THIRD-GENERATION DATABASE SYSTEM MANIFESTO

The Committee for Advanced DBMS Function[1]

Abstract

We call the older hierarchical and network systems **first generation** database systems and refer to the current collection of relational systems as the **second generation.** In this paper we consider the characteristics that must be satisfied by the next generation of data managers, which we call **third generation** database systems.

Our requirements are collected into three basis tenets along with 13 more detailed propositions.

1. INTRODUCTION

The network and hierarchical database systems that were prevalent in the 1970's are aptly classified as **first generation** database systems because they were the first systems to offer substantial DBMS function in a unified system with a data definition and data manipulation language for collections of records.[2] CODASYL systems [CODA71] and IMS [DATE86] typify such first generation systems.

In the 1980's first generation systems were largely supplanted by the current collection of relational

---

[1]The committee is composed of Michael Stonebraker of the University of California, Berkeley, Lawrence A. Rowe of the University of California, Berkeley, Bruce Lindsay of IBM Research, James Gray of Tandem Computers, Michael Carey of the University of Wisconsin, Michael Brodie of GTE Laboratories, Philip Bernstein of Digital Equipment Corporation, and David Beech of Oracle Corporation.

To achieve broad exposure this paper is being published in the United States in SIGMOD RECORD and in Europe in the Proceedings of the IFIP TC2 Conference on Object Oriented Databases.

[2]To discuss relational and other systems without confusion, we will use neutral terms in this paper. Therefore, we define a **data element** as an atomic data value that is stored in the database. Every data element has a **data type** (or **type** for short), and data elements can be assembled into a **record** which is a set of one or more named data elements. Lastly, a **collection** is a named set of records, each with the same number and type of data elements.

DBMSs which we term **second generation** database systems. These are widely believed to be a substantial step forward for many applications over first generation systems because of their use of a non-procedural data manipulation language and their provision of a substantial degree of data independence. Second generation systems are typified by DB2, INGRES, NON-STOP SQL, ORACLE and Rdb/VMS.[3]

However, second generation systems were focused on **business data processing** applications, and many researchers have pointed out that they are inadequate for a broader class of applications. Computer aided design (CAD), computer aided software engineering (CASE) and hypertext applications are often singled out as examples that could effectively utilize a different kind of DBMS with specialized capabilities. Consider, for example, a publishing application in which a client wishes to arrange the layout of a newspaper and then print it. This application requires storing text segments, graphics, icons, and the myriad of other kinds of data elements found in most hypertext environments. Supporting such data elements is usually difficult in second generation systems.

However, critics of the relational model fail to realize a crucial fact. Second generation systems do not support **most** business data processing applications all that well. For example, consider an insurance application that processes claims. This application requires traditional data elements such as the name and coverage of each person insured. However, it is desirable to store images of photographs of the event to which a claim is related as well as a facsimile of the original hand-written claim form. Such data elements are also difficult to store in second generation DBMSs. Moreover, all information related to a specific claim is aggregated into a **folder** which contains traditional data, images and perhaps procedural data as well. A folder is often very complex and makes the data elements and aggregates of CAD and CASE systems seem fairly routine by comparison.

Thus, almost everybody requires a better DBMS, and there have been several efforts to construct prototypes with advanced function. Moreover, most current DBMS vendors are working on major functional enhancements of their second generation DBMSs. There is a surprising degree of consensus on the desired

[3]DB2, INGRES, NON-STOP SQL, ORACLE and Rdb/VMS are trademarks respectively of IBM, INGRES Corporation, Tandem, ORACLE Corporation, and Digital Equipment Corporation.

capabilities of these next-generation systems, which we term **third generation** database systems. In this paper, we present the three basic tenets that should guide the development of third generation systems. In addition, we indicate 13 propositions which discuss more detailed requirements for such systems. Our paper should be contrasted with those of [ATKI89, KIM90, ZDON90] which suggest different sets of tenets.

2. THE TENETS OF THIRD-GENERATION DBMSs

The first tenet deals with the definition of third generation DBMSs.

TENET 1: Besides traditional data management services, third generation DBMSs will provide support for richer object structures and rules.

Data management characterizes the things that current relational systems do well, such as processing 100 transactions per second from 1000 on-line terminals and efficiently executing six way joins. Richer object structures characterize the capabilities required to store and manipulate non-traditional data elements such as text and spatial data. In addition, an application designer should be given the capability of specifying a set of **rules** about data elements, records and collections.[4] Referential integrity in a relational context is one simple example of such a rule; however, there are many more complex ones.

We now consider two simple examples that illustrate this tenet. Return to the newspaper application described earlier. It contains many non-traditional data elements such as text, icons, maps, and advertisement copy; hence richer object structures are clearly required. Furthermore, consider the classified advertisements for the paper. Besides the text for the advertisement, there are a collection of business data processing data elements, such as the rate, the number of days the advertisement will run, the classification, the billing address, etc. Any automatic newspaper layout program requires access to this data to decide whether to place any particular advertisement in the current newspaper. Moreover, selling classified

[4] See the previous footnote for definitions of these terms.

advertisements in a large newspaper is a standard transaction processing application which requires traditional data management services. In addition, there are many rules that control the layout of a newspaper. For example, one cannot put an advertisement for Macy's on the same page as an advertisement for Nordstrom. The move toward semi-automatic or automatic layout requires capturing and then enforcing such rules. As a result there is need for rule management in our example application as well.

Consider next our insurance example. As noted earlier, there is the requirement for storing non-traditional data elements such as photographs and claims. Moreover, making changes to the insurance coverage for customers is a standard transaction processing application. In addition, an insurance application requires a large collection of rules such as

> Cancel the coverage of any customer who has had a claim of type Y over value X.
> Escalate any claim that is more than N days old.

