Extensible Database Management Systems

Michael Carey
University of Wisconsin-Madison

Laura Haas
IBM Almaden Research Center

1 Introduction

Work in the area of extensible database management systems addresses two problems with today's (primarily relational) DBMSs. First, current systems are lacking when it comes to meeting the needs of emerging applications such as CAD/CAM, multi-media office systems, image management, statistical data management, and text manipulation. Such applications require support for new data types, functions, complex objects, storage techniques, and access methods for greater modeling power and performance. Extensible DBMS efforts aim to support the addition of such new features. Second, current systems tend to be monolithic in nature, making it difficult to incorporate recent advances in DBMS technology in a timely manner. Extensible DBMS efforts seek to ease the problem of tracking technology developments, simplifying the incorporation of new algorithms and new kinds of storage devices into a DBMS.

To meet these goals, a DBMS must be extensible at all levels. At the user interface level, it must be possible to add new data types and new operations on such data; this is sometimes referred to as abstract data type (or ADT) support. In addition, it should be possible to extend the query language with new set operators such as transitive closure. At a lower level, as new algorithm developments occur, the system should support the addition of new implementations of its operators. An example here might be the addition of a new, highly efficient join method. Similarly, but at a lower level still, the DBMS should support extensions to its repertoire of access methods; it should be possible to add a new index structure to support queries over spatial data, for example. Finally, the DBMS should allow for the definition of new data storage methods, thus making it possible to accommodate new storage media such as optical disks. Of course, most of these changes to the system will also affect the query optimizer in some way, as the optimizer must understand the entire set of operations, how they compose and interact with each other, and the cost of their various implementations.

In the rest of this paper we will briefly survey some of the major extensible DBMS efforts and issues. We begin by listing and discussing a number of major prototyping projects. We then explore extensibility at each level of a DBMS using examples drawn from these systems. Finally, we present some open questions in the extensible DBMS area and speculate on directions the field might take over the next few years. Due to limited space, our presentation is necessarily sketchy and incomplete; we suggest that interested readers follow the pointers available through the paper's references.

2 Extensible DBMS Prototypes

A number of projects have addressed extensibility at some level of the system [DBE87]. Major extensible DBMS prototyping projects have included ADT-INGRES [Ong84], RAD [Osbo86], DASDBS [Sche90], POSTGRES [Rowe87], PROBE [Mano86], GENESIS [Bato88b], EXODUS [Care89], Starburst [Haas90], R^2D^2 [Linn88], and Sabrina [Gard89]. As we will see, each of these efforts has taken one of two approaches to providing extensibility. Object-oriented database systems are also extensible at the data type level; however, these systems are beyond the scope of this article.

ADT-INGRES and RAD were two early efforts that provided support for ADTs and associated functions in the context of the relational data model. Sabrina is a similar but more recent system, and includes a novel approach to indexing over ADT fields. Starburst is an extensible relational DBMS effort that is distinguished

by its support for DBMS extensions at virtually all levels. POSTGRES, PROBE, and R^2D^2 are extensible DBMSs that start with richer data models. POSTGRES is based on an extended relational data model that includes "procedural" data, array fields, and rules; its ADT facility builds on that of ADT-INGRES, adding support for indexes on ADT fields and for extending the system with new access methods. PROBE is based on an object-oriented data model, and is best known for its approach to supporting the addition of spatial and temporal data types. R^2D^2 is based on an extension of the nested relational model and supports ADTs in the context of this model. Each of these systems offers users a full, end-user DBMS with "hooks" to support the relevant extensions. The systems differ in terms of their basic data models, their query languages, the levels at which extensions are permitted, and the amount of effort needed to add extensions.

In contrast to these systems, the DASDBS, GENESIS, and EXODUS efforts have taken more of a "do it yourself" or "toolkit" approach to extensibility. DASDBS offers a "database kernel" that provides storage and transaction management for nested relations; application needs are met by implementing application-specific DBMS layers on top of the kernel. GENESIS provides a highly layered DBMS architecture, a library of components that can be composed at each layer, and a graphical tool for generating a customized DBMS by selecting from the library of existing components. Finally, EXODUS provides a storage management kernel, a programming language with built-in support for persistent data, and a tool to aid in constructing query optimizers for new query languages. The toolkit approach can provide greater flexibility than a complete, extensible DBMS can, especially for applications that do not need all of the features of a full-function DBMS. On the other hand, for applications that do require a complete DBMS, the framework and guidance provided by an end-user data model, query language, and extension facility can significantly reduce the amount of effort needed to meet the application's needs (particularly if only small extensions, such as ADTs, are needed). To provide a better starting point for these applications, each of the toolkit-oriented projects have opted to offer "starter" data models and query languages [Bato88c, Care88, Sche90].

