

[Richardson 89b] J. E. Richardson, M. J. Carey, D. T. Schuh. *The Design of the E Programming Language.*


*Description:*

The goal of the EXODUS project is to provide tools for exploring alternate database designs. Thus, EXODUS itself just refers to the storage management support. A data model called EXTRA, EXCESS, a query language variant of QUEL based on that data model, and E, an imperative, statically typed language based on C++, have been implemented on top of EXODUS.

The EXTRA data model is object-oriented with multiple inheritance; conflict resolution requires explicit renaming. The E language provides embedded semantics, polymorphism through overriding, genericity through type parameters, and iterators. The language can be used to achieve orthogonality and transparency, but efficiency concerns lead to some impedance mismatch. The EXTRA data model does support the notions of shared and exclusive dependent object composition, value-based joins, and self-referential cyclic queries.

Queries may search over specified collections instead of entire classes and may contain user-defined operators. The query optimizer accepts hints via a rule-based system concerning the relative costs of query operators.

The storage manager supports variable-sized data, and multimedia may be represented in the E language. The manager also provides direct access to the buffered objects (for efficiency) and accepts hints concerning its buffer replacement policy and clustering objects. Storage reclamation must be specified explicitly in the E language (i.e. weak identity). The manager is also responsible for versioning (versions are apparently unavailable at higher levels), locking (at arbitrary granularities; again, apparently unavailable at higher levels), logging, and recovery.

Other issues: No support is given for transitive closure queries except through direct programming in the E language. The E language is also the only means for introducing new indexes, which may be imposed on arbitrary collections. The language assumes a single database for all transactions. Also, there is apparently no support for session control; each execution of an application constitutes a single transaction. No support is given for acquiring access to meta-information.

## 3.3 Altair

*Affiliation:*

Altair, France

*Publications:*


[Bancilhon 88] F. Bancilhon, et. al.
*The Design and Implementation of $O_2$, an Object-Oriented Database System.*

[Deux 90] O. Deux, et. al.
*The Story of $O_2$.*

[Lecluse 88] C. Lecluse, P. Richard, F. Velez.
*$O_2$, an Object-Oriented Data Model.*


*Description:*

The Altair project is responsible for the $O_2$ data model and the $CO_2$ programming language, a statically typed, imperative language based on C. The formal data model is object-oriented with multiple inheritance. The user is expected to resolve conflicts among identical names inherited from more than one superclass. The data model supports polymorphism through overriding.

The $CO_2$ language is not totally transparent since types, not values, are declared to be persistent. Data persistence is determined by reachability from named root objects. The model calls for strong identity; thus, the system provides garbage collection. Apparently, multimedia support may be achieved using the language. Queries may search over arbitrary collections, and the system supports simple, path-oriented, class hierarchy indexes. User-defined operators may be used in queries, which means that cyclic queries are allowed.

A mirror server process is instantiated for each client to handle data model issues and deal with communication with the secondary storage manager. Object identifiers constitute single-level indirection into secondary storage; i.e. they do not represent surrogates.

Other issues: The language is the only means for introducing new indexes that may be imposed on arbitrary collections; they will not be used, however, by the standard query mechanism. The system separates the persistence of class definitions from the persistence of normal objects. Also, there is apparently no support for session control; each execution of an application constitutes a single transaction (although save-points may be specified). No support is given for genericity, specifying clustering, triggers, composition, relationships, or acquiring access to meta-information.

## 3.4 ORION

*Affiliation:*

Microelectronics and Computer Technology Corporation (MCC)

*Publications:*


[Banerjee 87a] J. Banerjee, et. al.
*Data Model Issues for Object-Oriented Applications.*

[Banerjee 87b] J. Banerjee, W. Kim, H-J. Kim, H. F. Korth.
*Semantics and Implementation of Schema Evolution in Object-Oriented Databases.*

[Kim 87] W. Kim, J. Banerjee, H-T. Chou, J. F. Garza, D. Woelk.
*Composite Object Support in an Object-Oriented Database System.*

[Kim 88] W. Kim, et. al.
*Integrating an Object-oriented Programming System with a Database System.*

[Kim 89a] K-C. Kim, W. Kim, A. Dale.
*Cyclic Query Processing in Object-Oriented Databases.*

[Kim 90] W. Kim, J. F. Garza, N. Ballou, E. Woelk.
*Architecture of the ORION Next-Generation Database System.*

[Rabitti 88] F. Rabitti, D. Woelk, W. Kim.
*A Model of Authorization for Object-Oriented and Semantic Databases.*

[Woelk 87] D. Woelk, W. Kim.
*Multimedia Information Management in an Object-Oriented Database System.*

*Description:*

The ORION system presents an object-oriented data model that provides polymorphism through dynamic method resolution and genericity through dynamic typing (instead of through type parameters; thus, homogeneity cannot be guaranteed). Concurrency control is pessimistic, with transactions locking on an object or composite object basis. ORION provides very flexible composition facilities, allowing distinctions for shared vs. exclusive and dependent vs. independent references. It also supports object versions and generic vs. specific references.

The data model supports multiple inheritance in which conflicts are resolved in favor of the superclass inherited earliest. Weak identity is supported, forcing the application developer to manage storage reclamation explicitly. Multimedia data support has been integrated into the ORION data manager. User-defined operators may be associated with classes and used in queries, which may operate over arbitrary collections of objects. Cyclic nested queries (both self-referential and transitive closure) are allowed. Applications developers may provide hints for query optimization. Currently, the system manages only single attribute, class hierarchy indexes.

Authorization is addressed, with both positive and negative authorizations allowed. Composition affects the granularity of authorization here as well. Access is provided through an extension of Common LISP (thus, the dynamic typing). Orthogonality is limited since types must be declared as persistent.

ORION is the only system to address thoroughly automatic database conversion for evolution. Many transformations have been derived to re-establish invariants after class changes. These transformations occur lazily, by inserting checks into each operation for objects that must be updated.

The storage manager uses a double buffering scheme: the first buffer caches segments holding the file format of objects, while the second buffer caches the in-memory format. Each object is converted from its file format immediately when buffered. Forwarders are used to deal with dereferences to objects that are no longer in the object cache.

Other issues: Apparently, no nested transactions are allowed and no trigger facility is provided. No access to meta-information is described. Other than composition, no relationship support is provided. Also, class membership is apparently the only mechanism for determining clustering. The only configuration support is the ability to refer to either a generic or specific version of an object.

## 3.5 ENCORE

*Affiliation:*

Brown University

*Publications:*


[Hornick 87] M. F. Hornick, S. B. Zdonik. *A Shared, Segmented Memory System for an Object-Oriented Database.*

[Skarra 86] A. H. Skarra, S. B. Zdonik. *The Management of Changing Types in an Object-Oriented Database.*

[Smith 87] K. E. Smith, S. B. Zdonik. *Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems.*

[Zdonik 86] S. B. Zdonik, P. Wegner. *Language and Methodology for Object-Oriented Database Environments.*

[Zdonik 88] S. B. Zdonik. *Data Abstraction and Query Optimization.*


*Description:*

The ENCORE project was started in response to the needs of other projects, notably the Intermedia hypermedia project. It presents a standard object-oriented data model that supports multiple inheritance, strong identity, versioning, and triggers. Simple exclusive composition is provided, which is also used to specify granularity for locking, versioning, and storage clustering. No indication is given as to how or whether polymorphism and genericity are provided. Also, although support for multimedia is a stated goal, no indication is given as to how ENCORE provides such support.

The system uses a standard architecture for distribution, providing a data model server for each client. The storage manager transforms each object from its storage format on each access by an application instead of when the object is buffered.

Queries may employ user-defined operators and may operate on arbitrary object collections. The set of indexing structures may be extended, and optimization hints are accepted via rule systems.

Database evolution is accomplished through versions. Classes are objects; thus, class changes are reflected through versions. No provision, however, is given for managing configurations.

Other issues: Apparently, no support is given for meta-information access, nested transactions, relationship support (other than by using triggers), or authorization.

## 3.6 GemStone

*Affiliation:*

Servio Logic Corporation

*Publications:*


[Maier 86] D. Maier, J. Stein, A. Otis, A. Purdy.
*Development of an Object-Oriented DBMS.*

[Penney 87] D. J. Penney, J. Stein.
*Class Modification in the GemStone Object-Oriented DBMS.*

[Purdy 87] A. Purdy, B. Schuchardt, D. Maier.
*Integrating an Object Server with Other Worlds.*


*Description:*

The GemStone system presents an object-oriented data model with single inheritance (as of their publications). Polymorphism is provided through dynamic method resolution and genericity through dynamic typing. The primary access to GemStone is through an extension of Smalltalk, although limited access can be achieved in C and Pascal via libraries. Orthogonality and transparency are ignored, as all objects are considered persistent. The system provides strong identity, so storage reclamation occurs when objects become inaccessible.

