JUNE 1987 VOL. 10 NO. 2

a quarterly bulletin of the
Computer Society of the IEEE
technical committee on

Database Engineering

CONTENTS

SPECIAL ISSUE ON EXTENSIBLE DATABASE SYSTEMS

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering . Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the Editor

The theme of this issue of **Database Engineering** is "Extensible Database Systems." Now that relational database technology is well understood, a number of database researchers have turned their attention to applications that are not well served by relational systems. Applications such as computer-aided software engineering, CAD/CAM, scientific/statistical data gathering, image processing, and data-intensive AI applications all have requirements that exceed the capabilities of traditional relational database systems. One approach to providing the new data types, operations, access methods, and other features needed for such applications is through developing an *extensible database system* — a database system that can be customized to fit the needs of a wide range of potential applications.

I asked eight research groups working on database system extensibility to contribute papers to this issue, and to my delight, all eight of them agreed. In addition, a ninth group agreed to submit a short research summary; because their paper was a "late entry," time and page count constraints made it impossible for their contribution to be a full-length paper. I found all nine of these papers to be very informative, and I hope that the **Database Engineering** readership will agree.

The first two papers are excellent lead-in papers for this issue. In *CASE Requirements for Extensible Database Systems*, Phil Bernstein and David Lomet describe database requirements for computer-aided software engineering, a challenging application area for extensible database systems. They also describe their work in progress at Wang Institute. The second paper, *Extensible Databases and RAD*, is by Silvia Osborn of the University of Western Ontario. This paper categorizes various approaches to supporting new applications and then describes the author's experiences with the RAD system, a relational DBMS extended with a facility for defining new data types for domains.

The next three papers describe three extensible database system projects. *Extendability in POSTGRES*, by Mike Stonebraker, Jeff Anton, and Michael Hirohama, discusses the POSTGRES project at UC Berkeley. The paper describes the POSTGRES facilities for adding user-defined functions, data types, operators, aggregate functions, and access methods. In *Extensibility in the PROBE Database System*, David Goldhirsch and Jack Orenstein present an overview of the PROBE project at CCA. They describe PROBE's data model and algebra, how new object classes can be added to PROBE, and PROBE's support for spatial applications. John McPherson and Hamid Pirahesh describe the Starburst project at IBM Almaden in *An Overview of Extensibility in Starburst*. They emphasize the overall Starburst architecture and how it supports extensions in the areas of storage management, access methods, data types, and complex objects.

The next two papers also describe extensible database system projects; they differ from the previous three in being oriented more towards supporting the production of customized database systems than towards providing one particular DBMS which can then be extended. In *Principles of Database Management System Extensibility*, Don Batory describes the GENESIS project at UT-Austin. He describes how extensibility is provided, from low-level storage structures up through user-level data types and operators, via standardized interfaces and careful layering. In *An Overview of the EXODUS Project*, David DeWitt and I describe the EXODUS extensible database system project at the University of Wisconsin. We describe the collection of components and tools that EXODUS provides, explaining how they simplify application-specific DBMS development and extensions.

In the next paper, *An Extensible Framework for Multimedia Information Management*, Darrell Woelk and Won Kim describe an extensible component of the ORION object-oriented database system at MCC. They explain how object-oriented concepts are used in ORION's multimedia information manager, enabling extensions to be added for handling new storage and I/O devices and new types of multimedia information. The final paper, *DASDBS: A Kernel DBMS and Application-Specific Layers* is by Hans Schek (who is currently on leave at IBM Almaden). He briefly summarizes how the DASDBS project at the Technical University of Darmstadt is addressing extensibility via their DBMS kernel approach.

I would like to thank each of the authors for agreeing to participate in this special issue of **Database Engineering**. I hope that the readers will find the authors' contributions to be as interesting as I did.

Michael J. Carey
May, 1987

CASE Requirements for Extensible Database Systems

Philip A. Bernstein
David B. Lomet
Wang Institute of Graduate Studies[1]

1. Introduction

A database system (DBMS) offers functions for storage managment, type management, associative access, and transaction management. State-of-the-art relational DBMSs provide storage management for small records, type management for simple record structures, associative access using relational algebraic operators, and transaction management for short-duration transactions. These facilities are inadequate to support most engineering design applications, such as mechanical design, electronic design, and software design [10, 21, 22, 24, 31, 79].

In particular, these facilities are inadequate for computer-aided software engineering (CASE). A CASE environment consists of integrated tools to assist the individual programmer (e.g., editors and debuggers), to manage multi-component product configurations (e.g., version control and automated regression testers), and to plan and track large projects (e.g., schedulers). Such tools must store source and object versions of programs, internal forms of programs (such as syntax trees and flow graphs), documents, test data, and project management information. The absense of appropriately powerful and flexible database facilities is a major problem for developers of these tools. Many regard it as the *most* serious impediment to qualitative improvements in CASE environments.

Much of the current research in database management is focused on repairing this inadequacy for design environments in general. Features being studied include: storage structures for large objects and for multiple versions of each object, multi-dimensional search structures, rich data typing, type inheritance, user-defined operators, more powerful set-oriented operators (such as transitive closure), long-duration transactions, nested transactions, triggers, and semantic integrity constraints. To support software engineering tools, a DBMS needs many, perhaps most, of these features.

We recently summarized our view of current research on these features, especially as they relate to CASE requirements, in [10]. An abridged version of this paper and its bibliography appears below, followed by a summary of our project to respond to these requirements.

2. Database Facilities for CASE

This section lists facilities that *should* be offered in a DBMS for CASE, but are not available in the right form in current relational DBMSs. For each facility, it briefly explains the desired functionality and why it is useful.

2.1. Storage Management

Large Objects

A DBMS for CASE must be able to store large variable-length objects, such as documents and programs. Some large storage objects that are today stored as a single unit should be decomposed into smaller pieces, to take full advantage of DBMS facilities. For example, one could store each of a program's procedures as a separate object, rather than storing all of the program's procedures together in a single object, as is typically done with file systems. Nevertheless, the requirement to store large objects is hard to circumvent in all cases.

Conventional relational DBMSs are designed to store small objects, namely records, and sets of small objects, namely files. Often the system has a small maximum length for either records or fields, which makes it impossible to store a large object as a single record.

File systems can store each large object as a file. However, opening a file is a rather expensive operation, which is mostly oriented to the needs of end users. A DBMS probably should circumvent the open operation's ordinary access control and name service functions.

Versioning

A CASE environment needs to store many

[1]Authors' address: Wang Institute of Graduate Studies, 72 Tyng Road, Tyngsboro, MA 01879-2099

versions of documents, programs, and other objects [32]. Tools for this purpose have been in widespread use for many years (e.g., SCCS [63] and RCS [78]). The main technical problem in designing such a tool is the tradeoff between storage space and execution time; more compact representations usually require more execution time to store and retrieve data.

However, if versioning is implemented in the DBMS's storage subsystem, then it can be done on fine granularity components, such as records or blocks. When creating a new version, only those components that have changed since the previous version need to be stored. In the new version, unchanged components can be represented by pointers to their previous versions. This is efficient in storage because only the changed components are repeated in the new version. It is also efficient in execution time; since versions are not encoded, they can be retrieved directly, instead of being reconstructed using change logs, as in RCS and SCCS.

Versions may also be correlated to time. The sequence of versions of an object may be thought of as the object's history. One may want to retrieve versions based on the time they were created (e.g., the versions of the dump utility that were created while Jones was running the utilities project).

Data Representation

The DBMS must cope with different representations of atomic types. The differences may be matters of machine architecture (e.g., byte-ordering or the representation of integers), programming language (e.g., representation of strings), or tools (e.g., representation of trees). The DBMS must know the representation of each type as it is stored and as it must be presented to its users. This knowledge is maintained by type management facilities, described in the next section. After the DBMS knows the source and destination representation of an object, it must translate between those representations. There are two issues here: *when* to perform the translation and *how* to do it efficiently. Experience has shown that the main efficiency consideration here is reducing memory-to-memory copying of data.

2.2. Type Management

Tools for a CASE environment need a full range of atomic and composite data types. Since tool developers want to share their data, they must be able to express these data types to the DBMS. Data types in current relational DBMSs are usually limited to (non-recursive) record types, each of which consists of a sequence of atomic fields. Union types, arrays, and nested structures are ordinarily not supported [45].

Some data is more conveniently represented in procedural form [49]. For example, the "length" attribute of a program could be calculated by a procedure, rather than stored explicitly. Limited forms of this facility have been available in database view mechanisms for many years. However, a view is a derived object, defined by a data dictionary entry; it is not a "base-level" object, physically stored in the database. Performance improvements may be possible by making procedural attributes base-level objects.

One important type that appears in many CASE applications is directed graphs [17]. For example, tools store parse trees, flow graphs, dependency graphs, and PERT charts. In today's DBMSs, one stores graphs by storing their edges in relations, and manipulates graphs using standard relational operators. This is unfortunate, because many fast algorithms for manipulating directed graphs are not easily expressed using relational data structures and operations.

The DBMS should check the integrity of objects relative to their type definition. However, when and where should this integrity be checked? It could be checked in the application's workspace, every time an object is updated. However, this could be quite slow. Alternatively, integrity could be checked whenever an updated object is passed back to the DBMS. This is more efficient, but allows users to manipulate invalid objects for some time before the integrity violation is caught.

Database objects may have integrity constraints that are more complex than can be expressed in the type definition language. For example, one may have a style checking program that determines whether a document is consistent with an organization's standards. To assist with this process, a DBMS may offer a *trigger* mechanism,. For example, one could define a trigger on document objects, which is activated by invoking the "check-in" operation, and causes the style-checker to be invoked.

Triggers can also be used for alerting. For example, a user U can check-out an object from the database for reading, and leave a trigger there in case someone else wants to check-out the same object for writing. The action part of the trigger simply sends a message to U.

2.3. Associative Access

A popular and important feature of virtually all DBMSs is the ability to retrieve data objects based on their contents. Content-based retrieval is valuable for many CASE tools: a debugger may want to find programs that modify variable

x; a configuration management tool may want to find modules that are checked out by a programmer who is on vacation; a project management tool may want to find unfinished modules that are blocking the completion of release 3.2. In today's record-oriented DBMSs (i.e., relational, network, and many inverted file systems), there are four main considerations in implementing content-based access.

Indexing and Clustering

The changing nature of hardware has changed performance tradeoffs Much work has been done in recent years on efficient access structures. There are now linear hashing algorithms that cope gracefully with file growth [27, 37, 39, 41]. There are variations of B-trees that use large and variable-sized nodes to exploit fast sequential disk access [40, 43]. And there are multi-attribute index structures that cluster data based on combinations of fields, so that data that is frequently retrieved by such combinations can be retrieved in fewer disk accesses [8, 9, 56, 57, 62, 67]. All of these mechanisms are potentially valuable for improving associative access to data in a CASE environment, but few of them have been incorporated in commercial DBMS products.

There have been so many new and useful indexing techniques developed in recent years that it may be too expensive for a DBMS vendor to supply a wide enough variety to suit most users. Therefore, researchers are considering DBMS architectures in which users can add their own. In this approach, the DBMS defines an access method interface. A user may implement an access method that conforms to this interface and register it with the DBMS.

Set-at-a-time Operators

A major innovation of relational DBMSs over their predecessors is support for set-at-a-time operators: project, select, join, and aggregation. The operators abstract iteration over all elements of a set (i.e., all tuples of a relation); the user provides a property over individual objects, and the operator retrieves all objects in the set that satisfy the property. One may want to encapsulate iteration over other structured collections of objects, such as graphs or two-dimensional objects in a plane. Relational operators are often inconvenient for this purpose. Like index structures, the range of possible operators may be too great for the vendor to build into the DBMS. Thus, it would be desirable if new operators could be added to a DBMS without the vendor's assistance. This entails giving (sophisticated) users the ability to program their own operators.

2.4. Transaction Management for CASE

It is inappropriate to regard each activity of an engineer in a CASE environment to be a transaction in the usual DBMS sense. Here, an engineer reads data into a workspace, and may operate on the contents of that workspace for many days. If the system fails during that period, it is unacceptable for all work on that workspace to be lost. Moreover, the work of two design engineers may not be serializable. They may work on shared objects that they pass back and forth in a way that is not equivalent to performing their work serially.

There are several ways to modify the notion of a transaction to suit the needs of CASE. One way is to ensure that every transaction is short -- so short that aborting a transaction is only a minor inconvenience. For example, checking-in or checking-out objects from a database satisfies this notion of transaction. Longer activities, such as fixing a bug, may consist of many transactions.

Higher level transaction notions may also be valuable [35]. A transaction to fix a bug may involve running transactions to check out certain programs, modify them, test them, and check them back into the database. The fix-a-bug transaction is therefore a long transaction within which smaller transactions are nested. Should one of the subtransactions fail (e.g., a certain modification to a program is found to be infeasible), one may want to abort the fix-a-bug transaction; or, one might want to push ahead by trying different subtransactions (i.e., a different approach to fixing the bug).

Many popular transaction recovery algorithms use a form of logging. The log keeps track of the operations that have executed so far. The recovery algorithm uses the log to obliterate the effects of aborted transactions, and to redo the effects of committed transactions whose updates are lost in a failure.

Today, the DBMS's logging mechanism is rarely offered directly to the DBMS user. However, it could be quite useful this way, e.g., so that an interactive tool could create a private log of its actions. It could use generic DBMS functions to append log entries, and could subsequently interpret them in a tool-specific way.

3. Our Project

We are currently building a DBMS to support the development and use of CASE tools. We intend to implement the DBMS to an industrial standard of reliability and documentation that will make it easily usable outside Wang Institute. Working with CASE researchers at

Wang Insitute, we will evaluate the effectiveness of the DBMS in satisfying the needs of CASE tool developers.

There are many projects to build new DBMSs in support of engineering design applications. However, the main goal of most of these projects is to produce a testbed for research into DBMS implementation. Such testbeds are needed if the field is to move forward. However, when building such a testbed, it is difficult to meet research goals and users' needs at the same time. We chose to reverse these priorities, focusing first on meeting users' needs, and secondarily on advancing the state-of-the-art.

In the user's view, the most important property of a DBMS is that it works. It must reliably perform its advertised functions and must not, under any circumstances, corrupt data. (Even commercial products have had bugs that irreparably corrupted user data.) To avoid such disasters, we decided to focus especially hard on system quality, devoting much attention to coding standards, testing, and documentation.

Simplicity

The major theme of our DBMS development is design simplicity: Wherever possible, we will strive to use a small number of abstractions and simple mechanisms to support those abstractions. To meet a given set of DBMS requirements, it is often more difficult to find simple system designs than complex ones. However, once found, a simple design has many benefits.

First, a simple design is easier to validate and implement, and the resulting implementation is easier to test. Given our focus on quality, this is an especially important advantage. This improves the productivity of our development group, and helps shorten the development time to produce a DBMS with a given level of functionality.

Second, the goals of modularity and information hiding are easier to reach in a simple design. Using information hiding, we can isolate data representations and algorithms used in each module. This enhances our ability to maintain and improve the implementation.

Third, by using relatively few mechanisms, and mechanisms that are relatively simple, we are better able to predict the performance of the system. This also helps us tune the system; bottlenecks are easier to locate (due to modularity), and once found, they are easier to repair (due to information hiding).

Overall, a simple design breeds reliability. Users will not entrust their data to a DBMS unless it is extremely reliable. The simpler the design, the more quickly we will be able to reach a high level of reliability.

This theme of simplicity of mechanism is analogous to that of reduced instruction set computers (RISC). We intend to use general-purpose low-level mechanisms that can be implemented with good and predictable performance. Tailoring the mechanisms to particular applications is left as a higher-level function. This gives a builder of an application (in our case, software engineering tool) the flexibility to tailor the mechanisms for the particular higher-level functions needed.

The Design

Borrowing a good idea from Exodus, we intend to use B-trees to store large objects, to identify sets of objects, to define an index on a set of objects, and to define versions of objects [15]. Using a single B-tree structure for all of these purposes is a good illustration of how we intend to reach our primary goal of design simplicity.

To support versioning of objects, we will use a modification of Easton's write-once B-trees [26]. Initially, we will use a relatively standard B-tree, modified using Easton's scheme. Later, we hope to modify the basic B-tree structure for better performance [40, 42, 43].

Status

We have implemented a B-Tree manager, a basic data dictionary facility, a lock manager[2], and a logging-based recovery manager. The B-Tree manager and data dictionary facility have been integrated, and thoroughly tested. We hope to release them this summer, along with our tool for regression testing and the test scripts themselves (so that others may conveniently modify and re-test the system). This summer, we intend to implement write-once B-trees, and to integrate the lock and recovery managers with the current (non-versioned) B-tree manager.

Due to the recent merger of Wang Institute with Boston University, the status of our research group is in doubt. We therefore do not have firm development plans for the system beyond the summer.

Acknowledgments

This paper was greatly influenced by the authors' discussions with Mark Ardis and Richard Fairley, and Russ Holden, and by the mountain of previous work cited in the bibliography. We also thank Michael Carey for his help in editting the paper for this issue.

[2]The lock manager was produced under the direction of Prof. Billy G. Claybrook.

Bibliography

1. Afsarmanesh, H., Knapp, D., McLeod, D., Parker., A. An Extensible, Object-Oriented Approach to Databases for VLSI/CAD. Proc. of the International Conference on Very Large Databases, August, 1985.

2. Ahlsen, M., Bjornerstedt, A., and Hulten, C. "OPAL: An Object-Based System for Application Development". *Database Engineering 8*, 4 (December 1985), 31-40.

3. Anderson, T., Lougenia, E., Jr., Earl F., and Maier, D. PROTEUS: Objectifying the DBMS User Interface. 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

4. Bancilhon, F., R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. Proc. ACM-SIGMOD Conf. on Management of Data, NY, 1986, pp. pp.16-52.

5. D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise. Genesis: a reconfigurable database management system. Tech. Rept. 86-07, Univ of Texas at Austin, March, 1986.

6. Batory, D.S., and Buchmann, A.P. Molecular Objects, Abstract Data Types and Data Models: A Framework. Proc. International Conference on Very Large Databases, August, 1984.

7. Batory, D.S., and Kim, W. "Modeling Concepts for VLSI CAD Objects". *ACM Transactions on Database Systems 10*, 3 (September 1985), 322-346.

8. J.L. Bentley. "Multidimensional search trees used for associative searching". *Communications ACM 18*, 9 (Sept 1975), 509-517.

9. J.L. Bentley. "Multidimensional binary search trees in database applications". *IEEE Transactions in Software Engineering SE-5*, 4 (July 1979), 333-340.

10. Bernstein, P.A. Database System Support for Software Engineering -- An Extended Abstract. Proc. 9th Int.'l Conf. on Software Engineering, Monterrey, CA, April, 1987.

11. P. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1986.

12. Brodie, M., Blaustein, B., Dayal, U., Manola, F., and Rosenthal, A. "CAD-CAM Database Management". *Database Engineering 7*, 2 (June 1984).

13. Buchmann, A.P. and Celis, C.P. An Architecture and Data Model for CAD Databases. Proc. International Conference on Very Large Databases, Singapore, August, 1985.

14. Buneman, P., and M. Atkinson. Inheritance and Persistence in Database Programming Languages. Proc. 1986 ACM-SIGMOD International Conference on Management of Data, Washington, DC, May, 1986, pp. 4-15.

15. Carey, M.J., DeWitt, D.J., Richardson, J.E., and Shekita, E.J. Object and File Management in the EXODUS Extensible Database System. Proc. of the Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, August 25-28, 1986, pp. 91-100.

16. M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E. Richardson, E.J. Shekita, and M. Muralikrishna. The architecture of the EXODUS extensible DBMS: a preliminary report. Tech. Rept. 644, Univ of Wisconsin-Madison, May, 1986.

17. Clarke, L.A., Wileden, J.C., and Wolf, A.L. Graphite: A Meta-tool for ADA Environment Development. Proc. Second International Conference on ADA Applications and Environments, Miami Beach, FL, April 8-10, 1986, pp. 81-90.

18. Cockshott, W., Atkinson, M., Chisholm, K., Bailey, P., and Morrison, R. "Persistent Object Management Systems". *Software--Practice and Experience 14* (), 49-71.

19. Copeland, G. and Maier, D. Making Smalltalk a Database System. Proc. 1984 ACM-SIGMOD International Conference on Management of Data, Boston, June, 1984, pp. 316-325.

20. Dayal, U., Buchmann, A., Goldhirsch, D., Heiler, S., Manola, F.A., Orenstein, J.A., and Rosenthal, A.S. PROBE: A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis. Technical Report CCA-85-03, Computer Corporation of America, July, 1985.

21. Linn, J.L. and Winner, R.I. (Editors). *The Department of Defense Requirements for Engineering Information Systems.* Volume 1: Operational Concepts edition, The Institute for Defense Analyses, Alexandria, VA, 1986.

22. Linn, J.L. and Winner, R.I. (Editors). *The Department of Defense Requirements for Engineering Information Systems.* Volume 2: Requirements edition, The Institute for Defense Analyses, Alexandria, VA, 1986.

23. Derrett, N., Kent, W., and Lyngbaek, P. "Some Aspects of Operations in an Object-Oriented Database". *Database Engineering 8* (December 1985), 66-74.

24. Dittrich, K.R., Gotthard, W. and Lockemann, P.C. DAMOKLES - A Database System for Software Engineering Environments. IFIP Workshop on Advanced Programming Environments (June, 1986). proceedings to appear.

25. Eastman, C.M. System Facilities for CAD Databases. Proc. IEEE 17th Design Automation Conference, June, 1980.

26. M. Easton. "Key-sequence data sets on indelible storage". *IBM Journal on Research Development 30,* 3 (May 1986), 230-241.

27. R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong. "Extendible hashing-a fast access method for dynamic files.". *ACM Transactions on Database Systems 4,* 3 (Sept 1979), 315-344.

28. Gray, M.. *Databases for Computer-Aided Design, New Applications of Databases.* Academic Press, 1984.

29. Habermann, A.N., and Notkin, D. "Gandalf: Software Development Environments". *IEEE Transactions on Software Engineering SE-12,* 12 (December 1986), 1117-1126.

30. Haskin, R.L. and Lorie, R.A. On Extending the Functions of a Relational Database System. Proc. ACM-SIGMOD International Conference on the Management of Data, 1983.

31. Katz, R. H.. *Information Management for Engineering Design.* Springer-Verlag, 1985.

32. Katz, R.H., Chang, E., and Bhateja, R. Version Modelling Concepts for Computer-Aided Design Databases. Proc. 1986 ACM-SIGMOD International Conference on Management of Data, Washington DC, May, 1986, pp. 379-386.

33. Kempf, J., and Synder, A. Persistent Objects on a Database. Software Technology Laboratory Report STL-86-12, Hewlett-Packard, September 23, 1986.

34. Ketabchi, M.A., Berzins, V., and March, S.T. ODM: An Object-Oriented Data Model for Design Databases. Proc. of ACM Annual Computer Science Conference, 1986.

35. Kim, W., Lorie, R., McNabb, D., and Plouffe, W. A Transaction Mechanism for Engineering Design Databases. Proc. 10th Int'l. Conf. on Very Large Data Bases, Singapore, August 27-31, 1984, pp. 355-362.

36. Lamersdorf, W., Schmidt, J.W., and Muller, G. "A Recursive Approach to Office Object Modelling". *Information Processing and Management 22,* 2 (March 1986), 109-120. Pergamon Press Ltd., Oxford UK.

37. P. Larson. Linear hashing with partial expansions. Proceedings 6th Conference on VLDB, Montreal, Canada, 1980, pp. 224-232.