3 Levels of Extensibility

The goal of this section of the paper is to illustrate the problems and approaches involved in supporting extensions at different levels of a DBMS. We organize our presentation into three levels – query language, query processing, and storage and access methods.

3.1 Query Language Extensions

Support for ADTs, as pioneered by ADT-INGRES [Ong84] and RAD [Osbo86], is the primary example of DBMS extensibility at the query language level. In the Berkeley ADT-INGRES system, for example, users were permitted to define new ADTs for use as field types in a relational schema. To register an ADT, it was necessary to tell the system the name of the ADT, the amount of space an instance of the ADT would consume, and how to convert between the printable string representation of an ADT instance and its internal representation; conversions were handled by registering an appropriate pair of C routines. Users could then define new functions, again written in C, to operate on ADT fields in the context of relational queries. Important unary and binary functions could also be defined as having an "operator" invocation syntax, either by overloading existing operator symbols or by defining new ones. The RAD facilities for defining ADTs and their operations were fairly similar, though in RAD the implementor of an ADT could construct complex ADT functions by invoking tuple-oriented library routines to manipulate other relations in a RAD database if so desired.

Work on ADTs at Berkeley [Ston86, Ston87, Lync88, Ston90] has continued, primarily in the context of POSTGRES. For efficiency, it must be possible to process queries involving ADTs using standard optimizations such as indexing and efficient join algorithms. To ensure this, the definer of an ADT for POSTGRES can help the query optimizer by providing analogies between the new ADT's operators and those already understood by the system. For example, the definer of a polygon ADT for spatial applications might provide the boolean operators *ALT, ALE, AE, ANE, AGT,* and *AGE* for comparing the areas of polygons [Ston90] and inform the optimizer that these are analagous to $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$. This makes it possible to index relations on polygon field areas using B+ trees, or to process equi-area joins between polygon fields

of relations using sort-merge or hash join techniques. Further extensions were proposed in [Lync88] to allow relations to be indexed on single-valued or multi-valued functions of ADT fields; this was shown to be useful for supporting textual databases.

In addition to ADTs, it is also desirable to support more general extensions at the query language level, such as new set operations. One interesting class of set operations is aggregates (e.g., *avg* or *count*), and both RAD [Osbo86] and POSTGRES [Ston87] allow users to define new aggregate operations. In POSTGRES, for example, such operations can be added by writing and registering two C functions with the system. The first function is to be called once per element being aggregated over, accumulating the information needed for the aggregate computation in a state record. The other function is to be called at the end of the operation to compute the actual aggregate value from the accumulated state. With this facility, it is possible for a user who wishes to add a new aggregate operation, such as an operation to compute the *median* of a set of integer values, to do so. More general set operation extensions are supported by RAD, Starburst, and GENESIS. In RAD, users can extend the system with new *transformation* operations that consume and produce relations; such operations fit easily into the RAD query language because it is based on the relational algebra. Starburst provides a similar facility, called *table functions* [Haas89], which can also be used to import data into the system. GENESIS supports set operation extensions through the addition of new *stream translators* [Bato88c]. Finally, it should be mentioned that this sort of extension is also supported in several object-oriented DBMSs with functional data models, including PROBE [Mano86] and also Iris [Wilk90].

3.2 Query Processing

Beneath the query language, the query processing engine of an extensible DBMS should support the addition of new execution strategies, including new implementations of operations (operators) and new ways of combining existing operators. As an example of the first type of extension, one might add a new operator with a better algorithm for handling the join operation (e.g., hybrid hash join). As an example of the second, in a relational DBMS, one might wish to add a strategy that, if many records are to be retrieved using an unclustered index, will use the index to retrieve qualifying record identifiers and sort them on page number before retrieving the records. One problem common to both kinds of extensions is informing the query optimizer about the extension.

