



Database Research Directions -- Four Position Papers:

- (1) Laguna Beach Report
- (2) Object Oriented Database System Manifesto
- (3) Third Generation Database System Manifesto
- (4) Database Systems Achievements and Opportunities

Technical Report 90.10
September 1990
Part Number: 50120



Database Research Directions -- Four Position Papers:

- (1) Laguna Beach Report
- (2) Object Oriented Database System Manifesto
- (3) Third Generation Database System Manifesto
- (4) Database Systems Achievements and Opportunities

Technical Report 90.10
September 1990
Part Number: 50120

Database Research Directions -- Four Position Papers:

1. *Future Directions in DBMS Research*, edited by E. Neuhold and M. Stonebraker, International Computer Science Institute, TR-88-001, May, 1988., Berkeley, CA.
2. *Object-Oriented Database System Manifesto*, M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonick, Altair/INRIA, T.R. 30-89, August, 1989, Le Chesnaym France. also appeared in Proceedings of 1st International Conference on Deductive and Object-Oriented Databases, December, 1989, Japan.
3. *Third Generation Database System Manifesto*, edited by M. Stonebraker, U. C. Berkeley, Electronics Research Lab., M90/28, April, 1990, Berkeley, CA.
4. *Database Systems Achievements and Opportunities -- Report of the NSF Invited Workshop on Future Directions in DBMS Research, Feb. 1990.*, edited by A. Silbershatz, M. Stonebraker, J. Ullman, May 1990. To appear in ACM SIGMOD Record V. 19.3, Dec. 1990.

These reports are republished as Tandem Technical Report 90.10, Part Number 50120, Tandem Computers Inc., Cupertino, CA.

Abstract: This compendium of papers profiles the views of senior database researchers as they debate what to do next, now that the relational model is well established. The papers are collected here as a public service, since there is no other easy way for readers to get them all. None has ever been published in a widely available journal.

It is inappropriate for me to comment here on the papers. To summarize their overall message:

- There is pride in the accomplishments of database research. It has made major contributions both to theory and practice.
- There is unanimity that current database technology, although a great advance over the past, is inadequate for the demands of current and future business and scientific applications. Consequently, considerable research and experimentation is still needed.
- Several research issues recur in each paper: (1) user interfaces and data display, (2) support for complex objects and for data types such as time, image, and geometry, (3) extensible database systems, (4) active databases (rules, constraints, procedures, and triggers), (5) parallel algorithms for dabase processing, (6) algorithms and techniques to store and process very large databases (petabytes), (7) techniques to access heterogeneous databases, and, (8) a generalized transaction model.

Jim Gray, Tandem Computers Inc., September 1990.

Future Directions in DBMS Research

edited by

Erich Neuhold and
Michael Stonebraker

TR-88-001
May, 1988



INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street, Suite 600
Berkeley, California 94704-1105

FUTURE DIRECTIONS IN DBMS RESEARCH

*The Laguna Beach Participants**

Abstract

On February 4-5, 1988, the International Computer Science Institute sponsored a two day workshop at which 16 senior members of the data base research community discussed future research topics in the DBMS area. This paper summarizes the discussion which took place.

1. INTRODUCTION

A computer science research laboratory has been set up at the University of California, Berkeley called the International Computer Science Institute (ICSI). ICSI has been sponsored and has received its initial support through the German National Research Center for Computer Science (Gesellschaft fuer Mathematik und Datenverarbeitung - GMD). In addition to pursuing a research program in fundamental computer science, ICSI has the mandate to promote the interchange of ideas and the collaboration between computer scientists in the US and in other parts of the world. As such, ICSI sponsored a two day workshop in Laguna Beach, California, attended by 7 senior DBMS researchers from Germany and 9 from the USA. This workshop was organized by Erich Neuhold of GMD and Michael Stonebraker of Berkeley.

The purpose of the workshop was to discuss what DBMS topics deserve research attention in the future. During the first day, each participant presented four topics that he was not working on that he thought were important and that he would like to investigate. In addition, each participant was asked to present two topics that others were working on, which he thought were unlikely to yield significant research progress. All participants then cast five votes in support of research topics proposed by others. They were also given two votes to indicate agreement with overrated topics. In addition to the voting, time on the first day was devoted to "war stories", i.e. experiences of the participants in working on very hard real-world data base problems. The second day of the workshop was devoted to 40 minute discussions on a collection of specified controversial topics.

The discussion was far reaching and significant agreement was present on a number of issues. For example, the participants were nearly unanimous on the importance of research on user interfaces, active data bases and parallelism. All other topics received noticeably less support. The participants were unanimously negative on the prospective research contribution of hardware data base machines, general recursive query processing, and interfaces between a DBMS and Prolog. On other issues such as the importance of research into data base tool kits, the participants held divergent positions.

This report summarizes the discussion which took place. All participants contributed to its preparation and have approved its content.

The remainder of the report is organized as follows. Section 2 discusses the applications that the participants thought would drive future requirements of DBMSs. Then, Section 3 considers the hardware environment in which DBMSs will be required to run in the future and discusses the viability of hardware data base machines. In Section 4 the relationship of a DBMS with other system software including the operating system and language processors is treated. Section 5

* The Laguna Beach Participants were Philip A. Bernstein, Umeshwar Dayal, David J. DeWitt, Dieter Gawlick, Jim Gray, Matthias Jarke, Bruce G. Lindsay, Peter C. Lockemann, David Maier, Erich J. Neuhold, Andreas Reuter, Lawrence A. Rowe, Hans J. Schek, Joachim W. Schmidt, Michael Schrefl, and Michael Stonebraker.

turns to mechanisms for DBMS extensions including so-called object-oriented data bases. Active data base management systems are treated in Section 6 along with the attitude of the participants toward rule systems in general. Research in end user interfaces and application development tools is the subject of Section 7. Future research in techniques for implementing single-site DBMSs is considered in Section 8, while Section 9 turns to distributed data base management systems. Lastly, Section 10 consists of a collection of topics which did not fit well into one of the previous sections.

2. FUTURE APPLICATIONS

Several participants suggested investigating the needs of a specific application area as a fruitful research endeavor. Others used new application areas as examples to motivate the discussion of specific research topics or in exploring specific issues. Lastly, many of the war stories concerned new application areas. This section summarizes the themes that emerged.

2.1. CASE

Several participants pointed to Computer Aided Software Engineering (CASE) as a significant new application area for DBMS technology. Basically, all information associated with a computer program (source code, documentation, program design, etc.) should go into a data base. The participants thought that the needs of clients in this area were similar to many other engineering applications (e.g. versions, complex objects, inheritance, etc). Some current commercial systems are using data base management systems to manage CASE data while others are building custom DBMSs with needed capabilities. There was substantial agreement that CASE is an important application areas with meaty problems worthy of investigation. For example, a good way to build a generalized "make" facility (i.e one which would automatically recompile appropriate software modules when changes occur) would be to support a powerful trigger facility. As noted above, research on active data bases received nearly unanimous support from the participants.

2.2. CIM

A smaller number of participants pointed to Computer Integrated Manufacturing (CIM) as a significant new application area. Here, data on all phases of plant operation would be stored in a data base system (e.g. code for machine tools, test results, production schedules, etc.) Participants pointed to new data base capabilities that were needed to support applications in this area such as alerters and triggers. Considerable support for working on the problems in this area also emerged.

2.3. Images

Two of the participants pointed to image applications such as found in medical diagnosis, natural resource management (e.g. satellite data) as an important research area. The discussion pointed out the need for storing large (several megabyte) bit strings of images in a DBMS. Beyond storing large objects, the participants felt there was a substantial pattern recognition problem in this area that should be addressed by others. Nobody thought it was hard to put very large but relatively unstructured objects efficiently into data bases.

2.4. Spatial Data Bases

Several participants used spatial examples as hard applications. These were typically geographical data bases containing encoded information currently found on maps. The problems varied from providing urban services (e.g. how do I get from X to Y efficiently on public transportation) to conducting a military operation to environmental information systems integrating all kinds of data from under, on, and over the earth surface. There was widespread support for the importance of such applications, and the participants generally thought that this was a very good application area for extendible DBMSs.

2.5. IR

One participant thought it was important to investigate the needs of information retrieval applications. Basically, he thought that text would be a component in many data base applications and that the key-word oriented searching prevalent in today's text and information retrieval applications should be reconsidered. The objective would be to efficiently support the needs of clients in this area with semantic and object oriented models so that IR systems would not have to be written as "one-off" applications and the power of generalized DBMSs would come to bear in this field. Some interest for this point of view emerged.

3. THE FUTURE HARDWARE ENVIRONMENT

3.1. The Revolution

Several participants pointed out that there appears to be a "knee of technology" that is occurring. Specifically, individual high end CPUs have increased in speed from about 1 MIP to about 15 MIPS in the last two decades at a consistent rate of about a factor of two every five years. For the next several years RISC technology will allow CPUs to get faster at about a factor of two every 1-2 years. This factor of 2 1/2 - 5 acceleration in the growth rate of CPU technology is likely to have a dramatic impact on the way we think about DBMSs, operating systems and user interfaces. The participants discussed the machine of 1991 as having 50 MIPS and a gigabyte of main memory, selling for \$100,000 or less. Semi-serious comments were bandied about such as "if your data base doesn't fit in main memory, wait a few months and it will."

There was a wide-ranging discussion on whether the application needs of clients would increase at least as fast as the price of resources (CPUs, memory, disks) will be driven down over the next several years. The cost of an application with a constant need for resources will decrease by something approaching a factor of two per year for the next several years as noted above. Clearly, a client with a constant application will solve it with ever cheaper hardware, and performance will become uninteresting sooner or later. In order for data base management systems to continue as a vital research area it is necessary for the needs of clients to increase at least as fast as the hardware is getting cheaper.

There was substantial agreement that DBMS clients have large unmet needs that can take full advantage of cheaper resources. Several participants reported that decision support applications were frequently not being done because they were currently too expensive. Clearly, cheaper hardware will enable this class of applications. Others pointed out that clients justify substantially increased resource consumption on the grounds of human productivity. For example, they would use additional hardware to create real time responses to user interactions rather than in batch. Several participants pointed out that real world data bases are currently growing at 30-50 percent per year and the growth rate shows no signs of decreasing. Hence, space consumption is increasing at about the rate that disks are declining in price. Another participant noted the DBMS functionality would increase by at least the rate of decrease of hardware costs, and therefore performance of the DBMS will continue to be an issue. Lastly, one participant pointed to the scientific community who will consume any amount of available resources to refine the granularity at which they keep information.

The conclusion was that carefully managing resources applied to accessing data would be a significant problem for the foreseeable future. The only impediment to this scenario would be the inability of the DBMS community or their clients to write new software for enhanced function and new applications fast enough.

3.2. Data Base Machines

There was overwhelming sentiment that research on hardware data base machines was unlikely to produce significant results. Some people put it more strongly and said they did not want to read any more papers on hardware filtering or hardware sorting. The basic point they made was that general purpose CPUs are falling in price so fast that one should do DBMS functions in software on a general purpose machine rather than on custom designed hardware. Put differently, nobody was optimistic that custom hardware could compete with general purpose CPUs any time in the near future.

4. THE FUTURE SOFTWARE ENVIRONMENT

4.1. Operating Systems

Operating systems are much maligned by data base researchers for providing the wrong kinds of services, especially in the area of file management. The informed consensus of the participants was that this problem is not likely to go away any time soon. It was felt that operating system designers would have their hands full for the next several years with networking software and substantial kernel modifications caused by the large increase in CPU speed and amount of memory that will be present in future machines. These new communication protocols will include ISO and special purpose protocols such as NFS. In addition, fast restart, non-stop operation, failure management and restructuring the operating system to have a small kernel and a layered collection of services on top of it are likely to consume the attention of the operating system designers for the foreseeable future.

Any hope that OSs will get more responsive to the needs of the DBMS are probably optimistic. The only area where there is hope is in transaction management. Several participants pointed out the need for transactions which spanned subsystems (e.g. two different data managers or the data manager and the network manager). Such needs can only be addressed by transaction support built into the kernel or at least kept outside the DBMS. A minority position was that this class of problems is naturally addressed as a heterogeneous distributed data base system problem. This solution clearly is only acceptable if all need for transaction behaviour can be kept inside the DBMS environment.

Several people pointed out that OS support for data bases was an important research topic. One participant discussed an application which required 10 mbyte per second access to his data. Such performance is only attainable by striping data over a number of disks and reading them in parallel. There was support for additional research on high performance file systems for DBMS applications.

The discussion also focused on whether current operating system interfaces (e.g. MVS) would be preserved. Everybody thought that we were stuck with current interfaces for a long time, just as our clients are stuck with SQL.

4.2. Programming Language Interfaces

There was a discussion of the difficulty of interfacing current DBMSs to current programming languages. Some participants felt that it was important to clean up the current pretty awful SQL interface for both embedded and dynamic queries. Others said that most users would be coding in application development languages (4GLs) and that this level would be hidden, and thereby it would be less important. The consensus was that there would be a lot of programs written in normal programming languages that made calls on data base services, and that a better interface would be a worthwhile area of investigation. Moreover, somebody has to write the 4GL interpreters and compilers and this person should see a cleaner interface.

The possibility of a standard 4GL was briefly discussed. Nobody expressed any interest on working on this essentially political problem. Any 4GL standard is expected to be many years away.