38. LeBlang, D.B., and Chase, R.P., Jr. Computer-Aided Software Engineering in a Distributed Workstation Environment. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 23-25, 1984, pp. 104-112.

39. Litwin, W. Linear Hashing: A New Tool for File and Table Addressing. Proc. 6th Int.'l Conf. on Very Large Data Bases, Montreal, 1980, pp. 212-223.

40. Litwin, W. and Lomet, D.B. "A New Method for Fast Data Searches with Keys". *IEEE Software 4,* 2 (March 1987), 16-24.

41. Lomet, D.B. "Bounded index exponential hashing". *ACM Transactions on Database Systems 8*, 1 (March 1983), 136-165.

42. Lomet, D. A simple bounded disorder file organization with good performance. Tech. Rept. 86-13, Wang Institute, September, 1986. Submitted for Publication.

43. Lomet, D.B. "Partial expansions for file organizations with an index". *ACM Transactions on Database Systems 12*, 1 (March 1987), 65-84".

44. Lorie, R., Kim, W., McNabb, D., Plouffe, W., and Maier, A. Supporting Complex Objects in a Relational System for Engineering Databases. In *Query Processing in Database Systems*, Springer Verlag, Berlin, 1985.

45. Lorie, R. and Plouffe, W. Complex Objects and Their Use in Design Transactions. Engineering Design Applications, Proceedings of Annual Meeting, San Jose, May 23-26, 1983, pp. 115-121.

46. Maier, D., Otis, A.,and Purdy, A. "Object-Oriented Database Development at Servio Logic". *Database Engineering 8*, 4 (December 1985).

47. Maier, D., and Stein, J. Development of an Object-Oriented DBMS. Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), Portland, September 29 - October 2, 1986, pp. 472-482.

48. Maier, D., and Stein, J. Indexing in an Object-Oriented DBMS. 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

49. Manola, F., and Dayal, U. PDM: An Object-Oriented Data Model. 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

50. Marzullo, K., and Wiebe, D. Jasmine: A Software System Modelling Facility. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, CA, December 9-11, 1986, pp. 121-130.

51. Mishkin. Managing Permanent Objects. Technical Report YALEU/DCS RR-338, Department of Computer Science, Yale University, New Haven, CT, November, 1984.

52. J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, 1985.

53. Nestor, J.R. Toward a Persistent Object Base. In *International Workshop on Programming Environments*, Springer-Verlag, 1986.

54. Kierstrasz, O.M. "Hybrid: A Unified Object-Oriented System". *Database Engineering 8*, 4 (December 1985), 49-57.

55. Nierstrasz, O.M., and Tsichritzis, D.C. An Object-Oriented Environment for OIS Applications. Proc. 11th Int.'l Conference on Very Large Databases, Stockholm, 1985.

56. J. Nievergelt, H. Hinterberger and K. Sevcik. "The gridfile: an adaptable, symmetric multikey file structure". *ACM Transactions on Database Systems 9*, 1 (March 1984), 38-71.

57. J.A. Orenstein. "Multidimensional TRIEs used for associative searching". *Information Processing Letters 14*, 4 (June 1982), 150-157.

58. Orenstein, J. A., S. K. Sarin, U. Dayal. Managing Pesistent Objects in Ada: Final Technical Report. Computer Corp. of America, 4 Cambridge Center, Cambridge, MA 02142, May, 1986.

59. Plouffe, W., Kim, W., Lorie, R., and McNabb, D. "A Database System for Engineering Design". *Database Engineering 7*, 2 (June 1984).

60. Purdy, A., Maier, D., and Schuchardt, B. Integrating an Object Server with Other Worlds. Technical Report CS/E-86-013, Oregon Graduate Center, December 9, 1986. To appear in ACM Transactions on Office Information Systems, April 1987.

61. Riddle, W.E. and Williams, L.G. "Software Environments Workshop Report". *ACM SIGSOFT - Software Engineering Notes 11*, 1 (January 1986), 73-102.

62. J.T. Robinson. The k-d-B-tree; a search structure for large multidimensional dynamic indexes. Proceedings SIGMOD Conference on MOD, New York, 1981, pp. 10-18.

63. Rochkind, M. J. "The Souce Code Control System". *IEEE Trans. on Software Engineering 1*, 4 (Dec. 1975), 364-370.

64. Rowe, L.A. A Shared Object Hierarchy. 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

65. Rowe, L.A., and Shoens, K.A. Data Abstraction, Views and Updates in RIGEL. Proc. ACM-SIGMOD International Conference on Management of Data, 1979.

66. Sathi, A., Fox, M.S. and Greenberg, M. "Representation of Activity Knowledge for Project Management". *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-7*, 5 (September 1985), 531-552.

67. Scheuermann, P. and M. Ouksel. "Multidimensional B-trees for associative searching in database systems". *Information Systems 7*, 2 (1982), 123-137.

68. Schwarz, P., Chang, W., Freytag, J.C., Lohman, G., McPherson, J., Mohan, C., and Pirahesh, H. Extensibility in the Starburst Database System. 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

69. Shipman, D. "The Functional Data Model and Data Language DAPLEX". *ACM Transactions on Database Systems 6*, 1 (March 1981), 140-173.

70. Sidle, T.W. Weaknesses of Commercial Database Management Systems in Engineering Applications. Proc. IEEE 17th Design Automation Conference, June, 1980.

71. Skarra, A. H., and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. OOPSLA '86 Proceedings, 1986, pp. pp. 483 -495.

72. Skarra, A.H., Zdonik, S.B., and Reiss, S.P. An Object Server for an Object-Oriented Datbase System. 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

73. Spooner, D., Milicia, M., and Faatz, D. Modeling Mechanical CAD Data with Abstract Data Types and Object-Oriented Techniques. Proc. 2nd International Conference on Data Engineering, Los Angeles, February, 1986.

74. Stonebraker, M. and L.A. Rowe. The design of POSTGRES. Proc. SIGMOD Conf. on Management of Data, June, 1986, pp. 340-355.

75. Stonebraker, M., and Rowe, L.A. The Postgres Papers. Memorandum No. UCB/ERL M86/85, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, November 5, 1986.

76. Stonebraker, M., Rubenstein, B., and Guttman, A. Applications of Abstract Data Types and Abstract Indices to CAD Databases. Engineering Design Application Proceedings from SIGMOD Database Week, May, 1983.

77. Thatte, S.M. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. 1986 International Conference on Object-Oriented Database Systems, Pacific Grove, CA, September, 1986.

78. Tichy, W. F. "RCS - A System for Version Control". *Software Practice and Experience 15* (1985).

79. Wiederhold, G. "Views, Objects, and Databases". *Computer 19*, 12 (December 1986), 37-44.

80. Woelk, D., Kim, W., and Luther, W. An Object-Oriented Approach to Multimedia Databases. Proc. 1986 ACM-SIGMOD International Conference on Management of Data, Washington DC, May, 1986, pp. 311-325.

81. Zdonik, S.B. Object Management Systems Concepts. Proc. ACM/SIGOA Conference on Office Information Systems, 1984.

82. Zdonik, S.B. Language and Methodology for Object-Oriented Database Environments. Proc. Nineteenth Annual Hawaii International Conference on System Sciences, January, 1986.

Extensible Databases and RAD

Sylvia Osborn

Department of Computer Science

The University of Western Ontario

London, Ontario, Canada N6A-5B7

April 28, 1987

1 Definitions

The 1980s have seen a new generation of database management systems being developed. The main motivation for this activity is the need for database management system functionality in new application areas such as computer aided design, office information systems, pictorial and graphic databases, software engineering systems, etc. Many researchers are studying these problems, some looking at only parts of the problem while others are designing and building systems which will make up a new generation of database systems.

These new DBMS's can be classified into 3 categories: extensible database systems, object-oriented database systems and database system generators. We use the term *extensible database system* to refer to a DBMS which allows an applications programmer to add new data types and new operations on these new data types to an existing DBMS. These new types would be added to the system by an application programmer or data base administrator. In an extensible database system, the underlying DBMS will always be the same in such modules as concurrency control, recovery, basic storage, and query language. Extensible databases may also allow the addition of access methods and hooks to allow the query optimizer make use of these new access methods. Note that the new operations added usually pertain to operations on what would be the domains in the relational model, and that the major operators of the query language would not be modifiable. Note also that we are assuming that we already have a *database* system so that persistence of data and data structures from one instantiation of a program to another is already taken care of.

Persistence only becomes an issue if such a system is implemented by augmenting a traditional programming language. Some examples of extensible database systems are RAD, ADT-Ingres and POSTGRES [Osb86, Ston86a, Ston86b]. An *object oriented database system* is an extensible database system which incorporates a semantic data model, which in turn is sufficiently powerful to allow reasonably straightforward modelling of complex objects. Complex objects are objects which have a highly nested structure, like a large piece of software or an engineering design. They may also be very large. The semantic data model should be able to model such things as arbitrary levels of aggregation [Smith77], components of aggregates which may be sets of other objects or ordered sequences of other objects, and IS-A or generalization hierarchies with inheritance of object components and of operations. Some object-oriented database systems also model versions and knowledge. One should be able to define new operations on these objects, which could have the effect of changing the query interface. It is also reasonable to expect that the other pieces of the database management system work at the object level, for example that the objects are passed to the storage manager as a unit, that locking and recovery are done on a per-object basis, etc. Since this definition inherits all the properties of an extensible database system, object oriented database systems also allow the definition of new data types, operations on them and access methods. They will have the same underlying concurrency control methods, recovery etc. from one instantiation to another, although, being object oriented, these modules may differ quite a lot from those found in traditional DBMS's. Some examples of object oriented database systems are GemStone, Probe, and Iris [Cope84, Man86, Lyng86].

The third category of new systems is the *database system generators, customizers* or *compilers*. These systems allow a database system implementor or architect to design and implement a new type of database management system without having to write all the code for this new system from scratch. Using one of these database generators, one could generate different database management systems which differ in virtually all of their modules. The resulting system could be a traditional (non-extensible) DBMS, an extensible one or an object-oriented one. Examples of database generators are EXODUS, the Data Model Compiler and GENESIS [Carey86, Mary86, Bat86].

All three kinds of new database systems highlight some new roles for the people who use them. With traditional DBMS's, database administrators designed the data structures for an application, having a good understanding of the data model and its use. Application programmers used these structures to carry out tasks which were either non-trivial or not possible using the interactive

query language. End users were not usually expected to be programmers, but would use some transactions set up by an applications programmer, or would express queries in an interactive query language such as SQL. With these new systems, we will also have these roles, but there will be some new roles to be played as well. All three of these new types of systems require code to be written and attached to the database system. For the extensible systems the code for the operations on abstract data types and new access methods is required. For object-oriented systems, code is needed for the operations on objects, which may be more complicated than that for an abstract data type. This code should be written by an experienced programmer, but not someone who has implemented a DBMS. Another challenge is to find ways for the end users, who are professionals, say, like engineers, to specify non-trivial operations unanticipated when the operations were defined for the complex object, the way end users of current relational systems can specify things in SQL. The database compilers provide the greatest challenge. They may require a data model designer (how many people have ever designed successful data models?), a database system architect, etc. One of the goals of these systems is to provide tools to make these tasks simpler.

2 RAD

RAD (which stands for relations with abstract data types) was conceived in the early 1980s as an attempt to address the problems of these challenging application areas, using a simple extension to the relational model, namely allowing the definition of new domains [Osb86]. The application programmer is allowed to issue a CREATE DOMAIN statement which registers a new domain with the relational database. Along with this goes some code for managing the bytes which these domain values will occupy. These primitive operations include inserting new values, outputing values, updating values, constant validation, and testing for equality and comparisons. The definition of an arbitrary number of predicates involving an arbitrary number of parameters is also allowed. RAD also has a method for defining aggregates on columns of these new data types and for defining arbitrary transformations. Aggregates provide a way of mapping a relation onto a single value of any type. A transformation maps a relation to a relation, thus providing a very general tool for, say, transforming images from one format to another, or coding such things as the group-by construct of SQL or the transitive closure operation of QBE.

A single user version of RAD was partially implemented and a small experiment was run involving four student projects: a menu-driven calendar manager, a random quiz generator for a

first-year programming course, a drug information system for a hospital pharmacy giving doses, side effects, other names for medications, etc., and a system for a real estate agent which matches descriptions of houses in the database with the requirements of a customer. Two versions of each project were implemented: one as a stand-alone program, and one using RAD. Some problems were encountered because RAD was not fully debugged. The general consensus was, however, that the RAD version took significantly less time to develop, involved less code, and would be easier to modify and extend because it involved a more general approach and was shorter. Further development on RAD has been abandoned for reasons which should become apparent below.

RAD and systems like it are not suitable for modelling really complex objects. The reason for this is the absence of a semantic data model, i.e. because they are *not* object-oriented. Let us consider for a moment a software engineering environment. A software project consists of designs, requirements, programs, modules, abstract data types, submodules, procedures, authors of all of these things, which pieces go with each other in what ways, and so on. It would be possible to define RAD domain types for designs, requirements, programs, etc., which would then be associated with their authors, submodules, etc. by appropriate relations.

The trick with a system like RAD, is to pick the right place in the complex structure to define the domains. One could define a domain for the whole software project, which would tend to force the implementation of a semantic data model in a single domain. One can flatten the whole structure into atomic domains – the smallest units one ever needs to talk about, which could be less than a line of code in one of the programs – and construct a complex set of first normal form relations to describe the projects. Since the relational model only provides the operations on domain values, and the relational operators on relations, the first choice means that we can not apply the relational operators on parts of the software project (for example "find all modules written by J. Smith"). The second choice means that when we want to look at a significant portion of a project, the data for it will be in dozens of relations. Although RAD would allow us to define a transformation to "gather up all the modules written after Jan. 1, 1985", this transformation would be very costly and inefficient. Any compromise between these two extremes will be very difficult to choose and will still suffer some of the drawbacks of the two extreme solutions. If one could use a semantic data model insead of the relational model, there are probably more than two object classes to use to model a complex application. As well, type constructors other than the flat tuple would be available, say to allow a set or sequence of modules as a component of a program.

Relational systems also lack an inheritance mechanism. Although inheritance is not ideal

[Snyd86], it is a very useful tool for managing complexity. The applications for which these systems are intended are very large and complex. Inheritance of components of an object class by its subclasses is a natural way to model many applications and is becoming well understood. Inheritance of operations on objects can provide an economical implementation of a complex set of activities. It seems that for any of these new DBMS's to be really successful in supporting these application areas, it must include inheritance in its data model. It is difficult to add inheritance to the relational model because, as it stands, the relational model only allows one to declare relation instances, not types. Inheritance is usually defined between object classes or data types. Even Codd found he had to add the concept of an entity *type* to the relational model before talking about inheritance [Codd79].

RAD was very successful in providing a simple mechanism whereby a programmer could define new domain types. The four students who took part in the experiment were in their senior undergraduate year, so they could barely be called professional programmers and they certainly had never implemented a database system, nor did they know the internals of the RAD implementation. In spite of the problems they had because of bugs and a lack of good manuals, they all were able to write the code necessary to implement the abstract data types and operations required for their application and link it in to the database management system.

3 References

Bat86 Batory, D.S. GENESIS: A Project to Develop an Extensible Database Management System. Proc. 1986 International Workshop on Object-oriented Database Systems, Sept. 1986, 207-208.

Carey86 Carey, M. et al. The Architecture of the EXODUS Extensible DBMS. Proc. 1986 International Workshop on Object-oriented Database Systems, Sept. 1986, 52-65.

Codd79 Codd, E.F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (1979), 397-434.

Cope84 Copeland, G. and Maier, D. Making Smalltalk a database system. Proc. 1984 ACM-SIGMOD International Conference on Management of Data, Boston, Ma. June, 1984, 316-325.

Lyng86 Lyngbaek, P. and Kent, W. A Data Modeling Methodology for the Design and Implementation of Information Systems. Proc. 1986 International Workshop on Object-oriented Database Systems, Sept. 1986, 6-17.

Man86 Manola, F. and Dayal, U. PDM: an Object-oriented Data Model. Proc. 1986 International Workshop on Object-oriented Database Systems, Sept. 1986, 18-25.

Mary86 Maryanski, F. et al. The Data Model Compiler: A tool for generating object-oriented database systems. Proc. 1986 International Workshop on Object-oriented Database Systems, Sept. 1986, 73-84.

Osb86 Osborn, S.L. and Heaven, T.E. The design of a relational database system with abstract data types for domains. *ACM Trans. Database Syst. 11*, 3 (1986), 357-373.

Smith77 Smith, J.M. and Smith, D.C.P. Database abstractions: Aggregation. *Communications ACM 20*, 6 (1977), 405-414.

Snyd86 Snyder, A. Encapsulation and Inheritance in Object-oriented Programming Languages. OOPSLA '86, *Sigplan Notices 21*, 11 Nov. 1986, 38-45.

Ston86a Stonebraker, M. Inclusion of new types in relational data base systems. Proceedings 1986 International Conference on Data Engineering, Los Angeles, Ca. Feb., 1986, 262-269.

Ston86b Stonebraker, M. and Rowe, L. The Design of POSTGRES. Proc. 1986 ACM-SIGMOD International Conference on Management of Data, Washington, D.C., May, 1986, 340-355.

EXTENDABILITY IN POSTGRES

Michael Stonebraker, Jeff Anton, and Michael Hirohama

Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

This paper presents the mechanisms being designed into the next-generation data manager POSTGRES to support extendability. POSTGRES supports user defined data types, operators, functions, aggregate functions and access methods. This paper sketches the specification and implementation of each of these constructs.

1. INTRODUCTION

POSTGRES is a next-generation data manager being built at Berkeley. Although formally a relational DBMS because duplicate tuples are not allowed in any relation, the system contains substantial new facilities to support procedural objects, rules, versions, inheritance, and user extendability. The POSTGRES data model is discussed in [ROWE87], the rules system is described in [STON87a], and the storage system is presented in [STON87]. A (now obsolete) overview of the system appears in [STON86a], and a preliminary description of the extendable type system is contained in [STON86b]. In turn this builds upon the original paper on the subject [STON83]. Although much of the material in the current paper previously appeared in [STON86b], extendable aggregates are new, and the other mechanisms have changed somewhat. Hence, this paper should be viewed as an update on POSTGRES extendability and an indication of the status of the implementation in April 1987.

POSTGRES contains facilities for extendability in five areas, namely functions, data types, operators, aggregates, and access methods. In the next five sections we discuss each area. Then, in Section 7 we make some general comments and indicate the status of the code.

2. EXTENDABLE FUNCTIONS

POSTGRES allows the query language to be extended with new functions which can be user defined. For example, if ``arrogance'' has been previously registered as such a function taking an integer argument and returning a floating point number, then the following POSTQUEL command becomes permissable:

retrieve (EMP.name) where arrogance (EMP.salary) > 5.3

Another function, high-arrogance, could return a boolean and allow the following expression:

retrieve (EMP.name) where high-arrogance (EMP.salary)

To register a new function, a user utilizes the following syntax:

DEFINE <cachable> PROCEDURE proc-name (type-1, type-2, ..., type-n)
RETURNS type-r
LANGUAGE IS {C, LISP}
FILE = some-file

This research was sponsored by the Naval Electronics Systems Command Contract N00039-84-C-0039.

Basically, the implementor of the function indicates the input data types and the return data type. Then, he indicates the language in which the function is coded, and the current options are C and LISP. Other than putting a shell script for the compiler in a particular system catalog relation and ensuring that routines written in the new language can be called from C, there is no other difficulty to adding new languages. The source code for the function appears in the indicated file, and the ``cachable'' flag indicates whether the function can be precomputed. Since query language commands can be stored in the data base and POSTGRES is planning to cache the results of such procedures, it must know whether a function used in such a query language statement can be executed before the user specifically requests execution. Most functions, such as arrogance and high-arrogance, can be evaluated earlier than requested; however some functions such as time-of-day cannot. The cachable flag is present to indicate which option is appropriate for the function being defined.

POSTGRES accepts the source code in the indicated file and compiles it for storage in a system catalog. Then, the code is dynamically loaded when necessary for execution. Repeated execution of a function will cause no extra overhead, since the function will remain in a main memory cache. Functions can be either called in the POSTGRES address space or a process will be forked for the function and a remote procedure call executed. The choice of **trusted** or **untrusted** operation is controlled by the data base administrator of the data base in question who can set the **trusted** flag in the same system catalog.

Lastly, POSTGRES automatically does type-checking on the parameters of a function and signals an error if there is a type mismatch. Hence

retrieve (EMP.name) where arrogance (EMP.name) > 5.0

results in an error because arrogance does not accept a character string argument.

3. EXTENDABLE DATA TYPES

POSTGRES contains a collection of built-in data types which are required to define the system catalogs. These include many of the standard ones such as integers and character strings. In addition, a user can define new data types of the following forms:

new base data types
arrays of base data types
procedural data types
parameterized procedural data types

The last three kinds of types are described in [ROWE87]; hence we concentrate only on the first one in this paper.

A user can create a new base data type using the following syntax:

```
DEFINE TYPE type-name (
LENGTH = N,
DEFAULT = ``string'',
INPUT = proc-name-1,
OUTPUT = proc-name-2,
SEND = proc-name-3,
RECEIVE = proc-name-4,
<by-value>)
```

Data types can be fixed length, in which case LENGTH is a positive integer or variable length, in which case LENGTH is negative and POSTGRES assumes that the new type has the same format as the POSTGRES data type, **text**, which is an arbitrary length byte string containing its length as the first 4 bytes. A default value is optionally available in case a user wants some specific bit pattern to mean ``data not present''. The first two registered procedures are required to convert from ascii representation to the internal format of the data type and back. The next two registered procedures are used when the application program requesting POSTGRES services resides on a different machine. In this case, the machine on which POSTGRES runs may use a different format for the data type than used on the remote machine. In this case it is appropriate to convert data items on output to a standard form and on input from the standard format to the machine specific format. The send and receive procedures perform these functions. The optional by-value clause indicates that operators and functions which use this data type should be passed an argument by value rather than by reference.

An example use of the above syntax is now shown to create the BOX data type:

DEFINE TYPE box (LENGTH = 8,
INPUT = my-procedure-1,
OUTPUT = my-procedure-2)

A user can then create relations using the type. For example, the following relation contains box descriptions along with their associated identifier.

create MYBOXES (id = integer, description = box)

4. USER DEFINED OPERATORS

For existing built-in types as well as for user defined data types, a user can define new operators with the following syntax:

DEFINE OPERATOR opr-name-1 AS proc-name(
PRECEDENCE = number,
ASSOCIATIVITY = string,
COMMUTATOR = opr-name-2,
NEGATOR = opr-name-3,
HASHES,
SORT = opr-name-4,
RESTRICT = proc-name-2,
JOIN = proc-name-3)

For example, the following syntax defines a new operator, area-equal, for the BOX data type:

DEFINE OPERATOR AE AS my-procedure-1(
PRECEDENCE = 3,
ASSOCIATIVITY = yes,
COMMUTATOR = AE,
NEGATOR = ANE,
HASHES,
SORT = ALT,
RESTRICT = my-procedure-2,
JOIN = my-procedure-3)

Here, AE is defined by the registered procedure, my-procedure-1. The precedence level is used to resolve non-parenthesized expressions, e.g.:

5 ** MYBOXES.description AE MYBOXES2.description

Here ** is a second operator and might be the one with higher precedence and therefore applied in preference to AE.