Distribution is achieved by interposing any number of data model servers between the storage server (which is responsible for authorization and indexing) and clients. The data model servers execute queries written in their extended Smalltalk (called OPAL). Thus, arbitrary queries are allowed. Queries may be executed over arbitrary collections. Indexes may also be defined over arbitrary collections. No other indication of query optimization has been discussed.

Concurrency control is optimistic, except for class changes. Database evolution is generally constrained by integrity constraints, but some automatic changes to objects are understood; these changes occur immediately.

Other issues: Apparently, no nested transactions are allowed and no trigger facility is provided. No access to meta-information is described, nor any mechanism for extending types for multimedia (other than what is provided in Smalltalk). No relationship, composition, clustering, or versioning support is provided.

## 3.7 Iris

*Affiliation:*

Hewlett-Packard Laboratories

*Publications:*


[Derrett 89] N. Derrett, M-C. Shan.
*Rule-Based Query Optimization in IRIS.*

[Fishman 87] D. H. Fishman, et. al.
*Iris: An Object-Oriented Database Management System.*

[Wilkinson 90] K. Wilkinson, P. Lyngbaek, W. Hasan.
*The Iris Architecture and Implementation.*


*Description:*

The Iris database system does not adhere strictly to the object-oriented model. Its data model is a form of entity-relationship model where the entity classes may inherit one or more other classes. Polymorphism is achieved through compile-time overload resolution, although overriding is provided when explicitly requested. Genericity applies only to the built-in collection types.

Although there is no support for composition, full relationship support is provided. Weak identity is provided, requiring applications to deal with storage reclamation explicitly. Object identifiers act as surrogates and do not indicate where an object is stored. Simple versioning is supported.

An extension of SQL, OSQL, provides access to the database. Currently, this language may be embedded within C or FORTRAN. Thus, no transparency or orthogonality is provided. All clients communicate with the storage system (which is relational) through one data model server. Transactions may be nested and use pessimistic concurrency control.

Objects may acquire and lose types dynamically, which is how it is expected databases will evolve. Objects retain their identity across type changes. Access to meta-information is provided. Multimedia objects can be accommodated. Queries may include user-defined operators. Thus, queries may be self-referential, although transitive closures do not seem to be supported. Queries and indexes apply to entire classes; the query optimizer will accept hints in the form of a rule system.

Other issues: Apparently, no support is provided for triggers, configurations, authorization, or clustering.

## 3.8 VBase

*Affiliation:*

Ontologic, Inc.

*Publications:*

[Andrews 87] T. Andrews, C. Harris.
*Combining Language and Database Advances in an Object-Oriented Development Environment.*

[Duhl 88] J. Duhl, C. Damon.
*A Performance Comparison of Object and Relational Databases Using the Sun Benchmark.*

*Description:*

VBase, since renamed Ontos, presents an object-oriented data model with single inheritance (as of their writings). The database supports embedded semantics, triggers, optional attributes, some relationship support (i.e. inverse management), access to meta-information, clustering hints at object creation, genericity through type parameters, and polymorphism through overriding. Although it is not clear, it appears that VBase supports weak identity, requiring the application to manage storage reclamation explicitly.

All access to the database is through two proprietary languages, a declarative schema definition language (TDL) and an extension of C for writing semantics and applications (COP). The C extension is statically typed, although run-time type assertions may be used. The language also includes support for exception handling. It is not clear which values are persistent and how a database is specified.

Queries and indexes must be written in COP; apparently, no explicit support is otherwise provided. As such, queries and indexes may therefore apply to arbitrary collections, and the application developer may extend the set of available indexes. No consideration is given to query optimization.

Other issues: Apparently, no support is provided for session control, nested transactions, authorization, logging, composition, or database evolution.

## 3.9 GEM

*Affiliation:*

Bell Laboratories

*Publications:*

[Zaniolo 83] C. Zaniolo.
*The Database Language GEM.*

*Description:*

The GEM database is an early attempt to extend the relational data model. It does not present an object-oriented data model. Its data model can be described as a tuple-set model or a typed, entity-relationship model. As such, GEM supports object identity, relationships, optional attributes through null values, union types, and nested path expressions in queries (in an extension of QUEL). Apparently, cyclic queries are not supported.

## 3.10 Coral3

*Affiliation:*

System Concepts Laboratory, Xerox Palo Alto Research Center

*Publications:*

[Merrow 87] T. Merrow, J. Laursen.
*A Pragmatic System for Shared Persistent Objects.*

*Description:*

Coral3 is also not an object-oriented database system. It is an attempt to introduce persistence into Smalltalk. Thus, Smalltalk's object-oriented model is presented: strong identity, polymorphism through dynamic method resolution, genericity through dynamic typing, and triggers through Smalltalk's "dependents" mechanism. Persistent objects are accessed through special holders, which must be dereferenced before operating on the data. Thus, no orthogonality or transparency is achieved. For performance, caching of persistent objects is supported.

In addition, no concept of session control is provided. In the absence of transactions, an application is provided with a facility for setting locks explicitly in order to achieve concurrency control. The assumption is that the objects manipulated are large enough so that no complex interactions are necessary. No other database-like facility is provided.

## 3.11 Telesophy

*Affiliation:*

Bell Communications Research

*Publications:*

[Caplinger 87] M. Caplinger.
*An Information System Based on Distributed Objects.*

[Schatz 89] B. R. Schatz, M. A. Caplinger.
*Searching in a Hyperlibrary.*

*Description:*

*Telesophy* is a term coined by Bruce Schatz to mean "wisdom at a distance". The system implemented to demonstrate telesophy, tsl, is primarily oriented toward hypertext, although search is also an inherent component. The system is not strictly object-oriented in that inheritance is not supported, although object identity is.

The primary contribution of tsl is that it presents a model for distributed access to distributed data. Objects may be placed in storage servers. Indexes for arbitrary object collections are managed by other servers. An application executes a query by first searching one or more indexes, then submitting the resulting object identifiers to the appropriate storage servers.

The information space is flat in that no abstraction is allowed. No restriction, however, is placed on the form of objects; in fact, support for multimedia data is an explicit goal. Objects are stored in contiguous chunks and optimistic concurrency control is provided on the assumption that the data will be changed very infrequently. Interoperability is supported by requiring each node in the network to provide pack and unpack routines for each data type. No other database-like facility is provided.

## 3.12 POMS

*Affiliation:*

University of Edinburgh

*Publications:*

[Atkinson 82] M. Atkinson, K. Chisholm, P. Cockshott.
*PS-algol: an Algol with a Persistent Heap.*

[Cockshott 84] W. P. Cockshott, et. al.
*Persistent Object Management System.*

*Description:*

The Persistent Object Management System (POMS) is the storage management facility for PS-Algol, which integrates persistent values into a variant of Algol-68 called S-Algol. PS-Algol provides dynamic connections to databases (apparently only one at a time), transaction support, associative access, and implicit storage reclamation (i.e. garbage collection). The data model is that of Algol-68, which does not support inheritance. Orthogonality and transparency, however, are achieved. Persistent objects are determined by their reachability from a specified database root object. POMS uses a double hash table so that no forwarding is required for caching persistent objects. Finally, POMS stores data type information in each database so that it is self-contained.

# 4 Conclusions

One of the primary objectives of a database system is to handle the storage of persistent data for applications. By using a database system, application developers do not need to worry about how persistent data is organized in secondary storage. They must, however, be concerned with managing persistent data using the data model presented by the database system. In the past, data models have been designed for such qualities as data independence, ease of real-world modeling (for limited real-world domains), and ease of providing other, desirable database features. Little or no consideration was given to the need of applications to compute with persistent data or to the computational models used to implement applications.

Object-oriented databases, then, constitute an attempt to integrate most of the desirable features of database systems (see Figure 1) with desirable features of the object-oriented model of computation (see Figure 2).

Data persistence
Storage management
Concurrency control
Session control
Atomicity
Recoverability
Authorization
Versions
Configurations
Associative access
Triggers
Distribution
Interoperability

**Figure 1:** *Features provided by database systems*

Object identity
Direct references
Inheritance
Polymorphism
Genericity
Encapsulation
Name space control
Data abstraction extensibility
Data type extensibility
Procedural extensibility
Imperative execution model

**Figure 2:** *Features provided by object-oriented languages*

In doing so, three issues arise. First, integration should occur without impedance mismatch. In particular, language support for object-oriented database services should be orthogonal and transparent. Second, integration should not lose any advantages of existing data models. For instance, object-oriented programming does not support data independence inherently, so features such as relationship support and query joins should

be provided. Finally, integration presents an opportunity for introducing new desirable features. For example, this survey includes descriptions of the following features:

- composition
- property propagation
- cyclic queries
- indexing extensibility
- database self-containment
- access to meta-information
- database evolution
- database independence

All but the last are affected by object-oriented concepts. Composition, property propagation, and cyclic queries take advantage of the ability to specify direct references betweeen objects. Indexing extensibility and database self-containment derive from the ability to embed semantics. Last, access to meta-information and database evolution depend on the data model supported by the database system.

