A Data Management Extension Architecture

Bruce Lindsay, John McPherson, and Hamid Pirahesh

IBM Almaden Research Center, San Jose, CA 95120

Abstract

A database management system architecture is described that facilitates the implementation of data management extensions for relational database systems The architecture defines two classes of data management extensions alternative ways of storing relations called relation "storage methods", and access paths, integrity constraints, or triggers which are "attachments" to relations Generic sets of operations are defined for storage methods and attachments, and these operations must be provided in order to add a new storage method or attachment type to the system The data management extension architecture also provides common services for coordination of storage method and attachment execution This article describes the data management extension architecture along with some implementation issues and techniques

Introduction

In order to provide effective and efficient support for the many, diverse database applications of the next decade, database management systems must be able to incorporate new facilities specialized to the application and hardware environment Recently there has been considerable interest in "extensible" database management systems [STONEBRAKER86,BATORY86,CAREY86,DAYAL85] Database extensions can be (grossly) classified as *user data* extensions or *data management* extensions User data extensions concentrate on providing support for user-defined abstract data types and functions for fields of database records Data management extensions provide alternative implementations of database storage and access paths

User data type extensions provide support for domains [CODD79] and allow application developers to specify data type representations, conversions, and functions on user data types, which can be performed by the database management system during the execution of database queries and updates User data types not only enhance application development productivity and integrity, but also improve system performance because predicates involving user-defined types can be evaluated by the DBMS to eliminate items which would otherwise have to be returned to the application

Data management extensions provide alternative data storage methods, data access paths, and integrity constraint or trigger facilities Data management extensions allow a database system to evolve to support new hardware and application opportunities and requirements As database management systems are applied to an ever broader range of applications, it is often desirable to provide database facilities which are specialized to the application requirements, in order to enhance application performance or simplify the application implementation For example, spatial database applications can make use of an R-tree access path [GUTTMAN84] to efficiently compute certain spatial predicates Exploiting the evolution of the hardware facilities suggests data management extensions such as main memory data storage methods for selected high traffic relations, and special facilities to support (read-only) optical disk database publishing applications Data management extensions also allow the installation of improved, but representation incompatible, versions of data storage or access paths without impacting existing applications and data

This article describes a database management system architecture which facilitates the implementation of data management extensions for relational database systems The difficulty of implementing data management extensions stems from the fact that such extensions interact with almost all components of the DBMS Not only must data management extensions implement low-level, recoverable representations and operations, but they must also be integrated into the query planning process and the data definition facility The objective of the data management extension architecture is to simplify the integration of alternative data storage, access path, integrity constraint, and trigger facilities into the DBMS while preserving user and internal interfaces for data access and manipulation The challenge is to devise mechanisms for incorporating a broad range of data management extensions *without* degrading system performance

The principle features of the data management architecture 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 structures, integrity constraint, 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 idea of defining a common set of interfaces that is implemented by each relation storage type or access path is not new System R supports permanent and temporary relations through a common Relation Storage System (RSS) interface [ASTRAHAN76] INGRES has a similar interface called the Access Method Interface (AMI) [STONEBRAKER76] Our experience with System R and R* has shown that well defined external data management interfaces are not sufficient to make adding new storage and access structures easy and efficient It is also necessary to have well defined internal

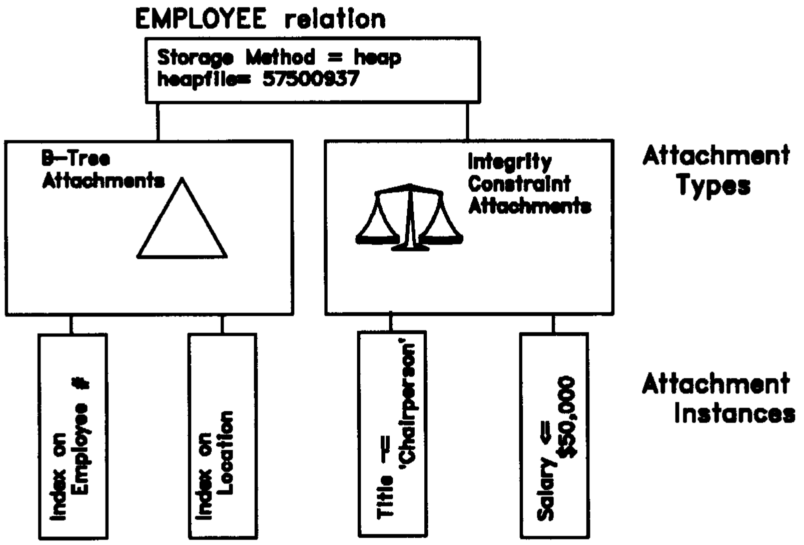

Figure 1 Relationship of Storage Methods and Attachments

interfaces and common services, along with mechanisms that are inherently extensible for locating and calling new storage and access structure routines This also appears to be the experience of INGRES Many extensions have been made to that system since its initial implementation, but the resulting system is "'hacked up enough' to make the inclusion of substantial new function extremely difficult" [STONEBRAKER86] This is one of the motivations stated for designing and implementing POSTGRES which is an entirely new database system