4.3. Prolog

There is considerable current research activity in interfacing Prolog to a DBMS. Many participants pointed to this area as one unlikely to have a significant impact. They felt that too many research groups were doing essentially the same thing. Moreover, most felt that the real problem was to integrate some portions of logic and rules systems into a data manager. As such figuring out efficient "glue" to interface an external rule manager to a DBMS is not very interesting. Many participants expressed strong sentiment to stop funding research projects in this area.

5. EXTENDIBLE DATA MANAGERS

Topics in this area were revisited many times during the two day workshop. The participants pointed out that CASE, CIM and spatial applications need new kinds of objects. Moreover,

there is no agreement in these areas on a small set of primitive object types, and extendible DBMSs seem the only solution to these application needs. The discussion naturally divided into extension architectures and object-oriented data bases, and we treat each area in turn.

5.1. Extension Architectures

There was substantial controversy in this area. Some favored building a complete DBMS containing a parser, query optimizer, access methods, transaction management, etc. and then supporting user extensions within the context of a full-function DBMS. This approach is being taken, for example, in POSTGRES, PROBE, STARBURST, and VODAK. Others favored a tool kit approach where an erector set of modules would be provided to allow a sophisticated application designer to put together a custom system. This approach is being investigated, for example, by DASDB, GENESIS, and EXODUS. The discussion was heated between these options and we summarize some of the points below.

The advocates of the tool kit approach stressed that a sophisticated user could install his own concurrency control or recovery modules from perhaps several that were available in the tool kit. In this way, a tailored transaction manager could be built. They also pointed to application areas like CASE where a "lean and mean" system could be built containing only required capabilities.

The advocates of extendible systems argued that transaction management code is among the hardest to make fast and bug free. Hence, it should be done once by the DBMS super-wizard, and it is unrealistic for even sophisticated users to expect to do better than the super-wizard can do. Moreover, performance knobs on the transaction manager should be installed by the super-wizard if they are needed. The second point argued by the advocates of extendible systems was that most applications seemed to require a full-function system with a query language, optimizer and general purpose run-time system. If a user needs a full-function system, then the tool kit approach is not appropriate.

The participants were about equally divided between the merits of these two points of view. Moreover, one participant pointed out that there is a spectrum of extensions, and that the tool kit and extendible systems are merely at different places in a continuum of possibilities. For example, extendible systems disallow replacing the query language with a different one while allowing the inclusion or replacement of data types. The tool kit approach, on the other hand, offers both kinds of customization. Consequently, the tool kit offers a superset of the extension possibilities of extendible systems.

Clearly, the architecture of extendible systems is a good research area. Moreover, several participants discussed the importance of research in optimizing extendible data base systems. Hence, research on query optimizers and run time systems which do not have hard coded knowledge of the low-level access paths or join algorithms was widely believed to be important. Several participants pointed at the advantage of a distinction between the design environment, e.g. for configuring the extendible DBMS, for evaluating schemas and for turning performance knobs, and the usage environment where the complexities of the extendible systems are compiled and optimized away. Both environments, however, should be reflected in a single architecture and system.

5.2. Object-Oriented Data Bases

There was a long discussion on this topic. Some of the participants thought this whole area was misguided, while others thought it was a highly significant research area. In fact, virtually all participants held a strong opinion one way or the other on this research area. About half thought it held no promise of significant results while the other half thought it was a area where major results could be expected. This seeming contradiction can be explained by the fact that the proponents and detractors invariably discovered they were talking about different capabilities when they used the words "object-oriented data base" (OODB).

Hence, it is clear that the term OODB does not have a common definition to the research community. The participants lamented the fact that there does not seem to be a visible spokesperson who can coerce the researchers in this area into using a common set of terms and defining a common goal that they are hoping to achieve. This situation was contrasted with the

early 1970s when Ted Codd defined the relational model nearly singlehandedly.

The proponents of OODB research made two major points. First, they argued that an OODB was an appropriate framework in which to construct an extendible DBMS. Researchers in this area are constructing systems which allow a user to write his own types and operations using the data language of the OODB itself. One goal of the OODB community is thereby similar to the goal of the extendible data management advocates. However, in these databases extensions are frequently accomplished through writing new data types and operations in the implementation language of the OODB and not in its data language. Some of the OODB proponents, however, pointed out that a OODB should be capable of invoking and linking to procedures written in a variety of programming languages making it again more similar to extendible data managers. In addition it can be argued that the object-oriented paradigm provides a framework where constraints on the defined operations can be enforced and managed. This will avoid all the problems that arise when extendible systems allow uncontrolled operational specifications.

The second major point was that inheritance is considered central to an OODB. A discussion of inheritance centered around the question of why it has to be so complex. It was felt that a good idea was needed to coalesce inheritance into something understandable to an average application programmer. Today different kinds of inheritance are frequently defined only via examples and have to be expressed using the same specification mechanics of the data language. In multi-person systems this leads to all kinds of semantic confusion and handling errors.

The opponents of OODB research made two points. First, they said that many decision support applications are largely built around the paradigm of browsing data collections constructed in ad-hoc ways. Hence, it is important to be able to easily do unstructured joins between disparate objects on any conditions that might be of interest. Systems which do not allow value-based joins between all types of objects are thereby uninteresting for decision support. Consequently, most (but not all) systems currently advertised as OODBs are uninteresting because they do not allow ad-hoc joins.

The second point made by the detractors is that many researchers use the term OODB to mean a storage manager which supports the notion of objects that can have fields which contain identifiers of other objects and which performs the following tasks:

- given an object-identifier, fetch the object instance
- given an object instance, find any field in the object

This level of interface requires a programmer to obtain an object identifier and then "navigate" to other objects by obtaining object identifiers that are fields in the object. This reminded several participants of CODASYL systems, whose severe problems were discussed at great length in the mid 1970s. This class of interfaces was consequently seen by some as a throw back to the 1970s and a misguided effort.

These positions are not really in conflict with each other. Most of the proponents of object-oriented data bases did not defend record-at-a-time navigation. Only a couple of participants felt there were application areas where this interface might be appropriate. All thought that an OODB should have a query language with value-based joins or other means of set-at-a time linking operations along with a substantial data manager to process such queries. The OODB opponents generally thought inheritance was a good idea, and everybody was in favor of extensions.

Hence, the real problem is that "object-oriented" means too many different things. Until the terminology stabilizes and some coherency of system goals emerges, we fear that others will consume much discussion time, as we did, in realizing that we didn't have a common meaning of the terms.

6. ACTIVE DATA BASES AND RULE SYSTEMS

Many participants pointed out the need for so-called active data bases. Hence, triggers, alerters, constraints, etc. should be supported by the DBMS as services. This capability was one of the needs most frequently mentioned as a requirement of future applications of data bases. There was extremely strong consensus that this was a fertile research area and that providing this service would be a major win in future systems.

Several participants noted that current commercial systems are installing procedures as data base objects which can be executed under DBMS control. In many cases such procedures are directly invoked by an application program. It is only marginally more complex to allow such procedures to be invoked as a result of conditions in the data base. Hence, active procedures and active data bases should probably be considered together as a single service. The participants noted a clear need for researchers to take the lead in defining better syntax for procedures, because current commercial systems seem to do a poor job of syntax and semantics in this area.

There was also unanimity that an active data base system should have a fairly simple rules system that would have extremely high performance. Questions normally addressed by AI researchers under the rubric of expert systems, (e.g. implementing a theorem prover to prove safety of a rule or mutual consistency of a rule set) should be completely avoided. One participant pointed out that a DBMS could simply fire rules at will and remember DBMS states as rules were activated. If the run time system ever returned to the same state again, then inconsistency or unsafety was present. In such a case the DBMS need only abort the current transaction to back out of the situation. Simple solutions such as this were universally preferred by the participants to attempts at theorem provers.

There is substantial current research activity in efficiently solving general recursive queries that might be posed by a user or result from activating a recursive rule. The participants felt that there are a number of clients with parts explosion or other transitive closure queries. There are also a number of clients with linear recursive queries such as shortest path problems in graphs. However, none of the participants had seen an application that was best solved by coding a general rule set and then optimizing it. As such, it is not clear that there is a client for solutions to general recursive queries. The overwhelming sentiment of the majority of participants is that they did not want to see any more papers in this area. Moreover, they complained that current conferences were accepting too many papers on recursive query processing.

An analogy was drawn to dependency theory which was explored at length a few years ago primarily by the theoretical research community. Most participants felt that little of practical significance had been contributed by this lengthy research beyond the original introduction of the theory, and that general recursive query processing would meet the same fate.

One participant noted that knowledge acquisition was a central and difficult problem in rule based applications. The sentiment of the workshop was that this task is extremely difficult. A small number of participants felt that significant advances would result from research in this area.

7. END USER INTERFACES

The participants noted that there are virtually no researchers investigating better end user interfaces to data bases or better database application development tools. In the last few years there have been only a scattering of papers published in this area. Moreover, there was universal consensus that this was an extremely important area, and it received more support from the participants than any other area. In addition, several participants noted that major advances have been made in non-database related user interfaces in the last decade. They noted that spreadsheets, WYSIWYG interfaces, Hypercard, etc. have caused a complete revolution in human interfaces.

The discussion turned to the obvious question "Why has major progress been made in a vitally important area with no participation from the database research community and what can be done to change the situation?" Several points emerged from the discussion. First, it was noted that publishing papers on user interfaces is inherently difficult. As one participant noted "The best user interface requires no manual." Hence, how can one write papers that capture the essence of such interfaces? A second point was that the academic community (especially program committees) is hostile to user interface papers because they typically have no equations or "hard intellectual" content. A third point was that user interface contributions must be built to assess their merits. One participant advocated "demo or die" to prove an idea. However, to build real systems there is a mammoth amount of low level support code which must be constructed before the actual interface can be built. This "infrastructure" has only recently become available in the form of toolkits such as X11.

The participants noted that because of the last point it is now easier to work on user interfaces and there may be more work in this area in the future. Almost one-third of them said they

were working on end-user interfaces of one sort or another. A constructive comment which emerged from the discussion was that international meetings such as SIGMOD and VLDB should set up a video track, through which researchers could contribute an idea in video rather than paper form. Such a stream with its own program committee would clearly be oriented to user interface contributions and has been used successfully by recent OOPSLA conferences. Collectively, we would strongly encourage the organizers of such meetings to implement this suggestion.

8. SINGLE SITE DBMS TECHNOLOGY

The undertone of discussions on this topic was that technology is in the process of changing the rules. Contributing factors are the imminent arrival of increasingly high performance cheap processors sometimes in shared memory configurations, gigabytes of main memory, and large arrays of 3 1/2" or 5 1/4" drives. This technology will require rethinking the query optimizer, execution techniques and run time system for the new environment.

A second point made by several participants was the need for high availability and much better failure management. Nobody thought that problems in this area would be easy because significant results might entail considerable low level "nuts and bolts" engineering.

There was widespread consensus that no more concurrency control papers should be written for the traditional database transaction model. Most participants felt that papers in this area tended to be "epsilon variants" on previous work, and future research on algorithms in this area should be discouraged. On the other hand, there was great enthusiasm for work on new transaction models. The standard notion of transactions as serializable and atomic has not changed in 15 years. There have been occasional wrinkles such as nested transactions and sagas. However, the sense of the meeting was that new application areas such as CASE and CIM would benefit from a more comprehensive transaction model. Such a model might encompass

- 1) check-in check-out protocols with version control,
- 2) weaker semantics than serializability or
- 3) transactions which can be **UNDONE** after committing

There was no consensus on what the model would look like, just that it was needed. The participants were vocal in their support for research in this area.

Strong support was also voiced for research on parallelism. Basically, this amounts to running multiple query plans for a single query language command. This tactic is valuable in a single CPU system in order to keep multiple disks busy. In a shared memory system it is desirable to utilize the multiple processors that are present, and obviously it is a central idea in tightly coupled distributed systems. The big benefit from research in this area is the ability to give a user the illusion of real time response as well as the ability to hold locks for shorter periods of time. Next to user interface research and active data bases, this topic received the most support.

There was some interest in research on tricks to achieve better performance. Work on removing hot spots, caching the answers to queries, and precomputed joins of one sort or another were mentioned as interesting areas. Each achieved some interest but was considered a limited scope idea, (i.e. something that a few people should probably look into).

Lastly, there were participants who were very hostile to access method research. They suggested that they had seen enough variants on B-trees and extendible hashing and were not interested in further papers in this area. The sentiment was that the recent papers are exploiting the same collection of themes, and that only a 10-20 percent increase in performance is likely to result from such ideas. A couple of participants noted that several commercial vendors have rewritten their data base systems in recent years and have not installed extendible hashing or enhanced B-tree schemes. The apparent reason is that they do not see the cost-benefit of writing code in this area. However, some participants noted that research on new access methods to support new classes of objects (e.g. spatial ones) is needed.

9. DISTRIBUTED DBMS

The participants felt that distributed data base management systems had been investigated at length by the research community over the last ten years. Now there is intense commercial

activity in this area, and several vendors are hard at work on heterogeneous (or federated) distributed DBMSs. Moreover, the really hard problems in this area center around system administration of a large (say 50,000 terminal) distributed computing system. Managing this environment (e.g. installing new updates of modules, new users) is a major nightmare on current systems. It was felt that researchers probably have little to contribute to the solution to this problem because one must have experience with the needs in this area in order to make a contribution. Few researchers are in a position to closely study large complex systems of this sort.