We have briefly considered two applications and demonstrated that a DBMS must have data, object and rules services to successfully solve each problem. Although it is certainly possible that niche markets will be available to systems with lesser capabilities, the successful DBMSs of the 90's will have services in all three areas.

We now turn to our second fundamental tenet.

TENET 2: Third generation DBMSs must subsume second generation DBMSs.

Put differently, second generation systems made a major contribution in two areas:

> non-procedural access
> data independence

and these advances must not be compromised by third generation systems.

Some argue that there are applications which **never** wish to run queries because of the simplicity of their DBMS accesses. CAD is often suggested as an example with this characteristic [CHAN89]. Therefore, some suggest that future systems will not require a query language and consequently do not need to subsume second generation systems. Several of the authors of this paper have talked to **numerous** CAD application designers with an interest in databases, and all have specified a query language as a necessity. For example, consider a mechanical CAD system which stores the parts which compose a product such as an automobile. Along with the spatial geometry of each part, a CAD system must store a collection of **attribute** data, such as the cost of the part, the color of the part, the mean time to failure, the supplier of the part, etc. CAD applications require a query language to specify ad-hoc queries on the attribute data such as:

> How much does the cost of my automobile increase if supplier X raises his prices by Y percent?

Consequently, we are led to a query language as an absolute requirement.

The second advance of second generation systems was the notion of data independence. In the area of physical data independence, second generation systems automatically maintain the consistency of **all** access paths to data, and a query optimizer automatically chooses the best way to execute any given user command. In addition, second generation systems provide **views** whereby a user can be insulated from changes to the underlying set of collections stored in the database. These characteristics have dramatically lowered the amount of program maintenance that must be done by applications and should not be abandoned.

Tenet 3 discusses the final philosophical premise which must guide third generation DBMSs.

TENET 3: Third generation DBMSs must be open to other subsystems.

Stated in different terms, any DBMS which expects broad applicability must have a fourth generation language (4GL), various decision support tools, friendly access from many programming languages, friendly access to popular subsystems such as LOTUS 1-2-3, interfaces to business graphics packages, the ability to run the application on a different machine from the database, and a distributed DBMS. All tools and the DBMS must run effectively on a wide variety of hardware platforms and operating systems.

This fact has two implications. First, any successful third generation system must support most of the tools described above. Second, a third generation DBMS must be **open,** i.e. it must allow access from additional tools running in a variety of environments. Moreover, each third generation system must be willing to participate with other third generation DBMSs in future distributed database systems.

These three tenets lead to a variety of more detailed propositions on which we now focus.

3. THE THIRTEEN PROPOSITIONS

There are three groups of detailed propositions which we feel must be followed by the successful third generation database systems of the 1990s. The first group discusses propositions which result from Tenet 1 and refine the requirements of object and rule management. The second group contains a collection of propositions which follow from the requirement that third generation DBMSs subsume second generation ones. Finally, we treat propositions which result from the requirement that a third generation system be open.

3.1. Propositions Concerning Object and Rule Management

DBMSs cannot possibly anticipate all the kinds of data elements that an application might want. Most people think, for example, that time is measured in seconds and days. However, all months have 30 days in bond trading applications, the day ends at 15:30 for most banks, and "yesterday" skips over weekends and holidays for stock market applications. Hence, it is imperative that a third generation DBMS manage a diversity of objects and we have 4 propositions that deal with object management and consider type constructors, inheritance, functions and unique identifiers.

PROPOSITION 1.1: A third generation DBMS must have a rich type system.

All of the following are desirable:

1) an abstract data type system to construct new base types
2) an array type constructor
3) a sequence type constructor
4) a record type constructor
5) a set type constructor
6) functions as a type
7) a union type constructor
8) recursive composition of the above constructors

The first mechanism allows one to construct new base types in addition to the standard integers, floats and character strings available in most systems. These include bit strings, points, lines, complex numbers, etc. The second mechanism allows one to have arrays of data elements, such as found in many scientific applications. Arrays normally have the property that a new element cannot be inserted into the middle of the array and cause all the subsequent members to have their position incremented. In some applications such as the lines of text in a document, one requires this insertion property, and the third type constructor supports such sequences. The fourth mechanism allows one to group data elements into records. Using this type constructor one could form, for example, a record of data items for a person who is one of the "old guard" of a particular university. The fifth mechanism is required to form unordered collections of data elements or records. For example, the set type constructor is required to form the set of all the old guard. We discuss the sixth mechanism, functions (methods) in Proposition 1.3; hence, it is desirable to have a DBMS which naturally stores such constructs. The next mechanism allows one to construct a data element which can take a value from one of several types. Examples of the utility of this construct are presented in [COPE84]. The last mechanism allows type constructors to be recursively composed to support **complex objects** which have internal structure such as documents, spatial geometries, etc. Moreover, there is no requirement that the last type constructor applied be the one which forms sets, as is true for second generation systems.

Besides implementing these type constructors, a DBMS must also extend the underlying query language with appropriate constructs. Consider, for example, the SALESPERSON collection, in which each salesperson has a name and a quota which is an array of 12 integers. In this case, one would like to be able to request the names of salespersons with April quotas over $5000 as follows:

```
select name
from SALESPERSON
where quota[4] > 5000
```

Consequently, the query language must be extended with syntax for addressing into arrays. Prototype syntax for a variety of type constructors is contained in [CARE88].

The utility of these type constructors is well understood by DBMS clients who have data to store with a richer structure. Moreover, such type constructors will also make it easier to implement the persistent programming languages discussed in Proposition 3.2. Furthermore, as time unfolds it is certainly possible that additional type constructors may become desirable. For example, transaction processing systems manage **queues** of messages [BERN90]. Hence, it may be desirable to have a type constructor which forms queues.