The next several fields are for the use of the query optimizer. The associativity flag is used to denote that multiple instances of the operator can be evaluated in any order. For example, consider the area-intersection operator, ^A^, and the following expression:

MYBOXES2.description ^A^ "0,0,1,1" ^A^ MYBOXES.description

The associativity flag indicates that

(MYBOXES2.description ^A^ "0,0,1,1") ^A^ MYBOXES.description

is the same as

MYBOXES2.description ^A^ ("0,0,1,1" ^A^ MYBOXES.description)

The commutator operator is present so that POSTGRES can reverse the order of the operands if it wishes. In the case of AE, the order of operands is irrelevant. However, with the operator ALT, one would have a commutator operator, area-greater-than, AGT. Hence the query optimizer could freely convert:

``0,0,1,1'' ALT MYBOXES.description

to

MYBOXES.description AGT ``0,0,1,1''

This allows the execution code to always use the latter representation and simplifies the access method interface somewhat.

The negator operator allows the query optimizer to convert

not MYBOXES.description AE ``0,0,1,1''

to

MYBOXES.description NAE ``0,0,1,1''

The next two specifications are present to support the query optimizer in performing joins. POSTGRES can always evaluate a join (i.e. processing a clause with two tuple variables separated by an operator that returns a boolean) by iterative substitution [WONG76]. In addition, POSTGRES is planning on implementing a hash-join algorithm along the lines of [SHAP86]; however, it must know whether this strategy is applicable. For example, a hash-join algorithm is usable for a clause of the form:

MYBOXES.description AE MYBOXES2.description

but not for a clause of the form

MYBOXES.description ALT MYBOXES2.description.

The HASHES flag gives the needed information to the query optimizer concerning whether this strategy is usable for the operator in question.

Similarly, the merge-sort operator indicates to the query optimizer whether merge-sort is a usable join strategy. For the AE clause above, the optimizer must sort the relations in question using the operator, ALT. On the other hand, merge-sort is not usable with the clause:

MYBOXES.description ALT MYBOXES2.description

If other join strategies are found to be practical, POSTGRES will change the optimizer and run-time system to use them and will require additional specification when an operator is defined. Fortunately, the research community invents new join strategies infrequently, and the added generality of user-defined join strategies was not felt to be worth the complexity involved.

In a similar vein, there has been considerable interest in adding operators, such as outer join and transitive closure, to a DBMS which do not readily fit the paradigm described above. POSTGRES takes the position that the number of such operators is quite small and they can be safely added to the query language and hardcoded into the optimizer. If new ones are desired, the syntax can be extended and the optimizer changed. Other groups (e.g. [CARE86]) have reached a different conclusion on these issues.

The last two pieces of the specification are present so the query optimizer can estimate result sizes. If a clause of the form:

MYBOXES.description ALT ``0,0,1,1''

is present in the qualification, then POSTGRES may have to estimate the fraction of the tuples in MYBOXES that satisfy the clause. The procedure, my-procedure-2, must be a registered procedure which accepts one argument of the correct data type and returns a floating point number. The query optimizer simply calls this function, passing the parameter ``0,0,1,1'' and multiplies the result by the relation size to get the desired expected number of tuples.

Similarly, when the operands of the operator both contain tuple variables, the query optimizer must estimate the size of the resulting join. The procedure, my-procedure-3, will return another floating point number which will be multiplied by the cardinalities of the two relations involved to compute the desired expected result size.

The difference between the function

my-procedure-1 (MYBOXES.description, ``0,0,1,1'')

and the operator

MYBOXES.description AE ``0,0,1,1''

is that POSTGRES attempts to optimize operators and can decide to use an index to restrict the search

space when operators are involved. However, there is no attempt to optimize functions, and they are performed by brute force. Moreover, functions can have any number of arguments while operators are restricted to one or two.

5. AGGREGATE OPERATORS

POSTGRES also supports the inclusion of user defined aggregate operators. Traditional systems hard code aggregates like sum, average, count, etc., and therefore cannot handle a new aggregate such as median or 2nd largest. POSTGRES, on the other hand, allows a user to define new aggregates. Syntactically, POSTQUEL supports the notion of a collection of tuples and can compare sets of tuples for equality. The following query illustrates this construct by testing whether the toy department exists on all floors:

```
retrieve (true = 1)
where {DEPT.floor} == {DEPT.floor where DEPT.dname = ``toy''}
```

Such collections of tuples are indicated by the brace notation {...}. The mechanism chosen for aggregates is to allow them to be a function with one argument which is a collection of tuples. Hence, the following expression computes the average salary of the shoe department:

```
retrieve (comp = avg{EMP.salary where EMP.dept = ``shoe''})
```

The notion of an aggregate function, expressed clumsily in SQL and QUEL, can be more elegantly expressed in POSTQUEL. The following command computes the average salary for each department:

```
retrieve (EMP.dept,
      comp = avg{E.salary from E in EMP where E.dept = EMP.dept})
```

In the former case, the {...} evaluate to a single collection of tuples, each with a salary. In the second case, the {...} evaluates to a collection of collections of salaries, one for each value of EMP.dept. POSTGRES will transform the expressions above to the internal form currently used for aggregates in INGRES [EPST79] and will apply the standard aggregate processing techniques developed previously. Hence, a user defined aggregate, e.g:

```
retrieve (EMP.dept,
      comp = foobar{E.salary using E in EMP where E.dept = EMP.dept})
```

must consist of two functions, a **state transition** function, T of the following form:

```
T( internal-state, next-salary) ---> next-internal-state
```

and then a **final calculation** function, C:

```
C(internal-state) ---> aggregate-value
```

For example, the average aggregate consists of a state transition function which uses as its state the sum computed so far and the number of values seen so far. It accepts a new salary and increments the count and adds the new salary to produce the next state. The state transition function must also be able to initialize correctly when passed a null current state. The final calculation function divides the sum by the count to produce the final answer. A second example is the aggregate 2nd largest which must have a state consisting of the current two largest values. It uses a state transition function which accepts a new value and returns the current two largest by discarding either the new input or one of the two elements in the current state. The final calculation function outputs the smaller of the two objects in the state.

Consequently, an aggregate is defined in the following way:

```
DEFINE AGGREGATE agg-name AS state-transition-function, final-calculation-function
```

These functions are required to have the following properties:

1) the output of state-transition-function and the input of final-calculation-function must be the same type, S.

2) the output of final-calculation-function can be of arbitrary type.

3) the input to state-transition-function must include as its first argument a value of type S. The other arguments must match the data types of the objects being aggregated.

6. EXTENDABLE ACCESS METHODS

It is expected that sophisticated users will be able to customize POSTGRES by adding their own access methods. Because we expect users to add access methods much less frequently than operators, functions, data types, or aggregates, we have not attempted to define a user-friendly syntax for this task. Hence, the reader should simply note that this task is intended to be performed by an expert. In this section we will explore the steps to integrating a new access method into POSTGRES and then discuss the steps a user must go through to use an access method defined by someone else.

Access methods are used only for secondary indexes in POSTGRES, and there is a POSTGRES supplied storage manager which handles data records. Hence, an access method stores a collection of keys and a pointer to a tuple which is stored by the storage manager. In addition, to assist the rule manager each access method must store a collection of rule locks. Consequently, all access methods must manage records that are variable length. In addition, POSTGRES also assumes that any access method is willing to store fixed or variable length keys. Collections of records managed by an access method will be called relations since POSTGRES treats them in the same way as collections of data records.

It is generally expected that an access method will choose to use the POSTGRES buffer manager. If so, the access method implementor must call POSTGRES supplied routines to read and write pages in the buffer pool and to pin and unpin buffer pages. This is a standard interface with the buffer manager, and those four procedures are registered so this step is straightforward. In addition to being aware of buffer management, the access method implementor must implement any required locking for objects in his index. He can use the POSTGRES lock manager or code one of his own choosing. If he does the latter, there may be deadlocks between his lock manager and the POSTGRES lock manager which are undetectable. Lastly, the access method implementor must specify any required ordering on the times that particular index pages are forced to stable memory. Such orderings may be required to preserve the consistency of the index when failures occur, and are supported by a call to a last buffer manager routine. Because of the POSTGRES storage architecture there is no need to write any other crash recovery functions. Although these interfaces are somewhat complicated, the access method implementor is assumed to be an expert and capable of mastering them.

The first step for the access method implementor is to write a collection of procedures that correspond to a POSTGRES defined access method interface which includes routines to open a relation, close a relation, get a unique tuple by key, get the next tuple in a scan, insert a new tuple, delete a tuple, begin a scan of a relation, end a scan of a relation, mark the position in a scan of a relation, and restore a previously marked position. The specifications of these routines are somewhat lengthy, and we restrict ourselves in this paper to making a few comments. First, there is no replace function because access methods are used only for secondary indexes and updating a secondary index always requires deleting a tuple and reinserting it somewhere else. It should also be noted that an access method implementor who adheres to the page layout of the access methods built by the POSTGRES designers (so far just B-trees; however R-trees [GUTM84] are planned), can utilize all the B-tree routines except insert, delete, and get-next. Hence, there is much less code to write in this case.

Next, all access method routines are called with a ``relation descriptor'' as one argument. This descriptor indicates the type of each field and its length and position. Hence, each access method routine is passed complete information about the tuple it will be manipulating. Lastly, the access method must be coded with a collection of **strategies** in mind. For example a B-tree access method can support efficient access for clauses of the form

relation.key operator value

where operator is one of

{=, <=, <, >, >=}

Hence, five strategy numbers must be assigned, one for each such operator.

The implementor of the access method must register each of his access method routines as a procedure and then enter the access method into an **access method** system catalog. This catalog contains the name of the access method, the name of each procedure to use, and a comment field indicating how many strategies there are and what each intuitively means. This information is used by the others to assist them in binding new operators to an existing access method as now discussed.

The next step is for the implementor or another user to define an **operator class** containing an operator for some or all of the strategies. Each operator in this class would be entered in another access method catalog, indicating its class, the access method to use, the strategy number for the operator, and two additional procedures. The first has the same meaning as the ``restrict'' procedure in the operator definition. It will return the expected fraction of the records which satisfy the clause

relation.key operator value

The second operator returns the expected number of pages examined in evaluating a clause of the above form. These two procedures are used by the query optimizer in the obvious way.

For example, by making direct entries into this system catalog, the class, area-ops can be defined containing the operators:

AE, ALT, ALE, AGT, AGE

The last step in utilizing an access method is to create the actual index. An index for the MYBOXES relation using the class, area-ops, would be defined as follows:

DEFINE INDEX box-index ON MYBOXES (description) using B-tree (area-ops)

After this step, the index is ready to use and is automatically kept consistent with the corresponding data records by POSTGRES.

7. DISCUSSION

There are several comments to be made at this time. First, no attempt has been made to allow POSTGRES to have user defined storage managers. One of the goals of POSTGRES is to remove all crash recovery code from the system by not overwriting data records. Hence, the storage manager is quite complex and uses coding techniques to differentially encode records. It was felt that allowing users to define alternate storage managers was a task far to complex to attempt.

However, users can create new storage managers external to POSTGRES. The mechanism is to define a data type and then store the actual data externally and have the POSTGRES value of each item simply be a pointer to the object stored elsewhere. For example, a document data type can use this technique to store the actual bytes in each document in an external file. The conversion routines discussed earlier in the paper and all the operators which use the document data type must be coded carefully to use the POSTGRES supplied value as a pointer to an external location. The only missing function in this utilization of the data type facility is recovery of document data in case of a transaction failure. To support this capability, POSTGRES would have to pass begin and end transaction statements to such managers so they can provide crash recovery for their objects. This is a straightforward step which we may attempt if there is sufficient demand.

Another comment concerns performance. In some applications one wants to directly call the access methods and is unwilling to pay any extra overhead for a high level data management system. In such applications POSTGRES allows a user to run queries such as the following:

retrieve (result = get-unique (tuple-id))

Although some of the parameters have been purposely omitted for simplicity, this illustrates that a user can directly call access method functions and incur very low overhead. To further reduce overhead, POSTGRES will special case queries such as these to go directly to the low level execution routines and bypass intermediate layers of the data base system. With this architecture in which low level routines are functions that are semantically identical to user defined functions, very high performance appears possible.

At the current time, POSTGRES supports user defined access methods (although only one has been written by us), user defined data types, and user defined operators. The parser and query optimization routines are all operational. The only restriction is that all the operator code must be preloaded, since our dynamic loader is not yet running. Moreover, the remote procedure call interface has not been started. Execution of user defined functions and aggregates is not yet possible; implementation of the former is nearly working, but aggregates have not been started. We expect to have a complete system with all capabilities we have defined toward the end of this year.

REFERENCES

[CARE86] Carey, M., et. al., "The Architecture of the EXODUS Extensible DBMS," Proc. International Workshop on Object-Oriented Database Systems, Pacific Grove, Ca., September 1986.

[EPST79] Epstein, R. and Ries, D., "Aggregate Processing in INGRES," Electronics Research Laboratory, University of California, Technical Report, M79-42, October 1979.

[GUTM84] Gutman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[ROWE87] Rowe, L. and Stonebraker, M., "The POSTGRES Data Model," (submitted for publication).

[SHAP86] Shapiro, L., "Join Processing in Database Systems with Large Main Memories," ACM-TODS, Sept. 1986.

[STON83] Stonebraker, M., et. al., "Application of Abstract Data Types and Abstract Indexes to CAD Data," Proc. Engineering Applications Stream of 1983 Data Base Week, San Jose, Ca., May 1983.

[STON86a] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON86b] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Engineering, Los Angeles, Ca., Feb. 1986.

[STON87a] Stonebraker, M. et. al., "The Design of the POSTGRES Rules System," Proc. 1987 IEEE Data Engineering Conference, Los Angeles, Ca., Feb. 1987.

[STON87b] Stonebraker, M., "The POSTGRES Storage System", (submitted for publication).

[WONG76] Wong, E., and Youseffi, K., "Decomposition; A Strategy for Query Processing," ACM-TODS, Sept. 1976.

Extensibility in the PROBE Database System

David Goldhirsch
Jack A. Orenstein

Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142

1. Introduction

It is widely recognized that existing database systems do not address the needs of many "non-traditional" applications such as geographic information systems (GISs), solid modeling and VLSI design. The underlying data models, query languages, and access methods were designed to deal with simple data types such as integers and strings, while the new applications are characterized by spatial data, temporal data, and other forms of data having both complex structure and semantics. Performance is likely to be a problem because of the mismatch between the requirements of the application and the capabilities of the database system. (For a more complete discussion of the motivations for the PROBE project and the range of issues addressed see [DAYA85].)

In response to these problems, several new database system architectures have been developed over the past ten years. These systems provide different answers to the following questions: What extensions should be added to a database system? How should these extensions be incorporated? These questions require answers at the implementation level and at the level of the data model and query language. For example, in GIS and CAD applications, the "convex hull" operation is important. How can the database system be augmented with a convex hull subroutine? What construct in the data model represents the convex hull subroutine, and how does the subroutine get invoked from the query language? In this paper we present a framework into which previously described systems can be placed, and then discuss how extensibility is achieved in the PROBE database system.

2. Extensibility in PROBE

Work on the problem of extending database system capabilities has led to three strategies:

Use an Existing Database System: One approach to extending database system functionality is to embed all the new functionality in the application program as in [CHOC84, SMIT84]. This requires the application programmer to map the types and operations of the application into those of the database system. The mapping of representations is usually not difficult. For example, any representation of spatial data can usually be mapped onto a relational schema without difficulty. However, it is often difficult or impossible to formulate even the simplest queries (for example, given some relational schema that stores the vertices and edges of polygons, try writing a query that retrieves all polygons overlapping a given polygon).

Providing adequate performance is another problem with this approach. The data structures and access paths provided by the database system may not be suitable for the data types being represented, and the DDL may not provide the means to specify the desired physical organization. Also, the complexity of the queries will probably cause the optimizer to miss many opportunities for improvement.

Add a 'Hardwired' Extension: In this plan, application-specific features are explicitly added to an otherwise general-purpose data model. Such extensions have been proposed for image data [CHAN81, LIEN77], text retrieval [SCHE82, STON82], geographic information processing [AUTO87, IEEE77, MORE85], and VLSI design [KATZ85].

The problem with this approach is that, in each case, the specific extensions added are application-specific and limited in generality. For example, the spatial capabilities required for geographic data would be, at best, of limited usefulness in a mechanical CAD application. Moreover, even for a single type of data, e.g., geographic data, there are many ways to represent and manipulate the data, and each way may be the best in some specific application. It does not seem possible to select one approach to build in and maintain generality. At the same time, it is clearly impossible to provide all useful approaches in the same database system.

Use an Extensible Database System: The problem common to the strategies described above is that database systems do not provide, in their query languages or architectures, facilities for the addition of extensions. *Object-oriented* or *extensible* database provide this feature explicitly. The architecture of an extensible database sytem consists of these two components:

1. A *database system kernel*: A query processor designed to manipulate objects of arbitrary types, not just the numeric and string types of conventional database systems.
2. A collection of *abstract data types* or *object classes* (we will use the latter term): The object class specifies the representation of objects of a new type, and provides operations for manipulating objects of the type.

The difference between an EDB and the hardwired approach is that an extensible database system is much simpler to extend, because the interface between the database system and the extension is clearly defined. This also facilitates the handling of multiple extensions.

However, the process of adding an object class to an EDB is not trivial. The designers of EXODUS describe their system as a *database generator* and discuss the need for a *database implementer* (DBI) -- a person who creates a database system for an application by adding object classes to EXODUS [CARE86]. This is an accurate view of how any EDB would be used, and the PROBE and POSTGRES [STON86] systems would also benefit from the services of a DBI, since this process of customization requires expertise beyond what could reasonably be expected of an application programmer. The other duties of a DBI will depend on the particular EDB being customized.

In designing PROBE (as in the design of any EDB system), it was necessary to specify an exact *division of labor* between the database kernel and the object classes. We believe that the database system should provide the operations of some data model, e.g., *select* and *project*, suitably generalized for dealing with collections of objects of the most generic types. It should therefore be the responsibility of each object class to implement specialized operations on individual objects. Note that for reasons similar to those that led us to the extensible database system approach, this division of labor corresponds well with a desirable division of labor among implementors -- application specialists (who supply the object classes) will not be concerned with database system implementation issues (e.g., relating to the management of collections of objects), while the EDB implementers do not have to consider application-specific issues (that might

limit the generality of the system).

The PROBE architecture (as shown in figure 1) reflects both this basic paradigm and a need to efficiently serve certain kinds of applications. To support basic extensibility, the data model supports the notion of generalization. At the root of the hierarchy is the most generic type, ENTITY. Below this are types with more semantics such as numeric types, string types and spatial types. Application-specific types can be added. To efficiently support spatial and temporal applications, a generic PTSET (point set) type was included. This type provides a very general (yet optimizable) notion of both spatial and temporal data (see [MANO86a]).

This architecture is consistent with our chosen division of labor: the database system kernel handles sets of generic objects, while the application-specific object classes handle individual objects of specialized types. The application program invokes database operations as usual. What is different is that the invocation will refer to functions provided by the object classes. For example, a VLSI CAD database built on top of PROBE might invoke a selection where the selection predicate checks a given instance of the GATE object class for overlap with all objects on a given layer. The selection is done in the database system kernel, and the overlap predicate is supplied by the object class that represents the GATE's geometry (e.g., the BOX object class).

3. The PROBE Data Model

A brief discussion of the PROBE data model (PDM) is necessary before proceeding (see [MANO86b, MANO86c] for detailed information). The basic modeling constructs of PDM are *entities* and *functions*. An entity is used to represent a real-world object. An entity is a member of one or more types, and types are organized into a generalization hierarchy. Functions are used to represent properties of entities (e.g., the age of a person), as well as relationships among entities (e.g., the containment of an entity by another). Functions can take any number of arguments (including 0) and return any number of results. The semantics of functions do not distinguish between those that are stored and those that are computed. The distinction is important to an implementer and to a query optimizer, but not to an application programmer. A stored function is essentially the same thing as a relation in the relational model.

Functions and entities are manipulated by the PDM algebra. This algebra can be viewed as an enhanced relational algebra. The differences include the notion of an entity, the ability to manipulate entities of arbitrary types, support for computed functions, and support for spatial and recursive queries. An operation of the PDM algebra that will be important in a later section is *apply-append*. This operations is essentially a natural join adapted to work with PDM functions.

Example: F is a stored function with three arguments, x, y and z. Since it is stored, any subset of the arguments can be used as input and the others can be treated as output. PLUS is a computed function that takes y and z as input and returns w as output. Conceptually, PLUS is the infinite set of (y, z, w) tuples for which w = y + z. If F(x, y, z) = {[a, 1, 2], [b, 5, 8], [c, 7, 3]} then apply-append(F, PLUS) returns this stored function (with arguments x, y, z and w): {[a, 1, 2, 3], [b, 5, 8, 13], [c, 7, 3, 10]}. (This is equivalent to a natural join between the relation corresponding to F and the infinite relation corresponding to PLUS.) When given two stored relations, apply-append returns the natural join. When given two computed relations, the result is a new computed relation (i.e., the composition of the two functions).

4. Adding an Object Class to PROBE

One of the tasks of the DBI is to "customize" an EDB by incorporating specialized object classes. To do this, the DBI must produce an *adapter*. What is described here is oriented towards the current implementation of PROBE; but the issues are applicable to any EDB.

Consider a geographic application that manipulates polygons. The PDM function *edges*: POLYGON → set of LINE-SEGMENT returns the line segments comprising the edges of polygon p. For purposes of this discussion, assume that *edges* is computed. Associated with the POLYGON type is an object class giving the implementations of all the computed functions of polygons. These subroutines compute their results from the representation of polygons which are stored in the database and accessed (but not interpreted) by the database system kernel. For a polygon, the representation might be an array of points defining the vertices of a polygon (i.e., *edges* returns an instance of the LINE-SEGMENT object class for each pair of adjacent array elements).

To support *edges*, the DBI must supply an adaptor routine that reconciles the following two views of this function:

1. PROBE sees *edges* as a PDM function. When invoked by apply-append, this function receives a tuple (containing a POLYGON instance) and returns a relation (containing a set of LINE-SEGMENT instances). (Internally, the PROBE breadboard uses relations and tuples.)
2. The object class view of *edges* is that of a subroutine (written in C for example). The array storing the polygon is passed to the function by the usual parameter passing mechanism, and the result (multiple line segments) is returned using a data structure supported in the language, e.g., an array or a linked list.

There are three things to be reconciled. First, PDM functions have labelled arguments (in the style of Ada), while most programming languages uses "positional" notation (i.e., the order of arguments is important). Therefore the adapter has to make sure that each labelled argument appears in the correct position in the argument list of the subprogram. Second, below the level of the query language, PROBE works with tuples and relations, while the subprogram works with integers, pointers, arrays, records, etc. The adapter has to convert from one representation to the other. The problem is most severe for set-valued functions. The implementation of apply-append passes a tuple to the subprogram (via the adapter) and expects a relation in return. That relation may be a singleton (in case of a single-valued function), but for the edges function it will not be since each polygon will return multiple line segments. The adapter is responsible for traversing the data structure returned by the subprogram and constructing the relation expected by the implementation of apply-append. The third incompatability has to to with the processing of *null values*. The POLYGON object class may not have any such notion, or may have its own specialized representation of null values, with its own semantics. Furthermore, the treatment of nulls may differ from one object class to another. PDM on the other hand, specifies a particular treatment of null values. One aspect of this is that a *"don't care"* null value is defined as being a member of every object class. The adapter has to intercept these null values from the database and handle them, not allowing them to reach the subprograms. The adapter may substitute the correct null value for that object class or handle the null input without invoking any object class functions at all.