The one area of distributed data bases that evoked support was the problem of scale. Clearly, data bases will be set up which involve hundreds or even thousands of nodes. CIM environments will have large numbers of nodes as well as environments where workstation data bases are present. There was support for rethinking algorithms for query processing, copies, transaction management, and crash recovery from the point of view of scalability to large numbers of nodes. However, detractors thought this might be a waste of time because commercial companies will make progress in this area faster than the research community.

10. MISCELLANEOUS TOPICS

10.1. Physical Data Base Design

There was a consensus that researchers should build automatic physical data base design tools that would choose a physical schema and then monitor the performance of the schema making changes as necessary. This would include adding and dropping indexes, load balancing arm activity across a substantial number of disk arms, etc. Hence, tuning knobs should be removed from the domain of the data base administrator and manipulated by a system demon. There was widespread support for this idea, and most people thought it was only a moderately hard problem.

10.2. Design Tools

Several participants noted that current data base design tools are merely graphical drawing systems. Moreover, there are a large variety of such products commercially available at the present time. As such, they have little research appeal. A couple of participants expressed interest in enhanced design tools that include methodological concepts for objects, operations and constraints.

10.3. Real Time Data Bases

The needs of applications which must run in real time was discussed at some length. Monitoring applications typically input a continuous stream of data into a data base. Often they discard data once it is no longer timely. Many times there are deadlines to deal with (i.e the DBMS must respond by the time the part goes by the robot arm on the assembly line). The notion of a transaction in this environment is unclear, and the need for triggers is apparent. There was support for research in this area to look at this class of issues.

10.4. Data Models

There was no support for any more data models. The participants universally felt we had enough already. There was also little support for an attempt to standardize on a common "next generation" data model. This was felt to be an exercise in frustration as well as merely the invention of yet another data model. It was, however, pointed out that OODB's and active data-bases in a way provide their own data model. In contrast to traditional models these data models will not be fixed but dynamically changeable through the definitional capabilities of the respective data languages.

10.5. Data Translation

The problem of data translation in a heterogeneous computing environment was raised by one participant. This area was discussed and most people believed it to be a solved research problem. The literature of the early 1970s contains appropriate architectures and real world data translation systems have been deployed in the field for quite a number of years.

10.6. Information Exchange Via Data Bases

A couple of participants raised the problem of information exchange and collaborative work. They argued that there are many application areas where the different end-user participants must utilize various tools which should run on a common representation. In CAD, simulation, and document applications, the absence of a common representation is seen as a severe drawback. There was support for information interchange via a data base in these areas in preference to the current techniques as this would provide automatically features like synchronisation, concurrency, recovery, versioning, and security. This approach would also naturally lead to more sophisticated data dictionary systems represented through the schemas of those DBMSs.

**The Object-Oriented Database System
Manifesto**

**Malcolm Atkinson, François Bancilhon
David DeWitt, Klaus Dittrich
David Maier, Stanley Zdonik**

**Rapport Technique *Altair* 30-89
21 août 1989**



**GIP ALTIR
IN 2 - INRIA - LRI**

Domaine de Voluceau BP 105 - Rocquencourt 78153 Le Chesnay Cedex
Tel. 39 63 54 17 Telex 697 033 F

The Object-Oriented Database System Manifesto

Malcolm Atkinson
University of Glasgow

François Bancilhon
Altaïr

David DeWitt
University of Wisconsin

Klaus Dittrich
University of Zurich

David Maier
Oregon Graduate Center

Stanley Zdonik
Brown University

August 19, 1989

Abstract

This paper attempts to define an *object-oriented database system*. It describes the main features and characteristics that a system must have to qualify as an object-oriented database system.

We have separated these characteristics into three groups:

- *Mandatory*, the ones the system must satisfy in order to be termed an object-oriented database system. These are complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility, computational completeness, persistence, secondary storage management, concurrency, recovery and an *ad hoc* query facility.
- *Optional*, the ones that can be added to make the system better, but which are not mandatory. These are multiple inheritance, type checking and inferencing, distribution, design transactions and versions.
- *Open*, the points where the designer can make a number of choices. These are the programming paradigm, the representation system, the type system, and uniformity.

We have taken a position, not so much expecting it to be the final word as to erect a provisional landmark to orient further debate.

1 Introduction

Currently, object-oriented database systems (OODBS) are receiving a lot of attention from both experimental and theoretical standpoints, and there has been considerable debate about the definition of such systems.

Three points characterize the field at this stage: (i) the lack of a common data model, (ii) the lack of formal foundations and (iii) strong experimental activity.

Whereas Codd's original paper [Codd 70] gave a clear specification of a relational database system (data model and query language), no such specification exists for object-oriented database systems [Maier 89]. Opinion is slowly converging on the gross characteristics of a family of object-oriented systems, but, at present, there is no clear consensus on what an object-oriented system is, let alone an object-oriented database system.

The second characteristic of the field is the lack of a strong theoretical framework. To compare object-oriented programming to logic programming, there is no equivalent of [Van Emdem and Kowalski 76]. The need for a solid underlying theory is obvious: the semantics of concepts such as types or programs are often ill defined. The absence of a solid theoretical framework, makes consensus on the data model almost impossible to achieve.

Finally, a lot of experimental work is underway: people are actually building systems. Some systems are just prototypes [Bancilhon *et al.* 88], [Nixon, *et al.* 87], [Banerjee *et al.* 87], [Skarra *et al.* 86], [Fishman *et al.* 87], [Carey *et al.* 86], but some are commercial products, [Atwood 85], [Maier, *et al.* 84], [Caruso and Sciore 87], [G-Base 88]. The interest in object-oriented databases seems to be driven by the needs of design support systems (e.g., CAD, CASE, Office Information Systems). These applications require databases that can handle very complex data, that can evolve gracefully, and that can provide the high-performance dictated by interactive systems.

The implementation situation is analogous to relational database systems in the mid-seventies (though there are more start-ups in the object-oriented case). For relational systems, even though there were some disagreements on a few specific points, such as the form of the query language, or whether relations should be sets or bags, these distinctions were in most cases superficial and there was a common underlying model. People were mainly developing implementation technology. Today, we are simultaneously choosing the specification of the system and producing the technology to support its implementation.

Thus, with respect to the specification of the system, we are taking a Darwinian approach: we hope that, out of the set of experimental prototypes being built, a fit model will emerge. We also hope that viable implementation technology for that model will evolve simultaneously.

Unfortunately, with the flurry of experimentation, we risk a system emerging as *the* system, not because it is the fittest, but because it is the first one to provide a significant subset of the functionality demanded by the market. It is a classical, and unfortunate, pattern of the computer field that an early product becomes the *de facto* standard and never disappears. This pattern is true at least for languages and operating systems (Fortran, Lisp, Cobol and SQL are good examples of such situations). Note however, that our goal here is not to standardize languages, but to refine terminology.

It is important to agree now on a definition of an object-oriented database systems. As a first step towards this goal, this paper suggests characteristics that such systems should possess. We expect that the paper will be used as a straw man, and that others will either invalidate or confirm the points mentioned here.

We have separated the characteristics of object-oriented database systems into three

categories: *mandatory* (the ones that the system must satisfy to deserve the label), *optional* (the ones that can be added to make the system better but which are not mandatory) and *open* (the places where the designer can be select from a number of equally acceptable solutions). In addition, there is some leeway how to best formulate each characteristic (mandatory as well as optional).

The rest of this paper is organized as follows. Section 3 describes the mandatory features of an OODBS. Section 4 describes its optional features and Section 5 presents the degrees of freedom left to the system designers.

2 Mandatory features: the Golden Rules

An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an *ad hoc* query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness.

2.1 Complex objects

Thou shalt support complex objects

Complex objects are built from simpler ones by applying constructors to them. The simplest objects are objects such as integers, characters, byte strings of any length, booleans and floats (one might add other atomic types). There are various complex object constructors: tuples, sets, bags, lists, and arrays are examples. The minimal set of constructors that the system should have are set, list and tuple. *Sets* are critical because they are a natural way of representing collections from the real world. *Tuples* are critical because they are a natural way of representing properties of an entity. Of course, both sets and tuples are important because they gained wide acceptance as object constructors through the relational model. *Lists* or *arrays* are important because they capture order, which occurs in the real world, and they also arise in many scientific applications, where people need matrices or time series data.

The object constructors must be orthogonal: any constructor should apply to any object. The constructors of the relational model are not orthogonal, because the set construct can only be applied to tuples and the tuple constructor can only be applied to atomic values. Other examples are non-first normal form relational models in which the top level construct must always be a relation.

Note that supporting complex objects also requires that appropriate operators must be provided for dealing with such objects (whatever their composition) as a whole. That is, operations on a complex object must propagate transitively to all its components. Examples include the retrieval or deletion of an entire complex object or the production of a "deep" copy (in contrast to a "shallow" copy where components are not replicated,

but are instead referenced by the copy of the object root only). Additional operations on complex objects may be defined, of course, by users of the system (see the extensibility rule below). However, this capability requires some system provided provisions such as two distinguishable types of references ("is-part-of" and "general").

2.2 Object identity

Thou shalt support object identity

Object identity has long existed in programming languages. The concept is more recent in databases, e.g., [Hall *et al.* 76], [Maier and Price 84], [Khoshafian and Copeland 86]. The idea is the following: in a model with object identity, an object has an existence which is independent of its value. Thus two notions of object equivalence exist: two objects can be identical (they are the same object) or they can be equal (they have the same value). This has two implications: one is object sharing and the other one is object updates.

Object sharing: in an identity-based model, two objects can share a component. Thus, the pictorial representation of a complex object is a graph, while it is limited to be a tree in a system without object identity. Consider the following example: a Person has a name, an age and a set of children. Assume Peter and Susan both have a 15-year-old child named John. In real life, two situations may arise: Susan and Peter are parent of the same child or there are two children involved. In a system without identity, Peter is represented by:

`(peter, 40, {(john, 15, {})})`

and Susan is represented by:

`(susan, 41, {(john, 15, {})})`.

Thus, there is no way of expressing whether Peter and Susan are the parents of the same child. In an identity-based model, these two structures can share the common part `(john, 15, {})` or not, thus capturing either situations.

Object updates: assume that Peter and Susan are indeed parents of a child named John. In this case, all updates to Susan's son will be applied to the object John and, consequently, also to Peter's son. In a value-based system, both sub-objects must be updated separately. Object identity is also a powerful data manipulation primitive that can be the basis of set, tuple and recursive complex object manipulation, [Abiteboul and Kanellakis 89].

Supporting object identity implies offering operations such as object assignment, object copy (both deep and shallow copy) and tests for object identity and object equality (both deep and shallow equality).

Of course, one can simulate object identity in a value-based system by introducing explicit object identifiers. However, this approach places the burden on the user to insure the uniqueness of object identifiers and to maintain referential integrity (and this burden can be significant for operations such as garbage collection).

Note that identity-based models are the norm in imperative programming languages: each object manipulated in a program has an identity and can be updated. This identity

either comes from the name of a variable or from a physical location in memory. But the concept is quite new in pure relational systems, where relations are value-based.

2.3 Encapsulation

Thou shalt encapsulate thine objects

The idea of encapsulation comes from (i) the need to cleanly distinguish between the specification and the implementation of an operation and (ii) the need for modularity. Modularity is necessary to structure complex applications designed and implemented by a team of programmers. It is also necessary as a tool for protection and authorization.

There are two views of encapsulation: the programming language view (which is the original view since the concept originated there) and the database adaptation of that view.

The idea of encapsulation in programming languages comes from abstract data types. In this view, an object has an interface part and an implementation part. The interface part is the specification of the set of operations that can be performed on the object. It is the only visible part of the object. The implementation part has a data part and a procedural part. The data part is the representation or state of the object and the procedure part describes, in some programming language, the implementation of each operation.

The database translation of the principle is that an object encapsulates both program and data. In the database world, it is not clear whether the structural part of the type is or is not part of the interface (this depends on the system), while in the programming language world, the data structure is clearly part of the implementation and not of the interface.

Consider, for instance, an Employee. In a relational system, an employee is represented by some tuple. It is queried using a relational language and, later, an application programmer writes programs to update this record such as to raise an Employee's salary or to fire an Employee. These are generally either written in a imperative programming language with embedded DML statements or in a fourth generation language and are stored in a traditional file system and not in the database. Thus, in this approach, there is a sharp distinction between program and data, and between the query language (for *ad hoc* queries) and the programming language (for application programs).

In an object-oriented system, we define the Employee as an object that has a data part (probably very similar to the record that was defined for the relational system) and an operation part, which consists of the *raise* and *fire* operations and other operations to access the Employee data. When storing a set of Employees, both the data and the operations are stored in the database.

Thus, there is a single model for data and operations, and information can be hidden. No operations, outside those specified in the interface, can be performed. This restriction holds for both update and retrieval operations.

Encapsulation provides a form of "logical data independence": we can change the implementation of a type without changing any of the programs using that type. Thus, the application programs are protected from implementation changes in the lower layers of the system.

We believe that proper encapsulation is obtained when only the operations are visible and the data and the implementation of the operations are hidden in the objects.

However, there are cases where encapsulation is not needed, and the use of the system can be significantly simplified if the system allows encapsulation to be violated under certain conditions. For example, with ad-hoc queries the need for encapsulation is reduced since issues such as maintainability are not important. Thus, an encapsulation mechanism must be provided by an OODBS, but there appear to be cases where its enforcement is not appropriate.

