1. Introduction and Background

Organizations of all sizes are increasingly attempting to employ database technology to address increasingly complex application requirements. In some cases, these requirements involve specialized application domains, such as:

- Computer-Aided Design (CAD)
- Computer-Aided Manufacturing (CAM)
- Geographic Information Systems (GIS)
- Computer-Aided Software Engineering (CASE)
- office automation
- multimedia content development and presentation

In other cases, these requirements involve more conventional business applications, such as billing, customer/employee record keeping, order processing, or maintenance of outside plant, but with extended requirements of various sorts, for example:

- outside plant information extended with geographic or graphic information about sites where equipment or facilities are located

- insurance claim information extended with photographs of property damage involved in the claim

- general business applications that must be integrated into distributed client/server architectures, or that have graphical user interfaces developed using object-oriented techniques

In attempting to apply Database Management Systems (DBMSs) to these applications, it became widely recognized that extensions to conventional DBMS technology were required to address the requirements of these applications. Developing the extended DBMS technology to address these requirements has been an extremely active area of both database research and product development activity. The result of attempts to address these requirements has been the emergence of the Object-Oriented DBMS (OODBMS) and, more recently, its extended relational counterpart, the Object/Relational DBMS (ORDBMS). These classes of products, which this report refers to generally as Object DBMSs (ODBMSs), attempt to provide a combination of the flexibility and extensibility associated with object-oriented programming, together with the facilities expected of a generalized DBMS.

The purpose of this report is to describe the motivation for the extended DBMS technology represented by ODBMSs, to describe representative ODBMS products and related standardization activities, and to discuss the general concepts and facilities of ODBMSs and some of the variants of these that appear in the various products. This report is an update of [Man89a], which surveyed the state of the art of OODBMS technology, and the development of extensions to relational DBMSs, as of 1989. The current report provides updated descriptions of current OODBMS products, and also covers the more recent developments in ORDBMS products, and ODBMS-related standardization activities.

This report does not attempt to be exhaustive in covering all ODBMS products, or in describing the details of any individual products. Moreover, the report does not attempt to "rate" the various systems covered against some standard set of features, or according to some other rating system, in order to choose the "best product". Applications and application contexts differ so widely that it is impossible to do this and be fair to the various products. Potential users of the products should evaluate more detailed information about

each product against the requirements of particular applications in order to make product selections. More is said on this subject in later sections.

The rest of this report is organized as follows. The rest of Section 1 describes the limitations of conventional relational DBMSs that led to the emergence of ODBMS technology. Section 2 describes the basic concepts of object-oriented software, and the general characteristics to be expected in an ODBMS. It also provides examples of how these ODBMS characteristics apply to aspects of advanced applications, and attempts to clarify the distinction made above between OODBMSs and ORDBMSs. (This and subsequent sections also discuss why this distinction may be rather temporary.) Section 3 provides descriptions of a number of representative OODBMS products that illustrate the state of commercial OODBMS technology. Section 4 then provides descriptions of several examples of a new class of database products, the Object/Relational DBMS (ORDBMS). These products represent attempts to extend relational DBMS technology with object capabilities. Section 5 discusses key standardization activities that affect the development of ODBMS technology, including the object extensions to SQL (SQL3) being developed by ANSI and ISO standards committees. Section 6 then provides an overview of the different ways that ODBMS products provide key features, and points out some of the key technical issues represented by the various choices. Section 7 describes a categorization of various classes of ODBMS applications, and discusses specific examples of applications. Finally, Section 8 describes some overall conclusions that can be reached about ODBMSs. The report also contains an extensive reference list.