Object-oriented databases also present the potential for being more efficient than earlier systems that support other data models. Direct references, the ability to represent complex designs, and the ability to execute user-defined operators all contribute to building more efficient database applications. The caveat, of course, is that these benefits must offset the overhead caused by the need to interpret these higher levels of abstraction [Duhl 88].

Finally, the list of capabilities provided by object-oriented database systems compares favorably to those required by multimedia applications (see the discussion at the end of section 1.1 on Motivation), and information management applications (e.g. Intermedia [Smith 87]; see also the discussion in section 1.2 concerning our own project, Alexandria), as well as engineering applications, such as CAD/CAM, and other design applications. No other data model supported by database systems in current use can make this claim.

# 5 References


[Ada 83] *American National Standard Reference Manual for the Ada Programming Language.*
ANSI/US Dept. of Defense, 1983.
ANSI/MIL-STD 1815A-1983.

[Andrews 87] T. Andrews, C. Harris.
Combining Language and Database Advances in an Object-Oriented Development Environment.
In *Conference Proceedings, OOPSLA '87 (SIGPLAN Notices)* 22(12):430-440, December 1987.

[Atkinson 82] M. Atkinson, K. Chisholm, P. Cockshott.
PS-algol: an Algol with a Persistent Heap.
*SIGPLAN Notices* 17(7):24-31, July 1982.

[Bancilhon 88] F. Bancilhon, et. al.
The Design and Implementation of $O_2$, an Object-Oriented Database System.
In *Advances in Object-Oriented Database Systems: 2nd International Workshop*, ed. K. R. Dittrich, pp. 1-22, Springer-Verlag, 1988.

[Banerjee 87a] J. Banerjee, et. al.
Data Model Issues for Object-Oriented Applications.
*ACM Transactions on Office Information Systems* 5(1):3-14, January 1987.

[Banerjee 87b] J. Banerjee, W. Kim, H-J. Kim, H. F. Korth.
Semantics and Implementation of Schema Evolution in Object-Oriented Databases.
In *International Conference on the Management of Data (SIGMOD Record)* 16(3):311-322, December 1987.

[Banerjee 88] J. Banerjee, W. Kim, K-C. Kim.
Queries in Object-Oriented Databases.
In *Proceedings 4th International Conference on Data Engineering*, pp. 31-38, 1988.

[Bertino 89] E. Bertino, W. Kim.
Indexing Techniques for Queries on Nested Objects.
*IEEE Transactions on Knowledge and Data Engineering* 1(2):196-214, June 1989.

[Caplinger 87] M. Caplinger.
An Information System Based on Distributed Objects.
In *Conference Proceedings, OOPSLA '87 (SIGPLAN Notices)* 22(12):126-137, December 1987.

[Carey 86] M. J. Carey, D. J. DeWitt, J. E. Richardson, E. J. Shekita.
Object and File Management in the EXODUS Extensible Database System.
In *Proceedings of the 12th International Conference on Very Large Databases*, August 1986.

[Carey 88] M. J. Carey, D. J. DeWitt, S. L. Vandenberg.
A Data Model and Query Language for EXODUS.
In *International Conference on the Management of Data (SIGMOD Record)* 17(3):413-423, September 1988.

[Chen 76] P. P. Chen.
The Entity-Relationship Model -- Toward a Unified View of Data.
*ACM Transactions on Database Systems* 1(1):9-36, March 1976.

[Cockshott 84] W. P. Cockshott, et. al.
Persistent Object Management System.
*Software-Practice and Experience* 14(1):49-70, January 1984.

[Derrett 89] N. Derrett, M-C. Shan.
Rule-Based Query Optimization in IRIS.
In *ACM 17th Annual Computer Science Conference*, pp. 78-86, February 1989.

[Deux 90] O. Deux, et. al.
The Story of $O_2$.
*IEEE Transactions on Knowledge and Data Engineering*, 2(1):91-108, March 1990.

[Donahue 86] J. Donahue, C. Hauser, J. Kent.
*A Client Interface to an Entity-Relationship Database System.*
Technical Report CSL-86-4, Xerox Palo Alto Research Center, Xerox Corporation, September 1986.

[Duhl 88] J. Duhl, C. Damon.
A Performance Comparison of Object and Relational Databases Using the Sun Benchmark.
In *Conference Proceedings, OOPSLA '88 (SIGPLAN Notices)* 23(11):153-163, November 1988.

[Eppinger 89] J. L. Eppinger.
*Virtual Memory Management for Transaction Processing Systems.*
PhD thesis, CMU-CS-89-115, Carnegie Mellon University, February 1989.

[Fishman 87] D. H. Fishman, et. al.
Iris: An Object-Oriented Database Management System.
*ACM Transactions on Office Information Systems* 5(1):48-58, January 1987.

[Goldberg 83] A. Goldberg, D. Robson.
*Smalltalk-80: The Language and its Implementation.*
Addison-Wesley, Reading, Massachusetts, 1983.

[Graefe 87] G. Graefe, D. J. DeWitt.
The EXODUS Optimizer Generator.
In *Proceedings of the ACM SIGMOD Conference*, pp. 160-171, May 1987.

[Graefe 88] G. Graefe, D. Maier.
Query Optimization in Object-Oriented Database Systems: A Prospectus.
In *Advances in Object-Oriented Database Systems: 2nd International Workshop*, ed. K. R. Dittrich, pp. 358-363, Springer-Verlag, 1988.

[Guttman 84] A. Guttman.
R-Trees: A Dynamic Index Structure for Spatial Searching.
In *International Conference on the Management of Data (SIGMOD Record)* 14(2):47-57, 1984.

[Hornick 87] M. F. Hornick, S. B. Zdonik.
A Shared, Segmented Memory System for an Object-Oriented Database.
*ACM Transactions on Office Information Systems* 5(1):70-82, January 1987.

[Horowitz 88] M. L. Horowitz.
*Automatically Achieving Elasticity in the Implementation of Programming Languages.*
PhD thesis, CMU-CS-88-104, Carnegie Mellon University, January 1988.

[Jensen 74] K. Jensen, N. Wirth.
*PASCAL User Manual and Report: 2nd Edition.*
Springer-Verlag, Berlin, 1974.

[Kernighan 78] B. W. Kernighan, D. M. Ritchie.
*The C Programming Language.*
Prenuce-Hall, Englewood Cliffs, New Jersey, 1978.

[Kim 87] W. Kim, J. Banerjee, H-T. Chou, J. F. Garza, D. Woelk.
Composite Object Support in an Object-Oriented Database System.
In *Conference Proceedings, OOPSLA '87 (SIGPLAN Notices)* 22(12):118-125, December 1987.

[Kim 88] W. Kim, et. al.
Integrating an Object-oriented Programming System with a Database System.
In *Conference Proceedings, OOPSLA '88 (SIGPLAN Notices)* 23(11):142-152, November 1988.

[Kim 89a] K-C. Kim, W. Kim, A. Dale.
Cyclic Query Processing in Object-Oriented Databases.
In *Proceedings 5th International Conference on Data Engineering*, pp. 564-571, 1989.

[Kim 89b] W. Kim, E. Bertino, J. F. Garza.
Composite Objects Revisited.
In *International Conference on the Management of Data (SIGMOD Record)* 18(2):337-347, June 1989.

[Kim 90] W. Kim, J. F. Garza, N. Ballou, E. Woelk.
Architecture of the ORION Next-Generation Database System.
*IEEE Transactions on Knowledge and Data Engineering*, 2(1):109-124, March 1990.

[Laffra 90] C. Laffra, P. van Oosterom.
Persistent Graphical Objects.
In *TOOLS '90*, June 1990.

[Lecluse 88] C. Lecluse, P. Richard, F. Velez.
$O_2$, an Object-Oriented Data Model.
In *International Conference on the Management of Data (SIGMOD Record)* 17(3):424-433, September 1988.

[Macleod 89] I. A. Macleod.
*A Query Language for Retrieving Information from Hierarchic Text Structures.*
Technical Report TR 89-263, Department of Computing and Information Science,

Queen's University, Kingston, Ontario, August 1989.

[Maier 86] D. Maier, J. Stein, A. Otis, A. Purdy.
Development of an Object-Oriented DBMS.
In *Conference Proceedings, OOPSLA '86 (SIGPLAN Notices)* 21(11):472-482, November 1986.

[Merrow 87] T. Merrow, J. Laursen.
A Pragmatic System for Shared Persistent Objects.
In *Conference Proceedings, OOPSLA '87 (SIGPLAN Notices)* 22(12):103-110, December 1987.