In EXODUS, a *query optimizer generator* is used to produce a query optimizer from a rule-based description of possible execution strategies plus a set of C support routines [Grae87]. The generated optimizer takes a query as its input and produces an access plan as its output; it does this by repeatedly (i) applying an algebraic transformation to the query, and then (ii) choosing the best plan for each operation in the transformed query. Here, a query is a tree of algebraic operations (e.g., for a relational DBMS, the operations of the relational algebra) and an access plan is a tree of operators (implementations of operations, e.g., hash join). The input to the optimizer generator lists the operations of the query algebra, the available operators, a set of query transformation rules, and a set of implementation rules. Operations and operators are characterized by their names and arities. Transformation rules are equivalence-preserving query tree transformations, expressed as a pair of equivalent expressions and an optional condition of applicability written in C. An example is a rule to convert cross products into joins when join predicates are present. Implementation rules specify an operator, an expression involving one or more operations that the operator implements, and an optional condition of applicability. An example here is a rule specifying that joins are computable via the nested-loop index join method if an index exists on the inner relation. The optimizer generator's input also includes a number of C support routines, including a cost function for each operator. As a result, query processing extensions in EXODUS are handled by modifying or adding rules and then regenerating the optimizer.

Starburst also takes a rule-based approach to query optimization [Haas89]. In contrast to EXODUS, where avoiding data model specifics in the optimizer generator was a driving consideration, the Starburst optimizer is strongly oriented towards relational query optimization. As a result, it has a more extensive architecture, including a rich semantic network for representing relational queries and a search strategy tailored to optimizing N-way joins. The Starburst query optimizer includes two different rule-based components,

each with its own rule language and rule interpreter. The query rewrite module is responsible for semantic optimizations, such as predicate manipulation and query modification, while the plan generator enumerates the feasible join orderings for a query and uses grammar-like production rules to select an execution plan for each one.

To add a new operation, or a new implementation of an existing operation, to an extensible DBMS, the code for the new operator must first be written and integrated into the system. Once this is done, the optimizer can then be extended as just described. The Starburst [Haas90] and GENESIS [Bato88c] query evaluation engines interpret a plan of composable, algebraic, demand-driven, stream-based operators. New operators are written in C and must adhere to certain public interfaces and conventions. Query evaluation using the EXODUS toolkit is based on a similar approach, but operators are written in the E programming language, which is intended to ease the difficulty in writing new operators [Rich89]. Among other features, E has built-in support for manipulating persistent data and includes the notion of iterator functions to simplify the development of composable, stream-oriented operators. To support spatial operations, PROBE took quite a different approach [Oren88], exploiting the object-oriented nature of its data model. The PROBE system has a built-in object class called *POINT-SET* that specifies the operations that all spatial data types must provide. New spatial data types are added as subclasses of this generic class, and PROBE's query processing component understands how to process *POINT-SET* queries.

3.3 Storage and Access Methods

An extensible DBMS must be able to accommodate new ways of storing data and new index structures. As new storage media appear, new storage methods may be needed to take full advantage of the technology. Similarly, new data types may demand new index structures, and index technology is continually evolving in any case.

The relational storage component of Starburst, known as Core, supports two kinds of extensions – *storage methods* and new types of *attachments* [Lind87]. A storage method is responsible for managing records of relations. Examples include a standard "heap" storage method, sequential files, and B+ tree files. Storage methods have a well-defined operational interface, with calls to create and destroy storage method instances as well as calls to insert, update, and delete records. In addition, each storage method must have a record key concept, supporting direct-by-key and key-sequential record access. The structure of a key is defined by each storage method; a key might be concatenated field values in a B+ tree storage method, and a physical address in the case of a sequential file storage method. Every Starburst relation is stored using one of the available storage methods.

An attachment is code that can be associated with a relation and that is designed to react to changes in the relation's contents. Whenever a record insert, delete, or update occurs, each of the relation's attachments is invoked to analyze the change and take appropriate actions. A secondary B+ tree index is a simple example of one type of attachment. Attachments can opt to veto operations; thus, functions like integrity constraint checking and production rule triggering can be implemented as attachments as well. If needed, an attachment may span several relations [Care90]. Finally, the Core attachment interface includes direct-by-key and key-sequential record access calls for use in attachments such as indexes. Starburst relations may have instances of any available attachment type. As an example, a relation could be stored using a sequential file storage method, indexed on various fields by several instances of a B-tree attachment, and guarded by several instances of integrity constraint attachments.

POSTGRES also supports the addition of new access methods by skilled programmers via a clearly defined secondary index interface [Ston87]. GENESIS supports both storage and access method extensions via standardized interfaces [Bato88a, Bato88b]. Moreover, a given logical file in GENESIS can be implemented by multiple physical files and links since files can involve layers of *elementary transformations* such as indexing or transposition. Through this approach, GENESIS is capable of emulating many commercial file structures using only a small collection of primitive building blocks.