1.1 Limitations of Conventional Relational DBMSs

Conventionally, a DBMS is associated with a particular data model, and provides an implementation of that model. In particular, a relational DBMS implements the relational data model. The relational model defines

- a single data structuring concept, the *relation* (or *table*), for representing all data in the database
- a set of operations on relations called the *relational algebra* (these operations include, select, project, join, union, etc.)
- a set of constraints, e.g., values of columns in tables must be atomic

Relational DBMSs then provide:

- a set of *data types* for use in columns (e.g., CHAR, DATETIME, FLOAT)
- a language that can be defined in terms of the algebra (SQL)
- an implementation of the language

Figure 1.1 shows the general pattern of application access in a relational DBMS. The application program sends SQL queries to the DBMS. These queries specify qualifications and other manipulations of the various database tables, and result in other tables (in SQL, tables are technically multisets of rows) being returned to the program. The retrieved rows must be mapped to variables or other data structures in the application's programming language. The program operates on these data structures, possibly producing changes to them. If these changes must be reflected in the database, the program formulates SQL update requests to perform the necessary changes, using the application variables or data

structures to prepare the necessary arguments for these update requests. Most of the data semantics reside in the application program, since the ability of the DBMS to represent complex data structures is limited, and the DBMS provides only limited capability to associate particular behavior with the data. (This is so even though many relational DBMSs now support various facilities for storing procedures in the database, and for defining certain types of constraints on database data).

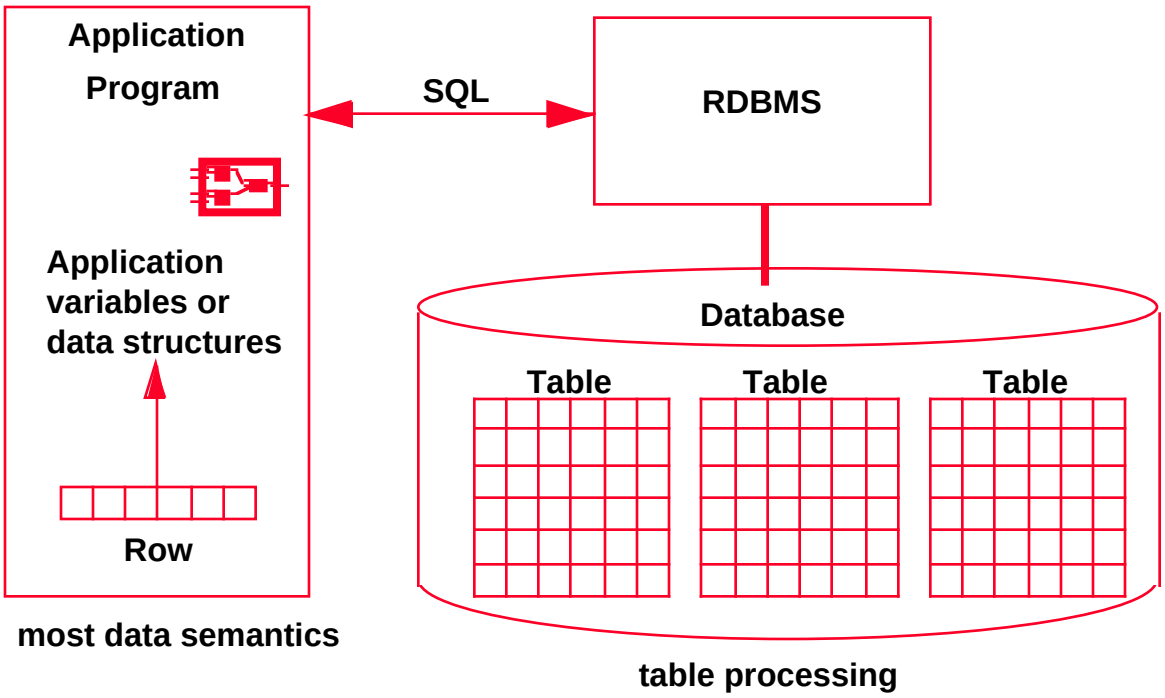


Figure 1.1 Relational DBMS Access

The capabilities of relational DBMSs, together with the pattern of application interaction with them described above, have proved highly effective in dealing with many mainstream business application areas. However, in attempting to apply relational DBMSs to the more advanced applications mentioned earlier, it has become widely recognized that the capabilities of conventional relational DBMSs required extensions to address the additional requirements of these applications.