Second generation systems have few of these type constructors, and the advocates of Object-oriented Data Bases (OODB) claim that entirely new DBMSs must come into existence to support these features. In this regard, we wish to take strong exception. There are prototypes that demonstrate how to add many of the above type constructors to relational systems. For example, [STON83] shows how to add sequences of records to a relational system, [ZANI83] and [DADA86] indicate how to construct certain complex objects, and [OSBO86, STON86] show how to include an ADT system. We claim that **all** these type constructors can be added to relational systems as natural enhancements and that the technology is relatively well understood.[5] Moreover, commercial relational systems with some of these features have already started to appear.

Our second object management proposition concerns inheritance.

PROPOSITION 1.2: Inheritance is a good idea.

Much has been said about this construct, and we feel we can be very brief. Allowing types to be organized into an inheritance hierarchy is a good idea. Moreover, we feel that multiple inheritance is essential, so the inheritance hierarchy must be a directed graph. If only single inheritance is supported, then we feel that there are too many situations that cannot be adequately modeled. For example, consider a collection of instances of PERSON. There are two specializations of the PERSON type, namely STUDENT and EMPLOYEE. Lastly, there is a STUDENT EMPLOYEE, which should inherit from both STUDENT and

[5]One might argue that a relational system with all these extensions can no longer be considered "relational", but that is not the point. The point is that such extensions are possible and quite natural.

EMPLOYEE. In each collection, data items appropriate to the collection would be specified when the collection was defined and others would be inherited from the parent collections. A diagram of this situation, which demands multiple inheritance, is indicated in Figure 1. While [ATKI89] advocates inheritance, it lists multiple inheritance as an optional feature.

Moreover, it is also desirable to have collections which specify no additional fields. For example, TEENAGER might be a collection having the same data elements as PERSON, but having a restriction on ages. Again, there have been prototype demonstrations on how to add these features to relational systems, and we expect commercial relational systems to move in this direction.

Our third proposition concerns the inclusion of functions in a third generation DBMS.

PROPOSITION 1.3: Functions, including database procedures and methods, and encapsulation are a good idea.

Second generation systems support functions and encapsulation in restricted ways. For example, the operations available for tables in SQL are implemented by the functions **create, alter,** and **drop.** Hence, the

---

```
              __________
             | PERSON  |
             |__________|
              /         |
             /          |
            /           |
  _________/_____     __|__________
 | EMPLOYEE  |       | STUDENT    |
 |_____________|       |___________|
        |                   /
        |                  /
        |                 /
     ___|________________/_____
    | STUDENT EMPLOYEE  |
    |______________________|
```

A Typical Multiple Inheritance Hierarchy
Figure 1

---

table abstraction is only available by executing one of the above functions.

Obviously, the benefits of encapsulation should be made available to application designers so they can associate functions with user collections. For example, the functions HIRE(EMPLOYEE), FIRE(EMPLOYEE) and RAISE-SAL(EMPLOYEE) should be associated with the familiar EMPLOYEE collection. If users are not allowed direct access to the EMPLOYEE collection but are given these functions instead, then all knowledge of the internal structure of the EMPLOYEE collection is encapsulated within these functions.

Encapsulation has administrative advantages by encouraging modularity and by registering functions along with the data they encapsulate. If the EMPLOYEE collection changes in such a way that its previous contents cannot be defined as a view, then all the code which must be changed is localized in one place, and will therefore be easier to change.

Encapsulation often has performance advantages in a protected or distributed system. For example, the function HIRE(EMPLOYEE) may make a number of accesses to the database while executing. If it is specified as a function to be executed internally by the data manager, then only one round trip message between the application and the DBMS is executed. On the other hand, if the function runs in the user program then one round trip message will be executed for each access. Moving functions inside the DBMS has been shown to improve performance on the popular Debit-Credit benchmark [ANON85].

Lastly, such functions can be inherited and possibly overridden down the inheritance hierarchy. Therefore, the function HIRE(EMPLOYEE) can automatically be applied to the STUDENT EMPLOYEE collection. With overriding, the implementation of the function HIRE can be rewritten for the for the STUDENT EMPLOYEE collection. In summary, encapsulated functions have performance and structuring benefits and are highly desirable. However, there are three comments which we must make concerning functions.

First, we feel that users should write functions in a higher level language (HLL) and obtain DBMS access through a high-level non-procedural access language. This language may be available through an embedding via a preprocessor or through direct extension of the HLL itself. Put differently, functions should run queries and not perform their own navigation using calls to some lower level DBMS interface. Proposition 2.1 will discuss the undesirability of constructing user programs with low-level data access interfaces, and the same discussion applies equally to the construction of functions.

There are occasional requirements for a function to directly access internal interfaces of a DBMS. This will require violating our admonition above about only accessing the database through the query language, and an example of such a function is presented in [STON90]. Consequently, direct access to system internals should probably be an allowable but highly discouraged (!) way to write functions.

Our second comment concerns the notion of **opaque** types. Some OODB enthusiasts claim that the only way that a user should be able to access a collection is to execute some function available for the collection. For example, the only way to access the EMPLOYEE collection would be to execute a function such as HIRE(EMPLOYEE). Such a restriction ignores the needs of the query language whose execution engine requires access to each data element directly. Consider, for example:

```
select *
from EMPLOYEE
where salary > 10000
```

To solve this query, the execution engine must have direct access to the salary data elements and any auxiliary access paths (indexes) available for them. Therefore, we believe that a mechanism is required to makes types **transparent,** so that data elements inside them can be accessed through the query language. It is possible that this can be accomplished through an automatically defined "accessor" function for each data element or through some other means. An authorization system is obviously required to control access to the database through the query language.