2.4 Types and Classes

Thou shalt support types or classes

This issue is touchy: there are two main categories of object-oriented systems, those supporting the notion of class and those supporting the notion of type. In the first category, are systems such as Smalltalk [Goldberg and Robson 83], Gemstone [Maier, *et al.* 84], Vision [Caruso and Sciore 87], and more generally all the systems of the Smalltalk family, Orion [Banerjee *et al.* 87], Flavors [Bobrow and Steifik 81], G-Base [G-Base 88], Lore [Caseau 89] and more generally all the systems derived from Lisp. In the second category, are systems such as C++ [Stroustrup 86], Simula [Simula 67], Trellis/Owl [Schaffert, *et al.* 86], Vbase [Atwood 85] and O_2 [Bancilhon *et al.* 88].

A *type*, in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. It corresponds to the notion of an abstract data type. It has two parts: the interface and the implementation (or implementations). Only the interface part is visible to the users of the type, the implementation of the object is seen only by the type designer. The interface consists of a list of operations together with their signatures (i.e., the type of the input parameters and the type of the result).

The type implementation consists of a data part and an operation part. In the data part, one describes the internal structure of the object's data. Depending on the power of the system, the structure of this data part can be more or less complex. The operation part consists of procedures which implement the operations of the interface part.

In programming languages, types are tools to increase programmer productivity, by insuring program correctness. By forcing the user to declare the types of the variables and expressions he/she manipulates, the system reasons about the correctness of programs based on this typing information. If the type system is designed carefully, the system can do the type checking at compile-time, otherwise some of it might have to be deferred at compile time. Thus types are mainly used *at compile time* to check the correctness of the programs. In general, in type-based systems, a type is not a first class citizen and has a special status and cannot be modified at run-time.

The notion of *class* is different from that of type. Its specification is the same as that of a type, but it is more of a run-time notion. It contains two aspects: an object factory and an object warehouse. The object factory can be used to create new objects, by performing the operation *new* on the class, or by cloning some prototype object representative of the class. The object warehouse means that attached to the class is its extension, i.e., the set of

objects that are instances of the class. The user can manipulate the warehouse by applying operations on all elements of the class. Classes are not used for checking the correctness of a program but rather to create and manipulate objects. In most systems that employ the class mechanism, classes are first class citizens and, as such, can be manipulated at run-time, i.e., updated or passed as parameters. In most cases, while providing the system with increased flexibility and uniformity, this renders compile-time type checking impossible.

Of course, there are strong similarities between classes and types, the names have been used with both meanings and the differences can be subtle in some systems.

We do not feel that we should choose one of these two approaches and we consider the choice between the two should be left to the designer of the system (see Section 4.3). We require, however, that the system should offer some form of data structuring mechanism, be it classes or types. Thus the classical notion of database schema will be replaced by that of a set of classes or a set of types.

We do not, however, feel that is necessary for the system to automatically maintain the extent of a type (i.e., the set of objects of a given type in the database) or, if the extent of a type is maintained, for the system to make it accessible to the user. Consider, for example, the *rectangle* type, which can be used in many databases by multiple users. It does not make sense to talk about the set of all rectangles maintained by the system or to perform operations on them. We think it is more realistic to ask each user to maintain and manipulate its own set of rectangles. On the other hand, in the case of a type such as *employee*, it might be nice for the system to automatically maintain the employee extent.

2.5 Class or Type Hierarchies

Thine classes or types shalt inherit from their ancestors

Inheritance has two advantages: it is a powerful modeling tool, because it gives a concise and precise description of the world and it helps in factoring out shared specifications and implementations in applications.

An example will help illustrate the interest in having the system provide an inheritance mechanism. Assume that we have Employees and Students. Each Employee has a name, an age above 18 and a salary, he or she can die, get married and be paid (how dull is the life of the Employee!). Each Student has an age, a name and a set of grades. He or she can die, get married and have his or her GPA computed.

In a relational system, the data base designer defines a relation for Employee, a relation for Student, writes the code for the *die*, *marry* and *pay* operations on the Employee relation, and writes the code for the *die*, *marry* and *GPA computation* for the Student relation. Thus, the application programmer writes six programs.

In an object-oriented system, using the inheritance property, we recognize that Employees and Students are Persons; thus, they have something in common (the fact of being a Person), and they also have something specific. We introduce a type Person, which has attributes *name* and *age* and we write the operations *die* and *marry* for this type. Then, we declare that Employees are special types of Persons, who inherit attributes and operations, and have a special attribute *salary* and a special operation *pay*. Similarly, we declare that

a Student is a special kind of Person, with a specific *set-of-grades* attribute and a special operation *GPA computation*. In this case, we have a better structured and more concise description of the schema (we factored out specification) and we have only written four programs (we factored out implementation). Inheritance helps code reusability, because every program is at the level at which the largest number of objects can share it.

There are at least four types of inheritance: *substitution* inheritance, *inclusion* inheritance, *constraint* inheritance and *specialization* inheritance.

In substitution inheritance, we say that a type *t* inherits from a type *t'*, if we can perform more operations on objects of type *t* than on object of type *t'*. Thus, any place where we can have an object of type *t'*, we can substitute for it an object of type *t*. This kind of inheritance is based on behavior and not on values.

Inclusion inheritance corresponds to the notion of classification. It states that *t* is subtype of *t'*, if every object of type *t* is also an object of type *t'*. This type of inheritance is based on structure and not on operations. An example is a *square* type with methods *get*, *set(size)* and *filled-square*, with methods *get*, *set(size)*, and *fill(color)*.

Constraint inheritance is a subcase of inclusion inheritance. A type *t* is a subtype of a type *t'*, if it consists of all objects of type *t* which satisfy a given constraint. An example of such a inheritance is that *teenager* is a subclass of *person*: teenagers don't have any more fields or operations than persons but they obey more specific constraints (their age is restricted to be between 13 and 19).

With specialization inheritance, a type *t* is a subtype of a type *t'*, if objects of type *t* are objects of type *t'* which contains more specific information. Examples of such are persons and employees where the information on employees is that of persons together with some extra fields.

Various degrees of these four types of inheritance are provided by existing systems and prototypes, and we do not prescribe a specific style of inheritance.

2.6 Overriding, overloading and late binding

Thou shalt not bind prematurely

In contrast to the previous example, there are cases where one wants to have the same name used for different operations. Consider, for example, the *display* operation: it takes an object as input and displays it on the screen. Depending on the type of the object, we want to use different display mechanisms. If the object is a picture, we want it to appear on the screen. If the object is a person, we want some form of a tuple printed. Finally, if the object is a graph, we will want its graphical representation. Consider now the problem of displaying a set, the type of whose members is unknown at compile time.

In an application using a conventional system, we have three operations: *display-person*, *display-bitmap* and *display-graph*. The programmer will test the type of each object in the set and use the corresponding display operation. This forces the programmer, to be aware of all the possible types of the objects in the set, to be aware of the associated display operation, and to use it accordingly.

for *x* in *X* do

```

begin
  case of type(x)
    person: display(x);
    bitmap: display-bitmap(x);
    graph: display-graph(x);
  end
end

```

In an object-oriented system, we define the display operation at the object type level (the most general type in the system). Thus, display has a single name and can be used indifferently on graphs, persons and pictures. However, we *redefine* the implementation of the operation for each of the types according to the type (this redefinition is called *overriding*). This results in a single name (display) denoting three different programs (this is called *overloading*). To display the set of elements, we simply apply the display operations to each one of them, and let the system pick the appropriate implementation at run-time.

```

for x in X do display(x)

```

Here, we gain a different advantage: the type implementors still write the same number of programs. But the application programmer does not have to worry about three different programs. In addition, the code is simpler as there is no case statement on types. Finally, the code is more maintainable as when a new type is introduced as new instance of the type are added, the display program will continue to work without modification. (provided that we override the display method for that new type).

In order to provide this new functionality, the system cannot bind operation names to programs at compile time. Therefore, operation names must be resolved (translated into program addresses) at run-time. This delayed translation is called *late binding*.

Note that, even though late binding makes type checking more difficult (and in some cases impossible), it does not preclude it completely.

2.7 Computational completeness

Thou shalt be computationally complete

From a programming language point of view, this property is obvious: it simply means that one can express any computable function, using the DML of the database system. From a database point of view this is a novelty, since SQL for instance is not complete.

We are not advocating here that designers of object-oriented database systems design new programming languages: computational completeness can be introduced through a reasonable connection to existing programming languages. Most systems indeed use an existing programming language [Banerjee *et al.* 87], [Fishman *et al.* 87], [Atwood 85], [Bancilhon *et al.* 88]; see [Bancilhon and Maier 88] for a discussion of this problem.

Note that this is different from being “resource complete”, i.e., being able to access all resources of the system (e.g. screen and remote communication) from within the language.

Therefore, the system, even though computationally complete might not be able to express a complete application. It is, however, more powerful than a database system which only stores and retrieves data and performs simple computations on atomic values.

2.8 Extensibility

Thou shalt be extensible

The database system comes with a set of predefined types. These types can be used at will by programmers to write their applications. This set of type must be extensible in the following sense: there is a means to define new types and there is *no distinction in usage between system defined and user defined types*. Of course, there might be a strong difference in the way system and user defined types are *supported* by the system, but this should be invisible to the application and to the application programmer. Recall that this type definition includes the definition of operations on the types. Note that the encapsulation requirement implies that there will be a mechanism for defining new types. This requirement strengthens that capability by saying that newly created types must have the same status as existing ones.

However, we do not require that the collection of type constructors (tuples, sets, lists, etc.) be extensible.

2.9 Persistence

Thou shalt remember thy data

This requirement is evident from a database point of view, but a novelty from a programming language point of view, [Atkinson *et al.* 83]. Persistence is the ability of the programmer to have her/his data survive the execution of a process, in order to eventually reuse it in another process. Persistence should be *orthogonal*, i.e., each object, independent of its type, is allowed to become persistent as such (i.e., without explicit translation). It should also be implicit: the user should not have to explicitly move or copy data to make it persistent.

2.10 Secondary storage management

Thou shalt manage very large databases

Secondary storage management is a classical feature of database management systems. It is usually supported through a set of mechanisms. These include index management, data clustering, data buffering, access path selection and query optimization.

None of these is visible to the user: they are simply performance features. However, they are so critical in terms of performance that their absence will keep the system from performing some tasks (simply because they take too much time). The important point is that they be invisible. The application programmer should not have to write code to

maintain indices, to allocate disk storage, or to move data between disk and main memory. Thus, there should be a clear independence between the logical and the physical level of the system.

2.11 Concurrency

Thou shalt accept concurrent users

With respect to the management of multiple users concurrently interacting with the system, the system should offer the same level of service as current database systems provide. It should therefore insure harmonious coexistence among users working simultaneously on the database. The system should therefore support the standard notion of atomicity of a sequence of operations and of controlled sharing. Serializability of operations should at least be offered, although less strict alternatives may be offered.

2.12 Recovery

Thou shalt recover from hardware and software failures

Here again, the system should provide the same level of service as current database systems. Therefore, in case of hardware or software failures, the system should recover, i.e., bring itself back to some coherent state of the data. Hardware failures include both processor and disk failures.

2.13 Ad Hoc Query Facility

Thou shalt have a simple way of querying data

The main problem here is to *provide* the functionality of an *ad hoc* query language. We do not require that it be done in the form of a query language but just that the service be provided. For instance, a graphical browser could be sufficient to fulfill this functionality. The service consists of allowing the user to ask simple queries to the database simply. The obvious yardstick is of course relational systems, thus the test is to take a number of representative relational queries and to check whether they can be stated with the same amount of work. Note that this facility could be supported by the data manipulation language or a subset of it.

We believe that a query facility should satisfy the following three criteria: (i) It should be *high level*, i.e., one should be able to express (in a few words or in a few mouse clicks) non-trivial queries concisely. This implies that it should be reasonably declarative, i.e., it should emphasize the *what* and not the *how*. (ii) It should be efficient. That is, the formulation of the queries should lend itself to some form of query optimization. (iii) It should be application independent, i.e., it should work on any possible database. This last requirements eliminates specific querying facilities which are application dependent, or require writing additional operations on each user-defined type.

2.14 Summary

This concludes the list of mandatory features and the distinction between traditional and object-oriented database systems should be clear. Relational database systems do not satisfy rules 1 through 8. CODASYL database systems partially satisfy rules 1 and 2. Some people have argued that object-oriented database systems are nothing more than CODASYL systems. It should be noted that (i) CODASYL systems do not completely satisfy these two rules (the object constructors are not orthogonal and object identity is not treated uniformly since relationships are restricted to be 1:n), and (ii) they do not satisfy rules 3, 5, 6, 8 and 13.

There is a collection of features for which the authors have not reached consensus on whether they should be required or optional. These features are:

- view definition and derived data;
- database administration utilities;
- integrity constraints;
- schema evolution facility.

3 Optional features: the goodies

We put under this heading things which clearly improve the system, but which are not mandatory to make it an object-oriented database system.

Some of these features are of an object oriented nature (e.g. multiple inheritance). They are included in this category because, even though they make the system *more* object-oriented, they do not belong in the core requirements.

Other features are simply database features (e.g. design transaction management). These characteristics usually improve the functionality of a data base system, but they are not in the core requirement of database systems and they are unrelated to the object oriented aspect. In fact most of them are targeted at serving “new” applications (CAD/CAM, CASE, Office automation, etc.) and are more application oriented than technology oriented. Because many object-oriented database systems are currently aiming at these new applications, there has been some confusion between these features and the object-oriented nature of the system.