Generic Abstractions for Data Management Extensions

In relational database systems, the *relation* (or table) is the fundamental, user-visible object type A relation is a set of records, and operations for selecting, inserting, updating, deleting, and combining records from multiple relations are specified using a data manipulation language (such as QUEL or SQL) Relational systems also incorporate *access paths* that support record selection operations and *integrity constraints* that prevent relation updates which would violate user specified intra- or inter-record predicates Access paths, in general, replicate information from the relations to which they are applied The replicated data stored by access path implementations is organized in useful ways to accelerate access to specific subsets of the relation's data Access paths need not be limited to a single table (e g , join indexes [VALDURIEZ85]) and can be used to maintain alternative representations or aggregations of the data stored in a relation In contrast to access paths, which *reflect* the contents of their relations, integrity constraints are used to *control* the contents of their relations Integrity constraints can be used to either prevent relation updates which do not satisfy the integrity constraint or to trigger relation updates which establish or maintain the desired data consistency It is important to note that access path updates and integrity constraint enforcement are performed implicitly as side effects of operations which modify the contents of a relation

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 constraints, or triggers that are associated with relation instances Relation storage method extensions are known simply as "storage methods " Examples of storage methods include recoverable and temporary relations Different recoverable (or temporary) storage methods are possible For example, the records of the relation may be stored sequentially in a disk file or they may be stored in the leaves of a B-tree index Another relation storage method might support access to a foreign database by simulating relation accesses via (remote) accesses to relations in the foreign database 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) 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

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 The relationship of storage methods and attachments is shown in Figure 1 In this example, the Employee relation uses the heap storage method, and it has instances of B-tree and intra-record consistency constraint attachments

Attachments, like storage methods, must support a well-defined set of operations Unlike storage methods, however, attachment modification operations 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 action 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 Like storage methods, access path extensions support direct-by-key and

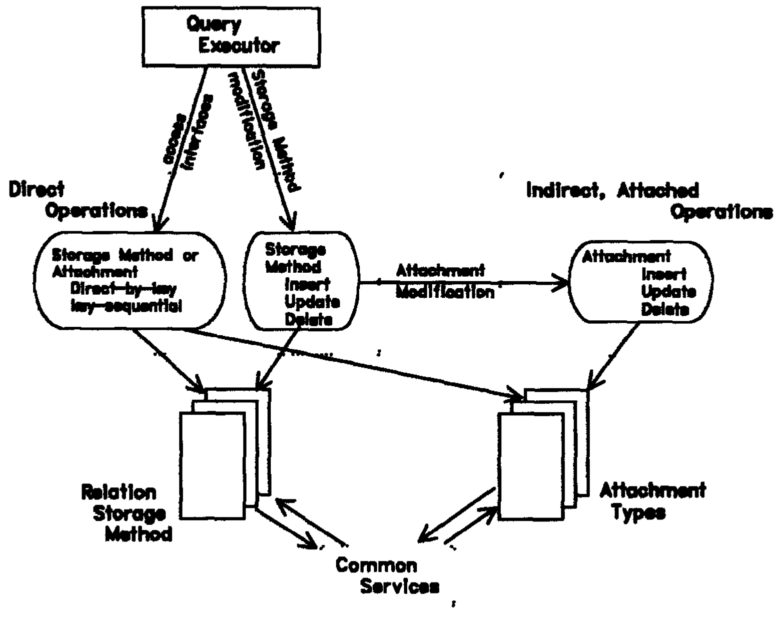
Query Executor
access interfaces
Storage Method modification
Direct Operations
Indirect, Attached Operations
Storage Method or Attachment Direct-by-key key-sequential
Storage Method Insert Update Delete
Attachment Modification
Attachment Insert Update Delete
Relation Storage Method
Attachment Types
Common Services

Figure 2 Generic Data Management Interfaces

(optionally) key-sequential accesses which return the storage method key of the corresponding relation records The attachment access operations can be invoked directly by the data management facility user

The attachment concept is more general than the concept of an access path since an attachment may insure that an integrity constraint is not violated, or trigger actions both inside and outside the database in addition to providing alternative means of accessing data stored in a relation Attachments are not merely triggers as System R [ASTRAHAN76] and POSTGRES [STONEBRAKER86] however, because they may have associated storage This storage can be used to maintain access structures, and even to maintain statistics about relations or precomputed function values for data stored in relations