Our last comment concerns the commercial marketplace. All major vendors of second generation DBMSs **already** support functions coded in a HLL (usually the 4GL supported by the vendor) that can make DBMS calls in SQL. Moreover, such functions can be used to encapsulate accesses to the data they manage. Hence, functions stored in the database with DBMS calls in the query language are already commonplace commercially. The work remaining for the commercial relational vendors to support this proposition is to allow inheritance of functions. Again there have been several prototypes which show that this is a relatively straightforward extension to a relational DBMS. Yet again, we see a clear path by which current relational systems can move towards satisfying this proposition.

Our last object management proposition deals with the automatic assignment of unique identifiers.

PROPOSITION 1.4: Unique Identifiers (UIDs) for records should be assigned by the DBMS only if a user-defined primary key is not available.

Second generation systems support the notion of a **primary key,** which is a user-assigned unique identifier. If a primary key exists for a collection that is known **never** to change, for example social security number, student registration number, or employee number, then no additional system-assigned UID is required. An **immutable** primary key has an extra advantage over a system-assigned unique identifier because it has a natural, human readable meaning. Consequently, in data interchange or debugging this may be an advantage.

If no primary key is available for a collection, then it is imperative that a system-assigned UID be provided. Because SQL supports update through a cursor, second generation systems must be able to update the last record retrieved, and this is only possible if it can be uniquely identified. If no primary key serves this purpose, the system must include an extra UID. Therefore, several second generation systems already obey this proposition.

Moreover, as will be noted in Proposition 2.3, some collections, e.g. views, do not necessarily have system assigned UIDs, so building a system that requires them is likely to be proven undesirable. We close our discussion on Tenet 1 with a final proposition that deals with the notion of rules.

PROPOSITION 1.5: Rules (triggers, constraints) will become a major feature in future systems. They should not be associated with a specific function or collection.

OODB researchers have generally ignored the importance of rules, in spite of the pioneering use of active data values and daemons in some programming languages utilizing object concepts. When questioned about rules, most OODB enthusiasts either are silent or suggest that rules be implemented by including code to support them in one or more functions that operate on a collection. For example, if one has a rule that every employee must earn a smaller salary than his manager, then code appropriate to this constraint would be inserted into both the HIRE(EMPLOYEE) and the RAISE-SAL(EMPLOYEE) functions.

There are two fundamental problems with associating rules with functions. First, whenever a new function is added, such as PENSION-CHANGE(EMPLOYEE), then one must ensure that the function in turn calls RAISE-SAL(EMPLOYEE), or one must include code for the rule in the new function. There is no way to guarantee that a programmer does either; consequently, there is no way to guarantee rule enforcement. Moreover, code for the rule must be placed in at least two functions, HIRE(EMPLOYEE) and RAISE-SAL(EMPLOYEE). This requires duplication of effort and will make changing the rule at some future time more difficult.

Next, consider the following rule:

> Whenever Joe gets a salary adjustment, propagate the change to Sam.

Under the OODB scheme, one must add appropriate code to both the HIRE and the RAISE-SAL functions. Now suppose a second rule is added:

> Whenever Sam gets a salary adjustment, propagate the change to Fred.

This rule will require inserting additional code into the same functions. Moreover, since the two rules interact with each other, the writer of the code for the second rule must understand all the rules that appear in the function he is modifying so he can correctly deal with the interactions. The same problem arises when a rule is subsequently deleted.

Lastly, it would be valuable if users could ask queries about the rules currently being enforced. If they are buried in functions, there is no easy way to do this.

In our opinion there is only one reasonable solution; rules must be enforced by the DBMS but not bound to any function or collection. This has two consequences. First, the OODB paradigm of "everything is expressed as a method" simply does not apply to rules. Second, one cannot directly access any internal interfaces in the DBMS below the rule activation code, which would allow a user to bypass the run time system that wakes up rules at the correct time.

In closing, there are already products from second generation commercial vendors which are faithful to the above proposition. Hence, the commercial relational marketplace is ahead of OODB thinking concerning this particular proposition.

3.2. Propositions Concerning Increasing DBMS Function

We claimed earlier that third generation systems could not take a step backwards, i.e. they must subsume all the capabilities of second generation systems. The capabilities of concern are query languages, the specification of sets of data elements and data independence. We have four propositions in this section that deal with these matters.

PROPOSITION 2.1: Essentially all programatic access to a database should be through a non-procedural, high-level access language.

Much of the OODB literature has underestimated the critical importance of high-level data access languages with expressive power equivalent to a relational query language. For example, [ATKI89] proposes that the DBMS offer an ad hoc query facility in any convenient form. We make a much stronger statement: the expressive power of a query language must be present in every programmatic interface and it should be used for essentially all access to DBMS data. Long term, this service can be provided by adding query language constructs to the multiple persistent programming languages that we discuss further in Proposition 3.2. Short term, this service can be provided by embedding a query language in conventional programming languages.

Second generation systems have demonstrated that dramatically lower program maintenance costs result from using this approach relative to first generation systems. In our opinion, third generation database systems **must not** compromise this advance. By contrast, many OODB researchers state that the applications for which they are designing their systems wish to **navigate** to desired data using a low-level procedural interface. Specifically, they want an interface to a DBMS in which they can access a specific record. One or more data elements in this record would be of type "reference to a record in some other collection" typically represented by some sort of pointer to this other record, e.g an object identifier. Then, the application would dereference one of these pointers to establish a new current record. This process would be repeated until the application had navigated to the desired records.

This navigational point of view is well articulated in the Turing Award presentation by Charles Bachman [BACH73]. We feel that the subsequent 17 years of history has demonstrated that this kind of interface is undesirable and should not be used. Here we summarize only two of the more important problems with navigation. First, when the programmer navigates to desired data in this fashion, he is replacing the function of the query optimizer by hand-coded lower level calls. It has been clearly demonstrated by history that a well-written, well-tuned, optimizer can almost always do better than a programmer can do by hand. Hence, the programmer will produce a program which has inferior performance. Moreover, the programmer must be considerably smarter to code against a more complex lower level interface.