[Meyer 88] B. Meyer.
*Object-Oriented Software Construction.*
Prentice Hall, New York, 1988.

[Osborn 88] S. L. Osborn.
Identity, Equality and Query Optimization.
In *Advances in Object-Oriented Database Systems: 2nd International Workshop*, ed. K. R. Dittrich, pp. 346-351, Springer-Verlag, 1988.

[Osborn 89] S. L. Osborn.
The Role of Polymorphism in Schema Evolution in an Object-Oriented Database.
*IEEE Transactions on Knowledge and Data Engineering* 1(3):310-317, September 1989.

[Penney 87] D. J. Penney, J. Stein.
Class Modification in the GemStone Object-Oriented DBMS.
In *Conference Proceedings, OOPSLA '87 (SIGPLAN Notices)* 22(12):111-117, December 1987.

[Palay 90] A. Palay, D. Anderson, M. Horowitz, M. McInerny
*The Alexandria Project: In Search of a Unified Environment for Information Access and Management.*
In preparation.

[Purdy 87] A. Purdy, B. Schuchardt, D. Maier.
Integrating an Object Server with Other Worlds.
*ACM Transactions on Office Information Systems* 5(1):27, January 1987.

[Rabitti 88] F. Rabitti, D. Woelk, W. Kim.
A Model of Authorization for Object-Oriented and Semantic Databases.
In *Proceedings of the International Conference on Extending Database Technology*, pp. 231-250, March 1988.

[Richardson 87] J. E. Richardson, M. J. Carey.
Programming Constructs for Database System Implementation in EXODUS.
In *International Conference on the Management of Data (SIGMOD Record)* 16(3):208-219, December 1987.

[Richardson 89a] J. E. Richardson, M. J. Carey.
Persistence in the E Language: Issues and Implementation.
*Software-Practice and Experience* 19(12):1115-1150, December 1989.

[Richardson 89b] J. E. Richardson, M. J. Carey, D. T. Schuh.
*The Design of the E Programming Language.*
Technical Report CS-TR 824, University of Wisconsin-Madison, February 1989.

[Robinson 81] J. T. Robinson.
The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes.
In *International Conference on the Management of Data (SIGMOD Record)* 10:10-18, April 1981.

[Rowe 87] L. A. Rowe, M. Stonebraker.
The POSTGRES Data Model.
In *Proceedings of the 13th International Conference on Very Large Data Bases*, pp. 83-96, September 1987.

[Rumbaugh 87] J. Rumbaugh.
Relations as Semantic Constructs in an Object-Oriented Language.
In *Conference Proceedings, OOPSLA '87 (SIGPLAN Notices)* 22(12):466-481, December 1987.

[Rumbaugh 88] J. Rumbaugh.
Controlling Propagation of Operations using Attributes on Relations.
In *Conference Proceedings, OOPSLA '88 (SIGPLAN Notices)* 23(11):285-296, November 1988.

[Schatz 89] B. R. Schatz, M. A. Caplinger.
Searching in a Hyperlibrary.
In *Proceedings 5th International Conference on Data Engineering*, pp. 188-197, 1989.

[Skarra 86] A. H. Skarra, S. B. Zdonik.
The Management of Changing Types in an Object-Oriented Database.
In *Conference Proceedings, OOPSLA '86 (SIGPLAN Notices)* 21(11):483-495, November 1986.

[Smith 77a] J. M. Smith, D. C. P. Smith.
Database Abstractions: Aggregation.
*Communications of the ACM* 20(6):405-413, June 1977.

[Smith 77b] J. M. Smith, D. C. P. Smith.
Database Abstractions: Aggregation and Generalization.
*ACM Transactions on Database Systems* 2(2):105-133, June 1977.

[Smith 87] K. E. Smith, S. B. Zdonik.
Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems.
In *Conference Proceedings, OOPSLA '87 (SIGPLAN Notices)* 22(12):452-465, December 1987.

[Steele 84] G. L. Steele, Jr.
*Common LISP: The Language.*
Digital Press, 1984.

[Stein 89] J. Stein, T. L. Anderson, D. Maier.
Mistaking Identity (Preliminary Report).
In *Proceedings of the 2nd International Workshop on Database Programming Languages*, pp. 161-164, 1989.

[Stonebraker 86a] M. Stonebraker, L. A. Rowe.
The Design of POSTGRES.
In *International Conference on the Management of Data (SIGMOD Record)* 15(2), May 1986.

[Stonebraker 86b] M. Stonebraker.
Inclusion of New Types in Relational Data Base Systems.
In *Proceedings Second International Conference on Data Base Engineering*, February 1986.
Also in *Readings in Database Systems*, ed. M. Stonebraker, pp. 480-487, Morgan Kaufman Publishers, Inc. (San Mateo, CA), 1988.

[Stonebraker 87a] M. Stonebraker.
The Design of the POSTGRES Storage System.
In *Proceedings of the 13th International Conference on Very Large Data Bases*, pp. 289-300, September 1987.

[Stonebraker 87b] M. Stonebraker, E. N. Hanson, S. Potamianos.
The POSTGRES Rule Manager.
*IEEE Transactions on Software Engineering* 14(7):897-907, July 1988.
Also in *Proceedings Third International Conference on Data Engineering*, February 1987.

[Stroustrup 86] B. Stroustrup.
*The C++ Programming Language.*
Addison-Wesley, Reading, Massachusetts, 1986.

[Ullman 82] J. D. Ullman.
*Principles of Database Systems.*
Computer Science Press, Rockville, Maryland, 1982.

[Van Wijngaarden 75] A. Van Wijngaarden, et. al.
Revised Report on the Algorithmic Language ALGOL-68.
*Acta Informatica*, 5:1-236, 1975.

[Wegner 89] P. Wegner, S. B. Zdonik.
Models of Inheritance.
In *Proceedings 2nd International Workshop on Database Programming Languages*, pp. 248-255, 1989.

[Wilkinson 90] K. Wilkinson, P. Lyngback, W. Hasan.
The Iris Architecture and Implementation.
*IEEE Transactions on Knowledge and Data Engineering*, 2(1):63-75, March 1990.

[Woelk 86] D. Woelk, W. Kim, W. Luther.
An Object-Oriented Approach to Multimedia Databases.
In *International Conference on the Management of Data (SIGMOD Record)* 15(2):311-325, May 1986.

[Woelk 87] D. Woelk, W. Kim.
Multimedia Information Management in an Object-Oriented Database System.
In *Proc. Very Large Data Bases*, September 1987.

[Zaniolo 83] C. Zaniolo.
The Database Language GEM.
In *International Conference on the Management of Data (SIGMOD Record)* 13(4):207-218, 1983.

[Zdonik 86] S. B. Zdonik, P. Wegner.
Language and Methodology for Object-Oriented Database Environments.
In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, vol. II, pp. 378-387, 1986.

[Zdonik 88] S. B. Zdonik.
Data Abstraction and Query Optimization.
In *Advances in Object-Oriented Database Systems: 2nd International Workshop*, ed. K. R. Dittrich, pp. 368-373, Springer-Verlag, 1988.

[Zdonik 90] S. B. Zdonik, D. Maier, eds.
*Readings in Object-Oriented Database Systems.*
Morgan Kaufman Publishers, Inc. (San Mateo, CA), 1990.

[Zhu 89] J. Zhu, D. Maier.
Computational Objects in Object-Oriented Data Models.
In *Proceedings 2nd International Workshop on Database Programming Languages*, pp. 139-160, 1989.

# I Object-Oriented Languages

In an object-oriented language, all manipulable program entities are *objects*. Each object consists of some private state and a set of operations that may manipulate that state. No operations other than the associated ones may manipulate the internal state of an object. This property is known as *encapsulation*. In object-oriented parlance, the operations are called *methods*. The method set of an object defines its *semantics*.

In most programs, many collections of objects share method sets. For example, all objects that represent pieces of mail in a mail handling application share the methods that manipulate the internal state of mail messages. Moreover, such objects also share the form of their internal state. Thus, the description of the common properties (i.e. form, behavior, and methods) of a collection of objects can be localized. Most object-oriented systems provide a facility called a *class* to define these common properties. Each object in the collection described by a class is called an *instance* of that class. Associated with each class may be a set of operations whose role is to create object instances; these operations are called *constructors*.

A class defines the internal state of its instances by specifying a set of *instance variables* or *attributes*. The set of attributes itself is referred to as an *aggregation*. Attributes that are shared by all instances of a class (e.g. the average age of all employees) are called *class variables* or *attributes*. Sometimes, the specification of each attribute may include a *default value* and/or the expected *domain* of values the attribute may assume. This domain, or *type*, represents the *behavior*,[23] or set of operations, that values within that domain possess. Typically, types and classes are identified, so that the set of operations for each domain is specified by referring to a class.