The Data Management Extension Architecture

The data management extension architecture consists of three major components 1) direct operations on storage methods and attachments, 2) procedurally attached, indirect operations on attachments, and 3) a common services environment The direct operations on storage methods and attachments support relation modification (record insert, update, and delete) and accesses to record keys and fields via relation storage methods or access attachments Direct operations are also defined for creating, destroying, and managing storage methods and attachments Procedurally attached, indirect operations are used to notify attachments when the relation is modified so the attachments can modify their state or initiate appropriate actions based on the modification to the relation storage method Common services provide a shared framework for storage method and attachment execution The generic interfaces of the data management extension architecture are shown in Figure 2 The remainder of this section discusses the components of the data management extension architecture in greater detail

Direct Generic Operations

The direct generic operations are internal DBMS operations on relations and access path instances Generic interface operations are dynamically selected to execute data management operations based on the storage method or attachment type being manipulated The generic operations are divided into two classes 1) relation *modification* and management operations, and 2) *accesses* to record fields or keys

The relation modification operations include insert, update, and delete record Relation and attachment management operations include create and destroy relation or attachment instances, and change mode or status of relation or attachment instances Special care must be taken with database data definition operations, such as create relation or attachment instance, to allow storage method and attachment type-specific attributes to be specified via the user-level data definition language and executed via generic storage method and attachment creation operations For example, some storage methods may support multiple devices and will need to be told where to put a specific instance of the storage method To this end, the data definition language of the DBMS has been extended to allow specification of a storage method or attachment type and an attribute / value list for extension-specific parameters Storage method and attachment implementations supply generic operations to validate and process the attribute lists during parsing and execution of the data definition operations

The second class of direct generic operations supports *accesses* to stored relations and access paths Every relation storage method defines *record keys* for the records it stores, while access path extensions supply a mapping from an input key to a record key Both relation storage methods and access path attachments support *direct-by-key* and *key-sequential accesses* Given a key, a direct-by-key access returns selected data fields from a record in the relation, or returns a record key from the access path mapping structure Some access path attachments may be able to return record fields when the access path key is a multi-field value and the access is specified using a partial key Key-sequential accesses return a sequence of data field or record key values, in the order defined by the storage or access path This approach to data accesses separates access path accesses from relation accesses First the access path is accessed to obtain a record key, which is then used to access the relation record in the storage method

Key-sequential accesses require support for the notion of *position* within a relation or access path, and a definition of the interaction between the position of a key-sequential access and the deletion of records (or access path entries) at the current position This is because key-sequential accesses occur one record or record key at a time, and control may return to the user application before the completion of the key-sequence The user application may then delete records which may be at the current position of the key-sequential access We introduce the term *scan* to denote the key-sequential access position A scan may be on, after, or before an item of the relation or access path After a successful return from a key-sequential access, the scan is *on* the returned item If an item at the scan position is deleted, the scan will be positioned just *after* the deleted item Key-sequential access operations always access the *next* item after the current scan position

Also included as access operations are relation storage method and access attachment operations to assist the query planner in estimating the cost of performing an access via the storage method or attachment Given a list of "eligible" predicates supplied by the query planner, the storage method or access attachment can determine the "relevance" of the predicates to the access path instance and then estimate the I/O and CPU costs to return the record fields or keys that satisfy the predicates For example, a B-tree access path will return a low cost if there is a predicate on the key of the B-tree, and the R-tree access path will recognize the ENCLOSES predicate and report a low cost In a similar manner, the query planner will be able to determine the cost of using a storage method or attachment to scan a relation in a random order or with the tuples ordered by particular record fields

In addition to accessing record fields or keys, accesses also support record *filtering* via predicate expressions passed to the relation storage or access path Using common services, the extension implementations can test a predicate against the current relation record or against record fields available in the access path key Entries which do not satisfy the predicate are skipped The envisioned common services predicate evaluator will be able to call functions that are passed to it, and use any combination of fields from a record as operands Additionally, both constant and variable data can be used by the predicate evaluator It is expected that the predicate will (usually) be evaluated without having to copy the data from the extension's buffer pool

Indirect, Attached Procedures

The second component of the data management extension architecture is the definition of *procedural attachments* to relation instances Attached procedures are indirectly invoked as side effects of operations that modify stored relations Whenever a relation record is inserted, updated, or deleted, the corresponding attached procedure is invoked for each attachment type having instances that are defined on the relation being modified Each attachment type is passed its own meta-data (descriptor), the record key, and the old and new record values