However, the real killer concerns **schema evolution.** If the number of indexes changes or the data is reorganized to be differently clustered, there is no way for the navigation interface to automatically take advantage of such changes. Hence, if the physical access paths to data change, then a programmer must modify his program. On the other hand, a query optimizer simply produces a new plan which is optimized for the new environment. Moreover, if there is a change in the collections that are physically stored, then the support for **views** prevalent in second generation systems can be used to insulate the application from the change. To avoid these problems of schema evolution and required optimization of database access in each program, a user should specify the set of data elements in which he is interested as a query in a non-procedural language.

However, consider a user who is **browsing** the database, i.e. navigating from one record to another. Such a user wishes to see all the records on any path through the database that he explores. Moreover, which path he examines next may depend on the composition of the current record. Such a user is clearly accessing a single record at a time algorithmically. Our position on such users is straight-forward, namely they should run a sequence of queries that return a single record, such as:

```
select *
from collection
where collection.key = value
```

Although there is little room for optimization of such queries, one is still insulated from required program maintenance in the event that the schema changes. One does not obtain this service if a lower level interface is used, such as:

```
dereference (pointer)
```

Moreover, we claim that our approach yields comparable performance to that available from a lower level interface. This perhaps counter-intuitive assertion deserves some explanation. The vast majority of current OODB enthusiasts suggest that a pointer be **soft,** i.e. that its value not change even if the data element that it points to is moved. This characteristic, **location independence,** is desirable because it allows data elements to be moved without compromising the structure of the database. Such data element movement is often inevitable during database reorganization or during crash recovery. Therefore, OODB enthusiasts recommend that location independent unique identifiers be used for pointers. As a result, dereferencing a pointer requires an access to a hashed or indexed structure of unique identifiers.

In the SQL representation, the pair:

```
(relation-name, key)
```

is exactly a location independent unique identifier which entails the same kind of hashed or indexed lookup. Any overhead associated with the SQL syntax will presumably be removed at compile time.

Therefore we claim that there is little, if any, performance benefit to using the lower level interface when a single data element is returned. On the other hand, if multiple data elements are returned then replacing a high level query with multiple lower level calls may degrade performance, because of the cost of those multiple calls from the application to the DBMS.

The last claim that is often asserted by OODB enthusiasts is that programmers, e.g. CAD programmers, **want** to perform their own navigation, and therefore, a system should encourage navigation with a low-level interface. We recognize that certain programmers probably prefer navigation. There were programmers who resisted the move from assembly language to higher level programming languages and others who resisted moving to relational systems because they would have a less complex task to do and therefore a less interesting job. Moreover, they thought they could do a better job than compilers and optimizers. We feel that the arguments against navigation are compelling and that some programmers simply require education.

Therefore, we are led to conclude that essentially all DBMS access should be specified by queries in a non-procedural high-level access notation. In Proposition 3.2 we will discuss issues of integrating such queries with current HLLs. Of course, there are occasional situations with compelling reasons to access lower levels of the DBMS as noted in Proposition 1.3; however, this practice should be strongly discouraged.

We now turn to a second topic for which we believe that a step backwards must also be avoided. Third generation systems will support a variety of type constructors for collections as noted in Proposition 1.1, and our next proposition deals with the specification of such collections, especially collections which are sets.

PROPOSITION 2.2: There should be at least two ways to specify collections, one using enumeration of members and one using the query language to specify membership.

The OODB literature suggests specifying sets by enumerating the members of a set, typically by means of a linked list or array of identifiers for members [DEWI90]. We believe that this specification is generally an inferior choice. To explore our reasoning, consider the following example.

ALUMNI (name, age, address)
GROUPS (g-name, composition)

Here we have a collection of alumni for a particular university along with a collection of groups of alumni. Each group has a name, e.g. old guard, young turks, elders, etc. and the composition field indicates the alumni who are members of each of these groups. It is clearly possible to specify composition as an array of pointers to qualifying alumni. However, this specification will be quite inefficient because the sets in this example are likely to be quite large and have substantial overlap. More seriously, when a new person is added to the ALUMNI collection, it is the responsibility of the application programmer to add the new person to all the appropriate groups. In other words, the various sets of alumni are specified **extensionally** by enumerating their members, and membership in any set is **manually** determined by the application programmer.

On the other hand, it is also possible to represent GROUPS as follows:

GROUPS(g-name, min-age, max-age, composition)

Here, composition is specified **intensionally** by the following SQL expression:

```
select *
from ALUMNI
where age > GROUPS.min-age and age < GROUPS.max-age
```

In this specification, there is one query for each group, parameterized by the age requirement for the group. Not only is this a more compact specification for the various sets, but also it has the advantage that set membership is **automatic.** Hence, whenever a new alumnus is added to the database, he is automatically placed in the appropriate sets. Such sets are guaranteed to be semantically consistent.

Besides assured consistency, there is one further advantage of automatic sets, namely they have a possible performance advantage over manual sets. Suppose the user asks a query such as:

```
select g-name
from GROUPS
where composition.name = "Bill"
```

This query requests the groups in which Bill is a member and uses the "nested dot" notation popularized by GEM [ZANI83] to address into the members of a set. If an array of pointers specification is used for composition, the query optimizer may sequentially scan all records in GROUPS and then dereference each pointer looking for Bill. Alternately, it might look up the identifier for Bill, and then scan all composition fields looking for the identifier. On the other hand, if the intensional representation is used, then the above query can be transformed by the query optimizer into:

```
select g-name
from GROUPS, ALUMNI
where ALUMNI.name = "Bill"
and ALUMNI.age > GROUPS.min-age and ALUMNI.age < GROUPS.max-age
```

If there is an index on GROUPS.min-age or GROUPS.max-age and on ALUMNI.name, this query may substantially outperform either of the previous query plans.