The adapter will also provide access to functions needed for aggregation. For example, in order to compute a sum (one kind of aggregation), the application program directs PROBE's aggregation operator to call a SUM function, provided by the adapter, which adds a value (from one of the tuples being accumulated) to the value of an accumulator. The object class view (from the INTEGER object class, for example) is to add two numbers using the binary '+' operator.

Continuing the POLYGON example, aggregation could be used to define an *assemble* function that converts a collection of LINE-SEGMENTs to a POLYGON (i.e., *assemble* is the inverse of *edges*). The adapter provides the assemble function which stores as input a single LINE-SEGMENT and an accumulator over a partially-constructed polygon. The POLYGON function actually adds the line segment to the polygon. After all the line segments of a polygon have been passed to assemble, the accumulator will contain the completed polygon.

5. An Example: A Spatial Query

We now briefly describe how PROBE's approach to extensibility is used in processing a spatial query. This data and this query are currently supported by the breadboard PROBE system. The point of the example is to show that (a) PROBE can efficiently process spatial queries and (b) under the proposed division of labor between the database kernel and the spatial object classes, it is relatively easy to implement this kind of application.

The database represents some of the major roads in the eastern part of Massachusetts. Consider the query "find all gas stations within a mile of the Massachusetts Turnpike". Two spatial object classes are used:

1. The ELEMENT object class. PROBE's geometry filter uses a special approximate representation that is grid-like but runs faster and uses less space (see [OREN86a, OREN86b] for details). The geometry filter algorithms are part of the database system kernel, but the ELEMENT object class is handled exactly as any other object class.
2. The LINESEG object class, to provide precise handling of geometric data, represented by line segments.

We also constructed a third object class, that implemented the operations of a graphical interface (e.g., construct a box, place the box on the screen, highlight a box). This allowed us to implement the entire graphical interface using PDM Algebra.

The query is evaluated by a series of nested PDM Algebra expressions that implement the following strategy:

1. From the name "Massachusetts Turnpike" the entity representing this road can be located. A road is a complex object, so the "road segments" comprising the road must be located. Next, find the spatial object representing each road segment. Each one of these steps requires a call to apply-append. Since all the functions involved are stored, all that is occurring is a selection over a series of natural joins.
2. The spatial object representing a road segment is used to locate the straight-line segments that represent the shape of the road. Apply-append is used here too. The function that returns these line segments can be stored or computed (it is very similar to the edges function discussed above). We chose to use a stored function.

3. Form a one-mile buffer around each line segment that will contain all gas stations of interest (and possibly some others). This can be done using apply-append and a function which computes a box (the buffer) surrounding a given line segment.

4. The geometry filter representation of the buffers is obtained by using apply-append and the *decompose* function defined for the ELEMENT object class (a spatial object is *decomposed* into elements.) *Decompose* does not assume anything about the type of the spatial object being decomposed. It calls two generic functions that check overlap and containment. The type-specific implementations of these operations are invoked by the corresponding generic functions after checking the type of the spatial object being decomposed.

5. The points representing the gas stations have been decomposed as part of the loading of the database, so the geometry filter can now be run. The filter is invoked via the *spatial join* operation of PDM algebra and returns a (stored) function of (gas station, line segment) pairs. These objects are likely to be within the required distance of one another. The output from this step is an approximate (but conservative) answer to the query, corresponding to the "Candidates" box in figure 1. Refinement of this result occurs in the next step.

6. Each (gas station, line segment) pair now has to be verified. If the distance between the two objects is more than one mile, it is rejected. This step is implemented by first using apply-append to compute the distance between the gas station and the line segment. Then a cartesian product with the function ONE-MILE() → "1 mile" is executed. Next, the select operation is used to compare the distance to 1 mile. Select takes as input the stored function just computed, and the computed function LESSTHAN which specifies the selection condition. Finally, a projection onto the gas station entity yields the result.

The data set used in our development has 500 gas stations and 178 line segments. The Massachusetts Turnpike is represented by 14 of those line segments. Of the 7000 possible gas station / line segment combinations (following selection for line segments of the Massachusetts Turnpike), the geometry filter identified 165 candidates and the final step reported 20 gas stations. There are many open issues regarding the optimal setting of parameters used by the geometry filter and regarding spatial query processing strategies. For example, the figure of 165, representing about 2% of the possible pairs, could certainly be lowered by adjusting the parameters, and this would speed up the refinement step, although other costs would increase (e.g., the time to do the decomposition in step 4).

6. Conclusions and Work in Progress

In order to provide a database system that can meet the needs of a wide variety of applications, it is necessary to build in extensibility, rather than a specific set of extensions. PROBE provides this capability by supporting the incorporation of arbitrary object classes. This facility supports a division of labor in which the database system kernel is concerned with manipulating sets of uninterpreted objects, while only the object classes are concerned with application-specific details. In particular, the implementer of a specialized object class does not have to be concerned with database issues such as secondary storage, concurrency control and recovery.

These ideas were demonstrated in the context of an application being developed on a "breadboard" implementation of PROBE. The breadboard currently supports the PDM algebra and the geometry filter (as well as some spatial object classes). The recursion facilities (described in [ROSE86a, ROSE86b]) will be added in the near future.

Acknowledgements

We are grateful to Alex Buchmann, Upen Chakravarthy, Umeshwar Dayal, Mark Dewitt, Sandra Heiler, Frank Manola and Arnie Rosenthal for their contributions to the development of PROBE.

7. References

[AUTO87]
Proc. 8th International Symposium on Computer-Assisted Cartography.

[CARE86]
M. J. Carey et al. The architecture of the EXODUS extensible DBMS. Proc. International Workshop on object-oriented database systems, (1986).

[CHAN81]
S.-K. Chang, ed. Pictorial Information Systems. Special issue, *Computer*, *14*, 11 (1981).

[CHOC84]
M. Chock et al. Database structure and manipulation capabilities of a picture database management system (PICDBMS). *IEEE Trans. on Pattern Analysis and Machine Intelligence* 6, 4 (1984).

[KATZ85]
R. H. Katz. Information management for engineering design. Springer-Verlag (1985).

[DAYA85]
U. Dayal et al. PROBE – a research project in knowledge-oriented database systems: preliminary analysis. Technical Report CCA-85-03 (1985), Computer Corporation of America.

[IEEE77]
Proc. IEEE Workshop on Picture Data Description and Management (1977).

[LIEN77]
Y. E. Lien, D. F. Utter Jr. Design of an image database. Proc. IEEE77 Workshop on Picture Data Description and Management, (1977).

[MANO86a]
F. Manola, J. Orenstein. Toward a general spatial data model for an object-oriented DBMS. Proc. 12th International Conference on Very Large Databases, (1986).

[MANO86b]
F. Manola, U. Dayal. PDM: An object-oriented data model. Proc. International Workshop on object-oriented database systems, (1986).

[MANO86c]
F. Manola. PDM: an object-oriented data model for PROBE. Computer Corporation of America technical report, to appear.

[MORE85]
Scott Morehouse, "ARC/INFO: A Geo-Relational Model for Spatial Information", *Proc. Seventh Intl. Symp. on Computer-Assisted Cartography*, American Congress on Surveying and Mapping, 1985.

[OREN86a]
J. A. Orenstein. Spatial query processing in an object-oriented database system. Proc. ACM SIGMOD, (1986).

[OREN86b]
J. A. Orenstein, F. A. Manola. Spatial data modeling and query processing in PROBE. CCA Technical Report CCA-86-05, (1986).

[ROSE86a]
A. Rosenthal, S. Heiler, U. Dayal, F. Manola. Traversal recursion: a practical approach to supporting recursive applications. Proc. ACM SIGMOD (1986).

[ROSE86b]
A. Rosenthal, S. Heiler, U. Dayal, F. Manola. Traversal recursion: a practical approach to supporting recursive applications. Computer Corporation of America technical report CCA-86-06, (1986).

[SCHE82]
H.-J. Schek, P. Pistor. Data structures for an integrated data base management and information retrieval system. Proc. VLDB 8, (1982), 197-207.

[SMIT84]
J. D. Smith. The application of data base management systems to spatial data handling. Project report, Department of Landscape Architecture and Regional Planning, University of Massachusetts, Amherst (1984).

[STON82]
M. Stonebraker et al. Document processing in a relational data base system. *ACM TOOIS 1*, 2 (1983), 143-158.

[STON86]
M. Stonebraker. Object management in POSTGRES using procedures. Proc. International Workshop on object-oriented database systems, (1986).

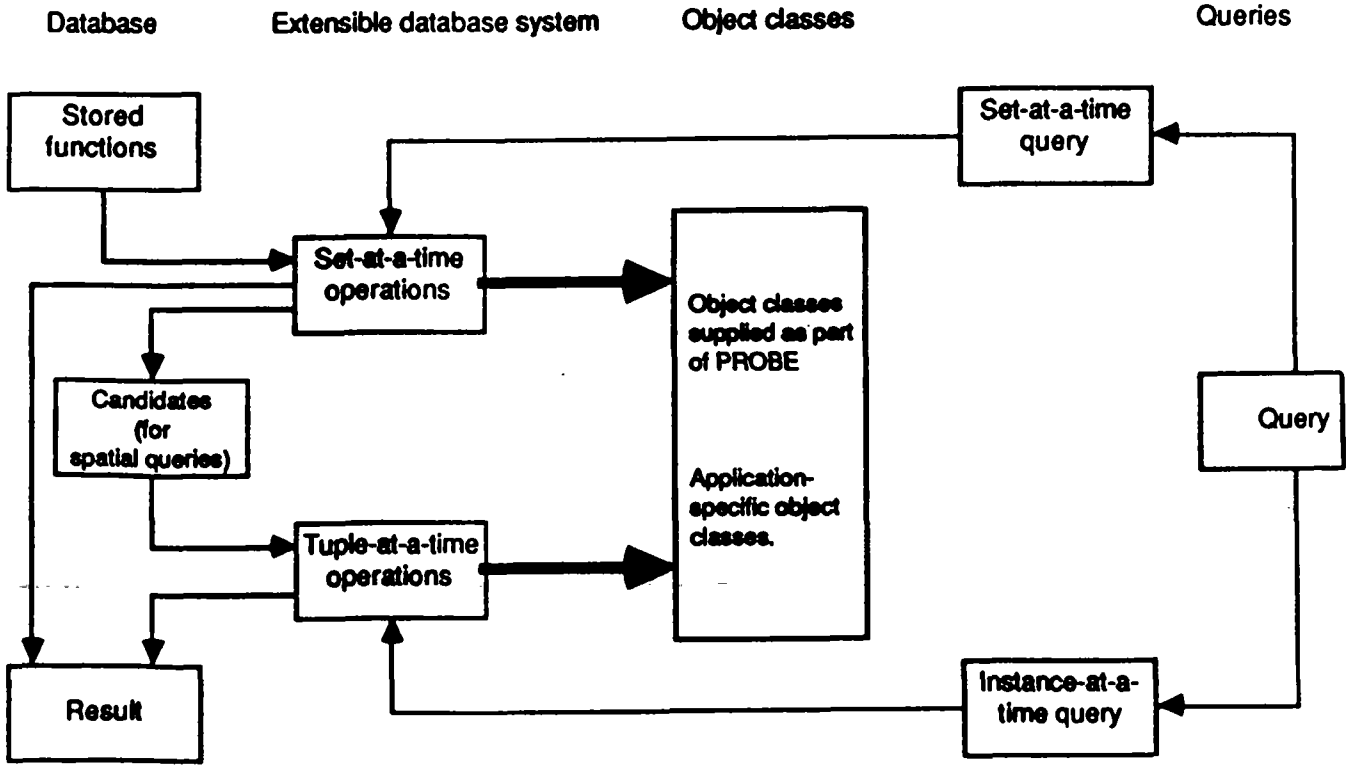

Figure 1: Architecture of the PROBE Database System

An Overview of Extensibility in Starburst

by

John McPherson

Hamid Pirahesh

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Abstract

One of the goals of the Starburst project is to design and implement an extensible database management system. It is felt that extensibility is required if one hopes to meet the diverse requirements of today's applications and to be able to adapt to the needs of future applications and technologies. This paper will provide a brief overview of the Starburst database management system followed by a discussion of several areas of extensibility in Starburst including relation storage management, access paths, abstract data types, and complex objects.

Starburst Overview

There are four primary research areas that are being addressed by Starburst: extensibility, system structure and performance, portability, and distributed data and function. Extensibility is the primary topic of this paper so we will defer discussion of this topic until later sections. In the area of system structure and performance, we are looking at new strategies for locking and recovery. Starburst will permit record level locking and will use a write ahead logging protocol. We are also looking at new optimization and execution strategies. Our portability goal is to design a system that can be easily ported to different hardware and operating systems. We are attempting to isolate hardware and operating system dependencies to a small set of clearly defined modules that will need to be modified to support different environments. Finally, we will be investigating techniques for distribution of data and function between workstations, departmental servers, and mainframes. Areas that need special attention include naming, security, and control. The emphasis in Starburst will not be on geographic distribution of data between homogeneous mainframes, as that was dealt with in R* [LINDSAY84] which was a predecessor project to Starburst.

The Starburst system is divided into two major subsystems named Core and Corona. Core provides tuple at a time access to stored relations. It includes support for record management, indexes, transaction management, buffer management, integrity constraints, and triggers. Corona provides an interface to applications and users. It includes support for query parsing and optimization, generation of runtime access plans, catalog management, and authorization. Core and Corona are similar to the Relational Storage System (RSS) and Relational Data System (RDS) in System R [ASTRAHAN76]. However, there is a more symbiotic relationship between Core and Corona than between the RSS and RDS. In

Starburst, for example, Core catalog information will be managed by Corona and passed to Core when it is called. This eliminates the need for Core access to any catalogs at query execution time. The "jagged" interface between Core and Corona reflects mutual trust and support between the two components, and is motivated primarily by performance considerations.

Extensibility

Database systems have traditionally provided support for the typical business data processing applications such as order entry, inventory control, and billing. In these applications, the database system has provided a repository for the applications' data that includes support for concurrency control, recovery, and in the case of relational databases, a query language that makes it easy to access and manipulate data. The database system has allowed these applications to concentrate on using data without worrying about the intricacies of data management. Other applications such as CAD/CAM, office information, statistical databases, expert systems, and text applications would also benefit from these database management services, but such applications require data models, storage techniques, and access methods that are not normally provided by traditional database management systems. For example, techniques for storing formatted records of modest size are inappropriate for storing large segments of text or images that may consume several megabytes of storage. Furthermore, standard access methods are inadequate for indexing structured documents or objects in an image.

While there is a desire to support a wider class of applications, one does not want to lose the benefits of a relational database management system including the set-oriented query language feature. The broad acceptance of relational databases demonstrates their usefulness for a large class of existing applications. It does not seem prudent, therefore, to completely abandon the existing relational database technology in attempting to support new applications with a database management system. We feel that non-traditional database applications can be supported with appropriate extensions to a relational database management system, resulting in a system that has the ease of use of a relational system and the performance and function required for the new application.

One approach to adding the new functions would be for us to design and build an *extended* database management system, that is, a system where we have determined in advance what function the system will contain. The major problems with the extended approach are that we would end up with a very large database system that tries to be all things to all applications, and there would undoubtedly be many applications and technologies that we would still be unable to support. Furthermore, the resulting system would be difficult to modify if we ever decided to try to add support for another type of application. Our approach, therefore, will be to build an *extensible* relational database management system, that is, a system that provides a full function relational database management system as a base and that can be easily extended to provide support for new applications and technologies. Any particular instance of the database system will be extended for those applications that are currently using the system. Other groups have also been investigating extensible database systems as witnessed by this issue of *Database Engineering* and other papers [STONEBRAKER86,BATORY86,CAREY86,DAYAL85].

Several types of extensions are being examined in Starburst and a few of them will be considered in this paper. Starburst will be able to support different methods of storing relations, different access paths for locating stored data, integrity constraints that control the consistency of the database, and other attachments to the database that are triggered by modifications to the database and result in actions either

inside or outside of the database. It is recognized that there will always be some data that is stored in external databases and Starburst will be able to access data in those external databases and coordinate concurrency control and recovery, as long as the external databases are capable of participating in two phase commit and in global deadlock detection protocols. Users will be able to define abstract data types, and we will provide support for complex objects. All of the Starburst extensibility features will be supported with appropriate changes to SQL. Some of the extensions to Starburst can only be made by knowledgeable programmers "at the factory". Such extensions include many of the Core extensions that, for performance reasons, require unprotected access to critical system resources. Other extensions, such as the definition and use of abstract data types based on existing types, can be made by casual database users. The remainder of this paper will discuss the data management extension architecture which provides support for alternative relation storage methods, access paths, integrity constraints and triggers. We will then discuss support for abstract types and complex objects.

Data Management Extension Architecture

Core provides extensible data management services for Starburst including support for alternative relation storage methods and alternative access paths. The principle features of the data management architecture [LINDSAY87] supported by Core are: a well defined set of interfaces for relation storage methods, access structures, integrity constraints, and triggers; an efficient and flexible way of determining what relation storage method, access paths, integrity constraints, and trigger routines need to be activated based on an extensible relation descriptor; an efficient way to activate these routines using vectors of routine entry points; and a formulation of common services such as logging, locking, event notification, and predicate evaluation, to coordinate the activities of the different extensions and to make their implementation easier.

The data management extension architecture treats extensions as alternative implementations of certain generic abstractions having generic interfaces. The architecture defines two distinct generic abstractions: relation *storage methods*; and access paths, integrity constraint, or triggers, called *attachments*, that are associated with relation instances.

Relation Storage Methods

Alternative relation storage method extensions, known simply as *storage methods*, allow different relations to be implemented in different ways to fit the needs of various applications and technologies. For instance, the records of a relation may be stored sequentially in a disk file or they may be stored in the leaves of a B-tree index. Read only relations on optical disks that are used for document storage might have very unique storage structures to facilitate efficient text searches. In any case, a storage method implementation must support a well-defined set of relation operations such as delete, insert, destroy relation, and estimate access costs (for query planning). Note that some of these operations, such as insert, delete, and update for read only relations, may be null operations. Additionally, storage method implementations must define the notion of a *record key* and support direct-by-key and key-sequential record accesses to selected fields of the records. The definition and interpretation of record keys is controlled by the storage method implementation. For example, record keys may be record addresses or may be composed from some subset of the fields of the records.

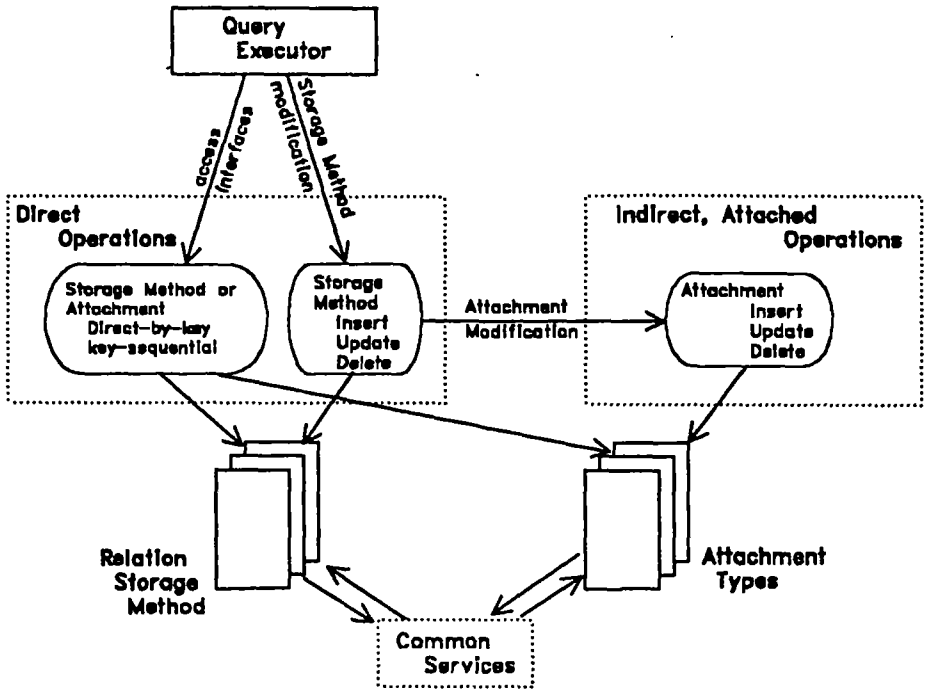

Figure 1: Data Management Interfaces

Attachments

Access path, integrity constraint, and trigger extensions are called *attachments*. Examples of attachment types include B-tree indexes, hash tables, join indexes, single record integrity constraints, and referential integrity constraints. In principle, any type of attachment can be applied to any storage method. However, some combinations (e.g., a permanent index on a temporary table) may not make sense. Attachment instances are associated with relation instances, and a single relation instance may have multiple attachment instances of the same or different types.

Attachments, like storage methods, must support a well-defined set of operations. The attachment access operations can be invoked directly by the data management facility user. Access path extensions support direct-by-key and (optionally) key-sequential accesses which return the storage method key of the corresponding relation records. Unlike storage methods, however, attachment modification operations, that is, insert, update, and delete, are not directly invoked by the data management facility user. Instead, attachment modification interfaces are invoked only as side effects of modification operations on relations. Whenever a record is inserted, updated, or deleted, the (old and new) record is presented by the data management facility to each attachment type with instances defined on the relation being modified. Attachments can then take actions to add the record keys to access path data structures, to check integrity constraints, or to trigger additional actions within the database or even outside of the database system. Any attachment can abort the relation operation if the operation violates any restrictions of the attachment.

In this way, the data management facility automatically ensures that all attachments are notified when a relation is modified.

Common Services

Storage method and attachment extensions, while isolated from each other by the extension architecture, are embedded in the database management system execution environment and must therefore obey certain conventions and make use of certain common services. A few of these will be discussed briefly.

The Core relation descriptor is stored by Corona and passed to Core at runtime. The Core relation descriptor contains a field for the storage method that implements the relation and a field for every attachment type defined to Core. Each storage method and attachment type is assigned an internal identifier, and field N in the Core relation descriptor is used by the attachment type with internal identifier N. If there are no instances of attachment N defined on the particular relation, then field N will be null. Field 0 is used for the storage method descriptor for the relation. The actual contents of each field is defined by the corresponding storage method and attachment, and for attachments, will describe all of the attachment instances that are defined on the relation. For example, the relation descriptor field for our B-tree index attachment will itself be a record with a field for each index defined on the associated relation. The relation descriptor is used by the common services to determine what attachments need to be notified for a particular relation modification operation. The relation descriptor is used by storage methods and attachments to store information that is necessary to perform the Core operations on relation and attachment instances.

In order to be able to coordinate the activity of multiple attachments during relation modification operations, it is necessary to be able to *undo* the effects of the storage method modification and the already-executed, attached procedures when a subsequently executed attachment vetoes the relation modification operation. The data management extension architecture relies on the use of a common recovery facility to drive, not only system restart and transaction abort, but also the *partial* rollback of the actions of the transaction. When a relation modification operation fails, for any reason, the *common recovery log* is used to drive the storage method and attachment implementations to undo the partial effects of the aborted relation modification. The same log-based driver also drives storage method and attachment implementations during transaction abort and during system restart recovery.

The data management extension architecture assumes that all storage method and attachment implementations will use a *locking-based* concurrency controller to synchronize the execution of their operations. While a system-supplied lock manager will be available to the storage method and attachment implementations, they can also provide their own lock controllers. However, all lock controllers must be able to participate in transaction commit and system-wide deadlock detection events.