The set of classes in an object-oriented system may be structured in two ways. One class may *use* another if it refers to the other, usually as a domain for an attribute. This organization is called an *aggregation* or *composition hierarchy*. Note that this organization may not be a strict hierarchy; cycles are allowed in the uses relationship.

A class may also *inherit* properties from one or more classes. That is, the definition of a class would include the properties of these other classes together with any new properties added in the class itself. This organization is referred to as a *generalization hierarchy*.[24] Note that this organization is only a strict hierarchy if each class is limited to inheriting from exactly one other class (i.e. *single inheritance*); the organization becomes a lattice, or directed acyclic graph, if *multiple inheritance* is allowed. Each class inherited from is called a *superclass*, and the inheriting class is called the *subclass*.

---

[23] Another term for behavior is *protocol*.

[24] Since the inverse relationship of generalization is *specialization*, this organization may sometimes be referred to as a specialization hierarchy. It all depends on which point of view is taken (i.e. which way the "arrows" should be drawn).

In standard object-oriented systems, the properties defined in a class need not just augment the properties inherited from its superclass(es). Some properties may replace inherited properties. When this occurs, we say that the defined property *overrides* the inherited property. Almost always, this occurs with the operations of a class. In the presence of overriding, the method executed when one operation invokes another depends on the actual class of the object of interest. That is, the invocation just states the name of the operation to be executed; a method of that name must be found in the class (or one of its ancestors) of the run-time value that is the focus of the invocation.

Because of this ambiguity, the invocation of an operation is referred to as a *message send* and the object of interest is called the *receiver*. Furthermore, the behavior of a class is really defined by a set of *messages*, which effectively are just the names and signatures of the operations to which instances of the class will respond.

From an application programmer's point of view, these concepts provide characteristics that aid in the application building process. In particular, the programmer may model application entities at appropriate levels of abstraction. Furthermore, classes comprise a good mechanism for decomposing complex design problems into manageable chunks. Class inheritance and method overriding support *polymorphism*, which allows attributes to assume values that are only weakly related; specifically, such values are related only by the messages they may receive. Finally, class inheritance also enables extensibility, so that new objects may be added consistently to an existing system without modifying existing code (or data).

# II Glossary

A ---

*abort* The act of terminating a transaction such that the state of the database remains as it was when the aborted transaction commenced.

*access control* See *authorization*. Usual kinds of access that are discriminated include read, write, append, execute, modify definition, etc.

*access list* A form of explicit authorization in which associated with each object is a set of user-permitted operation pairs. Contrast with *capability list*.

*access plan* An execution order for evaluating the subparts of a query.

*acyclic graph* A directed graph that contains no cycles.

*aggregation* A collection of data. Each data item is tagged with an attribute name.

*aggregation hierarchy* The graph that results among classes when considering the uses relationship. That is, each edge in this graph reflects that the source class uses the destination class. Note that this graph is rarely a strict hierarchy and often contains cycles.

*alerter* See *notifier*.

*algebraic equivalence* An equivalent expression based on equalities proven in some algebra.

*application* A program that requires the services of one or more subsystems and provides an interface to some interpretation of those services. For instance, a mail handling application provides an interpretation on the data in a database of messages.

*archiving* The storage of old object versions off-line (e.g. on tape or optical disk), which is not accessible without operator intervention.

*associative access* The ability to retrieve entities from a database based on associated data. For example, associative access allows one to retrieve all records for employees earning more than their manager. Contrast with *direct access*.

*associative memory* Memory in which accesses are made by association instead of by address. In other words, one looks for the item associated with a given value instead of, say, the tenth item in memory.

*atomic value* A value which has no subparts; for example, an integer, boolean, or floating point value.

*atomicity* The property that a database reflects either all changes engendered by a transaction or no changes, even in the presence of failures.

*attribute* A name to be used when dealing with (i.e. assigning to and fetching) a specific value in an aggregation. See, for instance, the concept of *instance variable* for objects.

*authorization* A protection mechanism provided by database systems to ensure that a given user is allowed to perform a specified operation on a particular object.

B ---

*behavior* The set of operations, or methods, that describe the semantics of a domain or type.

*behavior compatibility*
A form of inheritance in which inherited attributes always have the same semantics (i.e. as an algebra with interpretations).

*boundedness* A property that states that a computation's use of a resource (e.g. time or storage) has a reasonable limit.

*browsing* The ability to retrieve database entities through direct references as opposed to searching based on attribute values. See *direct access*, *search*, and *associative access*.

*buffering* The process by which databases provide access to persistent entities. The database system must transform the data from the format used for secondary storage to a form in main memory that applications can use. For object-oriented systems, this may involve translating unique object identifiers into direct references.

C ---

*caching* A buffering scheme that attempts to provide the most efficient access for data expected to be used by a client, especially when it is impractical to maintain such access for all data. Caches are used for buffering database data kept on secondary storage and for keeping local copies of data kept across a network. A popular caching scheme is to buffer a fixed number of data items that have been most recently used, on the assumption that something just used is likely to be used again soon.

*cancellation* A form of inheritance in which only some implementation is shared and some attributes may be eliminated by the inheritor.

*capability list* A form of explicit authorization in which associated with each user is a set of object-permitted operation pairs. Contrast with *access list*.

*Cartesian product*
The construction of virtual database entities from two collections of other entities (either actual or virtual) by unioning the attributes of each entity from one collection with the attributes of each entity from the other. Thus, if there are N and M entities in the two collections, the resulting collection has NxM entities.

*class* In object-oriented systems, a repository for the common definitions (i.e. instance variables and methods) of a collection of similar objects and for common properties (e.g. constructors, default values, and global variables) shared by the collection. Classes may inherit definitions and properties from one or more other classes.

*class attribute* See *class variable*.

*class hierarchy index*
An index on a collection representing the instances of a class that includes the instances of all the subclasses of that class.

*class variable* An attribute-value pair associated with a class and shared by all objects described by that class.

*clustering* A policy of grouping objects together on secondary storage that are likely to have similar access patterns in order to improve performance.

*column* In relational systems, a column is the same as an attribute in a relation. Relations are often compared to tables in which attributes head the columns and tuples comprise the rows.

*combination* See *Cartesian product.*

*commit* The act of terminating a transaction such that the state of the database is updated to reflect all changes brought about by the transaction.

*composite attribute*
See *composite reference.*

*composite reference*
A reference from one object to another reflecting a composition relationship between the objects.

*composition* A property of a set of objects when one refers to the others as its parts or subcomponents. That is, composition reflects the is-part-of relationship.

*composition hierarchy*
See *aggregation hierarchy.*

*computational paradigm*
A model for how data computation and manipulation occurs. A programming language presents a computational paradigm to programmers.

*concurrency control*
Any mechanism provided to ensure data consistency in the presence of concurrent access. See *optimistic concurrency* and *pessimistic concurrency.*

*concurrent access*
The ability of more than one user to access the same database at the same time.

*configuration* A set of object versions that reflects a consistent state for the set of objects as a whole.

*conflict resolution*
A mechanism that chooses one definition over another when both have been inherited and both have the same identifier name. Conflict resolution is necessary in the presence of multiple inheritance and constitutes a limited form of cancellation.

*constructor* A class property that is an operation which may be used to create an instance of that class.

*cycle* A situation in a graph in which a path exists from a given node to itself along one or more links.

*cyclic graph* A directed graph that contains one or more cycles.

*cyclic query* A query whose graph representation contains one or more cycles. This occurs when the query either refers to its own results (as in transitive closure) or relates a database entity to itself.

D ---

*DAG* An abbreviation for directed, acyclic graph.

*dangling reference*
A direct or value-based reference whose target entity is not in the database. Dangling references can arise in systems that provide weak identity.

*data abstraction extensibility*
The ability to extend the set of types recognized by a system.

*data consistency*
The property that changes are not lost and that data interrelationships make sense.

*exclusive reference*
A composite reference to an object indicating that no other root object may reference that object. See *shared reference*.

*explicit authorization*
An authorization that is kept in the database and can be consulted directly. See *implicit authorization*.

*extensibility* The ability to incorporate new functionality into an existing system. The best extensibility is when new functionality is treated just as built-in functionality.

F ---

*field* See *attribute*.

*file system* The facility provided by operating systems which permits applications to store data in discrete units called files for long periods of time. File systems often provide a hierarchically organized name space, but rarely provide any other structuring mechanisms.

*filter predicate* See *selection predicate*.

*flat value space* A value space that allows no composition. Composition in this sense occurs during aggregation or when constructing value sets.

*function library* A set of operators that may be called from any application that has access to library's name space.

*functional join* A join which is based on the value of a function (e.g. a simple attribute dereference).

G ---

*garbage collection*
See *storage reclamation*. Garbage collection generally refers to schemes that reclaim storage of inaccessible objects.