One area in which extensions are required is in the *data types* supported by DBMSs. The collection of built-in data types in most relational DBMSs (e.g., integer, character string) and built-in operators (e.g., +, -, average) were motivated by the needs of simple business data processing applications. However, in many more advanced applications this collection of types is not appropriate. For example, in a geographic application a user typically wants points, lines, line groups, and polygons as basic data types, and operators which include intersection, distance, and containment. In scientific applications, a user requires complex numbers, arrays, and time series with appropriate operations. Business applications also need specialized data types. [Sto86b] cites as an example a program that computes interest on bonds, and requires all months to have 30 days. Thus, this program would require a data type `DATE` in which the subtraction "April 15" - "March 15" would give 30, rather than 31, days. Other business applications involving even more complex data types have been mentioned earlier. Currently, applications must simulate these data types and

operators using the basic data types and operators provided by the DBMS, resulting in the potential for substantial inefficiency and complexity.

For example, consider the simple example of a relation consisting of data on two dimensional boxes [Sto86b]. If each box has an identifier, then it can be represented by the coordinates of two corner points as follows:

```
CREATE TABLE BOX
   (ID NUMBER(4),
    X1 FLOAT(8),
    X2 FLOAT(8),
    Y1 FLOAT(8),
    Y2 FLOAT(8));
```

Now consider a simple query to find all the (non-rotated) boxes that overlap the unit square, i.e., the box with coordinates (0, 1, 0, 1). This might be expressed in SQL as:

```
SELECT *
FROM BOX
WHERE NOT (X2 <= 0 OR X1 >= 1 OR Y2 <= 0 OR Y1 >= 1);
```

There are two basic problems with this representation. First, the query is hard to formulate and understand (the same query for *rotated* boxes would be even more complex). Second, the query would probably run slowly, both because there are numerous clauses that must be checked, and because the query optimizer would probably not be able to optimize the expression very well.

The problem would be considerably simplified if the DBMS provided a `BOX` data type, together with appropriate operations on that data type, allowing the relation to be defined as:

```
CREATE TABLE BOX
   (ID NUMBER(4),
    DESC BOX);
```

and the above query to be expressed as:

```
SELECT *
FROM BOX
WHERE DESC OVERLAPS '0, 1, 0, 1';
```

The definition of this new data type includes defining both a specialized representation for instances of the type, and also specialized operations that apply to instances of the type (e.g., the `OVERLAPS` operator in the example). The need for such types is more crucial as the individual types and their behavior become more complicated (e.g., images), and less expressible (or not expressible at all) using the DBMS's native capabilities.

The changes required to support the definition of new types are not restricted to those that can be seen by the user. Internal changes to the DBMS may also be required to support such data types. For example, current database systems implement hashing and B-trees as fast access paths for built-in data types. While some enhanced data types (e.g., date and time) might be able to use existing access methods, given certain extensions, other data types, such as polygons (or the `BOX` type above), would require (or, at least, could take

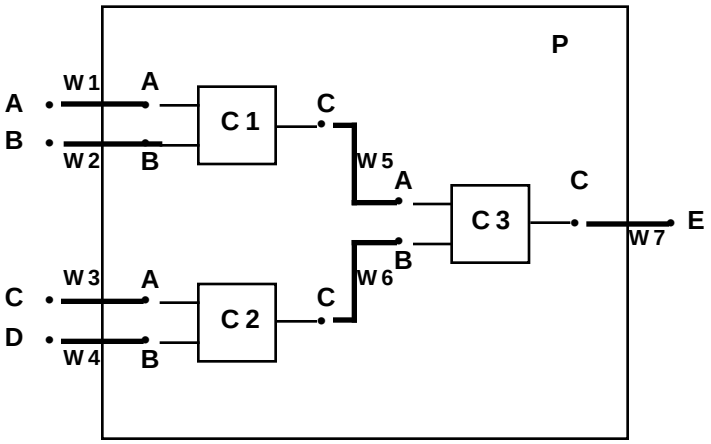
advantage of) entirely new access methods. A number of such access methods appropriate for spatial data have been developed. Specialized access methods for text searching have also been developed, and might be required to provide adequate performance.