It is possible for an attachment instance to defer an action until certain transaction events occur such as "before transaction enters the prepared state" or transaction commit. Deferred action queues are provided by common system services. An attachment instance can place an entry on the queue that will cause an indicated attachment procedure to be invoked with the indicated data when the event occurs. For example, certain integrity constraints cannot be evaluated when a single modification occurs but must be evaluated after all of the modifications have been made in the transaction. When the integrity constraint attachment is activated as a result of a modification to a relation on which the integrity constraint is

defined, the attachment can place an entry on the deferred action queue for the "before transaction enters prepared state" event. The entry would contain the address of the attachment routine that should be invoked to evaluate the integrity constraint and a pointer to data that, in this case, describes the integrity constraint that needs to be tested. After all database modifications have been made and before the transaction enters the prepared state, the corresponding deferred action queue will be processed. The entry that had been queued earlier will be removed and the indicated routine will be called and passed a pointer to the data. If the integrity constraint is not satisfied then the transaction can be aborted by the attachment.

Another common service interface supports the evaluation of filter predicates during direct-by-key and key-sequential accesses, and supports integrity constraint checking. In order to quickly reject unqualified entries during accesses, it is important to evaluate filter predicates as early as possible. The filter predicate expression, along with a list of fields needed from the current record, is passed to the access procedures. The access procedures, after isolating the needed fields, will invoke the filter expression evaluator on the filter predicate and the fields of the current record. The intention of this common service facility is to allow filter predicates to be evaluated while the field values from the relation storage or access path are still in the buffer pool. Figure 1 shows the relationship between storage methods, attachments, common services, and the operations on storage methods and attachments.

Abstract Data Types

The base system will provide support for basic types including integer, floating point, fixed length character string, variable length character string, and long field. It is often desirable, however, to permit a user or application to define new types and operations on those types in order to structure data in a more convenient form, to permit more control over how data is manipulated, and to increase performance by allowing user functions to be evaluated in the database management system instead of in the application. The support of abstract data types provides a way to offer extensible user data types.

An abstract data type is an encapsulation of data and operations on that data. The data associated with an abstract data type can only be accessed and manipulated by the operations that are associated with the abstract data type. We would like to support operations or functions that are written in general purpose programming languages such as C, and we would also like to support functions with embedded SQL statements. We will consider two kinds of abstract data types in Starburst: scalar types and structured types. *Scalar types* can be represented directly by an instance of one of the base types. For example, an angle can be represented as a floating point number. Functions for such an abstract data type might include trigonometric functions. *Structured types* allow collections of fields to be treated as a unit, and this concept can be generalized to *hierarchically-structured types*. For example, a "time" abstract data type can be composed of three fields for hours, minutes, and seconds with functions for elapsed time, conversion to number of seconds, etc.

Support for abstract data types requires support from many components of the DBMS. SQL support is required for their definition and use. The query processor must ensure that data within an abstract data type is only accessed and modified using the functions that are associated with the abstract data type. The query optimizer needs to know cost and selectivity for the functions. Finally, we would like to be able to evaluate functions in Core when possible in order to eliminate costs associated with the Core/Corona interface such as locking, latching, and the pinning of buffers in the buffer pool.

Complex Objects

The base system will provide support for complex objects; however, a database extension may provide special access path attachments or storage methods for better performance. The basic approach is similar to XSQL [HASKIN83], [LORIE83] where a complex object is a collection of heterogeneous components, represented by tuples, which are bound together with predicates, e.g., foreign key matching. This is in contrast to the abstract data type approach described in [BROWN83], where a complex object is defined as a set of nested abstract data types. In [BATORY84], two major attribute pairs are defined for complex objects: disjoint versus non-disjoint, and recursive versus non-recursive, forming four categories of objects. While many of the approaches reported in the literature lack support for at least one of these categories of objects, Starburst will support all four.

The system allows definition of complex object views. As with views over normal form tables, multiple views may overlap. Different views over the same object may structure the object differently. For example, a parent component in one view may be a child component in another view. We believe this capability is crucial for applications (e.g. CAD/CAM), where the same complex object (e.g. a VLSI chip) may be viewed differently by different applications (e.g. VLSI editor, simulator, or fabricator). The result of a query over a complex object is a complex object, i.e., the language has the closure property. Hence, the same language constructs are used to query a complex object as well as to define a new one. The closure property, which is an important attribute of normal form relational languages [DATE84], is preserved in the enhanced relational language for complex objects. In this language, the relational operations are enhanced for complex objects. Projection is enhanced to include component projection, selection can be done at the object level or at the subobject level, and a new type of join allows multiple complex objects to form a new composite complex object. Complex object traversal is allowed, and this capability is used, for example, in the tuple at a time application interface.

The data associated with complex objects and its components is kept in normal form tables. Clustering allows components of a complex object to share the same set of pages as much as possible to minimize I/O. Special access paths (attachments) may be added to speed up complex object retrieval.

Summary

The extension mechanisms that have been discussed are currently being implemented in the experimental Starburst database management system at IBM Almaden Research Center. These mechanisms support efficient execution through close coupling of the extensions to the operations they support and to the common service facilities they require. Additionally, some changes to SQL have been proposed to support extensions, providing an easy-to-use interface to the database and its extensions.

Acknowledgements

The authors would like to acknowledge the other members of the Starburst project, past and present, who contributed to ideas expressed in this paper: Walter Chang, Bill Cody, J. C. Freytag, Roberto Gagliardi, Laura Haas, George Lapis, Bruce Lindsay, Guy Lohman, C. Mohan, Kurt Rothermel, Peter Schwarz, Irv Traiger, Paul Wilms, and Bob Yost.

Bibliography

[ASTRAHAN 76] M. Astrahan, et al., System R: Relational Approach to Database Management, **ACM Trans. on Database Systems, Vol. 1**, No. 2 (June 1976), pp. 97-137.

[BATORY 84] D. Batory, A. Buchmann, Molecular Objects, Abstract Data Types, and Data Models: A Framework, **Proc. of the Tenth International Conference on Very Large Data Bases** (1984).

[BATORY 86] D. Batory, GENESIS: A Reconfigurable Database Management System, University of Texas at Austin Technical Report Number TR-86-07 (1986).

[BROWN 83] V. Brown, S. Navathe, S. Su, Complex Data Types and Data Manipulation Language for Scientific and Statistical Databases, **Proc. of International Workshop on Statistical Database Management**, Los Altos, California (1983).

[CAREY 86] M. Carey, D. DeWitt, J. Richardson, and E. Shekita, Object and File Management in the EXODUS Extensible Database System, **Proc. 6th International Conference on Very Large Data Bases**, Kyoto, Japan (August 1986), pp. 91-100.

[DATE 84] C. Date, Some Principles of Good Language Design, **ACM SIGMOD RECORD** (November 1984), pp. 1-7.

[DAYAL 85] U. Dayal, A. Buchmann, D. Goldhirsch, S. Heiler, F. Manola, J. Orenstein, and A. Rosenthal, PROBE - A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis, Computer Corporation of America Technical Report CCA-85-03 (July 1985).

[HASKIN 83] R. Haskin, R. Lorie, On Extending the Functions of a Relational Database System, **Proc. of ACM Engineering Design Applications** (1983).

[LINDSAY 84] B. Lindsay, L. Haas, C. Mohan, P. Wilms, and R. Yost, Computation and Communication in R*: A Distributed Database Manager, **ACM Trans. on Computing Systems, Vol. 2**, No. 1 (February 1984), pp. 24-38.

[LINDSAY 87] B. Lindsay, J. McPherson, H. Pirahesh, A Data Management Extension Architecture, To appear in **Proc. ACM SIGMOD '87**, San Francisco (May 1987).

[LORIE 83] R. Lorie, W. Plouffe, Complex Objects and Their Use in Design Transactions, **Proc. of ACM Engineering Design Applications** (1983).

[STONEBRAKER 76] M. Stonebraker, E. Wong, and P. Kreps, The Design and Implementation of INGRES, **ACM Trans. on Database Systems, Vol. 1**, No. 3 (September 1976), pp. 189-222.

[STONEBRAKER 86] M. Stonebraker and L. Rowe, The Design of POSTGRES, **Proc. of ACM SIGMOD '86**, Washington, D.C. (May 1986), pp. 340-355.

Principles of Database Management System Extensibility

D.S. Batory
Department of Computer Sciences
The University of Texas at Austin

1. Introduction

Database research has shifted away from traditional business applications. Support for VLSI CAD, graphics, statistical and temporal databases are today's 'hot topics'. Among the issues that are being addressed are new data types and operators, algorithms for transitive closure and sampling, specialized storage structures, and novel methods of concurrency control. It is well-known that to admit new algorithms, operators, or structures into existing DBMSs is a very difficult and costly process (if it is possible at all). The utility of new research results hinges on a technology by which DBMSs can be customized rapidly and cheaply. Such a technology raises fundamental issues on how DBMSs should be built. **Extensible** database systems address these architectural concerns.

GENESIS is a project to develop an extensible DBMS. Our approach is unique in that we are developing models of DBMS implementation, and validating the models by prototype development. Our work makes explicit fundamental principles of DBMS construction and reveal ways in which a practical technology for customizing DBMSs can be realized. We review some of these principles in this paper, and give more detailed explanations in [Bat82-87].

2. Simplest Common Interface

Extensible DBMSs require open architectures with standardized interfaces. The key to extensibility lies in how these interfaces are designed. It is our belief that declaring an ad hoc interface to be a standard is the worst of all possibilities. A better approach is to 1) identify a class of algorithms to be implemented, and 2) design the simplest interface that supports *all* algorithms of the class. The greater the number of algorithms, the more likely it is that *the interface captures fundamental properties of the algorithm class.* Such an interface is no longer ad hoc, but is justified by its demonstratable generality. We call this the **simplest common interface (SCI)** method for standardized interface design.

The SCI method provides a useful form of extensibility. As all (implementations of the) algorithms of the class support the same interface, they are interchangeable. Thus, if a particular algorithm doesn't provide the desired performance, it can be replaced by another without altering higher-level modules.

As an example, shadowing, page logging, and db-cache are three well-known database recovery algorithms. If one were to give each algorithm to a different implementor, three disparate interfaces would be designed. Algorithm interchangability would not be present. However, by defining an interface for all three, interchangeability is guarenteed.

The ingenuity of our colleagues ensures us that no single interface can encompass all future algorithms. (Note this is also true for 'extensible' DBMSs with ad hoc interfaces). However, SCIs have a distinct advantage since they capture properties of algorithm classes. If the initial class is sufficiently large to begin with, adding a new algorithm requires either no changes or simple, evolutionary changes. Radical modifications, which should be expected for ad hoc interfaces, are unlikely.

SCI is a necessary, but not sufficient, design principle for extensible DBMSs. As an example, one could build a monolithic file management system that provides an SCI interface to all file structures. While the interchangeability of different structures is an important and recognized goal in DBMSs, there are lower level primitives on which all file structures rely; the implementations of these primitives should not be duplicated. A better approach is to use a layered architecture, where each layer provides the primitives on which the next higher layer is defined. To provide maximum extensibility, each layer should have an SCI.

GENESIS uses SCIs. JUPITER, the file management system of GENESIS, has SCIs to file, node, block, recovery, and buffer management algorithms [Twi87]. Principles for layering DBMS architectures are addressed in the following sections.

3. Storage Structure Extensibility and Layered DBMS Architectures

Extensible DBMSs must be able to accommodate new storage structures. We have shown in earlier papers [Bat82-85] that storage structures are conceptual-to-internal mappings of data. Although the mappings (structures) for a particular DBMS can be very complicated, they can be described simply by a composition of primitive mappings (primitive storage structures). We explain the basic ideas in the context of a network data model.

Network models represent databases by files and links. Every file represents a set of records and every link represents a relationship between the records of two or more files. The implementation of a database can be specified by assigning an implementation to each file and an implementation to each link. Files have direct implementations on secondary storage as **simple file** structures. Classical examples are indexed-sequential, B+ trees, and heaps. Links can be implemented by **join algorithms** or by **linkset** structures. Classical linkset structures include pointer arrays, linear lists, and ring lists. Alternatively, files and links have indirect implementations in that they are mapped to lower-level files and links by **elementary transformations.** Classical transformations include indexing, long fields, transposition, compression, and horizontal partitioning. Lower-level files and links can be implemented by any of the above methods: simple files, linksets/join algorithms, or elementary transformations. These concepts lead to a building-blocks view of storage structures.

We assume readers are familiar with the concepts of simple files and linksets, but elementary transformations may be new. We digress briefly to illustrate the indexing and transposition mappings. Figure 1a shows indexing, which maps a higher-level or abstract file to an inverted file. An inverted file contains a data file and an index file for each attribute that is to be inverted. Each index file is connected to the data file by precisely one link (which is implemented by a pointer array or inverted list linkset). Figure 1b shows transposition, which maps an abstract file to a transposed file, which is a column partitioning of a file/relation.

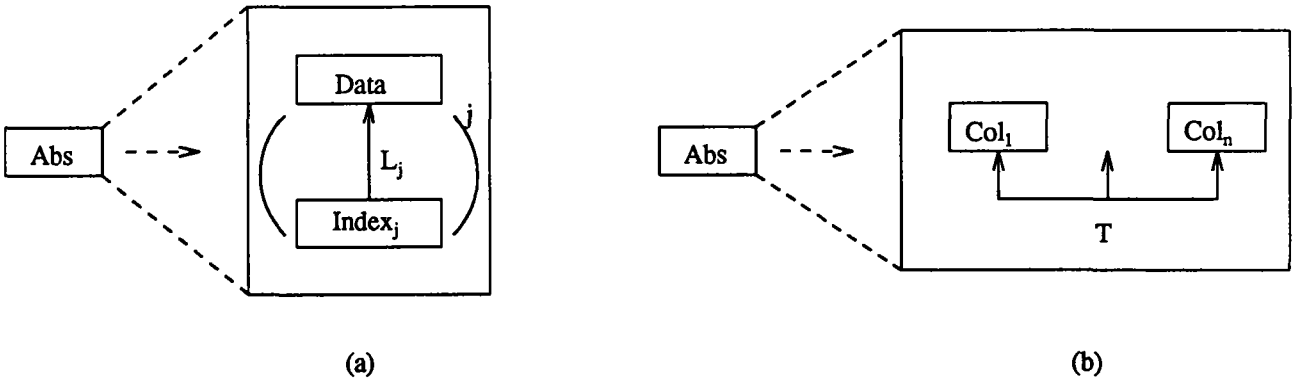

Figure 1. Indexing and Transposition Transformations

Every elementary transformation introduces some implementation detail that was not present previously. Indexing, for example, adds inverted file structure and transposition adds transposed file structure. The introduction of new structures must also be accompanied by algorithms to maintain them. This is accomplished by mapping operations.

Consider the indexing transformation of Figure 1a. A retrieve of abstract records invokes inverted file retrieval algorithms (i.e., scan the data file or access index records and follow pointers). A deletion of an abstract record causes its corresponding data record to be deleted and connecting index records to be updated. These and other operation mappings (algorithms) for inverted files are well-known. Similarly for transposition (Fig. 1b), a retrieve of abstract records invokes transposed file retrieval algorithms [Bat79]. The deletion of an abstract record causes the deletion of all of its subrecords. And so on for other operations.

The combination of data and operation mappings that are identified with an elementary transformation is a **layer.** There are indexing layers, transposition layers, etc. DBMS architectures are compositions of layers. Architectures differ in the layers that they use or the order in which layers are composed. For example, Figure 2a shows indexing occurring before transposition. This composition was first studied by Hammer and Niamir

[Ham79] and was first implemented by Statistics Canada as the RAPID DBMS [Tur79]. Figure 2b shows transposition occurring before indexing. This composition, called the decomposition storage model (DSM), was first studied by Copeland and Khoshafian [Cop85]. DSM was the starting point for the storage architecture of MCC's database machine. Even though the individual layers are the same in Figures 2a-b, the algorithms (if not the structures) that result from their compositions are not the same.

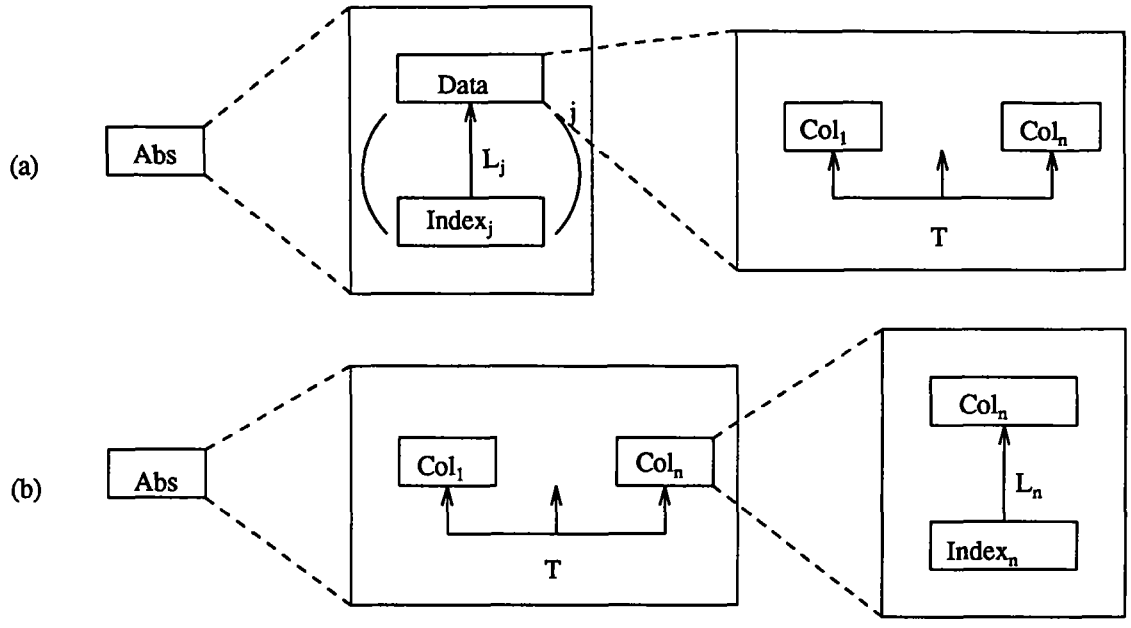

Figure 2. Compositions of Indexing and Transposition

It is important to understand that storage structure extensibility is not simply the ability to add, say, an R-tree or new isam structure to a DBMS. Rather, it is the ability to *compose* primitive structures and their algorithms. This point is not yet widely-appreciated.

Building a DBMS that has storage structure extensibility follows directly from these ideas. Let's look at a retrieve operation on an abstract file. RET(F,Q,O) retrieves the records in O order from abstract file F that satisfy query Q. RET is implemented as a case statement; one case for each simple file and elementary transformation implementation:

```
RET(F,Q,O)
{     case (F.implementation) of
      {     bplus:      RET_BPLUS(F,Q,O);       /* B+ tree retrieval */
            isam:       RET_ISAM(F,Q,O);        /* isam retrieval */
            ...
            index:      RET_INDEX(F,Q,O);       /* inverted file retrieval */
            xpose:      RET_XPOSE(F,Q,O);       /* transposed file retrieval */
      };
};
```

That is, if a retrieval on an abstract file F is to be performed, one calls the B+ tree retrieval algorithm if F is implemented as a B+ tree. If F is an isam file, the isam retrieval algorithm is called. If F is an inverted file, the inverted file retrieval algorithm is called. And so on. To achieve storage structure extensibility is simple: one identifies the basic operations on files and links. (These operations form the SCI to files and links). Each operation is realized as a case statement; one case for each file/link implementation. If a new file structure or

elementary transformation is invented, it and its algorithms are added to the system as new cases.

The GENESIS prototype implements these ideas in a table driven manner. Tables are used to specify the implementation of each file and each link in a database, and indicate which case to execute when an operation on an abstract file is to be performed. As lower-level files and links are generated by elementary transformations, they too are entered into these tables, along with their case identifiers.

Storage structure customization is achieved in GENESIS by filling in these tables by simple C programs, whose length is typically on the order of 100 lines. Because these programs are so short, changes to the target DBMS's architecture can be made very rapidly (in minutes or hours). If the required implementations are in the case statement library, the target system can be synthesized immediately. If not present, they'll need to be written. Although system development time is increased by several months, an advantage of our approach is that these algorithms are reusable; they should not need to be written again.

Further details on storage structure extensibility are given in [Bat86a].

4. Algorithm Extensibility

Extensible DBMSs must be able to accommodate new algorithms. Let's first consider the extensibility of storage structure algorithms; the extensibility of other algorithms follows similarly.

We explained in the last section how layers map operations. In general, there are many ways to accomplish such mappings; each is represented by a distinct algorithm. As an example, there are three standard algorithms (mappings) that implement the retrieve operation for inverted files:

```
RET_INDEX(F,Q,O)
{    case (choose_cheapest) of
     {    Alg1:    SCAN_DATA_FILE(F,Q,O);      /* scan data file */
          Alg2:    USE_1_INDEX(F,Q,O);         /* use one index file */
          Alg3:    USE_n_INDICES(F,Q,O);       /* use many indices */
     };
};
```

The first algorithm scans the data file. (It always works but is slow). The second uses one index file to process a query. (This is the strategy of System R). The third is the classical algorithm which accesses multiple index files to form the union and intersection of inverted lists. There are other algorithms, which we did not list, that exploit the indexing of compound attributes. To incorporate them into the above scheme is simple: add another case for each new algorithm. This is identical to the extensibility method of the previous section. However, there is a fundamental difference in the manner in which a particular case is chosen.

For storage structures, it is the elementary transformation (primitive storage structure) that maps the abstract file; the case which is executed is fixed at DBMS compilation-time. For algorithms, *the cheapest is selected at query execution time;* the actual choice is query dependent. Choosing the cheapest algorithm to execute is part of **query optimization.**

Not all algorithms are tied to storage structures. Consider the JOIN operation which produces the join of two abstract files A1 and A2. By analogy, we implement this operation as a case statement; one case for each basic join algorithm:

```
JOIN(A1,A2)
{    case (choose_cheapest) of
     {    Alg1:    NESTED_LOOPS(A1,A2);          /* nested loops join */
          Alg2:    MERGE_SCAN(A1,A2);            /* merge-scan join */
          Alg3:    GRACE_HASH(A1,A2);            /* GRACE hash join */
          ...
          Algn:    LOWER_LEVEL_JOIN(A1,A2);      /* lower-level join*/
     };
};
```

As before, the cheapest algorithm is selected at query-evaluation time. Nested loop, merge-scan, GRACE hash join, block nested loop algorithms, etc. could be used to compute the join of A1 and A2. Each calls RETrieve operations on A1 and A2 to get their records; these are the operations that are mapped to lower-levels.

It is important to note that none of the above-mentioned join algorithms exploit implementation details of files A1 and A2; these algorithms operate strictly at the conceptual level. There are, however, join algorithms that do exploit such details. The Blasgen and Eswaren join-index algorithm [Bla77], as an example, computes the join of A1 and A2 by joining index files of A1 and A2. This algorithm *cannot* be described in detail at the conceptual level as indicies are not visible. It can only be expressed as an operation, LOWER_LEVEL_JOIN, which is mapped to lower layers. (LOWER_LEVEL_JOIN is another of the basic operations on files that is supported by all layers). At the indexing layer where indices are visible, the Blasgen and Eswaren algorithm would appear as an implementation of LOWER_LEVEL_JOIN. Analogously, join algorithms that exploit transposition would be implementations of LOWER_LEVEL_JOIN at the transposition layer. In this way, our layered framework accommodates the spectrum of join algorithms.