In summary, there are at least two ways to specify collections such as sets, arrays, sequences, etc. They can be specified either **extensionally** through collections of pointers, or intensionally through expressions. Intensional specification maintains **automatic** set membership [CODA71], which is desirable in most applications. Extensional specifications are desirable only when there is no structural connection between the set members or when automatic membership is not desired.

Also with an intensional specification, semantic transformations can be performed by the optimizer, which is then free to use whatever access path is best for a given query, rather than being limited in any way by pointer structures. Hence, physical representation decisions can be delegated to the DBA where they belong. He can decide what access paths to maintain, such as linked lists or pointer arrays [CARE90].

Our point of view is that both representations are required, and that intensional representation should be favored. On the other hand, OODB enthusiasts typically recommend only extensional techniques. It should be pointed out that there was **considerable** attention dedicated in the mid 1970's to the advantages of automatic sets relative to manual sets [CODD74]. In order to avoid a step backwards, third generation systems must favor automatic sets.

Our third proposition in this section concerns **views** and their crucial role in database applications.

PROPOSITION 2.3: Updatable views are essential.

We see very few static databases; rather, most are dynamic and ever changing. In such a scenario, whenever the set of collections changes, then program maintenance may be required. Clearly, the encapsulation of database access into functions and the encapsulation of functions with a single collection is a helpful step. This will allow the functions which must be changed to be easily identified. However, this solution, by itself, is inadequate. If a change is made to the schema it may take weeks or even months to rewrite the affected functions. During this intervening time the database cannot simply be "down". Moreover, if changes occur rapidly, the resources consumed may be unjustifiable.

A clearly better approach is to support **virtual collections** (views). Second generation systems were an advance over first generation systems in part because they provided some support in this area. Unfortunately, it is often not possible to update relational views. Consequently, if a user performs a schema modification and then defines his previous collections as views, application programs which previously ran may or may not continue to do so. Third generation systems will have to do a better job on updatable views.

The traditional way to support view updates is to perform command transformations along the lines of [STON75]. To disambiguate view updates, additional semantic information must be provided by the definer of the view. One approach is to require that each collection be opaque which might become a view at a later time. In this case there is a group of functions through which all accesses to the collection are funneled [ROWE79], and the view definer must perform program maintenance on each of these functions. This will entail substantial program maintenance as well as disallow updates through the query language. Alternately, it has been shown [STON90B] that a suitable rules system can be used to provide the necessary semantics. This approach has the advantage that only one (or a small number) of rules need be specified to provide view update semantics. This will be simpler than changing the code in a collection of functions.

Notice that the members of a virtual collection do not necessarily have a unique identifier because they do not physically exist. Hence, it will be difficult to require that each record in a collection have a unique identifier, as dictated in many current OODB prototypes.

Our last point is that data independence cannot be given up, which requires that all physical details must be hidden from application programmers.

PROPOSITION 2.4: Performance indicators have almost nothing to do with data models and must not appear in them.

In general, the main determiners of performance using either the SQL or lower level specification are:

the amount of performance tuning done on the DBMS
the usage of compilation techniques by the DBMS
the location of the buffer pool (in the client or DBMS address space)

the kind of indexing available
the performance of the client-DBMS interface
and the clustering that is performed.

Such issues have nothing to do with the data model or with the usage of a higher level language like SQL versus a lower level navigational interface. For example, the tactic of clustering related objects together has been highlighted as an important OODB feature. However, this tactic has been used by data base systems for many years, and is a central notion in most IMS access methods. Hence, it is a physical representation issue that has nothing to do with the data model of a DBMS. Similarly, whether or not a system builds indexes on unique identifiers and buffers database records on a client machine or even in user space of an application program are not data model issues.

We have also talked to numerous programmers who are doing non traditional problems such as CAD, and are convinced that they require a DBMS that will support their application which is optimized for their environment. Providing subsecond response time to an engineer adding a line to an engineering drawing may require one or more of the following:

an access method for spatial data such as R-trees, hb-trees or grid files
a buffer pool on the engineer's workstation as opposed to a central server
a buffer pool in his application program
data buffered in screen format rather than DBMS format

These are all performance issues for a workstation/server environment and have nothing to do with the data model or with the presence or absence of a navigational interface.

For a given workload and database, one should attempt to provide the best performance possible. Whether these tactics are a good idea depends on the specific application. Moreover, they are readily available to **any** database system.

3.3. Propositions that Result from the Necessity of an Open System

So far we have been discussing the characteristics of third generation DBMSs. We now turn to the Application Programming Interface (API) through which a user program will communicate with the DBMS. Our first proposition states the obvious.

PROPOSITION 3.1: Third generation DBMSs must be accessible from multiple HLLs.

Some system designers claim that a DBMS should be tightly connected to a particular programming language. For example, they suggest that a function should yield the same result if it is executed in user space on transient data or inside the DBMS on persistent data. The only way this can happen is for the execution model of the DBMS to be identical to that of the specific programming language. We believe that this approach is wrong.

First, there is no agreement on a single HLL. Applications will be coded in a variety of HLLs, and we see no programming language **Esperanto** on the horizon. Consequently, applications will be written in a variety of programming languages, and a **multi-lingual** DBMS results.

However, an **open** DBMS must be multi-lingual for another reason. It must allow access from a variety of externally written application subsystems, e.g. Lotus 1-2-3. Such subsystems will be coded in a variety of programming languages, again requiring multi-lingual DBMS support.