One approach that has been taken to address this problem is to define specific extensions for various non-traditional data types, and add them to the DBMS. For example, various special-purpose extensions to DBMSs have been proposed for dealing with text, images, and geometric (e.g., CAD or geographic) data. The difficulty with this approach is that in each case the extensions added are application-specific, and limited in generality. For example, the spatial capabilities required for geographic data would be, at best, of limited use in a CAD application. Moreover, even for a single *category* of data, such as geographic data, there are many different ways to represent and manipulate the data, and each way may be the best in some specific application. It is not really feasible to select one set of data types as built in types, and also provide the required generality. At the same time, it is clearly impossible to build in all useful data types that might be required in any application in the same DBMS. Only a user-defined data type facility allows this type of customization. This would allow the inclusion of customized data types to support the required abstractions, while maintaining generality by allowing additional types to be defined as applications grow or change.

A second area in which extensions are required is in the *structures* supported by DBMSs. Many of the new DBMS applications deal with highly structured objects that are composed of other objects (such objects are frequently referred to as *complex objects* in the DBMS research literature [DMBC+87]). For example, a part in a part hierarchy may be composed of other parts; an integrated circuit module may be composed of other modules, pins and wires; a complex geographic feature such as an industrial park may be composed of other features such as buildings, smokestacks, and gardens; a program module may be composed of other program modules, each with a declaration part and body part; a document may be composed of sections and front matter, and the sections themselves may be composed of section headings, paragraphs of text, and figures.

In many applications these complex objects are the units of storage, retrieval, update, integrity control, concurrency control, and recovery. For instance, in a design application, it may be necessary to lock an entire part assembly (i.e., the part together with all of its component parts) if the part is to be redesigned. Similarly, if an instance of an integrated circuit module is to be deleted from a design, the deletion must be propagated atomically to all its components. The application programs that deal with such complex objects typically represent them as complex graph (pointer-based) structures in main memory, and, when a DBMS is not used, must linearize these structures for storage in conventional file systems.

It has been recognized for some time that such complex objects pose a potential problem for conventional relational DBMSs. This can be illustrated by Figure 1.2, which shows a simple example from [LP83], one of the early papers discussing this problem.



Gate Type	GT	description
	2AND	"C = A&B"
	4AND	"E = A&B&C&D"

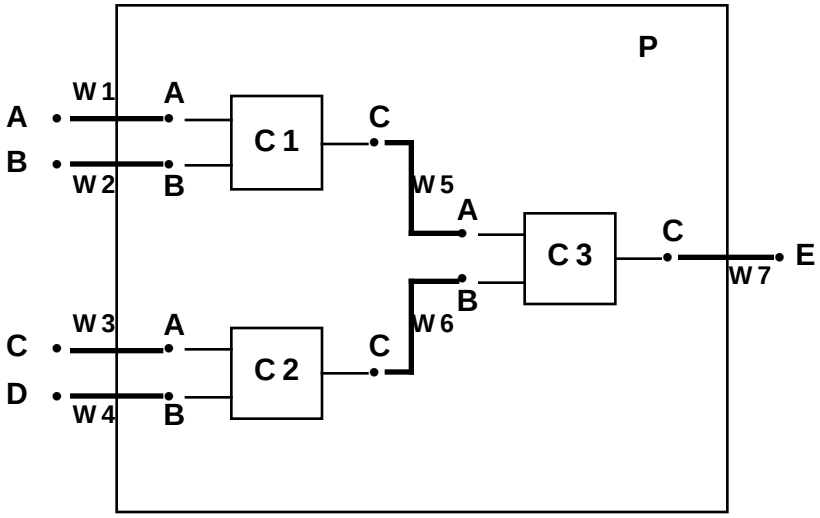
Figure 1.2 Simple Relational Representation of a Complex Object

The figure shows a 4-input AND gate built up of three 2-input AND gates, together with the relation that might be constructed *if no attempt were made to represent detailed information about the gates themselves, or their internal structure, in the database*.

There are two extreme approaches to storing the *details* of such objects in a relational DBMS. The first approach is to completely decompose the object into tuples (rows) so that the entire structure of the object can be directly represented in (and manipulated by) the DBMS. This approach is illustrated in Figure 1.3.