*generalization* The abstraction of common properties. In object-oriented systems, generalization is used to describe the relationship between a class and its subclasses that inherit it. The inverse relationship is called specialization.

*generalization hierarchy*
The graph that results when considering the inheritance relationships between classes. That is, each edge of this graph reflects that the source class is inherited by the destination class. Note that a strict hierarchy results only when single inheritance is allowed; otherwise, a lattice results.

*generic reference*
An inter-object reference indicating that the referenced object should be the most current, transient version. See also *specific reference*.

*genericity* The ability to handle different types using the same code but yet guaranteeing some form of homogeneity. For example, one module can implement both "set-of-employee" and "set-of-vehicle" and yet enforce that no vehicle be allowed in the former.

*granularity* The coverage of a given characteristic. When applied to data, granularity can range from the smallest identifiable unit of data to arbitrary data collections. See, for example, *lock granularity*.

H ---

*hierarchical data model*
A data model in which database entities (i.e. records of attribute-value pairs) may be related in strict hierarchies (i.e. each entity may have a single parent).

*hierarchy* A graph or network in which each node has at most one identifiable parent and possibly many children. In particular, this implies that there exists exactly one path from a root to any other node in the graph.

*horizontal extension*
The addition of new properties by the inheritor. Thus, for example, a subclass typically adds messages or instance variables to what is inherited from its superclass(es).

*host language* A language in which another language is embedded or on which a new language is based.

*hypertext* An organization (i.e. directed graph) for documents that effectively allows readers to follow related ideas as they wish.

I ---

*identity join* A join which is based on the equality of database entities in the two collections being joined.

*identity-based access*
See *direct access*.

*impedance mismatch*
The degree to which an application must handle persistent and transient values differently.

*imperative paradigm*
A computational paradigm in which desired results are specified by indicating exactly how to achieve those results. Contrast with *declarative paradigm*.

*implicit authorizations*
An authorization that is not stored in the database and is determined from other authorizations, whether explicit or implicit. See *explicit authorization*.

*independent reference*
A composite reference that may be assigned different objects at any time. The referenced object need not be destroyed when an owning object is. See *dependent reference*.

*index* A search structure in a database imposed on a collection of persistent entities. Thus, one collection may have several associated indexes which "optimize" searching that collection in different ways.

*indexing extensibility*
The ability to extend the kinds of indexing structures used in a database system. Most systems provide B-tree variants for indexing single attributes. Extensibility would allow the incorporation of radix and multi-dimensional indexing structures (e.g. k-d-b trees and R-trees).

*inheritance* The sharing of definitions of one class by another. That is, if one class inherits from another, the definitions of the first class include in some way the definitions of the second.

*data independence*

The degree to which two sets of data do not depend on each other. In the object-oriented model, inter-object references reduce data independence, whereas in the relational model, relations are independent of each other because of value-based identity.

*data model* A mathematical description of the allowable ways a user may organize the data held in a database. Several examples include the hierarchical data model, the network data model, the relational data model, and the object-oriented data model.

*data model paradigm*

A computational paradigm that includes just those operations specified by a data model. See the various database data models: *entity-relationship data model*, *relational data model*, *hierarchical data model*, *network data model*, *object-oriented data model*.

*data sharing* The ability to consider a database entity as a subcomponent of two or more other entities (e.g. a picture shared by two documents).

*data type extensibility*

The ability to extend the kind of data recognized by a system. In particular, this allows new interpretations of data (e.g. an array of bytes may represent an image, a matrix, a sequence of audio, a collection of graphic objects, or a sequence of video).

*database* A subsystem that provides services relating to the maintenance of persistent data.

*database conversion*

The process of changing existing database entities to match changed schema or class definitions. Conversion may be lazy (as entities are encountered during subsequent normal processing) or eager (i.e. off-line; immediately after the schema changes and before normal database processing resumes). Also, instead of conversion, database entities can be made to appear as instances of the changed class via emulation.

*database evolution*

The process of changing the definition of data instances and the support for transforming existing database instances to the new definition.

*database independence*

The property that applications can specify the source(s) of persistent data instead of assuming a source based on the environment in effect at the time of compilation or execution. Execution time is still better than compile time since then the application can work on different databases for different users. Full independence occurs when the application can specify the source(s) during execution.

*deadlock* A condition in a system allowing concurrency when a cycle exists in the graph of processes waiting on other processes to release needed resources. The simplest case occurs when two processes each possess a resource the other needs in order to complete its transaction. Locks constitute one such resource.

*declarative paradigm*

A computational paradigm in which the desired result is specified with little or no indication of how the result is to be achieved. Contrast with *imperative paradigm*.

*default value* A value that should be associated with an attribute until some other value is specified.

*dependent reference*

A composite reference to an object indicating that the object must be created and destroyed when its owner is. See *independent reference*.

*direct access* The ability to retrieve an entity from a database based on its name or unique identifier. In object-oriented systems, objects may refer to one another directly, and direct access is achieved when one retrieves an object referred to by another.

*directed graph* A graph in which each edge has a distinct source and target. See *undirected graph.*

*distribution* The ability of a database to operate over a network, to divide responsibilities or data among several processes (potentially on different machines), or to manage data replicated on several machines.

*domain* A characterization of the set of legal values that may be associated with an attribute. In object-oriented systems, the domain specification is almost always the class of objects that are allowed. See also *type compatibility.*

*dynamic method resolution*
Any mechanism that determines at run-time the method to execute for a message send.

*dynamic type acquisition*
The ability of an existing database object to assume and lose types during its lifetime.

*dynamic typing* Type checking that occurs during program execution instead of during translation. Dynamic typing requires the presence of type information at run-time. See *static typing.*

E ---

*eager conversion* The conversion of database instances of a modified class immediately after the change is committed.

*embedded language*
A self-contained language that has been made accessible inside of another language. For example, the language of expressions can be considered as embedded within a full, imperative language (such as Pascal).

*embedded semantics*
The storage of how to interpret data along with the data itself. Typically, "how to interpret" the data means either some form of data description (e.g. schemas in relational databases) or computer programs for the operations that manipulate the data (as in object-oriented database systems).

*emulation* The interpretation of database instances of a old class version to make them seem as if they are instances of a new class version.

*encapsulation* The property where all access to an entity's internal state must occur through a relatively small interface which abstracts away the details of its implementation

*entity* The smallest unit of data that may be retrieved from a database. In a relational database, a tuple is an entity. In an object-oriented database, an object is an entity. Contrast with values which are associated with *attributes.*

*entity-relationship*
See *relationship.*

*entity-relationship data model*
A data model in which there are entities with attributed properties and relationships. The attributed properties of each entity only contain atomic values. In a pure version, the attributes are all replaced by relationships; each entity only has an identity.

*evolution* See *database evolution.*

*instance* Any object in the collection of objects described by a given class.

*instance variable*
An attribute of an object and its storage. Typically, the set of instance variables for an object is defined by its class. Sometimes, the class also specifies the domain of legal values that each instance variable may assume.

*integrity constraint*
A restriction on the applicability of a database operation so that a given invariant (i.e. predicate) is maintained (i.e. kept true). Frequently, the invariant deals with some form of data consistency.

*interoperability* The ability to transmit data in a form understandable to its destination.

*invariant* A predicate that must be kept true. For example, the class inheritance lattice must contain no cycles is an invariant for the object-oriented data model.

*inverse relationship*
A relationship is typically specified in one direction only, although a true relationship is undirected. An inverse relationship is one specified as the other direction of another specified relationship, and thus must be maintained as such. For example, **parent-of** is the inverse relationship for **child-of**.

*inverse relationship support*
In an object-oriented system, inter-object references reflect uni-directional relationships (e.g. child refers to parent). A database can provide inverse relationship support if two such uni-directional relationships are declared to be inverses of each other (e.g. the parent maintains a set of children references).

*is-a hierarchy* See *generalization hierarchy*. The term comes from the relationship that one class "is a" another; e.g. a car "is a" vehicle.

*is-part-of hierarchy*
See *aggregation hierarchy*. The term comes from the relationship that one object "is part of" another.

*isomorphic embedding*
A property between two types when there exists a one-to-one correspondence between all values of one type and a subset of the values of the second type (e.g. all integers may be floating point values).

*iterator* A language mechanism for enumerating each element of an aggregation value (e.g. set, list, queue, tree) one at a time.

J ---

*join* The construction of virtual database entities from two collections of other entities (either actual or virtual) based on a relationship (usually equality on some attribute) that must hold on each entity of the Cartesian product of the two collections.

K ---

*key* An attribute set of a tuple or object whose associated set of values must be unique within a specified relation or object collection.

**L ---**

*lattice* A directed graph or network in which no cycles occur.

*lazy conversion* The conversion of database instances of a modified class to reflect the changes only when each instance is accessed.