As a result, a third generation DBMS will be accessed by programs written in a variety of languages. This leads to the inevitable conclusion that the type system of the HLL will not necessarily match the type system of the DBMS. Therefore, we are led to our next proposition.

PROPOSITION 3.2: Persistent X for a variety of Xs is a good idea. They will all be supported on top of a single DBMS by compiler extensions and a (more or less) complex run time system.

Second generation systems were interfaced to programming languages using a preprocessor partly because early DBMS developers did not have the cooperation of compiler developers. Moreover, there are certain advantages to keeping some independence between the DBMS language and the programming language, for example the programming language and DBMS can be independently enhanced and tested. However, the resulting interfaces were not very friendly and were characterized as early as 1977 as "like glueing an apple on a pancake". Also, vendors have tended to concentrate on elegant interfaces between their 4GLs and database services. Obviously it is possible to provide the same level of elegance for general purpose programming languages.

First, it is crucial to have a closer match between the type systems, which will be facilitated by Proposition 1.1. This is the main problem with current SQL embeddings, not the aesthetics of the SQL syntax. Second, it would then be nice to allow any variable in a user's program to be optionally **persistent.** In this case, the value of any persistent variable is remembered even after the program terminates. There has been considerable recent interest in such interfaces [LISK82, BUNE86].

In order to perform well, persistent X must maintain a **cache** of data elements and records in the program's address space, and then carefully manage the contents of this cache using some replacement algorithm. Consider a user who declares a persistent data element and then increments it 100 times. With a user space cache, these updates will require small numbers of microseconds. Otherwise, 100 calls across a protected boundary to the DMS will be required, and each one will require milliseconds. Hence, a user space cache will result in a performance improvement of 100 - 1000 for programs with high locality of reference to persistent data. The run time system for persistent X must therefore inspect the cache to see if any persistent element is present and fetch it into the cache if not. Moreover, the run time system must also simulate any types present in X that are not present in the DBMS.

As we noted earlier, functions should be coded by including calls to the DBMS expressed in the query language. Hence, persistent X also requires some way to express queries. Such queries can be expressed in a notation appropriate to the HLL in question, as illustrated for C++ by ODE [AGRA89]. The run-time system for the HLL must accept and process such queries and deliver the results back to the program.

Such a run time system will be more (or less) difficult to build depending on the HLL in question, how much simulation of types is required, and how far the query language available in the HLL deviates from the one available in the DBMS. A suitable run-time system can interface many HLLs to a DBMS. One of us has successfully built persistent CLOS on top of POSTGRES using this approach [ROWE90].

In summary, there will be a variety of persistent X's designed. Each requires compiler modifications unique to the language and a run time system particular to the HLL. All of these run time systems will connect to a common DBMS. The obvious question is "How should queries be expressed?" to this common DBMS. This leads to the next proposition.

PROPOSITION 3.3: For better or worse, SQL is intergalactic dataspeak.

SQL is the universal way of expressing queries today. The early commercial OODB's did not recognize this fact, and had to retrofit an SQL query system into their product. Unfortunately, some products did not manage to survive until they completed the job. Although SQL has a variety of well known minor problems [DATE84], it is necessary for commercial viability. Any OODB which desires to make an impact in the marketplace is likely to find that customers vote with their dollars for SQL. Moreover, SQL is a reasonable candidate for the new functions suggested in this paper, and prototype syntax for several of the capabilities has been explored in [BEEC88, ANSI89]. Of course, additional query languages may be appropriate for specific applications or HLLs.

Our last proposition concerns the architecture which should be followed when the application program is on one machine interfaced to a DBMS on a second server machine. Since DBMS commands will be coded in some extended version of SQL, it is certainly possible to transmit SQL queries and receive the resulting records and/or completion messages. Moreover, a consortium of tool and DBMS vendors, the SQL Access Group, is actively working to define and prototype an SQL remote data access facility. Such a facility will allow convenient interoperability between SQL tools and SQL DBMSs. Alternately, it is possible to communicate between client and server at some lower level interface.

Our last proposition discusses this matter.

PROPOSITION 3.4: Queries and their resulting answers should be the lowest level of communication between a client and a server.

In an environment where a user has a dedicated workstation and is interacting with data at a remote server, there is a question concerning the protocol between the workstation and the server. OODB enthusiasts are debating whether requests should be for single records, single pages or some other mechanism. Our view is very simple: expressions in the query language should be the lowest level unit of communication. Of course, if a collection of queries can be packaged into a function, then the user can use a remote procedure call to cause function execution on the server. This feature is desirable because it will result in less than one message per query.

If a lower level specification is used, such as page or record transfers, then the protocol is fundamentally more difficult to specify because of the increased amount of state, and machine dependencies may creep in. Moreover, any interface at a lower level than that of SQL will be much less efficient as noted in [HAGM86, TAND88]. Therefore, remote procedure calls and SQL queries provide an appropriate level of interface technology.

4. SUMMARY

There are many points upon which we agree with OODB enthusiasts and with [ATKI89]. They include the benefits of a rich type system, functions, inheritance and encapsulation. However, there are many areas where we are in strong disagreement. First, we see [ATKI89] as too narrowly focused on object management issues. By contrast, we address the much larger issue of providing solutions that support data, rule and object management with a complete toolkit, including integration of the DBMS and its query language into a mult-lingual environment. As such, we see the non-SQL, single language systems proposed by many OODB enthusiasts as appealing to a fairly narrow market.

Second, we feel that DBMS access should onlu occur through a query language, and nearly 20 years of history convinces us that this is correct. Physical navigation by a user program and within functions should be avoided. Third, the use of automatic collections whenever possible should be encouraged, as they offer many advantages over explicitly maintained collections. Fourth, persistence may well be added to a variety of programming languages. Because there is no programming language Esperanto, this should be accomplished by changing the compiler and writing a language-specific run-time system to interface to a single DBMS. Therefore, persistent programming languages have little to do with the data model. Fifth, unique identifiers should be either user-defined or system-defined, in contrast to one of the tenets in

[ATKI89].