In this case, the `Gate Type` and `Pin Type` relations describe the two types of gates, and the pins for each gate type, respectively. The `Gate Instance` relation indicates that each instance of a 4-input AND gate is built up of three instances of 2-input AND gates. The `Wire Instance` relation shows that each instance of a 4-input AND gate contains seven wires, each connecting a pair of pins. For instance, wire W1 connects the (external) input pin A of the 4-input AND gate to pin A of its first component 2-input AND gate.

There are a number of problems with this approach. As this example illustrates, in order to completely represent the structure of a complex object in a relational DBMS, the object must be broken up into many tuples scattered among several relations. This flattening into tuples of an inherently complex structure may be very unnatural from the user's perspective. Moreover, the manipulation of the object represented in this way may be both complicated and inefficient: First, because there is no way to explicitly specify to the DBMS that all these linked tuples form a single, complex object, operations on the complex object as a whole must typically consist of several relational commands. Second, these relational operations form a transaction that is very difficult to optimize, since the transaction typically involves a large number of joins, and is actually extracting a very few tuples (at most) from each relation, unlike the set-oriented operations for which conventional relational DBMSs are designed. Finally, the application program is again entirely responsible for converting this collection of tuples into the complex graph structure



Gate Type	GT	description
	2AND	"C = A&B"
	4AND	"E = A&B&C&D"

Pin Type	GT	PT	I/O
	2AND	A	I
	2AND	B	I
	2AND	C	O
	4AND	A	I
	4AND	B	I
	4AND	C	I
	4AND	D	I
	4AND	E	O

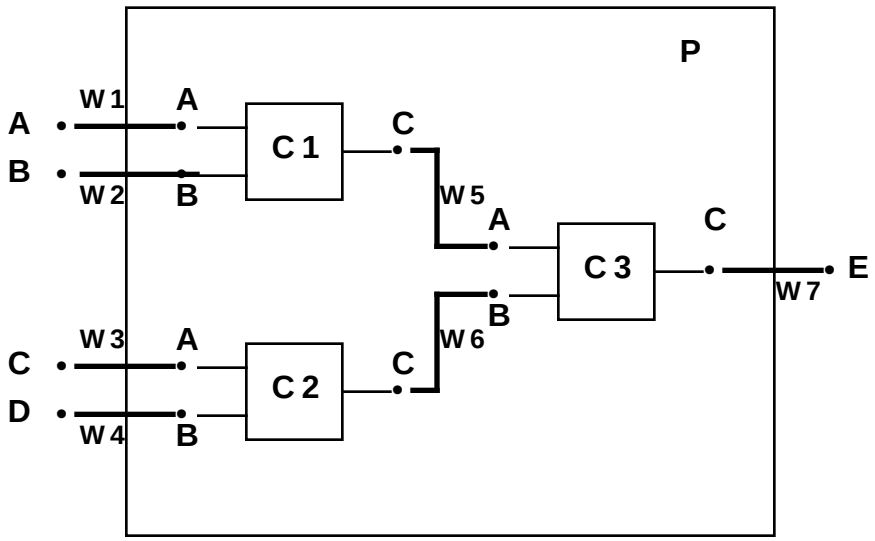
Gate Instance	GT	GI	Parent
	2AND	C1	4AND
	2AND	C2	4AND
	2AND	C3	4AND
	4AND	P	.....

Wire Instance	WI	GT1	GI1	Pin1	GT2	GI2	Pin2	Parent
	W1	4AND	P	A	2AND	C1	A	4AND
	W2	4AND	P	B	2AND	C1	B	4AND
	W3	4AND	P	C	2AND	C2	A	4AND
	W4	4AND	P	D	2AND	C2	B	4AND
	W5	2AND	C1	C	2AND	C3	A	4AND
	W6	2AND	C2	C	2AND	C3	B	4AND
	W7	2AND	C3	C	4AND	P	E	4AND

Figure 1.3 Complete Relational Representation of a Complex Object

that is typically required for internal processing (e.g., for operations that traverse the components of a gate, or follow connections to other gates; performing the database join operations required to simulate such traversals imposes an intolerable performance overhead on realistic applications employing such data types). All this results in considerable performance overhead. Moreover, this example is extremely simple. The situation is, of course, much more complicated when objects of realistic complexity are involved, and database-sized collections of such objects must be managed.