The GENESIS prototype implements operation mappings and algorithm extensibility in this manner. A model of query optimization for extensible DBMSs, based on the above, is forthcoming [Bat87].

5. Data Type and Operator Extensibility

Extensible DBMSs must be able to accommodate new data types and operators. A careful study of a variety of nontraditional database applications reveals that objects, not tuples or relations, are the primary entities that users want to deal with. Forcing users to deal with tuples or relations brings them closer to implementation details. Not only is this burdensome, but it makes application programs more difficult to write. For example, aggregation functions, multivalued functions, multivalued attributes, and recursive queries require special and (we feel) awkward treatment in relational data languages.

A conceptually cleaner approach is based on the functional data model. This model is object-based and has the following features: it is inherently open-ended, i.e., new data types and operations can be added easily; relationships between objects and operations on objects are treated uniformly; and recursive functions (which express recursive queries) can be defined. These features are ideal for extensible DBMSs.

The functional data model has been around for some time, but hasn't caught on. The primary reason, we feel, is the perception that functional implementations of systems are slow. Also, the seminal works on this topic, DAPLEX [Shi81] and FQL [Bun82], have computation models that are more complicated than they have to be. We have found a variant of the functional model which simplifies the computation model of DAPLEX and FQL, and has implementations that are demonstratably efficient.

The basic idea is the following distinction between sequences and streams. A **sequence** is a series of objects enclosed by braces. Here is a sequence of three objects:

{ d1 d2 d3 }

A **stream** is an encoding of a sequence. The above sequence is a stream of five tokens: a begin brace {, and end brace }, and three objects. Here's a more complicated example which deals with nested sequences. The first subsequence contains two objects, while the second is empty:

{ { d1 d2 } { } }

It is also a stream of eight tokens: three {'s, three }'s, and two objects.

Our data model is based, not on functions, but on **productions**, which are stream rewrite rules. A production maps an input stream of tokens to an output stream. An implementation of a production is a **stream translator.** Figure 3 shows a graphical depiction of the increment translator INC:INT→INT. It replaces integers in an input stream with their incremented value, and transmits brace tokens directly.

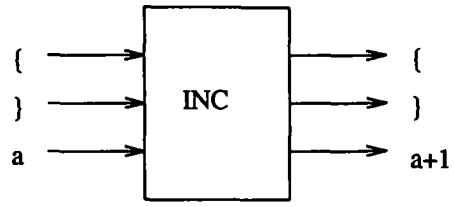

Figure 3. The INC Stream Translator

Thus, INC translates the sequence { 1 2 3 } to { 2 3 4 }. Similarly, { { 4 } { 5 6 } { } } is translated by INC to { { 5 } { 6 7 } { } }. In both examples, the nesting remains the same and only the integers are changed.

A more complicated production is COUNT:{*OBJ}→INT, which replaces a sequence of objects (recognized by a begin brace, a stream of objects, and an end brace) with the number of objects in that sequence. Thus, { 1 2 3 } is mapped by COUNT to 3, and { { 4 } { 5 6 } { } } is mapped to { 1 2 0 }. Note that COUNT always operates on the innermost sequences.

Composition of productions is straightforward. Let AVE:{*INT}→FLOAT be the production that replaces a sequence of integers with its floating point average. The composition of COUNT with AVE, written COUNT.AVE and executed from left to right, computes the average number of objects in a subsequence. Compositions are simple for stream translators, as one links the output of the COUNT translator to the input of AVE (see Fig. 4).

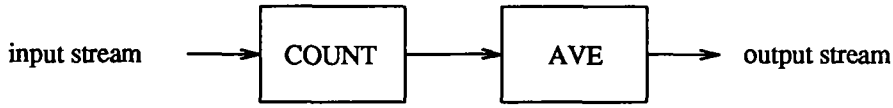

Figure 4. Composition of Stream Translators

Using this computation model, traditional and nontraditional database tasks can be processed by exploiting the inherent extensibility of the functional/production model. (In the following examples, system-defined productions are listed in capital letters). Consider the following expression that lists the names of departments that employ workers over the age of 55:

Dept.WHERE(Emp.Age>55).Dname.PRINT

Dept generates the sequence of all departments in the database. WHERE eliminates departments if they do not employ workers greater than 55. Dname replaces department objects with their names, and PRINT prints streams of names. In principle, handling traditional database operations (insertions, deletions, etc.) is straightforward.

The support for nontraditional database applications is handled in an identical manner - i.e., by introducing new productions. Consider the problem of displaying a 3-dimensional graphics database. Suppose each object in the database is described by collections of polygons. (A box, for example, would have six rectangles; each representing a side of the box). The following expression would display this database:

Graphics.Polygons.MATRIX_MULTIPLY.CLIP.SURFACE_REMOVAL.DISPLAY

Graphics generates the sequence of all graphics objects in the database. Polygons replaces each graphics object with its stream of polygons. MATRIX_MULTIPLY maps an untransformed polygon to a transformed polygon via matrix multiplication in homogeneous coordinates. CLIP clips polygons to a view screen,

SURFACE_REMOVAL eliminates hidden surfaces, and DISPLAY draws polygons on a display terminal.

Further details are given in [Bat86b].

6. Molecular Database Systems Technology

The principles that we have reviewed in the previous sections are leading us to a building-blocks technology for DBMS construction. The building blocks, or **atoms,** of data type and operator extensibility are stream translators, while layers are the atoms of storage structure extensibility. DBMSs are **molecules** of these primitive building blocks.

We are finishing the development of an algebra for molecular database system design which is based on these principles [Bat87]. The algebra reduces a wide spectrum of database algorithms to a few discrete points, called **basic algorithms. Variants,** which constitute the remainder of the spectrum population, can be generated from basic algorithms using simple rewrite rules. The algebra unifies query processing algorithms in centralized DBMSs, distributed DBMSs, and database machines by showing that different systems are described by different rewrite rules. In this way, significant aspects of DBMS architecture design can be reduced to a simple formalism, and rules by which atoms can be composed can be expressed algebraically.

References

[Bat79] D.S. Batory, 'On Searching Transposed Files', **ACM Trans. Database Syst.** 4,4 (Dec. 79), 531-544.

[Bat82] D.S. Batory and C.C. Gotlieb, 'A Unifying Model of Physical Databases', **ACM Trans. Database Syst.** 7,4 (Dec. 1982), 509-539.

[Bat84] D.S. Batory, 'Conceptual-To-Internal Mappings in Commercial Database Systems', **ACM PODS 1984,** 70-78.

[Bat85] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', **ACM Trans. Database Syst.** 10,4 (Dec. 1985), 463-528.

[Bat86a] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K.Tsukuda, B.C. Twichell, T.E. Wise, 'GENESIS: An Extensible Database Management System', to appear in **IEEE Trans. Software Engineering.**

[Bat86b] D.S. Batory and T.Y. Leung, 'Implementation Concepts for an Extensible Data Model and Data Language', TR-86-24, University of Texas at Austin, 1986.

[Bat87] D.S. Batory, 'A Molecular Database System Technology', to appear.

[Bla77] M.W. Blasgen and K.P. Eswaren, 'On the Evaluation of Queries in a Relational Database System', **IBM Systems Journal** 16 (1976), 363-377.

[Bun82] P. Buneman, R.E. Frankel, and R. Nikhil, 'An Implementation Technique for Database Query Languages', **ACM Trans. Database Syst.** 7,2 (June 1982), 164-186.

[Cop85] G.P. Copeland and S.N. Khoshafian, 'A Decomposition Storage Model', **SIGMOD 1985,** 268-279.

[Ham79] M. Hammer and B. Niamir, 'A Heuristic Approach to Attribute Partitioning', **SIGMOD 1979,** 93-100.

[Shi81] D. Shipman, 'The Functional Data Model and the Data Language DAPLEX', **ACM Trans. Database Syst.,** 6,1 (March 1981), 140-173.

[Tur79] M.J. Turner, R. Hammond, and P. Cotton, 'A DBMS for Large Statistical Databases', **VLDB 1979,** 319-327.

[Twi87] B.C. Twichell, 'Design Concepts for an Extensible File Management System', M.Sc. Thesis, Dept. Computer Sciences, University of Texas at Austin, 1987.

An Overview of the EXODUS Project

Michael J. Carey
David J. DeWitt
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

1. INTRODUCTION

In the 1970's, the relational data model was the focus of much of the research in the database area. At this point, relational database technology is well understood, a number of relational systems are commercially available, and they support the majority of business applications relatively well. One of the foremost database problems of the 1980's is how to support classes of applications that are not well served by relational systems. For example, computer-aided design systems, scientific and statistical applications, image and voice applications, and large, data-intensive AI applications all place demands on database systems that exceed the capabilities of relational systems. Such application classes differ from business applications in a variety of ways, including their data modeling needs, the types of operations of interest, and the storage structures and access methods required for their operations to be efficient.

The EXODUS project at the University of Wisconsin [Care85, Care86a, Care86b, Grae87, Rich87]] is addressing the problems posed in these emerging applications by providing tools that will enable the rapid implementation of high-performance, application-specific database systems. EXODUS provides a set of kernel facilities for use across all applications, such as a versatile storage manager and a general-purpose manager for type-related dependency information. In addition, EXODUS provides a set of tools to help the database implementor (DBI) to develop new database system software. The implementation of some DBMS components is supported by tools which actually generate the components from specifications; for example, tools are provided to generate a query optimizer from a rule-based description of a data model, its operators, and their implementations. Other components, such as new abstract data types, access methods, and database operations, must be explicitly coded by the DBI due to their more widely-varying and highly algorithmic nature.[1] EXODUS attempts to simplify this aspect of the DBI's job by providing a set of high-leverage programming language constructs for the DBI to use in writing the code for these components.

2. RELATED PROJECTS

A number of other database research efforts have recently begun to address the problem of building database systems to accommodate a wide range of potential applications via some form of extensibility. Related projects include PROBE at CCA [Daya85, Mano86], POSTGRES at Berkeley [Ston86a, Ston86b], STARBURST at IBM Almaden [Schw86], and GENESIS at UT-Austin [Bato86]. Although the goals of these projects are similar, and each uses some of the same mechanisms to provide extensibility, the overall approach of each project is quite different. STARBURST, POSTGRES, and PROBE are complete database systems, each with a (different) well-defined data model and query language. Each system provides the capability for users to add extensions such as new abstract data types and access methods within the framework provided by their data model. STARBURST is based on the relational model; POSTGRES extends the relational model with the notion of a procedure data type, triggers and inferencing capabilities, and a type hierarchy; PROBE is based on an extension of the DAPLEX functional data model, and includes support for spatial data and a class of recursive queries.

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant DCR-8402818, by IBM through a Fellowship and a Faculty Development Award, by DEC through its Initiatives for Excellence program, and by a grant from the Microelectronics and Computer Technology Corporation (MCC).

[1]Actually, EXODUS will provide a library of generally useful components, such as widely-applicable access methods including B+ trees and some form of dynamic hashing, but the DBI must implement components that are not available in the library.

The EXODUS project is distinguished from all but GENESIS by virtue of being a "database generator" effort as opposed to an attempt to build a single (although extensible) DBMS for use by all applications. The EXODUS and GENESIS efforts differ significantly in philosophy and in the technical details of their approaches to DBMS software generation. GENESIS has a stricter framework (being based on a "building block" plus "pluggable module" approach), whereas EXODUS has certain powerful fixed components plus a collection of tools for a DBI to use in building the desired system based around these components. The remainder of this paper describes the approach of EXODUS in more detail.

3. THE EXODUS ARCHITECTURE

3.1. Overview

Since EXODUS is basically a collection of components and tools that can be used in a number of different ways, describing EXODUS is more difficult than describing the structure/organization of other extensible database system designs that have appeared recently. We believe that the flexibility provided by the EXODUS approach will make the system usable for a much wider variety of applications (as we will discuss later).

The fixed components of EXODUS include the EXODUS *Storage Object Manager*, for managing persistent objects, and a generalized *Dependency Manager* (formerly called the *Type Manager*), for keeping track of information about various type-related dependencies. In addition to these fixed components, EXODUS also provides tools to aid the DBI in the construction of application-specific database systems. One such tool is the *E programming language*, which is provided for developing new database software components. A related resource is a *type-independent module library*; E's generator classes and iterators can be used to produce useful modules (e.g., various access methods) that are independent of the types of the objects on which they operate, and these modules can then be saved away for future use. Another class of tools are provided for *generating* components from a specification. An example is the EXODUS rule-based *Query Optimizer Generator*. We also envision providing similar generator tools to aid in the construction of the front-end portions of an application-specific DBMS. The components of EXODUS are described further in the following sections. More detail on the Storage Object Manager can also be found in [Care86a], the E programming language is described in [Rich87], and details regarding the Query Optimizer Generator and an initial evaluation of its performance can be found in [Grae87].

3.2. The Storage Object Manager

The Storage Object Manager provides *storage objects* for storing data and *files* for logically and physically grouping storage objects together. Also provided are a powerful buffer manager that buffers variable-length pieces of large storage objects, primitives for managing versions of storage objects, and concurrency control and recovery services for operations on storage objects and files.

A storage object is an uninterpreted container of bytes which can be as small (e.g., a few bytes) or as large (e.g., hundreds of megabytes) as demanded by an application. The distinction between small and large storage objects is hidden from higher layers of EXODUS software. Small storage objects reside within a single disk page, whereas large storage objects occupy potentially many disk pages. In either case, the object identifier (OID) of a storage object is an address of the form `(page #, slot #)`. The OID of a small storage object points to the object on disk; for a large storage object, the OID points to its *large object header*. A large object header can reside on a slotted page with other large object headers and small storage objects, and it contains pointers to other pages involved in the representation of the large object. Pages in a large storage object are private to that object (although pages are shared between versions of a large storage object). When a small storage object grows to the point where it can no longer be accommodated on a single page, the Storage Object Manager will automatically convert it into a large storage object, leaving its header in place of the original small object.

All read requests specify an OID and a range of bytes; the desired range of bytes is read into a contiguous region in the buffer pool (even if the bytes are distributed over several partially full pages on disk), and a pointer to the bytes is returned to the caller. Bytes may be overwritten directly, using this pointer, and a call is provided to tell the Storage Object Manager that a subrange of the bytes that were read have been modified (information needed for recovery to take place). For shrinking/growing storage objects, calls to insert bytes into and delete bytes from a specified offset within a storage object are provided, as is a call to append bytes to the end of an object. To make these operations efficient, large storage objects are represented using a B+ tree structure to index data pages on byte offset.

The Storage Object Manager also provides support for versions of storage objects. In the case of small storage objects, versioning is implemented by making a copy of the entire object before applying the update. Versions of large storage objects are maintained by copying and updating only those pages that differ from version to version. The Storage Object Manager also supports the deletion of a version with respect to a set of other versions with which it may share pages. The reason for only providing a primitive level of version support is that different EXODUS applications may have widely different notions of how versions should be supported. We do not omit version management altogether for efficiency reasons — it would be prohibitively expensive, both in terms of storage space and I/O cost, if clients were required to maintain versions of large objects externally by making entire copies.

For concurrency control, two-phase locking of byte ranges within storage objects is used, with a "lock entire object" option being provided for cases where object-level locking will suffice. To ensure the integrity of the internal pages of large storage objects during insert, append, and delete operations (e.g., while their counts and pointers are being changed), non-two-phase B+ tree locking protocols are employed. For recovery, small storage objects are handled by logging changed bytes and performing updates in place at the object level. Recovery for large storage objects is handled using a combination of shadowing and logging — updated internal pages and leaf blocks are shadowed up to the root level, with updates being installed atomically by overwriting the old object header with the new one. A similar scheme is used for versioned objects, but the before-image of the updated large object header (or entire small object) is retained as an old version of the object.

Finally, the Storage Object Manager provides the notion of a *file object*. A file object is an unordered set of related storage objects, and is useful in several different ways. First, the Storage Object Manager provides a mechanism for sequencing through all of the objects in a file, so related objects can be placed in a common file for sequential scanning purposes. Second, objects within a given file are placed on disk pages allocated to the file, so file objects provide support for objects that need to be co-located on disk.

3.3. The Dependency Manager

The EXODUS Dependency Manager is a repository for information related to persistent types.[2] It maintains information about all of the pieces (called *fragments*) that make up a compiled query, including type definitions and other E code, and about their relationships to one another. It also keeps track of the relationship between files and their types (by treating files in a manner similar to fragments). In short, the Dependency Manager keeps track of dependencies between types and most everything else that is related to or dependent upon such information.

More specifically, certain time ordering constraints must hold between the fragments constituting a complete query. For example, a compiled query plan must have been created more recently than the program text for any of the types or database operations that it employs, as otherwise out-of-date code will have been used in its creation. A given abstract data type, or a set of operations, is also likely to have multiple representations (e.g., E source code, an intermediate representation, and a linkable object file), and similar time ordering constraints must hold between these representations. The Dependency Manager's role is thus similar to the Unix™ `make` facility [Feld79]. Unlike `make`, which only examines dependencies and timestamps when it is started up, the Dependency Manager maintains a graph of inter-fragment dependencies at all times (and updates it incrementally).

The Dependency Manager also plays a role in maintaining data abstraction that distinguishes it from `make`. In particular, a given type used by a query plan is likely to use other types to constitute its internal representation. Strictly speaking, the first type is not *dependent* upon the linkable object code of its constituent types' operations; that is, while it must eventually be linked with their code, it is not necessary that their object code be up to date, or even compiled, until link time. We call fragments of this sort *companions*; `make` has no facilities for specifying and using companions. The Dependency Manager requires such a facility, as otherwise it would be unable to provide a complete list of the objects constituting the compiled access plan for a query, which is necessary when a query is to be linked.

The Dependency Manager maintains the correct time ordering of fragments via two mechanisms, *rules* and *actions*. The set of fragments constitutes the nodes of an acyclic directed graph; rules generate the arcs of this graph. When a fragment is found to be older than those fragments upon which it depends (with the dependencies being determined from the rules), a search is made for an appropriate action that can be performed to bring the

[2] As we will explain shortly, new types in EXODUS are defined using the `class` and `dbclass` constructs of the E programming language.

fragment up to date. Both rules and actions are defined using a syntax based on regular expressions to allow a wide range of default dependencies to be specifed conveniently.

3.4. The E Programming Language

A major tool provided by EXODUS is the E programming language and its compiler. E is a extension of C++ [Stro86] that aids the DBI in a number of problem areas related to database system programming, including interaction with persistent storage, accommodation of missing type (class) information, and query compilation. E is designed to be upward compatible with C++, and its extensions include both new language features and a number of predefined classes.

E was designed with the following database system architecture in mind: First, all access methods, data model operators, and utility functions are written in E. In addition to these modules, the Storage Object Manager, and the Dependency Manager, the database system includes the E compiler itself. At run time, database schema definitions (e.g., create relation commands) and queries are first translated into E programs and then compiled. One result of this architecture is a system in which the "impedance mismatch" [Cope84] between type systems disappears. Another is that the system is easy to extend. For example, the DBI may add a new data type by implementing it as an E class, storing its definition and implementation in files, and registering the resulting module with the Dependency Manager for later use.

The following paragraphs describe some of the more important features of E from the standpoint of the DBI. More details can be found in [Rich87].

3.4.1. Generator Classes for Unknown Types

One of the problems faced by the DBI is that many of the types involved in database processing are not known until well after the code needing those types is written. For example, the code implementing a hash-join algorithm does not know what types of entities it will have to join. Similarly, index code does not know what types of keys it will contain nor what type of entities it will index.

To address this problem, E augments C++ with *generator* classes, which are very similar to the parameterized clusters of CLU [Lisk77]. Such a class is parameterized in terms of one or more unknown types; within the class definition, these (formal) type names are used freely as regular type names. This mechanism allows one to define, for example, a class of the form `stack[ T ]` where the specific type (class) `T` of the stack elements is not known. The user of such a class must *instantiate* it by providing specific parameters to the class; e.g., one may declare `x` to be an integer stack via the declaration `stack[ int ] x`. Similarly, the DBI can define the type of a B+ tree node as a class in which both the key type and the type of entity being indexed are class parameters. Later, when the user builds an index over employees on social security number, the system generates and compiles a small E fragment which instantiates `BTnode[ SSN_type, EMP_type ]`. Such instantiation can be efficiently accomplished via a linking process [Atki78].

3.4.2. Class fileof[T] for Persistent Storage

Another problem in database system programming is that most file systems provide the DBI only with untyped storage. Thus, after being read from disk, all data must be explicitly type cast in the DBI's code before it can be operated upon. In addition, since the data resides on secondary storage, the DBI must include explicit calls to the buffer manager in order to use it. These factors increase the amount of code that the DBI must write, and they also provide increased opportunities for coding errors.

E's answer to this problem is the "built-in" generator class `fileof[ T ]` where `T` must be a `dbclass`. A dbclass is declared in the same way as a C++ class with the restriction that a dbclass may contain only other dbclasses. (Predefined dbclasses exist for the fundamental types int, float, char, etc.) Dbclasses were introduced so that the compiler can always distinguish between objects residing only on the heap and those that generally reside on disk (but may also reside in memory) since the implementation of the two is very different.

Instances of the `fileof` generator class are implemented as a descriptor (in memory) associated with a physical file (on disk). This implementation is hidden behind an operational interface that allows the user to bind typed pointers to objects in a file, to create and destroy objects in a file, etc. For example, the following function returns the sum of all the integers in a file of integers. (The file is passed by reference.)

```
int filesum( fileof[dbint]& f )
{
    dbint *p;    /* dbint is the predefined dbclass for int */
    int sum = 0;
    for( p = f.getfirst(); p != 0; p = f.getnext( p ) ) sum += *p;
    return sum;
}
```

Although this example is extremely simple, it illustrates the two features mentioned above. The first is that no casting is needed to use the integer pointer `p`; the second is that no buffer calls are necessary to access the objects in file `f`. Clearly, an important research direction related to the implementation of E is the optimization of the calls to the buffer manager generated by the E compiler (especially for files containing very large objects such as images).

3.4.3. Iterators for Scans and Query Processing

A typical approach for structuring a database system is to include a layer which provides *scans* over objects in the database. A scan is a control abstraction which provides a state-saving interface to the "memoryless" storage systems calls; this interface is needed for the record-at-a-time processing done in higher layers. A typical implementation of scans will allocate a data structure, called a *scan descriptor*, to save all needed state between calls; it is up to the user to pass the descriptor with every call.

The control abstraction of a scan is provided in EXODUS via the notion of an *iterator* [Lisk77, OBri86]. An iterator is a coroutine-like function that saves its data and control states between calls; each time the iterator produces (`yields`) a new value, it is suspended until resumed by the client. Thus, no matter how complicated the iterator may be, the client only sees a steady stream of values being produced. Finally, for implementation reasons, the client can only invoke an iterator within a new kind of structured statement, the `iterate` loop (which generalizes the `for ... in` loop of CLU).

The general idea for implementing scans should now be clear. For example, to implement a scan over B+ trees, we would write an iterator function which takes a B+ tree, a lower bound, and an upper bound as arguments. It would begin by searching down to the leaf level of the tree for the lower bound, keeping a stack of node pointers along the way. It would then walk the tree, yielding object references one at a time, until the upper bound is reached. At that point, the iterator would terminate.