3.1 Multiple inheritance

Whether the system provides multiple inheritance or not is an option. Since agreement on multiple inheritance in the object-oriented community has not yet been reached, we consider providing it to be optional. Note that once one decides to support multiple inheritance, there are many possible solutions for dealing with the problem of conflict resolution.

3.2 Type checking and type inferencing

The degree of type checking that the system will perform at compile time is left open but the more the better. The optimal situation is the one where a program which was accepted by the compiler cannot produce any run-time type errors. The amount of type inferencing is also left open to the system designer: the more the better, the ideal situation is the one in which only the base types have to be declared and the system infers the temporary types.

3.3 Distribution

It should be clear that this characteristic is orthogonal to the object-oriented nature of the system. Thus, the database system can be distributed or not.

3.4 Design transactions

In most new applications, the transaction model of classical business oriented database system is not satisfactory: transactions tend to be very long and the usual serializability criterion is not adequate. Thus, many OODBSs support design transactions (long transactions or nested transactions).

3.5 Versions

Most of the new applications (CAD/CAM and CASE) involve a design activity and require some form of versioning. Thus, many OODBSs support versions. Once again, providing a versioning mechanism this is not part of the core requirements for the system.

4 Open choices

Every system which satisfies rules 1 through 13 deserves the OODBS label. When designing such a system, there are still a lot of design choices to be made. These are the degrees of freedom for the OODBS implementors. These characteristics differ from the mandatory ones in the sense that no consensus has yet been reached by the scientific community concerning them. They also differ from the optional features in that we do not know which of the alternatives are more or less object-oriented.

4.1 Programming paradigm

We see no reason why we should impose one programming paradigm more than another: the logic programming style [Bancilhon 86], [Zaniolo 86], the functional programming style [Albano *et al.* 1986], [Banerjee *et al.* 87], or the imperative programming style [Stroustrup 86], [Eiffel 87], [Atwood 85] could all be chosen as programming paradigms. Another solution is that the system be independent of the programming style and support multiple programming paradigms [Skarra *et al.* 86], [Bancilhon *et al.* 88].

Of course, the choice of the syntax is also free and people will argue forever whether one should write "john hire" or "john.hire" or "hire john" or "hire(john)".

4.2 Representation system

The representation system is defined by the set of atomic types and the set of constructors. Even though we gave a minimal set of atomic types and constructors (elementary types from programming languages, and set, tuple and list constructors) available for describing the representation of objects, can be extended in many different ways.

4.3 Type system

There is also freedom with respect to the type formers. The only type formation facility we require is encapsulation. There can be other type formers such as generic types or type generator (such as $\text{set}[T]$, where T can be an arbitrary type), restriction, union and arrow (functions).

Another option is whether the type system is second order. Finally, the type system for variables might be richer than the type system for objects.

4.4 Uniformity

There is a heated debate on the degree of uniformity one should expect of such systems: is a type an object? is a method an object? or should these three notions be treated differently? We can view this problem at three different levels: the implementation level, the programming language level and the interface level.

At the implementation level, one must decide whether type information should be stored as objects, or whether an ad hoc system must be implemented. This is the same issue relational database systems designers have to face when they must decide whether to store the schema or in some ad hoc fashion. The decision should be made based on performance and ease of implementation grounds. Whatever, decision is made is, however, independent from the one taken at the next level up.

At the programming language level, the question is the following: are types first class entities in the semantics of the language. Most of the debate is concentrated on this question. There are probably different styles of uniformity (syntactical or semantical). Full uniformity at this level is also inconsistent with static type checking.

Finally, at the interface level, another independent decision must be made. One might want to present the user with a uniform view of types, objects, and methods, even if in the semantics of the programming language, these are notions of a different nature. Conversely, one could present them as different entities, even though the programming language views them as the same thing. That decision must be made based on human factor criteria.

5 Conclusions

Several other authors, [Kim 88] and [Dittrich 1986] argue that an OODBS is a DBMS with an underlying object-oriented data model. If one takes the notion of a data model in a broad sense that especially includes the additional aspects going beyond record-orientation, this view is certainly in accordance with our approach. [Dittrich 1986] and [Dittrich 1988] introduce a classification of object-oriented data models (and, consequently, of OODBS): if it supports complex objects, a model is called structurally object-oriented; if extensibility is provided, it is called behaviorally object-oriented; a fully object-oriented model has to offer both features. This definition also requires persistence, disk management, concurrency, and recovery; it at least implicitly assumes most of the other features (where applicable, according to the various classes); in total, it is thus slightly more liberal than our approach. However, as most current systems and prototypes do not fulfill all requirements mandated by our definition anyway, this classification provides a useful framework to compare both existing and ongoing work.

We have proposed a collection of defining characteristics for an object-oriented database system. To the best of our knowledge, the golden rules presented in this paper are currently the most detailed definition of an object-oriented database system. The choice of the characteristics and our interpretation of them devolves from our experience in specifying and implementing the current round of systems. Further experience with the design, implementation, and formalization of object-oriented databases will undoubtedly modify and refine our stance (in other words, don't be surprised if you hear one of the authors lambasting the current definition in the future). Our goal is only to put forth a concrete proposal to be debated, critiqued and analyzed by the scientific community. Thus, our last rule is:

Thou shalt question the golden rules

6 Acknowledgements

We wish to thank Philippe Bridon, Gilbert Harrus, Paris Kanellakis, Philippe Richard, and Fernando Velez for suggestions and comments on earlier drafts of the paper.

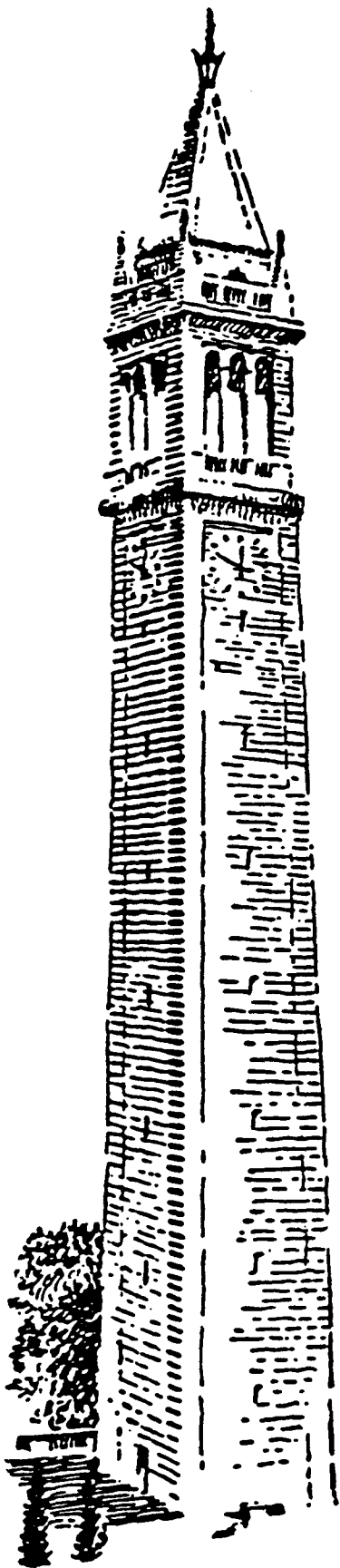
References

- [Abiteboul and Kanellakis 89] S. Abiteboul and P. Kanellakis, "Object identity as a query language primitive", *Proceedings of the 1989 ACM SIGMOD*, Portland, Oregon, June 89
- [Albano et al. 1986] A. Albano, G. Gheli, G. Occhiuto and R. Orsini, "Galileo: a strongly typed interactive conceptual language", *ACM TODS*, Vol 10, No. 2, June 1985.

- [Atkinson *et al.* 83] M. Atkinson, P.J. Bayley, K. Chilsom, W. Cockshott and R. Morrison, "An approach to persistent programming", *Computer Journal*, 26(4), 1983, pp 360-365.
- [Atwood 85] T. Atwood, "An object-oriented DBMS for design support applications", *Ontologic Inc. Report*.
- [Bancilhon 86] F. Bancilhon, "A logic programming object oriented cocktail", *ACM SIG-MOD Record*, 15:3, pp. 11-21, 1986.
- [Bancilhon and Maier 88] F. Bancilhon and D. Maier, "Multilanguage object-oriented systems: new answer to old database problems", in *Future Generation Computer II*, K. Fuchi and L. Kott editors, North-Holland, 1988.
- [Bancilhon *et al.* 88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Garmann, C. Lecluse, P. Pfeffer, P. Richard et F. Velez, "The design and implementation of O_2 , an object-oriented database system", *Proceedings of the ooDBS II Workshop*, Bad Munster, FRG, September 1988.
- [Banerjee *et al.* 87] J. Banerjee, H.T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou and H.J. Kim, "Data model issues for object-oriented applications", *ACM TOIS*, January 1987.
- [G-Base 88] "G-Base version 3, Introductory guide", *Graphael*, 1988.
- [Bobrow and Steifik 81] D. Bobrow and M. Steifik, "The Loops Manual", *Technical Report LB-VLSI-81-19*, Knowledge Systems Area, Xerox Palo Alto Research Center, 1981.
- [Carey *et al.* 86] M. Carey, D. DeWitt, J.E. Richardson and E.J. Shekita, "Object and file management in the EXODUS Extensible Database System", *Proceedings of the 12th VLDB*, pp 91-10, August 1986.
- [Caruso and Sciore 87] "The VISION Object-Oriented Database Management System", *Proceedings of the Workshop on Database Programming Languages*, Roscoff, France, September 1987
- [Caseau 89] "A model for a reflective object-oriented language", *Sigplan Notices*, Special issue on Concurrent Object-Oriented Programming, March 1989.
- [Codd 70] E. F. Codd, "A relational model for large shared data banks", *Communication of the ACM*, Volume 13, Number 6, (June 1970), pp 377-387.
- [Dittrich 1986] K.R. Dittrich, "Object-Oriented Database System : The Notions and the issues", in : *Dittrich, K.R. and Dayal, U. (eds): Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, IEEE Computer Science Press

- [Dittrich 1988] K. R. Dittrich, "Preface", In : Dittrich, K.R. (ed): *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science, Vol, 334, Springer-Verlag. 1988
- [Eiffel 87] "Eiffel user's manual", Interactive Software Engineering Inc., TR-EI-5/UM, 1987.
- [Fishman et al. 87] D. Fishman et al, "Iris: an object-oriented database management system", *ACM TOIS* 5:1, pp 48-69, January 86.
- [Hall et al. 76] P. Hall, J. Owlett, S. Todd, "Relations and Entities", in *"Modeling in Data Base Management Systems"*, G.M. Nijssen (ed.), pp 201-220, North-Holland, 1976.
- [Goldberg and Robson 83] A. Goldberg and D. Robson, "Smalltalk-80: the language and its implementation", *Addison-Wesley*, 1983.
- [Kim 88] W. Kim, "A foundation for object-oriented databases", *MCC Technical Report*, 1988.
- [Khoshafian and Copeland 86] S. Khoshafian and G. Copeland, "Object identity", *Proceedings of the 1st ACM OOPSLA conference*, Portland, Oregon, September 1986
- [Maier and Price 84] D. Maier and D. Price, "Data model requirements for engineering applications", *Proceedings of the First International Workshop on Expert Database Systems*, IEEE, 1984, pp 759-765
- [Maier 89] D. Maier, "Why isn't there an object-oriented data model?" *Proceedings IFIP 11th World Computer Conference*, San Francisco. CA, August-September 1989.
- [Maier, et al. 84] D. Maier, J. Stein, A. Otis, A. Purdy, "Development of an object-oriented DBMS" *Report CS/E-86-005*, Oregon Graduate Center, April 86
- [Nixon, et al. 87] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos and M. Stanley, "Design of a compiler for a semantic data model", *Technical note CSRI-44*, University of Toronto, May 1987.
- [Simula 67] "Simula 67 Reference Manual"
- [Schaffert, et al. 86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, "An introduction to Trellis/Owl", *Proceedings of the 1st OOPSALA Conference*, Portland, Oregon, September 1986
- [Skarra et al. 86] A. Skarra, S. Zdonik and S. Reiss, "An object server for an object oriented database system," *Proceedings of the 1986 International Workshop on Object Oriented Database System*, Computer Society Press, IEEE, pp. 196-204, 1986
- [Stroustrup 86] B. Stroustrup, "The C++ programming language", *Addison-Wesley*, 1986.

- [Van Emdem and Kowalski 76] M. Van Emdem and R. Kowalski, "The semantics of predicate logic as a programming language", *JACM*, Vol 23, No. 4, pp. 733-742, October 1976,
- [Zaniolo 86] C. Zaniolo, "Object-oriented programming in Prolog ", *Proceedings of the first workshop on Expert Database Systems*, 1985.



**THIRD-GENERATION DATA BASE SYSTEM
MANIFESTO**

by

The Committee for Advanced DBMS Function

Memorandum No. UCB/ERL M90/28

9 April 1990

**ELECTRONICS RESEARCH LABORATORY
College of Engineering
University of California, Berkeley, CA 94720**

THIRD-GENERATION DATA BASE SYSTEM MANIFESTO

The Committee for Advanced DBMS Function¹

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 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.

¹The 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.