A second approach is to store the entire object as an uninterpreted byte string, often called a *binary* (or *basic*) *large object*, or *BLOB*, in a column of a relation, usually together with some descriptive information in other columns, such as the part number or (here) the object type, that can be used to identify it. This approach is illustrated in Figure 1.4. In the `Gate Type` relation shown in the figure, the values of the `layout` column (depicted in the figure by graphics) would actually be byte strings that, once loaded into an application program's memory space, represent the entire structure of the gate.



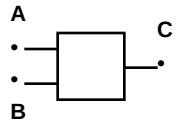
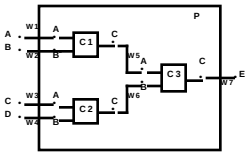
Gate Type	GT	description	layout
	2AND	"C = A&B"	
	4AND	"E = A&B&C&D"	

Figure 1.4 Relational Representation of a Complex Object Using a BLOB

This approach might be perfectly acceptable if all that is required is to display the layout in its entirety (or to display an employee's photograph contained in a similar column in an EMPLOYEE relation). However, in this approach the DBMS cannot be used for any manipulation or retrieval involving the content or structure of the object. For example, the subcomponents of the object cannot be identified by the DBMS, since this information is "buried" within the byte string representing the entire object, and cannot be extracted by the DBMS. Moreover, the DBMS is not able to perform any type checking on this data, and may not be able to perform other useful administrative functions. In effect, the DBMS is being used as a (somewhat complex) file system. Finally, if the main-memory form of the object has a complex, pointer-based internal structure (as is true in many advanced applications), the application program must deal with the problem of converting between this structure and a relocatable database structure that can be reasonably represented as a byte string.

Even in fairly conventional business applications, additional structural capabilities would sometimes be desirable. For example, frequently the most natural structure (and the one that might also yield the best performance) for an application would involve a nested or hierarchical structure, such as:

```
Person Name
     First
     Middle
     Last
Person Address
     Street
     City
     State
     ZIP
```

etc., rather than the requirement to normalize or flatten the structure imposed by a relational DBMS. Similarly, sometimes it would be desirable to define general categories of objects, such as class EMPLOYEE having attributes like NAME, AGE, and SALARY, together with *subclasses* of those objects, such as MANAGER, SUMMER_STAFF, PART_TIME, CONTRACT_EMPLOYEE, etc., with attributes specific to those subclasses, such that the subclasses can be treated as instances of the generic EMPLOYEE class in specific cases (such as printing all employees) without having to introduce redundant data or additional complexity in query formulation (e.g., numerous case statements) or processing. The more complex business applications mentioned at the beginning of this section, e.g., those involving document and multimedia types, impose even more stringent requirements, and can involve complex structures similar to those illustrated in Figures 1.2 through 1.4. Examples of these structures are presented in Section 2, and characteristics of ODBMS applications are discussed further in Section 7.

Approximations of the required capabilities can be constructed by users, using combinations of facilities provided by relational DBMSs such as BLOBs and stored procedures, and possibly object-oriented front-ends. However, this is generally recognized as a stop-gap measure. As the next section will show, the overall trend in the industry is to extend the DBMS itself with object-oriented facilities to provide the required capabilities, resulting in the ODBMSs described in Sections 3 and 4. Such facilities are recognized as being the only real way to providing both the required functionality and performance, even in relational DBMSs (as evidenced by the object extensions being developed for the relational SQL standard described in Section 5).

1.2 A Review of the Problem and Solution Approaches

DBMSs exist to provide persistent storage for program data. *Persistence* is a property of data that determines how long it should remain in existence as far as the computer system is concerned. With most programming languages, data in the program's address space is transient, i.e., it exists only as long as the program executes. Some data, such as locally declared data or procedure parameters, has an even shorter lifetime (the execution of the individual block or procedure).