*lock granularity* The amount of data covered by a lock. In object-oriented systems, locks might be placed on a specific instance variable of a single object or a collection of objects, on the entire object, or on a collection of objects. The collection may be specified in special ways; see, for example, *object composition.*

*locking* The most basic pessimistic concurrency control mechanism. A lock establishes some protection so that the owner of the lock can assume that it "owns" the locked data (i.e. no one else can change the data) until it releases the lock. Locks may establish different capabilities for the lock owner: read locks allow only reading, read-write locks allow modifications, etc. See also *lock granularity.*

*logging* The act of recording the current state of all transactions and the updates performed by committed transactions on persistent data. Committed transactions whose updates have been stored may be removed from the log.

*logical pinning* The condition that arises when unreachable objects exist in the database. Logical pinning can happen if a database uses strong identity and an object cycle becomes unreachable from any database root.

**M ---**

*media failure* A failure of the subsystem that stores persistent data during the execution of the database system (e.g. head crash or bus failure).

*message* The name and signature of a method. Since the actual method executed depends on the actual value of an invocation's receiver, the set of all methods that may be executed by an invocation is called a message.

*message send* An invocation of a method on an object. Because of the possibility of overriding, the actual method executed depends on the actual class of the receiver (i.e. the object of interest).

*meta-information*
Information that describes other information. In database systems, meta-information consists of schema or class definitions, indexes, and the directory (i.e. root objects) of entities held by the database. For object-oriented databases, the properties of each attribute (e.g. value domain, composite-ness, key-ness, relationships, and default value) also comprise meta-information.

*method* The implementation of an operation that can manipulate the internal state of an object.

*method resolution*
A rule for determining the actual method to execute for a given message send. In simple imperative systems, the method is exactly the operation whose name is the same and visible in the invocation scope (see *static method resolution*). In object-oriented systems, the method to be executed is the first one defined when searching from the actual class of the receiver to the root of the class generalization hierarchy (see *dynamic method resolution*).

*multi-dimensional sort*
A sort that does not yield a total ordering among the items. Typically, however, there are dimensions along which the items can be totally ordered. For example, a multi-dimensional sort of points in the plane would yield efficient access based on Cartesian location.

*multi-index* A database index that is actually a set of nested indexes, each of which applies to a single link of a given path expression.

*multimedia* The interpretation of computer data as non-standard visual or audio information, e.g. music, raster, graphics, and video.

*multiple inheritance*
Inheritance relationship in which each class may inherit from several other classes.

N ---

*name compatibility*
A form of inheritance in which only implementation is shared.

*name space* A scope in which any given name has a uniquely determined association. In database systems, how name spaces for entities are organized affects the flexibility with which users can build applications.

*negative authorization*
An authorization that indicates the associated user is prohibited from performing the associated operation on the associated object.

*nested expression*
See *path.*

*nested index* A database index that associates all possible start values for each known end value of a given path expression.

*nested transaction*
A transaction that occurs within other transactions, and, when committed or aborted, affects only the surrounding transaction. Thus, if the top-level transaction aborts, no changes occur to the database. See *save-points.*

*nested tuple-set data model*
A data model similar to the network and object-oriented data models in which values may be atomic, tuples, or sets. The primary difference is that object reference cycles are not allowed.

*network* A graph whose links are undirected.

*network data model*
A data model in which database entities (i.e. records of attribute-value pairs) may be related in arbitrary networks (i.e. each entity may have more than one parent). A network is acyclic if an entity may not transitively refer to itself; otherwise, it is cyclic.

*notifier* A trigger whose action is to notify the user when the specified condition occurs within the database. Typically, notifiers are used to inform a user when a specified value in an entity changes. Also called an alerter. See *notifier.*

O ---

*object* Data, when treated as an individual entity. Most often, an object is totally responsible for the manipulation of its internal state; no object may directly alter the internal state of another.

*object composition*
See *composition.*

*object identifier*
A unique name assigned to each object in a database (i.e. unique within that database).

*object identity* The ability to refer from one object to another directly, without requiring any search (at the conceptual level; the implementation may involve search).

*object-oriented*
A characteristic of a system when it treats data as individual entities, called objects. Almost always, object-oriented systems associate procedural semantics with objects (i.e. only the objects themselves can modify their internal state) and some form of inheritance.

*object-oriented data model*
A data model much like the network data model in which database entities (i.e. records of attribute-value pairs) may be typed. In addition, operations reflecting type semantics may be associated with classes of objects. Finally, object types may be organized into a hierarchy or lattice through inheritance.

*optimistic concurrency*
Describes any concurrency control mechanism that allows transactions to proceed regardless and checks for collisions only when transactions are committed. Optimistic concurrency works best when most transactions don't interfere (e.g. read-only transactions) or have a high likelihood of being aborted.

*optional attribute*
An attribute of a tuple or object for which no value need be assigned. In other words, a special "null" value may be associated with such attributes.

*orthogonality* The degree to which one feature is independent of another. In database languages, orthogonality pertains specifically to program values and persistence. That is, full orthogonality is achieved when any manipulable program value may be persistent. Non-orthogonality results when persistent values and transient values must be manipulated in different ways.

*overloading* A static method resolution that allows multiple definitions for the message name in scope of the invocation. Resolution, therefore, must be based on other characteristics, such as the declared type of the receiver.

*override* When a subclass (i.e. the inheriting class) re-defines an attribute (e.g. the domain of an instance variable or the implementation of a method) that would otherwise be inherited from a superclass (i.e. the inherited class). Generally, systems establish rules that delineate whether a re-definition is legal.

*owner* An object that has a composite reference to another object. The term arises from the owner's role in the **is-part-of** relationship.

P ---

*partial ordering* A relationship that relates some proper subset of pairs of items in a collection and transitivity applies (this is not a strictly mathematical definition). In particular, there may exist a pair of items in a collection that are not related by the relationship. For example, multi-dimensional sorts yield a partial ordering, not a total ordering.

*path* A sequence of attributes and a class. The class specifies the head of the path, and each attribute in the sequence describes the next reference to follow in the path. An actual path, therefore, describes a target object when given a source object in the specified class.

*path index* A database index that associates for each end value of a given path expression all values that yield that end value when starting anywhere along the path.

*persistence* The ability to store data between distinct executions of an application.

*pessimistic concurrency*
Describes any concurrency control mechanism that ensures that any transaction that proceeds can commit safely. Pessimistic concurrency assumes that conflicts at commit time occur too often or that aborting transactions wastes too many resources. See *locking*.

*pinning* The act of indicating to a cache manager that the specified buffers must remain in the cache until unpinned.

*polymorphism* The ability to manipulate values of different types simultaneously. In object-oriented systems, polymorphism occurs during the assignment of values to attributes and during method resolution.

*positive authorization*
An authorization that indicates permission for the associated user to perform the associated operation on the associated object.

*predicate* An expression that always evaluates to either true or false.

*presence* A reference to an object by another implies that the object referred to is "present" in the database.

*procedural extensibility*
The ability to extend the operators known to a system. In a database system, procedural extensibility is equivalent to embedded semantics. Object-oriented systems provide procedural extensibility.

*process failure* An abnormal termination of a database application in the middle of a transaction.

*projection* The construction of virtual database entities from a collection of other entities (either actual or virtual) by ignoring certain attributes of the other entities and, perhaps, removing duplicates in the resulting collection.

*property propagation*
Rules concerning the computation of underspecified values. For example, a part can assume the color of the composition of which it is a component. Or, equality tests involving an object may also have to consider one of the objects it references.

*protocol* See *behavior*.

Q ---

*query* A description in some language of the nature of the objects to be retrieved from a database and the domain over which to search.

*query language* The language used to specify a query.

*query optimization*
The process of transforming a database query into an equivalent query that executes faster than the original. This term is actually a misnomer, since there is generally no guarantee that the transformed query has the fastest execution.

*query predicate* See *selection predicate*.

R ---

*radix sort* A sort that takes into account several values associated with each item, giving priority to each value in turn. Alphabetization is just a radix sort on letter positions: sort first on a word's first letter, then the second, and so on.

*reachability* The property that one object can be retrieved only by following direct links from another object, recursively.

*read-only compatibility*
A form of type compatibility that requires values to remain unchanged.

*receiver* The object on which an invoked method should operate. Since the internal state of an object may be manipulated only by its class's methods, the act of invoking a method can be considered as sending a message (i.e. the method's name) to the (receiver) object.

*recovery* The process a database system implements to complete updates of committed transactions to persistent data interrupted by a failure. See also *logging*, *process failure*, *media failure*, and *system failure*.

*referential integrity*
A property of a database indicating that every entity referenced by another exists in the database. Databases that support strong identity must also support referential integrity.

*relation* A set of tuples in a relational database.

*relational data model*
A data model in which database entities (i.e. records of attribute-value pairs) must be organized into sets called relations and all inter-entity references must be by attribute values instead of direct.