Unlike Starburst and GENESIS, which allow storage management to be tailored for specific applications, DASDBS and EXODUS provide storage management kernels that are intended to suffice for a wide range of

applications. DASDBS [Sche90] provides nested relation storage and operations. Non-nested tuple attributes are stored as uninterpreted byte sequences, and the contents of a nested tuple are clustered together on disk. The EXODUS storage manager provides files of uninterpreted, versionable objects as its storage abstraction, and hints are provided to support object clustering [Care86]. In both systems, the intent is for applications to implement appropriate secondary indexes on top of the storage management kernel. DASDBS includes a multi-level transaction manager to help with index concurrency control and recovery, while EXODUS offers the E programming language to simplify the task of writing new application-specific access methods.

4 Open Questions and Directions

Many challenging problems remain for extensible database management systems. In this section, we discuss challenges related to safety, the complexity of building and registering extensions, and attaining even greater extensibility. We also comment on the relationship between extensible and object-oriented database systems.

One fundamental and difficult issue is to guarantee the integrity of the DBMS and its data in the face of extensions. There are several aspects to this problem. First, the internal data structures of the DBMS must be shielded from "wild writes" by extensions. In addition, if data integrity is to be ensured, the correctness and completeness of each extension must be verified. For example, if a new storage method does not log its actions, it may be impossible to recover the database to a consistent state after a crash. Interactions between extensions must also be monitored. It is possible for extensions to conflict with each other, undoing one another's work or introducing unpredictable results. Extensions may also depend on one another, and inconsistencies may result if such dependencies are not recorded and checked by the system [Haas90, Care90].

Little work has been done in this area of extension safety. POSTGRES offers one approach to protecting the base system from ADT operations. In POSTGRES, ADT operations can be executed in an isolated process via remote procedure calls from the base system [Ston90]. If this is too expensive, then the extension must be registered as a "trusted" extension – in which case there is no protection. Sabrina achieves a degree of safety from errors in ADT operations written in Lisp by employing a Lisp interpreter that disallows access to global environment variables in ADT functions [Gard89]. Starburst allows inter-extension dependencies to be recorded, and provides mechanisms for discovering whether one extension meets the requirements of another one [Haas90, Care90], but does not automatically check such dependencies. Other systems have largely ignored these issues. While there may not be an elegant solution to this set of problems, some solution is necessary if extensible database management systems are to be commercially viable.

Another important issue for future study is how to reduce both the amount and complexity of the work required to add extensions. For example, to add a new, industrial-strength access method, the programmer of the access method (extender) must provide for concurrency control and recovery. While some extensible DBMS prototypes provide transaction management facilities (e.g., locking and logging) as services to the extender, just putting in correct calls to these services is a complex and error-prone process; this is particularly true if highly concurrent index operations are to be supported. Even such basic functions as keeping track of the state of an index scan can be complicated, and today's extensible DBMS prototypes provide little help here. Finally, almost any extension requires changes in multiple system components. Adding a new access method, for example, will also affect the query optimizer and the data definition language. In many cases, it should be possible to automatically generate the routine "hook-up" code for the different components, and it should certainly be possible to guide the extender as to where such changes are necessary. Ideally, many extensions would be possible without resorting to low-level database code.

In addition to these problems, striving for still greater extensibility is an important challenge. One area that needs more work is extensibility at the data model and query language level. While several systems support the definition of ADTs, including the latest release of commercial INGRES, truly efficient querying requires that the optimizer be able to predict the selectivity and cost of operations on new types of data and to exploit available access paths. While the POSTGRES optimizer can be told about functions that are registered as ADT operators, and even supports indexing for such operator classes, queries involving more general functions cannot yet be optimized effectively [Ston90]. Another challenging issue at the data model and query language level is the interface with applications. One valuable role of a DBMS is to promote sharing

across applications – regardless of the application programming language. To ease application development, it is important to provide a smooth bridge between the type system of the application and that of the DBMS. Little has been done towards reconciling the potentially rich type system of an extensible DBMS with the possibly more primitive type systems of multiple application programming languages. Another query language challenge is support for set operations that are parameterized by type (e.g., it should not be necessary to write a median aggregate for every new data type). Finally, in a truly extensible data model, it should be possible to define not only new types, but also new *type constructors*, and it should be possible to introduce support for them in the query language and throughout the system. This would greatly increase the modeling power of the DBMS; however, providing such a facility is a challenging open problem.