In a conventional programming language, to make data persist beyond the lifetime of a single program execution, the programmer must include explicit instructions to save the data on stable storage, such as in a file on disk or using a DBMS as an intermediary. Conversely, to reuse data that was created in an earlier execution, or by some other program, the programmer must include explicit instructions to fetch the data from stable storage (or from the DBMS).

The problems illustrated in Section 1.1 are examples of what is sometimes called the *impedance mismatch* between application requirements for handling complex data structures and the DBMS's capabilities to provide persistent support for them. In particular, this impedance mismatch involves differences between:

- application requirements for representing complex data structures (e.g., the complex objects described in Section 1.1) and DBMS capabilities for providing persistent storage for those data structures

- application requirements to access complex data structures in specific ways (e.g., follow pointers in complex graph structures) and DBMS capabilities for supporting those access patterns.

The impedance mismatch between application requirements and conventional DBMS capabilities is exacerbated by the increasing use of object-oriented programming languages, which provide increasing power to define and manipulate complex data structures in the program, without corresponding facilities in the DBMS.

The impedance mismatch problem is being addressed by attempts to provide a more *seamless* interface between programming languages and DBMSs. *Seamlessness* generally describes the situation where the DBMS's data model or type system is a persistent extension of that of one or more host programming languages. There are both ease-of-use and performance issues relating to a lack of seamlessness:

- It is the programmer's responsibility to decide when to save and fetch objects.

- The programmer must write code to translate between the data representations in the program's address space and their external representations on secondary storage (e.g., a file of records or a set of tuples in one or more relations), which may be quite different. This mapping is especially complicated in an object-oriented environment, where a complex object may be composed of many component objects linked together with pointers. The resulting code may make the application harder to understand, and add considerable performance overhead.

- Type-checking also becomes part of the programmer's responsibility. Modern programming languages have elaborate type systems, often with strong type checking. However, when data is written out to stable storage, there may be no type checking provided (if a conventional file system is used) or a different type system may exist (if a conventional DBMS is used). The data can be modified by another program that has access to the file, with no guarantee that the modified object will conform to the original type when it is reread into the address space of the program that created it.

- The capabilities provided by the DBMS to access data may not match the ways in which the application wants to access it. This mismatch may result in poor application performance. This is true even if the DBMS optimizes queries, as noted already.

Conversely, there are significant performance advantages in having a closer correspondence between programming language and persistent (DBMS) types. Basically they involve minimizing the processing required to translate between application program objects and database objects. These have been shown to be crucial in many advanced applications where attempts were made to use a relational DBMS. There are also advantages in program development, maintenance, and reuse. Ideally (depending on the degree of seamlessness obtained), these include:

- Programmers do not have to write the often-complex code required to convert from programming language representations to database representations. The same type system is used in both places, and the system can automatically handle any conversion that is required.

- Programmers can write code that uses persistent storage in the same way in which they write ordinary code, without learning new concepts.

- Code originally written in a non-persistent environment can be imported and used in a database environment without modification (or without significant modification).

In addition, it is increasingly desirable in large-scale database applications for the DBMS to be capable of supporting logically-centralized behavior, such as business rules, and standard procedures associated with certain data types, to ensure that data semantics are maintained and key rules are enforced on all applications. The ability provided by some relational DBMSs to support stored procedures, triggers, and alerters are simple examples of such facilities. However, generalized facilities for specifying user-defined types with their own type-specific behavior, and for specifying behavior in the database in the same way as it is specified in applications, would provide increased support for such large-scale applications.

Previous research on "database programming languages" attempted to address some of these problems by integrating a programming language and a DBMS. These languages allowed certain distinguished programming language data types to be persistent (e.g., the type "relation" in PASCAL/R [Sch77]; "entity types" in ADAPLEX [SFL83]). These languages also incorporated powerful query facilities that allowed the programmer to fetch sets of data objects from the database, in some cases following pointer-based data relationships, in one access. Also, by allowing database operations to be bracketed in transactions, they supported the controlled sharing of concurrently accessed data objects. However, the programmer still saw two type systems (and two address spaces), one for