However, perhaps the most important disagreement we have with much of the OODB community is that we see a natural evolution from current relational DBMSs to ones with the capabilities discussed in this paper. Systems from aggressive relational vendors are faithful to Tenets 1, 2 and 3 and have good support for propositions 1.3, 1.4, 1.5, 2.1, 2.3, 2.4, 3.1, 3.3 and 3.4. To become true third generation systems they must add inheritance, additional type constructors, and implement persistent programming languages. There have been prototype systems which point the way to inclusion of these capabilities.

On the other hand, current systems that claim to be object-oriented generally are not faithful to any of our tenets and support propositions 1.1 (partly), 1.2, 1.3 and 3.2. To become true third generation systems, they must add a query language and query optimizer, a rules system, SQL client/server support, support for views, and persistent programming languages. In addition, they must undo any hard coded requirement for UIDs and discourage navigation. Moreover, they must build 4th generation languages, support distributed databases, and tune their systems to perform efficient data management.

Of course, there are significant research and development challenges to be overcome in satisfying these propositions. The design of a persistent programming language for a variety of existing HLLs presents a unique challenge. The inclusion in such languages of pleasing query language constructs is a further challenge. Moreover, both logical and physical database design are considered challenging for current relational systems, and they will get much more difficult for systems with richer type systems and rules. Database design methodologies and tools will be required to assist users in this area. Optimization of the execution of rules poses a significant challenge. In addition, tools to allow users to visualize and debug rule-oriented applications are crucial to the success of this technology. We encourage the research community to take on these issues.

REFERENCES

[AGRA89] Agrawal, R. and Gehani, G., "ODE: The Language and the Data Model," Proc. 1989 ACM-SIGMOD Conference on Mangement of Data, Portland, Ore. June

1989.

[ANON85] Anon et. al., "A Measure of Transaction Processing Power," Datamation, 1985.

[ANSI89] ANSI-ISO Committee, "Working Draft, Database Languages SQL2 and SQL3," July 1989.

[ATKI89] Atkinson, M. et. al., "The Object-Oriented Database System Manifesto," ALTAIR Technical Report No. 30-89, GIP ALTAIR, LeChesnay, France, Sept. 1989, also in Deductive and Object-oriented Databases, Elsevere Science Publishers, Amsterdam, Netherlands, 1990.

[BACH73] Bachman, C., "The Programmer as Navigator," CACM, November 1973.

[BEEC88] Beech, D., "A Foundation for Evolution from Relational to Object Databases," Proc. Conference on Extending Database Technology, Venice, Italy, April 1988.

[BERN90] Bernstein, P. et. al., "Implementing Recoverable Requests Using Queues", Proc. ACM SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990.

[BUNE86] Buneman, P. and Atkinson, M., "Inheritance and Persistence in Programming Languages," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[CARE88] Carey, M., et. al., "A Data Model and Query Language for EXODUS," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Ill., June 1988.

[CARE90] Carey, M., et al, "An Incremental Join Attachment for Starburst," (in preparation).

[CHAN89] Chang, E. and Katz, R., "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-oriented DBMS," Proc. 1989 ACM-SIGMOD Conference on Management of Data, Portland, Ore., June 1989.

[CODA71] CODASYL Data Base Task Group Report, April 1971.

[CODD74] Codd, E. and Date, C., "Interactive Support for Non-Programmers: The Relational and Network Approaches," Proc. 1974 ACM-SIGMOD Debate, Ann Arbor, Mich., May 1974.

[COPE84] Copeland, G. and Maier, D., "Making Smalltalk a Database System," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[DADA86] Dadam, P. et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, DC, 1986.

[DATE84] Date, C., "A Critique of the SQL Database Language," ACM SIGMOD Record 14(3), November 1984.

[DATE86] Date, C., "An Introduction to Database Systems," Addison-Wesley, Reading, Mass., 1986.

[DEWI90] Dewitt, D. et. al., "A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems," ALTAIR Technical Report 42-90, Le Chesnay, France, January 1990.

[HAGM86] Hagmann, R. and Ferrari, D., "Performance Analysis of Several Back-End Database Architectures," ACM-TODS, March 1986.

[KIM90] Kim, W., "Research Directions in Object-oriented Databases," MCC Technical report ACT-OODS-013-90, MCC, Austin, Tx., January 1990.

[LISK82] Liskov, B. and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust Distributed Programs," Proc. 9th Symposium on the Principles of Programming Languages, January 1982.

[OSBO86] Osborne, S. and Heaven, T., “The Design of a Relational System with Abstract Data Types as Domains,” ACM TODS, Sept. 1986.

[ROWE79] Rowe, L. and Shoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.

[ROWE90] Rowe, Lawrence, "The Design of PICASSO," (in preparation).

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, May 1975.

[STON83] Stonebraker, M., "Document Processing in a Relational Database System," ACM TOOIS, April 1983.

[STON86] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Base Engineering, Los Angeles, Ca., Feb. 1986.

[STON90] Stonebraker, M., et. al., "The Implementation of POSTGRES," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[STON90B] Stonebraker, M. et. al., "On Rules, Procedures, Caching and Views in Data Base Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990.

[TAND88] Tandem Performance Group, "A Benchmark of NonStop SQL on the Debit Credit Transaction," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Ill., June 1988.

[ZANI83] Zaniolo, C., "The Database Language GEM," Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.

[ZDON90] Zdonik, S. and Maier, D., "Fundamentals of Object-oriented Databases," in Readings in Object-oriented Database Systems, Morgan-Kaufman, San mateo, Ca., 1990.