Iterators are also used to piece executable queries together from a parse tree. If we consider a query to be a pipeline of processing filters, then each stage can be implemented as an iterator which is a client of one or more iterators (upstream in the pipe) and which yields its results to the next stage (downstream in the pipe). Execution of the pipeline will be demand-driven in nature. For example, the DBI for a relational DBMS would write code for select, project, and join as iterators implementing filters. Given the parse tree of a user query, it is a fairly simple task to produce E code that implements the pipeline.

3.5. Type-Independent Access Methods and Operator Methods

Layered above the Storage Object Manager is a collection of access methods that provide associative access to files of storage objects and further support for versioning (if desired). For access methods, EXODUS will provide a library of type-independent index structures including B+ trees, Grid files [Niev84], and linear hashing [Litw80]. These access methods will be implemented using the class generator and iterator capabilities provided by the E programming language. This capability enables existing access methods to be used with DBI-defined abstract data types without modification — as long as the capabilities provided by the data type satisfy the requirements of the access methods. In addition, a DBI may wish to implement new types of access methods in the process of developing an application-specific database system. EXODUS provides mechanisms to greatly simplify this task. First, since new access methods are written in E, the DBI is shielded from having to map main memory data structures onto storage objects and from having to write code to deal with buffering. E will also simplify the task of handling concurrency control and recovery for new access methods.

Layered above the access methods is a set of operator methods that implement the operations of the application's chosen data model. As for access methods, the class generator and iterator facilities of E facilitate the development of operator methods. Generally useful methods (e.g., selection) will be made available in a type-independent library; methods specific to a given application domain will have to be developed by the DBI.

3.6. The Rule-Based Query Optimizer Generator

Since we expect that EXODUS will be used for a wide variety of applications, each with a potentially different query language, it is not possible for EXODUS to furnish a single generic query language, and it is accordingly impossible for a single query optimizer to suffice for all applications. As an alternative, a generator for producing query optimizers for algebraic query languages has been implemented. The input to the query optimizer generator is a collection of rules regarding the operators of the target query language, the transformations that can be legally applied to these operators (e.g., pushing selections before joins), and a description of the methods that can be used to execute each operator in the query language (including their costs and side effects). The Query Optimizer Generator transforms these description files into C source code[3], producing an optimizer for the application's query language. Later, to optimize queries using the resulting optimizer, a query is first parsed and converted into its initial form as a tree of operators; it is then transformed by the generated optimizer into an optimized execution plan expressed as a tree of methods. During the process of optimizing a query, the optimizer avoids exhaustive search by using AI search techniques and employing past (learned) experience to direct the search. As described above, each method in the tree produced by the optimizer is implemented as an iterator generator in E. Thus, a post-optimization pass over the plan tree is made to produce E code corresponding to the plan. For queries involving more than one operator, the iterators are nested in a manner that allows the query to be processed in a pipelined fashion, as mentioned earlier.

4. APPLICATION-SPECIFIC DBMS DEVELOPMENT

Figure 1 presents a sketch of the architecture of a functionally complete, application-specific database system implemented using EXODUS. The components in Figure 1 that are implemented by the DBI in E are the access methods and operator methods. As discussed above, EXODUS provides a library of type-independent access methods, so it might not be necessary for a DBI to actually implement any access methods. EXODUS will also provide a library of methods for a number of operators that operate on a single type of storage object (e.g., selection), but it will not provide application or data model specific methods. For example, since a method for examining objects containing satellite image data for the signature of a particular crop disease would not be useful in general, it does not belong in such a library. In general, the DBI will need to implement (using E) one or more methods for each operator in the query language associated with the target application. The DBI must also write code (i.e., dbclass member functions) for the operations associated with each new abstract data type that he or she wishes to define.

To clarify by using a familiar example, a DBI who wanted to implement a relational DBMS for business applications via the EXODUS approach would have to obtain code for the desired access methods (e.g., B+ trees and linear hashing) by extracting existing code from the library and/or by writing the desired code from scratch in E. Similarly, code must be obtained for the operator methods (e.g., relation scan, indexed selection, nested loops join, merge join, etc.) and for various useful types (e.g., date and money). A DBI implementing a database management system for an image application would have to implement an analogous set of routines, presumably including various spatial index structures, operations that manipulate collections of images, and an appropriate set of types. As discussed earlier, E is provided to greatly simplify these programming tasks.

Finally, the top level of the EXODUS architecture consists of a set of components that are generated from DBI specifications. One such component is the query optimizer and compiler. We also plan to investigate and develop tools to automate the process of producing new DML/DDL components, which are the query parser and DDL support components shown in Figure 1. (This idea is similar to the data model compiler notion of [Mary86].) DML components generate operator trees to be fed to the query optimizer, while DDL components produce compiled E code; that is, user-level schema definitions result in the definition of associated E types (which are stored away and registered with the Dependency Manager) and E code to create the associated EXODUS files.

Note that is also possible to use E as a lower-level mechanism for accessing a database directly, for applications needing such low-level access. Assume that one has used the tools provided by EXODUS to construct an application-specific database system. "Normal" accesses to the database would be processed through its ad-hoc or embedded query interfaces, while those applications needing direct access to storage objects would be developed using E. Since schema information for all storage objects is maintained internally in E form, the application

[3]Note: While E is the language that the DBI will use to implement a DBMS, we are implementing the various components of EXODUS in C.

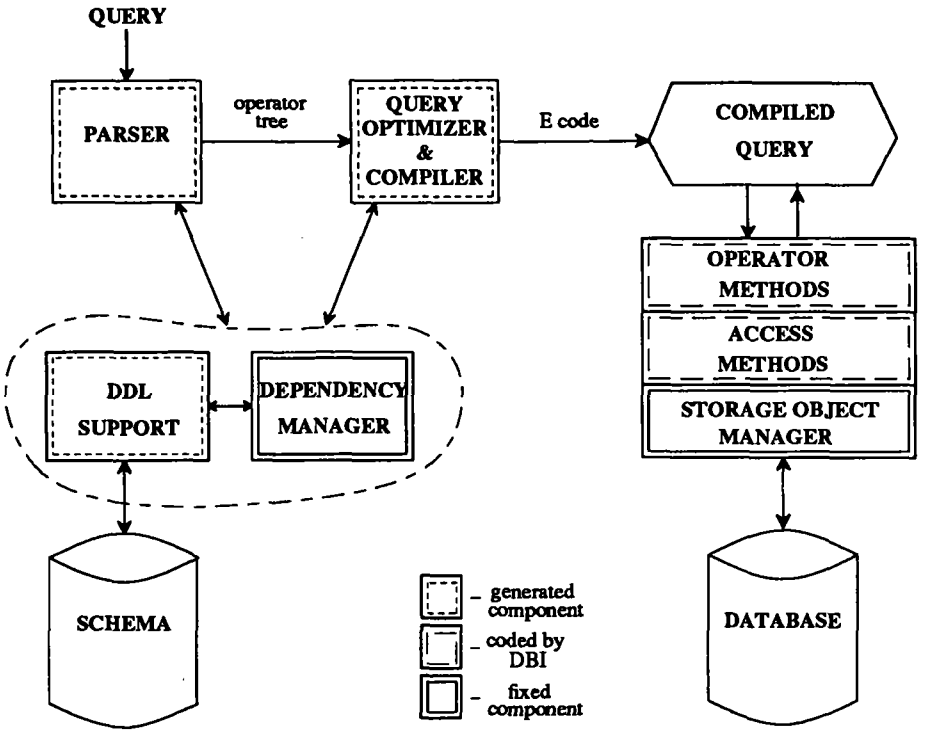

Figure 1: An EXODUS-Based DBMS.

programs can access storage objects corresponding to entity instances that were created via the ad-hoc query interface. One could also layer an application program on top of the access methods or operator methods layers without necessarily using the front-end portion of the system. Thus, one shared database can be used by both types of applications with little or no loss of efficiency and minimal loss of data independence. For certain applications, the availability of such a direct interface is critical to obtain reasonable performance [Rube87]. The flexibility of the EXODUS approach to extensible database systems will enable users to customize the system to fit such needs.

5. SUMMARY

In this paper we have briefly described the components of the EXODUS extensible database system and how it supports the development of application-specific DBMSs. The current state of the project is that the Query Optimizer Generator is running and is undergoing evaluation, much of the Dependency Manager is operational, and the Storage Object Manager should be completed soon as well. The E programming language design is finished, and the implementation of the E compiler is now underway. Our goal is to have all of the pieces ready for use later on this year. Once they are ready, we will use the EXODUS tools to build a relational DBMS prototype, after which we plan to apply the tools to data management problems that arise in more challenging application areas (e.g., image data processing, computer-aided software engineering, or data-intensive expert systems).

ACKNOWLEDGEMENTS

We wish to acknowledge the graduate students associated with the EXODUS project, as they have contributed significantly to the ideas described here: Daniel Frank, Goetz Graefe, Joel Richardson, Eugene Shekita, and Scott Vandenberg. Also, Dan Schuh and Isa Hashim are working on the implementation of the E language and portions of the Storage Object Manager (respectively).

REFERENCES

[Atki78] Atkinson, R., B. Liskov, and R. Scheifler, "Aspects of Implementing CLU," *Proc. of the ACM National Conf.*, 1978.

[Bato86] Batory, D., et al, "GENESIS: A Reconfigurable Database Management System," Tech. Rep. No. TR-86-07, Department of Computer Sciences, University of Texas at Austin, March 1986.

[Care85] Carey, M. and D. DeWitt, "Extensible Database Systems," *Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Care86a] Carey, M., et al, "Object and File Management in the EXODUS Extensible Database System," *Proc. of the 1986 VLDB Conf.*, Kyoto, Japan, August 1986.

[Care86b] Carey, M., et al, "The Architecture of the EXODUS Extensible DBMS," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Asilomar, CA, September 1986

[Cope84] Copeland, G. and D. Maier, "Making Smalltalk a Database System," *Proc. of the 1984 SIGMOD Conf.*, Boston, MA, May 1984.

[Daya85] Dayal, U. and J. Smith, "PROBE: A Knowledge-Oriented Database Management System," *Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Grae87] Graefe, G. and D. DeWitt, "The EXODUS Optimizer Generator," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Feld79] Feldman, S., "Make — A Program for Maintaining Computer Programs," *Software — Practice and Experience*, Vol. 9, 1979.

[Lisk77] Liskov, B., et al, "Abstraction Mechanisms in CLU," *Comm. ACM*, 20(8), August 1977.

[Litw80] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," *Proc. of the 1980 VLDB Conf.*, Montreal, Canada, October 1980.

[Mano86] Manola, F., and U. Dayal, "PDM: An Object-Oriented Data Model," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Mary86] Maryanski, F., et al, "The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Niev84] Nievergelt, J., H. Hintenberger, and K. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. on Database Systems*, Vol. 9, No. 1, March 1984.

[Obri86] O'Brien, P., B. Bullis, and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Rich87] Richardson, J., and M. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Rube87] Rubenstein, W. and R. Cattell, "Benchmarks for Database Response Time," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Schw86] Schwarz, P., et al, "Extensibility in the Starburst Database System," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Ston86a] Stonebraker, M., and L. Rowe, "The Design of POSTGRES," *Proc. of the 1986 SIGMOD Conf.*, Washington, DC, May 1986.

[Ston86b] Stonebraker, M., "Object Management in POSTGRES Using Procedures," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Stro86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, 1986.

An Extensible Framework for Multimedia Information Management

Darrell Woelk and Won Kim

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

1. Introduction

The management of multimedia information such as images and audio is becoming an important feature of computer systems. Multimedia information can broaden the bandwidth of communication between the user and the computer system. Although the cost of the hardware required for the capture, storage, and presentation of multimedia data is decreasing every year, the software for effectively managing such information is lacking. Future database systems must provide this capability if we are to be able to share large amounts of multimedia information among many users.

In our earlier work [WOEL86], we concluded that an object-oriented approach would be an elegant basis for addressing the data modelling requirements of multimedia applications. Subsequently, we developed an object-oriented data model by extracting a number of common concepts from existing object-oriented programming languages and systems, and then enhancing them with a number of additional concepts, including versions and predicate-based access to sets of objects. The data model, described in detail in [BANE87], has been implemented in a prototype object-oriented database system, which we have named ORION. ORION is implemented in Common Lisp [STEE84], and runs on a Symbolics 3600 Lisp Machine[SYMB85]. ORION adds persistence and sharability to the objects created and manipulated by object-oriented applications from such domains as artificial intelligence, computer-aided design, and office information systems. Important features of ORION include transaction management, versions [BANE87], composite objects [BANE87], and multimedia information management. The Proteus expert system [PETR86] developed by the MCC Artificial Intelligence Program has recently been modified to interface with ORION. The MUSE multimedia system [LUTH87] developed by the MCC Human Interface Program will be integrated with ORION in the near future.

The focus of this paper is multimedia information management in ORION. In particular, we will describe how we support extensibility (generalizability and modifiability) for the system developers and end users to extend the system, by adding new types of devices and protocols for the capture, storage, and presentation of multimedia information. To satisfy this requirement, we have implemented the multimedia information manager (MIM) as an extensible framework explicitly using the object-oriented concepts. The framework consists of definitions of class hierarchies and a message passing protocol for not only the multimedia capture, storage, and presentation devices, but also the captured and stored multimedia objects. Both the class hierarchies and the protocol may be easily extended and/or modified by system developers and end users as they see fit.

2. Extensibility Objectives

Extensibility is required to support new multimedia devices and new functions on multimedia information. For example, a color display device may be added to a system with relative ease, if at a high level of abstraction the color display can be viewed as a more specialized presentation device for spatial multimedia objects than a more general display device which is already supported in the system. The color display device may be further specialized by adding windowing software, and the win-

dows can in turn be specialized to create new display and input functionality. Future database systems should support the presentation of multimedia information on these presentation devices as described in [CHRI86a]. Further, database systems must also support the capture of multimedia information using such capture devices as cameras and audio digitizers.

It is also important to be able to add new multimedia storage devices, or to change the operating characteristics of storage devices. For example, read-only CD ROM [CDRO86] disks and write-once digital optical disks [CHRI86b] are both storage devices having desirable characteristics for the storage of certain types of multimedia information. The integration of these hardware devices into a system is becoming easier due to standard disk interfaces such as SCSI [KILL86]. A natural framework for logically accessing these devices must be provided by the database system.

Even multimedia information stored on magnetic disk may require special formatting for efficiency in storage and access. For example, an image may be stored using approximate geometry as described in [OREN86]. This storage format allows the expression of powerful spatial queries. The new storage format and the new query functionality can be defined as specializations of the more general capability for storing and presenting images.

3. Implementation of the Multimedia Information Manager

We have analyzed scenarios for the capture, storage, and presentation of many types of multimedia information and have generalized these into a framework of classes and a message protocol for interaction among instances of these classes. This framework is highly extensible, since it is based on the class lattice and message passing concepts of the object-oriented paradigm. In Section 3.1 we will describe some of the multimedia classes which are defined for ORION. Section 3.2 will present the message passing protocol among instances of these classes, in terms of the storage and presentation of a bit-mapped image.

3.1 Class Definitions

Multimedia information is captured, stored, and presented in ORION using lattices of classes which represent capture devices, storage devices, captured objects, and presentation devices. However, each instance of one of the device classes represents more than just the identity of a physical device as we will describe in the following paragraphs. The class lattices for the presentation and storage of multimedia information are described in this section. The class lattice for capture device classes is not described here due to space limitations. The capture device class lattice is described in [WOEL87].

3.1.1 Presentation-Device Classes

The MIM uses ORION classes to represent presentation devices available on the system. An *instance* of the presentation-device class, however, represents **more than just the identity of a physical presentation device**. Each instance also has attributes which further specify, for example, where on the device a multimedia object is to be presented and what portion of a multimedia object is to be presented. These pre-defined presentation-device instances can be stored in the database and used for presenting the same multimedia object using different presentation formats. Methods associated with a class are used to initialize parameters of a presentation device and initiate the presentation process. The class lattice for the presentation devices is shown in Figure 1. The shaded classes are provided with ORION. Other classes in the lattice are shown to indicate potential specializations for other media types by specific installations.

Figure 2 shows details of a portion of the class lattice for the presentation-device class. The screen-window subclass represents a window on a workstation screen that is to be used to display an image. An instance of the screen-window class has the attributes win-upper-left-x, win-upper-left-y,

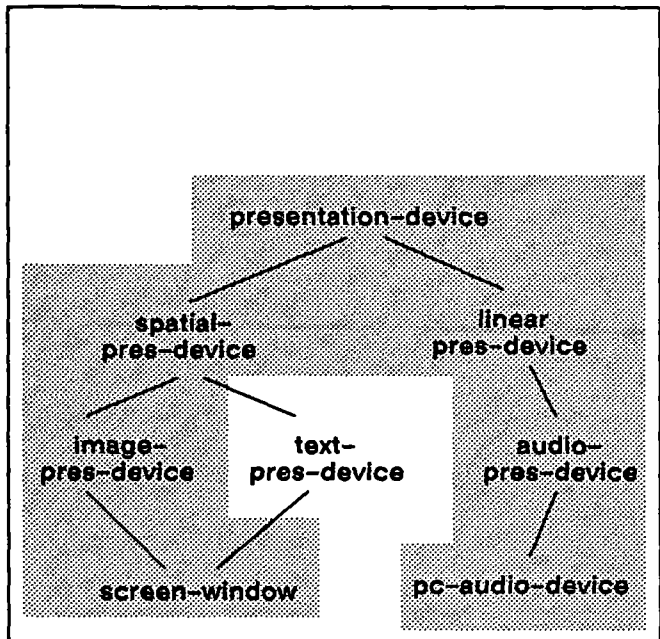

Figure 1. Presentation Device Class Lattice

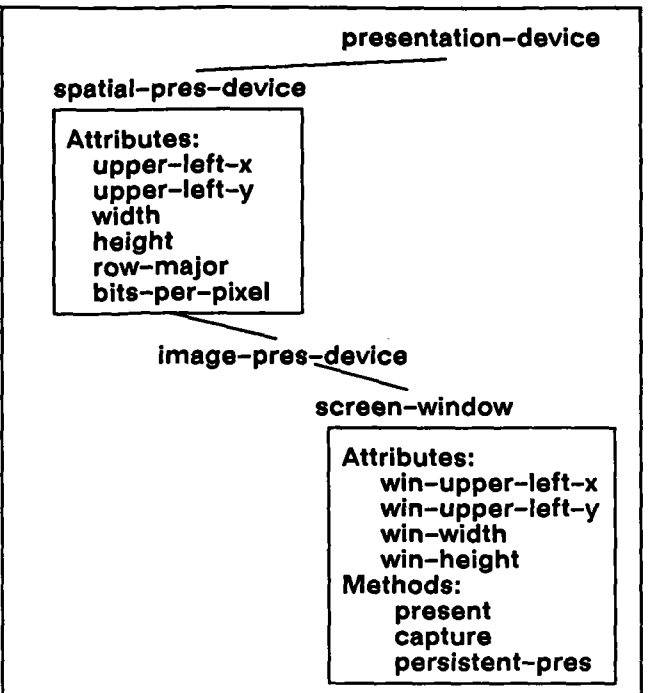

Figure 2. Details of Presentation Device Class Lattice

win-width, and win-height that represent where the window is positioned on the workstation screen. It inherits from the spatial-pres-device class the attributes upper-left-x, upper-left-y, width, and height that specify the rectangular area of an image that is to be displayed. This screen-window instance can be stored in the database and used whenever a specific rectangular area of an image is to be displayed in a specific position on the workstation screen.

3.1.2 Captured-Object, Storage-Device, and Disk Stream Classes

We have adapted the storage and access techniques for multimedia objects in ORION from previous research into the manipulation of long data objects [HASK82]. Every multimedia object stored in ORION is represented by an instance of the class captured-object or one of its subclasses. Figure 3 illustrates the class lattice for captured objects. The captured-object class defines an attribute named storage-object which has as its domain the class storage-device. The class lattice for storage devices and for disk streams are also shown in Figure 3. Transfer of data to and from storage-device instances is controlled through disk-stream instances. The shaded classes in Figure 3 are provided with ORION. Other classes in the lattice indicate potential specializations.

Figure 4 shows details of a portion of the class lattice for the captured-object class, the storage-device class, and the disk-stream class. Each instance of the captured-object class has a reference to a storage-device instance stored in its storage-object attribute. The spatial-captured-object class has attributes which further describe spatial objects. The attributes width and height describe the size and shape of the spatial object. The attribute row-major indicates the order in which the transformation from linear to spatial coordinates should take place. The attribute bits-per-pixel specifies the number of bits stored for each pixel in the spatial object.

As with presentation-device instances, each mag-disk-storage-device instance represents **more than just the identity of a magnetic storage device.** Each instance describes the portion of the device which is occupied by a particular multimedia object. The mag-disk-storage-device class has the block-list attribute which contains the block numbers of the physical disk blocks that make up a multi-

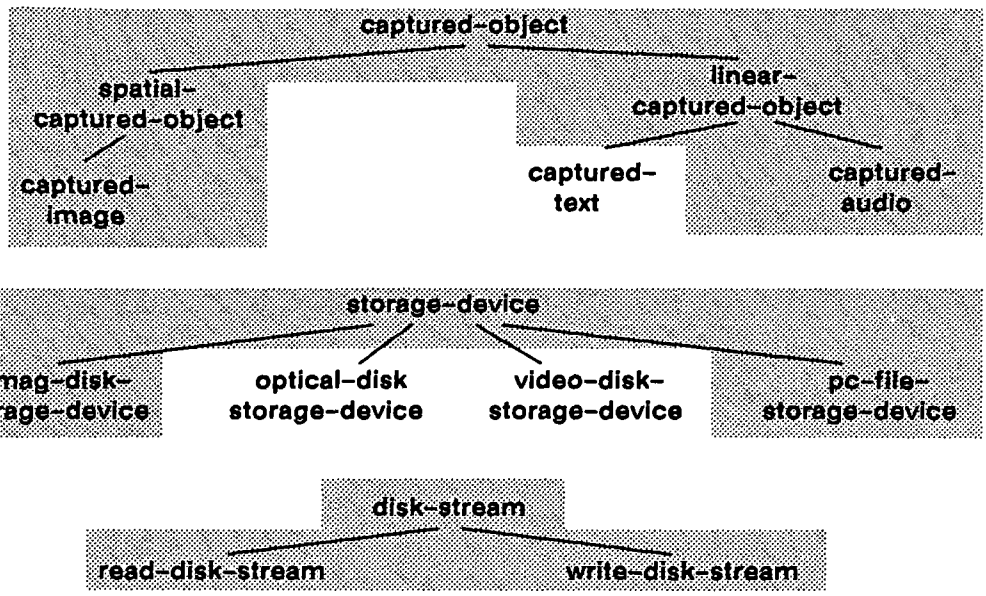

Figure 3. Captured-Object, Storage-Device, and Disk Stream Class Lattices

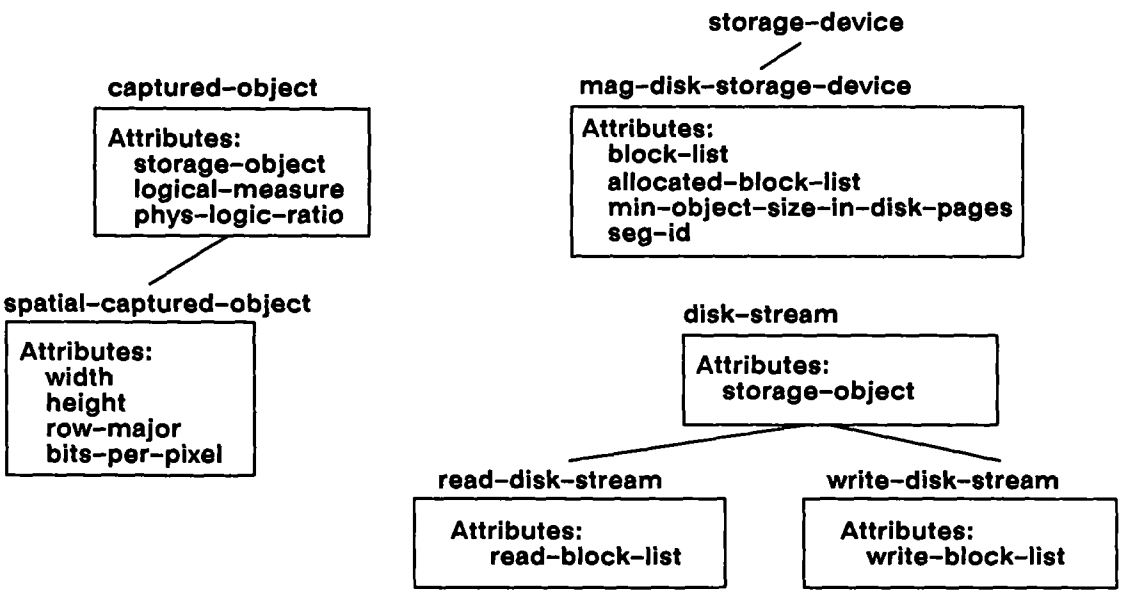

Figure 4. Details of Captured-Object, Storage-Device, and Disk Stream Class Lattices

media object. The allocated-block-list attribute specifies the blocks in the block-list which were actually allocated by this mag-disk-storage-device instance. The min-object-size-in-disk-pages attribute specifies the number of disk pages that should be allocated each time data is added to a multimedia object. The seg-id attribute specifies the segment on disk from which disk pages are to be allocated.