*relationship* A semantic connection among a set of entities or objects. A binary relationship is between two objects. If the set is of cardinality n, then the relationship is n-ary. A relationship is undirected; for example, a husband and wife are connected by a spouse relationship. Each entity may also have a cardinality; for example, the mother-children relationship is 1-to-many. The cardinality may hold semantic content; for instance, the relationship between biological parents and a child is 2-to-1.

*role* The function of an object in a system. For example, in an authorization system, users can be organized into a lattice of roles such that an edge between two roles indicates that a source role is also considered as the target role; thus, if an authorization exists for the source role, it implicitly applies to anyone that is a member of the target role as well.

*row* In relational systems, a row is the same as a tuple in a relation. Relations are often compared to tables in which attributes head the columns and tuples comprise the rows.

*rule* A pattern-action pair. A rule is said to be enabled when its pattern is matched be, or unified with, the global data. Once a rule is enabled, it may be "fired" by executing its associated action.

*rule-based paradigm*
A paradigm based on rules in which a set of rules are specified and applied. See *rule*.

S ---

*save-point* A declared moment within a transaction at which point all values must be saved because a later action may choose to "abort" all changes since then. See *nested transaction*.

*schema* The definition of a relation in a relational database. A schema reflects the attribute for each column of tuples in the relation and the domain for associated values. Sometimes, object-oriented systems use the term schema instead of class to describe the definition for object instances.

*schema evolution*

See *database evolution*. Relates specifically to modifying relation schemas in relational databases.

*search* The ability to retrieve entries from the database based on their characteristics (e.g. attribute values) instead of through direct references. See also *associative access* and *direct access*.

*secondary storage management*

The subsystem of a database system that handles the actual persistence of database entities, typically by managing their storage in a file system.

*segment* A unit of clustering.

*selection predicate*

A boolean expression that effectively selects the database items that satisfy a query. That is, each database entity retrieved should satisfy (i.e. make true) the selection predicate.

*self-containment*

A property of a database indicating that all information needed to understand the contained data is within the database. Such a database is considered to be self-describing. Embedded semantics is one method for achieving self-containment.

*semantics* Used to describe the interpretation of the model an object represents. Object semantics is defined by its associated set of methods.

*serializability* A property of transaction support indicating that the effects of a set of concurrent transactions are the same as if the transactions were performed one at a time in some order.

*session control* The management of the activities during the time an application is communicating with a database, including establishing a connection and managing transactions.

*shadowing* Technique used to ensure atomicity. Basically, a copy of the database portion affected by modifications is made, and then the database itself is changed in a single operation to refer to the new version.

*shared reference*

A composite reference to an object indicating that other root objects may also reference that object. See *exclusive reference*.

*sharing* See *data sharing*.

*signature compatibility*

A form of inheritance in which attributes may be extended horizontally by adding new attributes or vertically by constraining existing attributes (i.e. as a syntactic algebra).

*single class index* An index on a collection representing the instances of a class that includes only instances of that class.

*single inheritance*

Inheritance relationship in which each class may inherit from only one other class.

*sound* A rule is *sound* if it represents a valid transformation. A rule set is *sound* if all applicable rule sequences yield valid transformations.

*specialization* The addition of new properties to make a definition more specific. In object-oriented systems, specialization is used to describe the relationship between a subclass and its superclass(es) that it inherits. The inverse relationship is called generalization.

*specific reference* An inter-object reference indicating that the referenced object should be a specific working version. See also *generic reference*.

*stability* A property a database system must support which ensures that persistent data always reflects the results of committed transactions, even in the presence of failures. See also *recovery*, *process failure*, *media failure*, and *system failure*.

*static method resolution* Any mechanism that determines at translation time the method to execute for a message send. See *overloading*.

*static typing* Type checking that occurs during program translation instead of during execution. Static typing can obviate the need for the presence of type information at run-time. See *dynamic typing*.

*storage reclamation* The process by which unused storage is registered as being available. Unused storage arises when objects are deleted from the database or become inaccessible.

*strong identity* The property that any object exists as long as some direct reference exists to it from any other object in the database. See *logical pinning*.

*structure traversal* The ability to change focus from one entity directly to another, as is possible in the network or object-oriented data models.

*structured exception* An exception mechanism that organizes exceptions into a lattice, so that a handler for any ancestor of a raised exception is considered a match.

*subclass* In an inheritance relationship, the class doing the inheriting.

*subset subtyping* A form of type compatibility in which a type is characterized by the set of values it contains. Thus, one type is compatible with another if the values of the first is a subset of the values of the second (e.g. the positive integers are contained in the set of all integers).

*superclass* In an inheritance relationship between two classes, the class being inherited.

*surrogate* An object identifier that does not refer directly to the object's storage but requires the translation of at least one level of indirection.

*system failure* A failure of the database system itself, either because of a bug in its program, the machine it executes on crashes, or the network connection to it dies.

T ---

*table* Another name for relation in the relational data model, since a relation can be viewed as a table in which the columns are labeled by attributes and each row constitutes a tuple or entity.

*telesophy* A term coined by Bruce Schatz meaning "wisdom at a distance".

*termination* A property that states that a computation's use of a resource (usually time, but also storage) is not infinite.

*total ordering* A relationship that ensures that any two items in a collection are related and transitivity applies (this is not a strictly mathematical definition).

*transaction* A set of actions whose overall effect on persistent data should be considered atomic. A transaction may be committed or aborted.

*transaction support*
The mechanism provided by databases that implements transactions.

*transient value* A program value that is not persistent, i.e. a value that is not stored between executions of an applications. See *persistence.*

*transient version*
The current, modifiable version, or state, of an object.

*transitive closure*
The collection of items that results when satisfying a relationship on a universe collection recursively (that is, on every item added to the result collection). For example, the ancestors of a person can be found by finding the transitive closure of the parent-of relationship for that person.

*transparency* The degree to which distinctions between similar but different entities are hidden. In database languages, transparency pertains specifically to how well algorithms can be written that don't distinguish between persistent and transient values. Non-transparency occurs when values must be declared as either persistent or transient.

*trigger* A set of user-defined database actions that occur when a user-defined condition arises within the database. Typically, a trigger changes a value of an entity when a specified value of another entity changes. See also *notifier.*

*tuple* A set of attribute-value pairs. See also *aggregation.*

*Turing-equivalent paradigm*
Any paradigm that can compute anything a Turing machine can; used as a measure of power, since a Turing machine can compute all computable functions.

*two-phase commit*
(1) A method for coordinating multiple commits (first coordinate whether to commit and then decide to commit), or (2) the two phases involved in ensuring that a commit is atomic.

*type* See *domain.*

*type compatibility*
A rule for deciding whether a value computed during execution will be appropriate for an operation that is to be applied to it. The rule must be applied in at least three situations: during assignment, during function invocations, and during parameter evaluation. Normally, in object-oriented systems, a value is type compatible with a type if its actual class inherits the class associated with the type.

*type evolution* See *database evolution.* Relates specifically to modifying object types or classes in object-oriented databases.

*type generator* A parameterized type specification that actually represents a (potentially infinite) family of types, one for each combination of actual types replacing the type parameters.

*type parameterization*
A method for achieving genericity by allowing a subprogram to handle several different types using the same code, typically when implementing a data structure that uses those types (e.g. set-of-<type>).

**U ---**

*undirected graph*
: A graph in which the two ends of each edge are unordered (i.e. neither is the source). See *directed graph*.

*use*
: One class uses another if the first references the second as a domain (e.g. of an instance variable) or references an attribute of the second (e.g. invokes a method/message defined by the second).

**V ---**

*value-based identity*
: The method by which one database entity refers to another by specifying the values of its key attributes and the relation that contains it. Such a reference is called a value-based reference.

*value-based join*
: A join which is based on the equality of the values of a set of attributes in each of the participating collection of entities.

*value-based reference*
: A reference in a database entity to another specified by a set of attribute-value pairs which may be used to search a specific relation for the desired entity.

*version*
: The state of an object at some point in time. See also *transient version* and *working version*.

*vertical extension*
: The restriction of inherited properties by the inheritor. Thus, for example, a subclass can restrict the domain of an instance variable inherited from its superclass.

**W ---**

*weak identity*
: The property that objects may be removed explicitly from a database. See *dangling reference*.

*well-definedness property*
: An invariant that must hold on an instantiation (i.e. actual database) of a data model. For example, an object-oriented database possessing the strong identity property must ensure that every referenced object exists in the database.

*working version*
: A past, immutable version, or state, of an object.

*write-ahead logs*
: Technique used to ensure atomicity. Database changes are recorded in a non-volatile, single, linear stream. The changes are designed so that multiple attempts to incorporate them produce identical results. A pointer into the stream always refers to the boundary between those transactions that have been committed successfully and those yet to be committed completely.

**X ---**

**Y ---**

**Z ---**

# III Index