²To 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.

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 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 [ATKJ89, KIM90, ZDON90] which suggest different sets of tenets.

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

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 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

⁴See the previous footnote for definitions of these terms.

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 week-ends 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

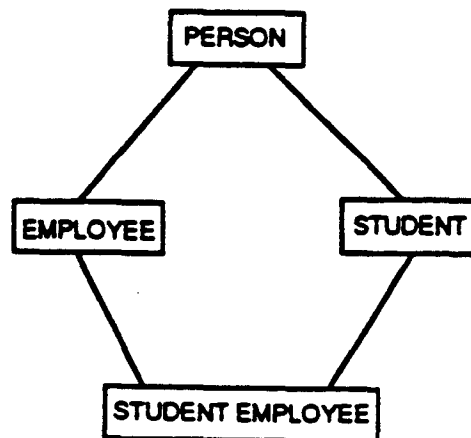
⁵One might argue that a relational system with all these extensions can no longer be considered "relational", but that is not the point. The point is that such extensions are possible and quite natural.

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 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.



A Typical Multiple Inheritance Hierarchy
Figure 1

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 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 `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 make 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 pro-

grammar.

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 represen-

tation 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 [LISK8, 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.

PROPOSITON 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 multi-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 only 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 Management 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*, Elsevier 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, NJ., 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 NF² 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 TOOLIS, 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.

DATABASE SYSTEMS: ACHIEVEMENTS AND OPPORTUNITIES

*Report of the NSF Invitational Workshop on
Future Directions in DBMS Research, February 1990¹*

I. EXECUTIVE SUMMARY

The history of database system research in the U.S. is one of exceptional productivity and startling economic impact. Barely twenty years old as a basic science research field, database research conducted with Federal support in the nation's universities and in its industrial research laboratories has fueled an information services industry estimated at \$10 billion per year in the U.S. alone. This industry has grown at an average rate of 20 percent per year since 1965 and is continuing to expand at this rate. Achievements in database research underpin fundamental advances in communications systems, transportation and logistics, financial management, knowledge-based systems, accessibility to scientific literature, and a host of other civilian and defense applications. They also serve as the foundation for considerable progress in basic science in various fields ranging from computing to biology.

As impressive as the accomplishments of basic database research have been, there is a growing awareness and concern in the U.S. and abroad that only the surface has been scratched in developing an understanding of the database principles and techniques required to support the advanced information management applications that are expected to revolutionize industrialized economies early in the next century. Rapid advances in areas such as manufacturing science, scientific visualization, robotics, optical storage, and high-speed communications already threaten to overwhelm the existing substrate of database theory and practice.

In February 1990, the National Science Foundation convened a 2-day workshop in Palo Alto, California for the purpose of identifying the *technology pull* factors that will serve as forcing functions for advanced database technology, and the corresponding basic research needed to enable that technology. The invited workshop participants included representatives from both the academic and industrial sides of the database research community. The primary conclusions of the workshop participants can be summarized as follows:

1. A substantial number of the advanced technologies that will underpin industrialized economies in the early twenty-first century depend on radically new database technologies that are currently not well understood, and that require intensive and sustained basic research.

¹ The workshop was attended by Michael Brodie, Peter Buneman, Mike Carey, Ashok Chandra, Hector Garcia-Molina, Jim Gray, Ron Fagin, Dave Lomet, Dave Maier, Marie Ann Niemat, Avi Silberschatz, Michael Stonebraker, Irv Traiger, Jeff Ullman, Gio Wiederhold, Carlo Zaniolo, and Maria Zemankova. Post-workshop contributors to this report include Phil Bernstein, Won Kim, Hank Korth, and Andre van Tilborg. The workshop was supported by NSF grant IRI-89-19556. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the panel and do not necessarily reflect the views of the National Science Foundation.

2. Next-generation database applications will have little in common with today's business data processing databases. They will involve much more data, require new capabilities including type extensions, multi-media support, complex objects, rule processing, and archival storage, and will necessitate rethinking the algorithms for almost all DBMS operations.
3. The cooperation between different organizations on common scientific, engineering, and commercial problems will require large-scale, heterogeneous, distributed databases. Very difficult problems await in the areas of inconsistent databases, security, and massive scale-up of distributed DBMS technology.
4. Basic research at universities and industrial research laboratories continues to serve as the bedrock on which current U.S. and global database systems technology rests.
5. The U.S. university infrastructure for conducting leading database research appears to be at risk of withering due to inadequate funding for work in this field.

Workshop participants reached a consensus that the challenges inherent in these conclusions demand energetic and proactive responses from U.S. industry, government, and academia. In particular, it is recommended that:

1. The National Science Foundation, together with the other Federal Government agencies represented in the Federal Coordinating Council on Science, Engineering, and Technology that fund basic research, should develop a strategy to ensure that basic research in the database area is funded at a level commensurate with its importance to scientific research and national economic well-being.
2. The National Research Council, through its Computer Science and Technology Board, should prepare a study that clearly indicates the appropriate level of Federal support for basic database research to maintain U.S. strength in this increasingly critical technology. Included should be a comparative study of international investments in database research, an assessment of the potential utility of international cooperation in selected areas of database research, and the delineation of a strategy for maintaining U.S. strength in database technology while allowing for potential international cooperation and joint projects.
3. Database research must constitute a substantial portion of the High Performance Computing Plan endorsed by the Office of Science and Technology Policy, and it must be a major component of Congressional legislation intended to ensure adequate support for computing, such as S. 1067 (the Gore bill) and S. 1976 (the Johnston bill). The Computing Research Board should help to ensure that database research is not neglected in these important plans.
4. U.S. industrial firms with a substantial stake in database technology must keep in mind the inherent linkage between the ongoing success of their commercial pursuits and university research and be vigorous in their support of academic research in the database area.

II. BACKGROUND AND SCOPE

The database research community has been in existence since the late 1960s. Starting with modest representation, mostly in industrial research laboratories, it has expanded dramatically over the last two decades to include substantial efforts at major universities, government laboratories and research consortia. Initially database research centered on a vital but limited class of applications, the management of data in business applications such as automated banking, record keeping, and reservation systems. These applications have four requirements that characterize database systems:

1. *Efficiency* in the access to and modification of very large amounts of data.
2. *Resilience*, the ability of the data to survive hardware crashes and software errors, without sustaining loss or becoming inconsistent.
3. *Access control*, including simultaneous access of data by multiple users in a consistent manner (called *concurrency control*) and assuring only authorized access to information (*security*).
4. *Persistence*, the maintenance of data over long periods of time, independent of any programs that access the data.

Database systems research has centered around methods for designing systems with these characteristics, and also around the languages and conceptual tools that help users to access, manipulate, and design databases.

From large corporate information systems to small personal databases, *database management systems* (DBMSs) are now used in almost every computing environment to organize, create and maintain important collections of information. The technology that makes these systems possible is the direct result of a successful program of database research. Section III of this report highlights some important achievements of the database research community over the past two decades, including the scope and significance of the technological transfer of database research results to industry. We focus on the major accomplishments of *relational databases*, *transaction management*, and *distributed databases*.

Today we stand at the threshold of applying database technology in a variety of new and important directions, including scientific databases, design databases, and universal access to information. Thus, in Section IV we pinpoint two key areas where targeted research funding will make a significant impact in the next few years: *next-generation database applications* and *heterogeneous, distributed databases*. Section V summarizes the need to maintain the momentum in this vital area of computer science and provides several concrete suggestions for doing so.

III. ACCOMPLISHMENTS OF THE LAST TWO DECADES

From among the various directions that the database research community has explored, the following three have perhaps had the most impact:

- Relational database systems,
- Transaction management, and

- Distributed database systems.

Each has fundamentally affected users of database systems, offering either radical simplifications in dealing with data, or great enhancement of their capability to manage information.

3.1. Relational Databases

In 1970, there were two popular approaches used to construct database management systems. The first approach, exemplified by IBM's Information Management System (IMS), has a *data model* (mathematical abstraction of data and operations on data) that requires all data records to be assembled into a collection of *trees*. Consequently, some records are *root* records and all others have unique *parent* records. IMS had a low-level query language, by which an application programmer could *navigate* from root records to the records of interest, accessing one record at a time.

The second approach was typified by the proposal of the Conference on Data Systems Languages (CODASYL). They suggested that the collection of DBMS records be arranged into a directed graph. Again, a navigational query language was proposed, by which an application program could move from a specific *entry point* record to desired information.

Both the tree-based (called *hierarchical*) and graph-based (*network*) approaches to data management have several fundamental disadvantages.

1. To answer a specific database request, an application programmer, skilled in performing disk-oriented optimization, must write a complex program to navigate through the database. For example, the company president cannot, at short notice, pose the query "How many employees in the Widget department will retire in the next three years?" unless he has the skill and patience to write a detailed program.
2. When the structure of the database changes, as it will whenever new kinds of information are added, application programs usually need to be rewritten.

As a result, the database systems of 1970 were costly to use because of the low-level interface between the application program and the DBMS, and because the dynamic nature of user data mandates repeated program maintenance.

The relational data model, pioneered by E. F. Codd in a series of papers in 1970-72, offered a fundamentally different approach to data storage. Codd suggested that conceptually all data be represented by simple tabular data structures (*relations*), and that users access data through a high-level, nonprocedural (or *declarative*) query language. Instead of writing an algorithm to obtain desired records one at a time, the application programmer is only required to specify a *predicate* that identifies the desired records, or combination of records. A *query optimizer* in the DBMS translates the predicate specification into an algorithm to perform database access to solve the query. These nonprocedural languages are dramatically easier to use than the navigation languages of IMS and CODASYL; they lead to higher programmer productivity and facilitate direct database access by end users.

During the 1970s the database research community extensively investigated the relational DBMS concept. They:

- Invented high-level relational query languages to ease the use of the DBMS by both end users and application programmers. The theory of higher level query languages has been developed to provide a firm basis for understanding and evaluating the expressive power of database language constructs.
- Developed the theory and algorithms necessary to “optimize” queries, that is, to translate queries in the high-level relational query languages into plans for accessing the data that are as efficient as what a skilled programmer would have written using one of the earlier DBMSs. This technology probably represents the most successful experiment in optimization of very high-level languages among all varieties of computer systems.
- Formulated a theory of “normalization” to help with database design by eliminating redundancy and certain logical anomalies from the data.
- Constructed algorithms to allocate tuples of relations to *pages* (blocks of records) in files on secondary storage, to minimize the average cost of accessing those tuples.
- Constructed buffer management algorithms to exploit knowledge of access patterns for moving pages back and forth between disk and a main memory buffer pool.
- Constructed indexing techniques to provide fast associative access to random single records and/or sets of records specified by values or value ranges for one or more attributes.
- Implemented prototype relational DBMSs that formed the nucleus for many of the present commercial relational DBMSs.

As a result of this research in the 1970s, numerous commercial products based on the relational concept appeared in the 1980s. Not only were the ideas identified by the research community picked up and used by the vendors, but also, several of the commercial developments were led by implementors of the earlier research prototypes. Today, commercial relational database systems are available on virtually any hardware platform from personal computer to mainframe, and are likely to become standard software on all new computers in the 1990s.

There is a moral to be learned from the success of relational database systems. When the relational data model was first proposed, it was regarded as an elegant theoretical construct but implementable only as a “toy.” It was only with considerable research, much of it focused on basic principles of relational databases, that large-scale implementations were made possible. The next generation of databases calls for continued research into the foundations of database systems, in the expectation that other such useful “toys” will emerge.

3.2. Transaction Management

During the last two decades, database researchers have also pioneered the transaction concept. A *transaction* is a sequence of operations that must appear “atomic” when executed. For example, when a bank customer moves \$100 from account *A* to account *B*, the database system must ensure that either both of the operations

1. Debit *A*

2. Credit *B*

happen or that neither happens (and the customer is informed). If only the first occurs, then the customer has lost \$100, and an *inconsistent* database state results.

To guarantee that a transaction transforms the database from one consistent state to another requires that:

- The concurrent execution of transactions must be such that each transaction appears to execute in isolation. *Concurrency control* is the technique used to provide this assurance.
- System failures, either of hardware or software, must not result in inconsistent database states. A transaction must execute in its entirety or not at all. *Recovery* is the technique used to provide this assurance.

We briefly elaborate on these two issues below.

Concurrent transactions in the system must be *synchronized* correctly in order to guarantee that consistency is preserved. For instance, while we are moving \$100 from *A* to *B*, a simultaneous movement of \$300 from account *B* to account *C* should result in a net deduction of \$200 from *B*. The normal view of “correct” synchronization of transactions is that they must be *serializable*; that is, the effect on the database of any number of transactions executing in parallel must be the same as if they were executed one after another, in some order.

During the 1970s and early 1980s the DBMS research community worked extensively on the transaction model. First, the theory of serializability was worked out in detail, and precise definitions of the correctness of *schedulers* (algorithms for deciding when transactions could execute) were produced. Second, numerous concurrency control algorithms were invented that ensure serializability. These included algorithms based on

1. *Locking data* items so conflicting accesses were prohibited. Especially important is a technique called *two-phase locking*, which guarantees serializability by requiring that a transaction obtain all the locks it will ever need before releasing any locks.
2. *Timestamping* accesses so the system could check that no violations of serializability were possible.
3. Keeping *multiple versions* of data objects available.

The various algorithms were subjected to rigorous experimental studies and theoretical analysis to determine the conditions under which each was preferred.