An instance of the read-disk-stream class is created whenever a multimedia object is read from disk. The read-disk-stream instance has a storage-object attribute which references the mag-disk-storage-device instance for the multimedia object. It also has a read-block-list attribute which main-

tains a cursor indicating the next block of the multimedia object to be read from disk. Similarly, an instance of the write-disk-stream class is created whenever data is written to a multimedia object.

3.2 Message Passing Protocol for Presentation

This section will describe the message protocol for the presentation of multimedia information using the ORION classes described in the previous section. The protocol will be discussed by using the example of a bit-mapped image; however, the protocol is similar for many types of multimedia information. There is a similar message passing protocol for the capture of multimedia information which is not described here because of space limitations. This capture protocol is described in [WOEL87].

Figure 5 shows an instance of a class called **vehicle** which has been defined by an application program. It also shows instances of the **image-pres-device**, **captured-image**, **read-disk-stream**, and **mag-disk-storage-device** classes described earlier. The arrows represent messages sent from one instance to another instance. The **vehicle** instance has an image attribute that specifies the identity of a **captured-image** instance that represents a picture of the vehicle. It also has a display-dev attribute that specifies the identity of an **image-pres-device** instance. This **image-pres-device** instance has attributes pre-defined by the user that specify where the image is to be displayed on the screen and what part of the image should be displayed. When the **vehicle** instance receives the *picture* message, the picture method defined for the class **vehicle** will send a *present* message shown below to the specified **image-pres-device** instance.

(*present* **presentation-device** captured-object [physical-resource])

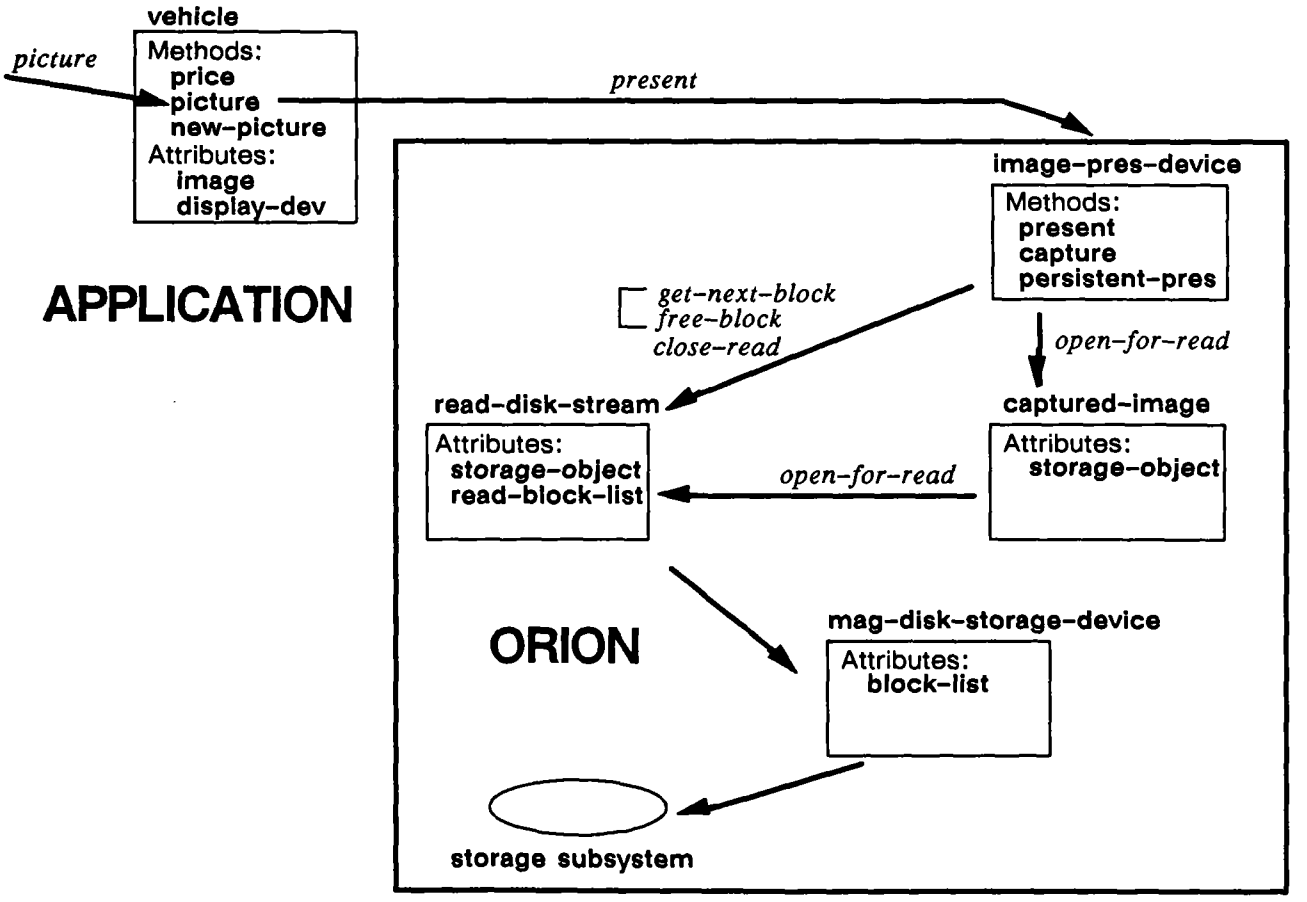

Figure 5. Message Passing Protocol for Presentation of Multimedia Information

The physical-resource parameter above specifies a physical resource, such as the address of the video frame buffer for image presentation. We use italics to denote the message name, bold-face for the object receiving the message, non-bold face for the parameters of a message, and square brackets for optional parameters. Classes specified for the parameters in these messages will always be the most general class acceptable. In the example above, the captured-object parameter will have a value which is an instance of the **captured-image** class, a subclass of the **captured-object** class.)

The *present* method of the **image-pres-device** class transfers image data from the **captured-image** instance and displays the image on a display device. The **image-pres-device** instance has attributes which specify the rectangular portion of the image to be displayed. It translates these rectangular coordinates into linear coordinates to be used for reading the image data from disk. It then initiates the reading of data by sending the following message to the **captured-image** instance:

(*open-for-read* **captured-object** [start-offset])

The start-offset is an offset in bytes from the start of the multimedia object.

The **captured-image** instance then creates a **read-disk-stream** instance and returns its identity to the **image-pres-device** instance. The **image-pres-device** will then send the following message to the **read-disk-stream**:

(*get-next-block* **read-disk-stream**)

The **read-disk-stream** instance calls the ORION storage subsystem to retrieve a block of data from disk. The address of the ORION page buffer containing the block is returned. The **image-pres-device** instance will transfer the data to a physical presentation device, and then send the following message to the **read-disk-stream**:

(*free-block* **read-disk-stream**)

A cursor will also be automatically incremented so that the next *get-next-block* message will read the next block of the multimedia object. When the data transfer is complete, the **image-pres-device** sends a *close-read* message to the **read-disk-stream** instance.

New methods may be written to extend the system so that the media type of the presentation-device and the captured-object may be different. For example, an audio-pres-device instance presenting a captured-text instance could result in text-to-speech translation.

4. Concluding Remarks

In this paper, we described our implementation of the Multimedia Information Manager (MIM) for the ORION object-oriented database system and how it met our design objectives for extensibility. A framework representing multimedia capture, storage, and presentation devices has been implemented using ORION classes. This framework may be specialized by system developers and end users to extend the functionality of the MIM. A message passing protocol was defined for the interaction among instances of these classes. This protocol may also be specialized.

Using the multimedia classes and message passing protocol described in this paper, we have implemented capture, storage, and presentation of bit-mapped images and audio with ORION on the Symbolics LISP Machine. We were able to use the Symbolics Flavors window system for displaying images but we did not wish to add special-purpose camera or audio-recording hardware to the Symbolics for capturing images, capturing audio, and presenting audio. We did have access over a local area network to other systems which had this type of multimedia capability. Therefore, we created new classes to represent remote capture and presentation devices by further specializations of the capture-device and presentation-device classes. The present and capture methods for these classes

were specialized in some cases to move captured data across the local area network and in other cases to actually capture multimedia data remotely, store it in the remote device, and present it remotely under the control of ORION.

References

[BANE87] Banerjee, J., H. T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H. J. Kim. "Data Model Issues for Object-Oriented Applications," to appear in *ACM Trans. on Office Information Systems*, April 1987.

[CDRO86] *CD ROM, The New Papyrus,* edited by S. Lambert and S. Ropiequet, Microsoft Press, Redmond, WA., 1986.

[CHRI86a] Christodoulakis, S., F. Ho, and M Theodoridou. "The Multimedia Object Presentation Manager of MINOS: A Symmetric Approach," *Proc. ACM SIGMOD Intl Conf. on the Management of Data*, May 1986, pp. 295-310.

[CHRI86b] Christodoulakis, S., and C. Faloutsos. "Design and Performance Considerations for an Optical Disk-Bases, Multimedia Object Server," *IEEE Computer*, Dec. 1986, pp. 45-56.

[HASK82] R. Haskin and R. Lorie. "On Extending the Functions of a Relational Database System," in *Proc. ACM SIGMOD Intl Conf. on Management of Data*, June 1982, pp. 207-212.

[KILL86] Killmon P. "For Computer Systems and Peripherals, Smarter is Better," *Computer Design*, January 15, 1986, pp. 57-70.

[LUTH87] Luther W., D. Woelk, and M. Carter. "MUSE: Multimedia User Sensory Environment," *IEEE Knowledge Engineering Newsletter*, February 1987.

[OREN86] Orenstein J. "Spatial Query Processing in an Object-Oriented System," *Proc. ACM SIGMOD Intl Conf. on the Management of Data*, May 1986, pp. 326-336.

[PETR86] Petrie, C., D. Russinoff, and D. Steiner. "Proteus: A Default Reasoning Perspective," *Fifth Generation Systems Conf.*, National Institute for Software, Washington, D.C., October, 1986.

[STEE84] Steele, G. Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb, "Common Lisp," *Digital Press*, 1984.

[SYMB85] Symbolics Inc., "User's Guide to Symbolics Computers," *Symbolics Manual #* 996015, March 1985.

[WOEL86] Woelk, D., Won Kim, and W. Luther. "An Object-Oriented Approach to Multimedia Databases," *Proc. ACM SIGMOD Intl Conf. on the Management of Data*, May 1986, pp. 311-325.

[WOEL87] Woelk, D. and W. Kim, "Multimedia Information Management in an Object-Oriented Database System," to appear in *Proc. 13th Intl Conf. on Very Large Data Bases,* September, 1987.

DASDBS: A Kernel DBMS and Application-Specific Layers

Hans-Joerg Schek, Technical University of Darmstadt[1]

1. History

The initial ideas about a new database system arose while the author worked on the AIM project at the IBM Heidelberg Scientific Center. This project started in 1978 with the objectives of integrating textual data with data base systems. The need for "complex objects" was observed there as well as similar requirements from other applications such as geographical information systems and engineering applications. After moving to the University of Darmstadt the DASDBS (Darmstadt Database System) project was established in early 1983 with the main objective to build a modular and configurable database system. This objective complemented the objectives of the ongoing AIM project /Da86/.

2. Motivation and Requirements

Our motivation for the project came from the insight that no single database system can satisfy "all" applications. Rather one should facilitate the development of specific database systems for different application classes. This position is justified by the requirements for new database systems supporting advanced applications such as **object orientation and knowledge representation** which must support notions of "molecules" (Buchmann) or "complex objects" (Lorie). The integration of "concepts" or "frames" known from AI is also desirable. **Extensibility and modularity** means that the DBMS must be extensible (ADT support) to application-specific needs. Suitable components of the system must be replaceable, depending upon the demands of the application. A requirement which strongly influenced the design of DASDBS is **set orientation:** Sets of complex objects are often needed in application programs rather than single data items with one call of the database. The database architecture must preserve the set orientation down to the operating system level in order to guarantee the required performance.

3. Kernel Architecture

Facing these requirements, several approaches to new DBMS architectures have been proposed /SL83/: Special-purpose systems for specific applications, full-scale next-generation DBMSs which try to identify an interface suitable for all new applications, and database kernel architectures or extensible DBMSs. We follow the latter direction: We pursue an architecture that builds application-specific front-ends on top of a common and (nearly) application- independent DBMS kernel. As shown in Fig. 1, DASBS is a database system family. Based on the kernel, different layers support various application classes as geo-sciences, office filing and retrieval, standard relational applications, and frames or molecule-oriented models. Within the DASDBS project we are working on these four application classes.

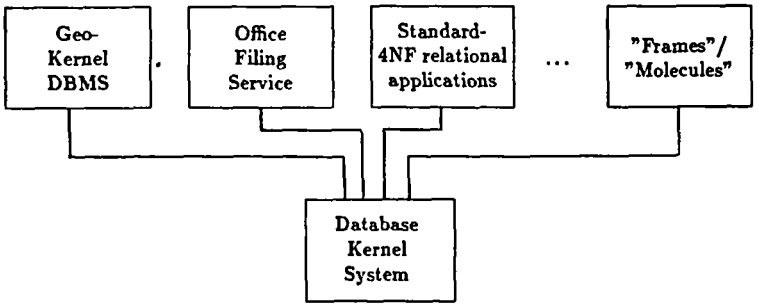

Figure 1: Architecture of the DASDBS family

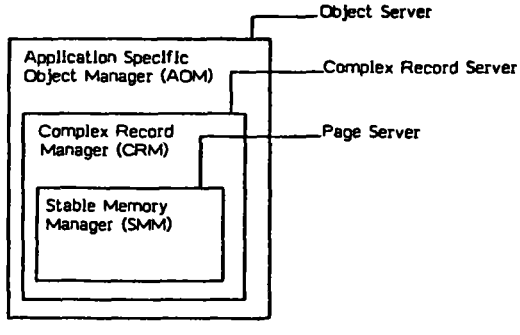

Figure 2: DASDBS as 3-level interface system

For one class, the architecture can be represented as shown in Fig. 2. The application-specific object manager (AOM) utilizes the services of the complex record manager (CRM) which in turn is supported by the functions provided by the stable memory manager (SMM). All three interfaces are set-oriented, i.e. application objects are

---

1 present address: IBM Almaden Research Center (SCHEK at ALMVMA.bitnet)

mapped to sets of complex records if necessary, a complex record in turn is mapped to a set of pages, which is ideally provided by the operating system in one call of the I/O subsystem. Overviews on the architecture are given in /DOPSSW85,SWe86/.

4. DASDBS Main Concepts

The Kernel: A difficult design decision was the functionality of a general-purpose DBMS kernel. Discussion of this topic is contained in /PSSWD87/. Basically, complex records offerd by the kernel are tuples of nested (NF2) relations. Operations at the kernel interface form a a subset of the NF2 algebra /SS86/, characterized by the single-pass property on a single (nested) relation /Sche85/. In addition, addresses of records and subrecords are made available in order to build access paths on top of the kernel. The storage scheme and addressing of records and components within a record as well as the management of the page set forming the storage cluster of a complex record are described in /DPS86/.

Transaction Management: In our multi-level architecture, lower levels provide basic services for higher levels. This is also valid for transaction management. In order to benefit from the semantics of operations in the definitions of conflicts at higher layers, and to get rid of so-called pseudo-conflicts at lower levels we provide transaction management (conceptually) at any level. Locks at a particular layer are only needed to make single actions of the next higher layer appear atomic, and are released as soon as possible. Only "semantic" locks at the top level are held until EOT. This is the generalization of the principle of "open nested transactions" applied in System R. More details can be found in /WS84,We86/.

5. Higher Layers

Geometric Objects and ADT Support:
Geometric objects roughly consist of usual attributes and of a geometric description which may have various formats. A tempting idea at the beginning of our project was to utilize our notion of complex records directly. Starting with geometric primitives like points or line segments or rectangles, we could have built higher-level objects by a repeated combination of primitives. However, in pursuing this idea, we were faced with the problem that any geometric description defined at some higher level must be mapped to some of the predefined primitives and their combinations. If this is not possible, e.g. for a new type of spline, its representation must be available to the DBMS. This is neither desirable nor necessary. We have shown in /SWa86/ that, instead of actually building access routines for each new type into the DBMS code, it is more elegant to call a few user-supplied functions (such as test, clip, and compose) defined on the various geometries a user wants support for. Such functions are necessary to create storage clusters according to spatial neighborhood or to create and to maintain indices for spatial queries. The nice observation here is that these functions can be defined for many different geometries which achieves extensibility simply by switching to a specific instance of one of these functions.

Office Filing and Retrieval: The office filing and retrieval layer is different from the geometric layer in that a different method of indexing (text search) must be applied. The mapping of office objects (according to a proposed filing and retrieval standard interface) to our kernel is described in /PSSW87/. A hierarchical signature search technique, also applicable for file directories, is described in /De86/. Similar to the direction chosen for geometries, the management of textual attributes is also regarded as a user-specific data type, an approach which was also taken in the AIM project.

4NF Relations: The kernel will also support classical flat relations for applications that want them. Although efficient implementations for relational databases exist, we see the following advantages when supported by our kernel. We may store the result of joining and nesting some of the 4NF relations. Therefore some of the expected joins are already materialized /SPS87/. In order to benefit from this observation, an algebraic optimizer has been developed which transforms 4NF expressions into "optimal" NF2 kernel expressions /Scho86/.

6. Discussion and Conclusion

Other projects on extensible database systems described in this issue of Database Engineering, such as STARBURST, POSTGRES, or PROBE, aim at an extension of a full-scale database system, i.e. all main

components of an (existing) database system must be extensible. EXODUS and GENESIS will provide even a database generator for specific databases. Our project is much more conservative: The higher layers, e.g. the query processor or optimizer, are not generated automatically. Rather, they have to be coded on top of the kernel. However, our kernel should provide a (much?) better basis for the development of special-purpose databases than current operating systems and their file management systems. Moreover, we expect to utilize some modules such as those supporting B-trees in many higher layers.

We are trying also to restrict extensibility, in the sense of supporting domain-level ADTs, to as few places as possible yet necessary for obtaining performance. As an example, we are implementing spatial indexes and spatial clusters according to user-defined geometries without incorporating new data storage structures. We are trying to show by developing various front-ends that the kernel is a sufficiently broad basis to cover a variety of storage schemes also supporting new data types. Our multi-level (sub-) transaction management facilitates the implementation of index structures, e.g. a user is not allowed to implement his own concurrency control or recovery scheme, as is possible, for example, in STARBURST. Currently we are evaluating a first version of the kernel written in Pascal and we are implementing an improved version in C.

Acknowledgement: I would like to thank to J.C. Freytag, G. Lohman, and G. Weikum for their spontaneous help in preparing this "last minute" summary.

7. References

This list contains references to more detailed descriptions of the DASDBS project. References to other work can be found there and are not listed here:

/Da86/ Dadam, P., et. al: *A DBMS prototype to support extended NF2 relations: An integrated view an flat tables and hierarchies,* ACM SIGMOD, Washington, 1986.
/De86/ Deppisch, U.: *S-Tree: A dynamic balanced signature index...* ACM Conf. Res. & Dev. in IR, Pisa, 1986.
/DOPSSW85/ Deppisch, U., Obermeit, V., Paul, H.-B., Schek, H.-J., Scholl, M., Weikum,G.: *The Storage Component of a Database Kernel System,* Techn. Rep. DVS1-1985-T1, in Springer IFB 94, 1985.
/DPS86/ Deppisch, U., Paul, H.-B., Schek, H.-J.: *A storage system for complex objects,* Proceedings of the International Workshop on Object Oriented Database Systems, Pacific Grove, Ca., Sept. 1986
/PSSW87/ Paul, H.-B., Schek, H.-J., Soeder, A., Weikum, G.: *Supporting the office filing and retrieval service by a database kernel system* (in German), Proc. GI Conf. on Database Systems for Office, Eng., and Scient. Appl., Springer, 1987.
/PSSSWD87/ Paul, H.-B., Schek, H.-J., Scholl, M., Weikum, G.: *Architecture and implementation of the Darmstadt database kernel system,* ACM SIGMOD, San Francisco, 1987.
/Sche85/ Schek, H.-J.: *Towards a basic NF2 algebra processor,* Conf. on Found. of Data Org., Kyoto, 1985.
/Scho86/ Scholl, M.H.: *Theoretical foundation of algebraic optimization utilizing unnormalized relations,* Int. Conf. on Database Theory, Rome, 1986.
/SL83/: Schek, H.-J., Lum, V. *Complex Data Objects: Text, Voice, Images: Can DBMS Manage Them?,* Panel Discussion at the VLDB, Florence, 1983
/SPS87/ Scholl, M.H., Paul, H.-B., Schek, H.-J.: *Supporting flat relations by a nested relational kernel,* Proc. VLDB, Brighton, 1987.
/SS86/ Schek, H.-J., Scholl, M.H.: *The relational model with relation-valued attributes,* Information Systems 11 (1986), No. 2, pp. 137-147.
/SWe86/ Schek, H.-J., Weikum, G.: *DASDBS: concepts and architecture of a database system for advanced applications,* Tech. Rep. DVSI-1986-T1, to appear in Informatik - Forschung und Entwicklung, Springer, 1987
/SWa86/ Schek, H.-J., Waterfeld, W.: *A database kernel system for scientific applications,* Proc. of the 2nd Intl. Symp. on Spatial Data Handling, Seattle, 1986.
/We86/ Weikum, G.: *A theoretical foundation of multi-level concurrency control,* ACM PODS, Cambridge, 1986.
/WS84/ Weikum, G., Schek, H.-J.: *Architectural issues of transaction management in layered systems,* VLDB, Singapore, 1984.

THE COMPUTER SOCIETY
OF THE IEEE
1730 Massachusetts Avenue, N W
Washington, DC 20036-1903