As we focus on making the data model still more extensible, we are really concerned with correctly modeling many different types of objects. Of course, this is a key concern of the object-oriented system community. In fact, distinguishing systems with highly extensible data models from object-oriented database systems is becoming difficult at best. This is understandable, as extensible database management systems have been adding modeling power to systems strong in traditional database technology (e.g., associative access, concurrency control, recovery), while work on object-oriented database systems has involved adding database technology to systems strong in modeling power. We conjecture that database management systems of the future will require both of these, as well as additional, technologies. The ability to extend the storage methods, indexing, and query optimization capabilities of next generation DBMSs will be the result of extensible DBMS research, while the object-oriented DBMS area will show us how to narrow the gap between real-world objects and the data model and how to more tightly integrate the DBMS with application programming languages. We see real promise for the database field as these two subareas advance and merge.

References

[Bato88a] Batory, D., *et al*, "GENESIS: An Extensible Database Management System," *IEEE Trans. on Software Eng.* 14(11), Nov. 1988.

[Bato88b] Batory, D., "Concepts for a Database System Compiler," *Proc. ACM Princ. of Database Sys. Conf.*, Austin, TX, March 1988.

[Bato88c] Batory, D., Leung, T., and Wise, T., "Implementation Concepts for an Extensible Data Model and Data Language," *ACM Trans. on Database Sys.* 13(3), Sept. 1988.

[Care86] Carey, M., *et al*, "Object and File Management in the EXODUS Extensible Database System," *Proc. 12th VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[Care88] Carey, M., DeWitt, D., and Vandenberg, S., "A Data Model and Query Language for EXODUS," *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Care89] Carey, M., *et al*, "The EXODUS Extensible DBMS Project: An Overview," *Readings in Object-Oriented Database Systems*, S. Zdonik and D. Maier, eds., Morgan-Kaufmann, 1989.

[Care90] Carey, M., *et al*, "An Incremental Join Attachment for Starburst," *Proc. 16th VLDB Conf.*, Brisbane, Australia, Aug. 1990.

[DBE87] *Database Engineering* 10 (2), Special Issue on Extensible Database Systems, M. Carey, ed., June 1987.

[Gard89] Gardarin, G., *et al*, "Managing Complex Objects in an Extensible Relational DBMS," *Proc. 15th VLDB Conf.*, Amsterdam, The Netherlands, Aug. 1989.

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

[Haas89] Haas, L., *et al*, "Extensible Query Optimization in Starburst," *Proc. ACM SIGMOD Conf.*, Portland, OR, June 1989.

[Haas90] Haas, L., *et al*, "Starburst Mid-Flight: As the Dust Clears," *IEEE Trans. on Data and Knowledge Eng.* 2(1), March 1990.

[Lind87] Lindsay, B., McPherson, J., and Pirahesh, H., "A Data Management Extension Architecture," *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987.

[Linn88] Linnemann, V., *et al*, "Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Lync88] Lynch, C., and Stonebraker, M., "Extended User-Defined Indexing with Application to Textual Databases," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

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

[Ong84] Ong, J., Fogg, D., and Stonebraker, M., "Implementation of Data Abstraction in the Relational Database System INGRES," *SIGMOD Record* 14(1), March 1984.

[Oren88] Orenstein, J., and Manola, F., "PROBE Spatial Data Modeling and Query Processing in an Image Database Application," *IEEE Trans. on Software Eng.* 14(5), May 1988.

[Osbo86] Osborn, S., and Heaven, T., "The Design of a Relational Database System with Abstract Data Types," *ACM Trans. on Database Sys.* 11(3), Sept. 1986.

[Rich89] Richardson, J., Carey, M., and Schuh, D., *The Design of the E Programming Language*, Tech. Rep. No. 824, Computer Sciences Dept., Univ. of Wisconsin, Madison, Feb. 1989.

[Rowe87] Rowe, L., and Stonebraker, M., "The POSTGRES Data Model," *Proc. 13th VLDB Conf.*, Brighton, England, Aug. 1987.

[Sche90] Schek, H., *et al*, "The DASDBS Project: Objectives, Experiences, and Future Perspectives," *IEEE Trans. on Data and Knowledge Eng.* 2(1), March 1990.

[Ston86] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," *Proc. 2nd Data Engineering Conf.*, Los Angeles, CA., Feb. 1986.

[Ston87] Stonebraker, M., *et al*, "Extendability in POSTGRES," in [DBE87].

[Ston90] Stonebraker, M, Rowe, L., and Hirohama, M., "The Implementation of POSTGRES," *IEEE Trans. on Data and Knowledge Eng.* 2(1), March 1990.

[Wilk90] Wilkinson, W, Lyngbaek, P., and Hasan, W., "The Iris Architecture and Implementation," *IEEE Trans. on Data and Knowledge Eng.* 2(1), March 1990.