Recovery is the other essential component of transaction management. We must guarantee that all the effects of a transaction are installed in the database, or that none of them are, and this guarantee must be kept even when a system crash loses the contents of main memory. During the late 1970s and early 1980s, two major approaches to this service were investigated, namely:

1. *Write-ahead logging*. A summary of the effects of a transaction is stored in a sequential file, called a log, before the changes are installed in the database itself. The log is on disk or tape where it can survive system crashes and power failures. When a transaction completes, the logged changes are then posted to the database. If a transaction fails to complete, the log is used to restore the prior database state.

2. *Shadow file techniques.* New copies of entire data items, usually disk pages, are created to reflect the effects of a transaction and are written to the disk in entirely new locations. A single atomic action remaps the data pages, so as to substitute the new versions for the old when the transaction completes. If a transaction fails, the new versions are discarded.

Recovery techniques have been extended to cope with the failure of the “stable” medium as well. A backup copy of the data is stored on an entirely separate device. Then, with logging, the log can be used to “roll forward” the backup copy to the current state.

3.3. Distributed Databases

A third area in which the DBMS research community played a vital role is distributed databases. In the late 1970s there was a realization that organizations are fundamentally decentralized and require databases at multiple sites. For example, information about the California customers of a company might be stored on a machine in Los Angeles, while data about the New England customers could exist on a machine in Boston. Such data distribution moves the data closer to the people who actually use it and reduces remote communication costs.

Furthermore, the decentralized system is more *available* when crashes occur. If a single, central site goes down, all data is unavailable. However, if one of several regional sites goes down, only part of the total database is inaccessible. Moreover, if the company chooses to pay the cost of *multiple copies* of important data, then a single site failure need not cause data inaccessibility.

In a multidatabase environment we strive to provide *location transparency*. That is, all data should appear to the user as if they are located at his or her particular site. Moreover, the user should be able to execute normal transactions against such data. Providing location transparency required the DBMS research community to investigate new algorithms for distributed query optimization, concurrency control, crash recovery, and support of multiple copies of data objects for higher performance and availability.

In the early 1980s the research community rose to this challenge. Distributed concurrency control algorithms were designed, implemented and tested. Again, simulation studies and analysis compared the candidates to see which algorithms were dominant. The fundamental notion of a *two-phase commit* to ensure the possibility of crash recovery in a distributed database was discovered. Algorithms were designed to recover from processor and communication failures, and *data patch* schemes were put forward to re-join distributed databases that had been forced to operate independently after a network failure. Technology for optimizing distributed queries was developed, along with new algorithms to perform the basic operations on data in a distributed environment. Lastly, various algorithms for the update of multiple copies of a data item were invented; these ensure that all copies of each item are consistent.

All the major DBMS vendors are presently commercializing distributed DBMS technology. Again we see the same pattern discussed above for relational databases and transactions, namely aggressive research funding by government and industry, followed by rapid technology transfer from research labs to commercial products.

IV. THE NEXT CHALLENGES

Some might argue that database systems are a mature technology and it is therefore time to refocus research onto other topics. Certainly relational DBMSs, both centralized and distributed, are well studied, and commercialization is well along. *Object management* ideas, following the philosophy of "object-oriented programming," have been extensively investigated over the last few years and should allow more general kinds of data elements to be placed in databases than the numbers and character strings supported in traditional systems. The relentless pace of advances in hardware technology makes CPUs, memory and disks drastically cheaper each year. Current databases will therefore become progressively cheaper to deploy as the 1990s unfold. Perhaps the DBMS area should be declared "solved," and energy and research money allocated elsewhere.

We argue strongly here that such a turn of events would be a serious mistake. Rather, we claim that solutions to the important database problems of the year 2000 and beyond are not known. Moreover, hardware advances of the next decade will not make brute force solutions economical, because the scale of the prospective applications is simply too great.

In this section we highlight two key areas where we feel important research contributions are required in order to make future DBMS applications viable:

1. *Next-generation database applications*
2. *Heterogeneous, distributed databases.*

In addition to being important intellectual challenges in their own right, their solutions offer products and technology of great social and economic importance, including, among many others, improved delivery of medical care, advanced design and manufacturing systems, enhanced tools for scientists, greater per capita productivity through increased personal access to information, and new military applications.

4.1. The Research Agenda for Next-Generation DBMS Applications

To motivate the discussion of research problems that follows, in this section we present several examples of the kinds of database applications that we expect will be built during the next decade.

1. For many years, NASA has been collecting vast amounts of information from space. They estimate that they require storage for 10^{16} bytes of data (about 10,000 optical disk jukeboxes) just to maintain a few years worth of satellite image data that they will collect in the 1990s. Moreover, they are very reluctant to throw anything away, lest it be exactly the data set needed by a future scientist to test some hypothesis. It is unclear how this database can be stored and searched for relevant images using current or soon-to-be available technology.
2. Databases serve as the backbone of computer-aided design systems. For example, civil engineers envision a facilities-engineering design system that manages all information about a project, such as a skyscraper. This database must maintain and integrate information about the project from the viewpoints of hundreds of subcontractors. For example, when an electrician puts a hole in a beam to let a wire through, the load-bearing soundness of the structure could be compromised. The design system should,

ideally, recalculate the stresses, or at the least, warn the cognizant engineer that a problem may exist.

3. The National Institutes of Health (NIH) and the U.S. Department of Energy (DOE) have embarked on a joint national initiative to construct the DNA sequence corresponding to the human genome. The gene sequence is several billion elements long and each element is a complex and somewhat variable object. The matching of individuals' medical problems to differences in genetic makeup is a staggering problem and will require new technologies of data representation and search.
4. Several large department stores already record every price-scanning action of every cashier in every store in their chain. Buyers run ad-hoc queries on this historical database, in an attempt to discover buying patterns and make stocking decisions. This application taxes the capacity of available disk systems. Moreover, as the cost of disk space declines, the retail chain will keep a larger and larger history to track buying habits more accurately. This process of "mining" data for hidden patterns is not limited to commercial applications. We foresee similar applications, often with even larger databases, in science, medicine, intelligence gathering, and many other areas.
5. Most insurance firms have a substantial on-line database that records the policy coverage of the firm's customers. These databases will soon be enhanced with *multimedia* data such as photographs of property damaged, digitized images of handwritten claim forms, audio transcripts of appraisers' evaluations, images of specially insured objects, and so on. Since image data is exceedingly large, such databases will become enormous. Moreover, future systems may well store video walk-throughs of houses in conjunction with a homeowners policy, further enlarging the size of this class of databases. Again, applications of this type are not limited to commercial enterprises.

These applications not only introduce problems of size, they also introduce problems with respect to all conventional aspects of DBMS technology (e.g., they pose fundamentally new requirements for access patterns, transactions, concurrency control, and data representation). These applications have in common the property that they will push the limits of available technology for the foreseeable future. As computing resources become cheaper, these problems are all likely to expand at the same or at a faster rate. Hence, they cannot be overcome simply by waiting for the technology to bring computing costs down to an acceptable level.

We now turn to the research problems that must be solved to make such *next-generation* applications work. Next-generation applications require new services in several different areas in order to succeed.

New Kinds of Data

Many next-generation applications entail storing large and internally complex objects. The insurance example, (5) above, requires storage of images. Scientific and design databases often deal with very large arrays or sequences of data elements. A database for software engineering might store program statements, and a chemical database might store protein structures. We need solutions to two classes of problems: data access and data type management.

Current databases are optimized for delivering small records to an application program. When fields in a record become very large, this paradigm breaks down. The DBMS should read a large object only once and place it directly at its final destination. Protocols must be designed to *chunk* large objects into manageable-size pieces for the application to process. A new generation of query languages will be required to support querying of array and sequence data as will mechanisms for easily manipulating disk and archive representations of such objects. In addition, extended storage structures and indexing techniques will be needed to support efficient processing of such data.

A second class of problems concerns type management. There must be a way for the programmer to construct the types appropriate for his application. The need for more flexible type systems has been one of the major forces in the development of object-oriented databases. One of the drawbacks of the systems developed so far is that type-checking is largely *dynamic*, which lays open the possibility that programming errors tend to show up at run-time, not during compilation. In order to provide the database application designer with the same safety-nets that are provided by modern high-level programming languages, we need to determine how we can combine static type disciplines with persistent data and evolution of the database structure over time.

Rule Processing

Next-generation applications will frequently involve a large number of *rules*, which take *declarative* ("if *A* is true, then *B* is true"), and *imperative* ("if *A* is true, then do *C*") forms. For example, a design database should notify the proper designer if a modification by a second designer may have affected the subsystem that is the responsibility of the first designer. Such rules may include elaborate constraints that the designer wants enforced, triggered actions that require processing when specific events take place, and complex deductions that should be made automatically within the system. It is common to call such systems "knowledge-base systems," although we prefer to view them as a natural, although difficult, extension of DBMS technology.

Rules have received considerable attention as the mechanism for triggering, data mining (as discussed in the department store example), and other forms of reasoning about data. Declarative rules are advantageous because they provide a *logical declaration* of what the user wants rather than a detailed specification of how the results are to be obtained. Similarly, imperative rules allow for a declarative specification of the conditions under which a certain action is to be taken. The value of declarativeness in relational query languages like SQL (the most common such language) has been amply demonstrated, and an extension of the idea to the next generation of query languages is desirable.

Traditionally, rule processing has been performed by separate subsystems, usually called "expert system shells." However, applications such as the notification example above cannot be done efficiently by a separate subsystem, and such rule processing must be performed directly by the DBMS. Research is needed on how to specify the rules and on how to process a large rule base efficiently. Although considerable effort has been directed at these topics by the Artificial Intelligence community, the focus has been on approaches that assume all relevant data structures are in main memory, such as RETE networks. Next-generation applications are far too big to be amenable to such techniques.

We also need tools that will allow us to validate and debug very large collections of

rules. In a large system, the addition of a single rule can easily introduce an inconsistency in the knowledge base or cause chaotic and unexpected effects and can even end up repeatedly "firing" itself. We need techniques to decompose sets of rules into manageable components and prevent (or control in a useful way) such inconsistencies and repeated rule firing.

New Concepts in Data Models

Many of the new applications will involve primitive concepts not found in most current applications, and there is a need to build them cleanly into specialized or extended query languages. Issues range from efficiency of implementation to the fundamental theory underlying important primitives. For example, we need to consider:

1. *Spatial Data.* Many scientific databases have two- or three-dimensional points, lines, and polygons as data elements. A typical search is to find the ten closest neighbors to some given data element. Solving such queries will require sophisticated, new multidimensional access methods. There has been substantial research in this area, but most has been oriented toward main memory data structures, such as quad trees and segment trees. The disk oriented structures, including K-D-B trees and R-trees, do not perform particularly well when given real world data.
2. *Time.* In many exploratory applications, one might wish to retrieve and explore the database state as of some point in the past or to retrieve the time history of a particular data value. Engineers, shopkeepers, and physicists all require different notions of time. No support for an algebra over time exists in any current commercial DBMS, although research prototypes and special-purpose systems have been built. However, there is not even an agreement across systems on what a "time interval" is; for example, is it discrete or continuous, open ended or closed?
3. *Uncertainty.* There are applications, such as identification of features from satellite photographs, where we need to attach a likelihood that data represents a certain phenomenon. Reasoning under uncertainty, especially when a conclusion must be derived from several inter-related partial or alternative results, is a problem that the Artificial Intelligence community has addressed for many years, with only modest success. Further research is essential, as we must learn not only to cope with data of limited reliability, but to do so efficiently, with massive amounts of data.

Scaling Up

It will be necessary to *scale* all DBMS algorithms to operate effectively on databases of the size contemplated by next-generation applications, often several orders of magnitude bigger than the largest databases found today. Databases larger than a terabyte (10^{12} bytes) will not be unusual. The current architecture of DBMSs will not scale to such magnitudes. For example, current DBMSs build a new index on a relation by locking it, building the index and then releasing the lock. Building an index for a 1-terabyte table may require several days of computing. Hence, it is imperative that algorithms be designed to construct indexes incrementally without making the table being indexed inaccessible.

Similarly, making a dump on tape of a 1-terabyte database will take days, and obviously must be done incrementally, without taking the database off line. In the event that a database is corrupted because of a head crash on a disk or for some other reason, the

traditional algorithm is to restore the most recent dump from tape and then to roll the database forward to the present time using the database log. However, reading a 1-terabyte dump will take days, leading to unacceptably long recovery times. Hence, a new approach to backup and recovery in very large databases must be found.

Parallelism

Ad-hoc queries over the large databases contemplated by next-generation application designers will take a long time to process. A scan of a 1-terabyte table may take days, and it is clearly unreasonable for a user to have to submit a query on Monday morning and then go home until Thursday when his answer will appear.

First, imagine a 1-terabyte database stored on a collection of disks, with a large number of CPUs available. The goal is to process a user's query with nearly *linear speedup*; that is, the query is processed in time inversely proportional to the number of processors and disks allocated. To obtain linear speedup, the DBMS architecture must avoid bottlenecks, and the storage system must ensure that relevant data is spread over all disk drives. Moreover, parallelizing a user command will allow it to be executed faster, but it will also use a larger fraction of the available computing resources, thereby penalizing the response time of other concurrent users, and possibly causing the system to thrash, as many queries compete for limited resources. Research on multiuser aspects of parallelism such as this one is in its infancy.

On the other hand, if the table in question is resident on an archive, a different form of parallelism may be required. If there are no indexes to speed the search, a sequential scan may be necessary, in which case the DBMS should evaluate as many queries as possible in parallel, while performing a single scan of the data.

In general, it remains a challenge to develop a realistic theory for data movement throughout the memory hierarchy of parallel computers. The challenges posed by next-generation database systems will force computer scientists to confront these issues.

Tertiary Storage

For the foreseeable future, ultra large databases will require both secondary (disk) storage and the integration of an *archive* or tertiary store into the DBMS. All current commercial DBMSs require data to be either disk or main-memory resident. Future systems will have to deal with the more complex issue of optimizing queries when a portion of the data to be accessed is in an archive. Current archive devices have a very long latency period. Hence, query optimizers must choose strategies that avoid frequent movement of data between storage media. Moreover, the DBMS must also optimize the placement of data records on the archive to minimize subsequent retrieval times. Lastly, in such a system, disk storage can be used as a read or write cache for archive objects. New algorithms will be needed to manage intelligently the buffering in a three-level system.

Long-Duration Transactions

The next-generation applications often aim to facilitate collaborative and interactive access to a database. The traditional transaction model discussed in Section III assumes that transactions are short, perhaps a fraction of a second. However, a designer may lock a file for a day, during which it is redesigned. We need entirely new approaches to maintaining

the integrity of data, sharing data, and recovery of data, when transactions can take hours or days.

Versions and Configurations

Some next-generation applications need *versions* of objects to represent alternative or successive states of a single conceptual entity. For instance, in a facilities-engineering database, numerous revisions of the electric plans will occur during the design, construction and maintenance of the building, and it may be necessary to keep all the revisions for accounting or legal reasons. Furthermore, it is necessary to maintain consistent *configurations*, consisting of versions of related objects, such as the electrical plan, the heating plan, general and detailed architectural drawings.

While there has been much discussion and many proposals for proper version and configuration models in different domains, little has been implemented. Much remains to be done in the creation of space-efficient algorithms for version management and techniques for ensuring the consistency of configurations.

4.2. Heterogeneous, Distributed Databases

There is now effectively one world-wide telephone system and one world-wide computer network. Visionaries in the field of computer networks speak of a single world-wide file system. Likewise, we should now begin to contemplate the existence of a single, world-wide database system, from which users can obtain information on any topic covered by data made available by purveyors, and on which business can be transacted in a uniform way. While such an accomplishment is a generation away, we can and must begin now to develop the underlying technology in collaboration with other nations.

Indeed, there are a number of applications that are now becoming feasible and that will help drive the technology needed for worldwide interconnection of information.

1. Collaborative efforts are underway in many physical science disciplines, entailing multiproject databases. The project has a database composed of portions assembled by each researcher, and a *collaborative* database results. The human genome project is one example of this phenomenon.
2. A typical defense contractor has a collection of subcontractors assisting with portions of the contractor project. The contractor wants to have a single project database that spans the portions of the project database administered by the contractor and each subcontractor.
3. An automobile company wishes to allow its suppliers to access new designs of cars that it is contemplating building. In this way, suppliers can give early feedback on the cost of components. Such feedback will allow the most cost-effective car to be designed and manufactured. However, this goal requires a database that spans multiple organizations, that is, an *intercompany* database.

These examples all concern the necessity of logically integrating databases from multiple organizations, often across company boundaries, into what appears to be a single database. The databases involved are *heterogeneous*, in the sense that they do not normally share a complete set of common assumptions about the information with which they

deal, and they are *distributed*, meaning that individual databases are under local control and are connected by relatively low-bandwidth links. The problem of making heterogeneous, distributed databases behave as if they formed part of a single database is often called *interoperability*; we now use two very simple examples to illustrate the problems that arise in this environment.

First, consider a hypothetical program manager at the NSF, who wishes to find the total number of Computer Science Ph.D. students in the U.S. There are over 100 institutions that grant a Ph.D. degree in Computer Science. In theory, all have an on-line student database that allows queries to be asked of its contents. Moreover, the NSF program manager can, in theory, discover how to access all of these databases and then ask the correct *local* query at each site.

Unfortunately, the sum of the responses to these 100+ local queries will not necessarily be the answer to his overall query. Some institutions record only full-time students; others record full- and part-time students. Furthermore, some distinguish Ph.D. from Masters candidates, and some do not. Some erroneously may omit certain classes of students, such as foreign students. Some may mistakenly include students, such as Electrical Engineering candidates in an EECS department. The basic problem is that these 100+ databases are *semantically inconsistent*.

A second problem is equally illustrative. Consider the possibility of an electronic version of a travel assistant, such as the Michelin Guide. Most people traveling on vacation consult two or more such travel guides, which list prices and quality ratings for restaurants and hotels. Obviously, one might want to ask the price of a room at a specific hotel, and each guide is likely to give a different answer. One might quote last year's price, while another might indicate the price with tax, and a third might quote the price including meals. To answer the user's query, it is necessary to treat each value obtained as *evidence*, and then to provide *fusion* of this evidence to form a best answer to the user's query.

To properly support heterogeneous, distributed databases, there is a difficult research agenda, outlined below, that must be accomplished.

Browsing

Let us suppose that the problems of access have been solved in any one of the scenarios mentioned above; that is, the user has a uniform query language that can be applied to any one of the individual databases or to some merged "view" of the collection of databases. If an inconsistency is detected, or if missing information appears to invalidate a query, we cannot simply give up. There must be some system for explaining to the user how the data arrived in that state and, in particular, from what databases it was derived. With this information, it may be possible to filter out the offending data elements and still arrive at a meaningful query. Without it, it is highly unlikely that any automatic agent could do a trustworthy job. Thus, we need to support *browsing*, the ability to interrogate the structure of the database and, when multiple databases are combined, interrogate the nature of the process that merges data.

Incompleteness and Inconsistency

The Ph.D. student and travel-advisor examples above indicate the problems with semantic inconsistency and with data fusion. In the Ph.D. student example there are 100+

disparate databases each containing student information. Since the individual participant databases were never designed with the objective of interoperating with other databases, there is no single *global schema* to which all individual databases conform. Rather there are individual differences that must be addressed. These include differences in units. For example, one database might give starting salaries for graduates in dollars per month while another records annual salaries. In this case, it is possible to apply a conversion to obtain composite consistent answers. More seriously, the definition of a part-time student may be different in the different databases. This difference will result in composite answers that are semantically inconsistent. Worse still is the case where the local database omits information, such as data on foreign students, and is therefore simply wrong.

Future interoperability of databases will require dramatic progress to be made on these semantic issues. We must extend substantially the data model that is used by a DBMS to include *much* more semantic information about the meaning of the data in each database. Research on extended data models is required to discover the form that this information should take.

Mediators

As the problems of fusion and semantic inconsistency are so severe, there is need for a class of information sources that stand between the user and the heterogeneous databases. For example, if there were sufficient demand, it would make sense to create a "CS Ph.D. mediator" that could be queried as if it were a consistent, unified database containing the information that actually sits in the 100+ local databases of the CS departments. A "travel advisor" that provided the information obtained by fusing the various databases of travel guides, hotels, car-rental companies, and so on, could be commercially viable. Perhaps most valuable of all would be a mediator that provided the information available in the world's libraries, or at least that portion of the libraries that are stored electronically.

Mediators must be accessible by people who have not had a chance to study the details of their query language and data model. Thus, some agreement regarding language and model standards is essential, and we need to do extensive experiments before standardization can be addressed. Self-description of data is another important research problem that must be addressed if access to unfamiliar data is to become a reality.

Name Services

The NSF program manager must be able to consult a national name service to discover the location and name of the databases or mediators of interest. Similarly, a scientist working in an inter-disciplinary problem domain must be able to discover the existence of relevant data sets collected in other disciplines. The mechanism by which items enter and leave such name servers and the organization of such systems is an open issue.

Security

Security is a major problem (failing) in current DBMSs. Heterogeneity and distribution makes this open problem even more difficult. A corporation may want to make parts of its database accessible to certain parties, as with the automobile company in (3) above offering preliminary design information to potential suppliers. However, the automobile company certainly does not want the same designs accessed by its competitors, and it doesn't want any outsider accessing its salary data.

Authentication is the reliable identification of subjects making database access. A heterogeneous, distributed database system will need to cope with a world of multiple authenticators of variable trustworthiness. Database systems must be resistant to compromise by remote systems masquerading as authorized users. We foresee a need for mandatory security and research into the analysis of covert channels, in order that distributed, heterogeneous database systems do not increase user uncertainty about the security and integrity of his data.

A widely distributed system may have thousands or millions of users. Moreover, a given user may be identified differently on different systems. Further, access permission might be based on role (e.g., current company Treasurer) or access site. Finally, sites can act as intermediate agents for users, and data may pass through and be manipulated by these intervening sites. Whether an access is permitted may well be influenced by who is acting on a user's behalf. Current authorization systems will surely require substantial extensions to deal with these problems.

Site Scale-up

The security issue is just one element of *scale-up*, which must be addressed in a large distributed DBMS. Current distributed DBMS algorithms for query processing, concurrency control, and support of multiple copies were designed to function with a few sites, and they must all be rethought for 1000 or 10,000 sites. For example, some query processing algorithms expect to find the location of an object by searching all sites for it. This approach is clearly impossible in a large network. Other algorithms expect all sites in the network to be operational, and clearly in a 10,000 site network, several sites will be down at any given time. Lastly, certain query processing algorithms expect to optimize a join by considering all possible sites and choosing the one with the cheapest overall cost. With a very large number of sites, a query optimizer that loops over all sites in this fashion may well spend more time trying to "optimize" the query than it would have spent in simply executing the query in a naive and expensive way.

Powerful desktop computers, cheap and frequently underutilized, must be factored into the query optimization space, as using them will frequently be the most responsive and least expensive way to execute a query. Ensuring good user response time becomes increasingly difficult as the number of sites and the distances between them increase. Local caching, and even local replication, of remote data at the desktop will become increasingly important. Efficient cache maintenance is an open problem.

Transaction Management

Transaction management in a heterogeneous, distributed database system is a difficult issue. The main problem is that each of the local database management systems may be using a different type of a concurrency control scheme. Integrating these is a challenging problem, made worse if we wish to preserve the local autonomy of each of the local databases and allow local and global transactions to execute in parallel.

One simple solution is to restrict global transactions to retrieve-only access. However, the issue of reliable transaction management in the general case, where global and local transactions are allowed to both read and write data, is still open.

V. CONCLUSIONS AND CALL TO ACTION

In brief, next-generation applications will little resemble current business data processing databases. They will have much larger data sets, require new capabilities such as type extensions, multi-media support, complex objects, rule processing, and archival storage, and they will entail rethinking algorithms for almost all DBMS operations. In addition, the cooperation between different organizations on common problems will require heterogeneous, distributed databases. Such databases bring very difficult problems in the areas of querying semantically inconsistent databases, security, and scale-up of distributed DBMS technology to large numbers of sites. Thus, database systems research offers

- A host of new intellectual challenges for computer scientists.
- Resulting technology that will enable a broad spectrum of new applications in business, science, medicine, defense, and other areas.


In order to address the crucial DBMS issues that face the world in the 1990s, and to ensure that the commercial DBMS business remains healthy on into the next century, we feel it is imperative that NSF initiate strong new programs of research in the areas outlined in Section IV: next-generation database applications and heterogeneous, distributed databases. We also call on government agencies charged with promotion of research in the target application areas, and on companies that today benefit from the previous round of database research, to support the next round of basic research. We specifically recommend the following:

- NSF, together with the other agencies represented in the Federal Coordinating Council on Science, Engineering, and Technology that fund basic research, should develop a strategy to ensure that basic database research is funded at a level commensurate with its importance to scientific research and national economic well-being.
- The Computer Science and Technology Board of the National Research Council should prepare a study indicating the level of Federal funding required for basic database research in order to maintain U.S. strength in this increasingly critical technology. Specific issues addressed should include a comparative study of international investments in database research, an assessment of the potential utility of international cooperation in selected areas of database research based on areas of actual and potential research overlap, and the delineation of a strategy for maintaining U.S. strength in database technology while allowing for potential international cooperation and joint projects.
- The Computing Research Board should help to ensure that database research constitutes a substantial portion of the High Performance Computing Plan endorsed by the Office of Science and Technology Policy and that it is a major component of Congressional legislation intended to ensure adequate support for computing, such as S. 1067 (the Gore bill) and S. 1976 (the Johnston bill).
- U.S. industrial firms with a substantial stake in database technology should vigorously support existing programs and the development of new programs that provide funding for basic university research in the database area.

To date, the database industry has shown remarkable success in transforming scientific ideas into major products, and it is crucial that advanced research be encouraged actively as the database community tackles the challenges ahead. The fate of a major industry in the year 2000 and beyond will be determined by the decisions made in the next few years.

ACKNOWLEDGEMENTS

This report represents the contributions of all participants at the workshop as well as several post-workshop contributors. Special thanks are owed to Avi Silbershatz and Jeff Ullman for initiating the workshop and performing the painstaking tasks of coordinating this work and editing the report, and to Mike Stonebraker, whose initial draft provided the organization and much of the content for the final report. We also thank Hewlett-Packard Laboratories for kindly hosting the workshop.

Distributed by
 **TANDEM**
Corporate Information Center
10400 N. Tantau Ave., LOC 248-07
Cupertino, CA 95014-0708