This method of procedural attachment allows access path attachments to update their representations, integrity constraints to be evaluated during record modification operations, and other actions to be triggered Any invoked attachment can *veto* the entire record modification operation if the operation would violate any constraints of the attachment Each attachment type is invoked at most once per relation modification and must service all instances of its attachment type that are currently defined on the relation Attachments may access or modify other data in the database by calling the appropriate storage method or attachment routines In this manner, modifications may cascade in the database

An example of procedural attachments for a B-tree access path is in order After a record is inserted into a relation having B-tree indexes defined on it, the B-tree attached procedure for insert will be invoked passing a copy of the inserted record along with the newly assigned tuple identifier or record key For each B-tree index defined on the relation being modified, the B-tree insert procedure will form an index key by projecting fields from the inserted record, and then insert the index key plus tuple identifier or record key into the B-tree index On update, the old record and record key will be used to determine which key to delete from the B-tree index and the new record and record key will be used to form the key to be inserted into the index Of course, the B-tree update operations should be able to detect when no indexed fields for a given index are modified

Attached procedures can perform complex operations, including the invocation of additional operations on the same or other relations For example, the referential integrity attachment to a "parent" relation would perform record delete operations on the "child" relation when a "parent" record is deleted If the "child" relation also has a referential integrity attachment, it would perform record delete operations on its "child" relation Thus, cascaded deletes can be supported On insert, the same attachment type on the "child" relation would test the "parent" relation for a record with matching referential integrity fields

Common Services Environment

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 The most obvious interface convention is the common record and field value representations needed to allow communication with the generic operations comprising the storage method and attachment extensions More subtle issues are raised by the recovery and concurrency aspects of the database management system as discussed below Also, data management extensions will need to participate in database events such as transaction commit

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 level 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 If some storage method or attachment implementation were to use serial validation or timestamp order based concurrency control while others used locking, a non-serializable transaction execution is possible in general [BERNSTEIN81] 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

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 The predicate evaluation facility is also available to the integrity constraint attachments and to the query execution engine

In order to correctly maintain scan positions in a relation or access path during key-sequential accesses, the storage methods and attachments must be notified of certain transaction events In particular, all key-sequential accesses must be terminated at transaction termination This is because all locks are released at transaction termination and the access procedures use locking to maintain the integrity of the scan position A common service facility will notify all storage methods and attachments which used key-sequential accesses during the transaction when the transaction completes so that they can clean up (i e , close) any open scans

Key-sequential access positions are also affected by partial transaction rollback Partial transaction rollback is used, not only to recover from vetoed relation modifications, but also to undo the partial effects of (complex) data definition operations and to support a sort of application-level sub-transaction To correctly return the database management system to an earlier point in the execution of the transaction, scan positions must be restored to their earlier status While the common log-driven rollback will correctly restore stored relations and access path data structures to an earlier state, scan positions are not restored automatically because their state changes are not logged (for performance reasons) Instead, when a transaction rollback point is established, the storage methods and attachments are driven by the system to obtain their key-sequential access positions The scan positions are retained until the rollback point is canceled or until they are used to restore the key-sequential positions following a partial rollback

Another event notification mechanism is needed to support deferred actions on storage methods and attachments In order to make storage method and attachment drop (destroy) operations undoable without logging the entire state of the relation or access path, the actual release of the relation or access path state is deferred until the transaction commits A common service facility will be available to allow extensions to request notification when (if) the transaction commits so that they can complete the execution of their deferred actions

Some areas of the database management system do not require special data management extension facilities because the mechanisms employed by the query planning and processing portions of the system are general enough to handle data management extensions Because extensions are alternative implementations of a common relation abstraction, a uniform authorization facility can be used to control user access to relations of all storage methods A common facility will also record and track the dependencies between "bound" query plans and the extensions used by the plans In order to provide good performance for production databases, it is important to retain the translations of queries into query execution plans that directly invoke the relation and access path operations, and to use the saved query execution plans whenever the queries are subsequently executed This query binding approach avoids the non-trivial costs of accessing the relation descriptions and optimizing the query at query execution time However, the validity of the translation of the query into its execution plan depends upon the continued existence of the relations and access paths used by the query plan A uniform mechanism for recording the dependencies of execution plans on the relations they use allows the system to invalidate any plans which depend upon relations or access paths that have been deleted from the system Invalidated execution plans are automatically re-translated, by the common system, the next time the query is invoked by an application

Data Management Extension Mechanisms

This section discusses some implementation issues and techniques for data management extensions One of our goals is to prove that an extensible database system can be more than just a research tool, extensible database system technology can be designed and implemented in a way that performance is good enough to permit its use in production systems The interfaces to storage methods and attachments are tuple-at-a-time interfaces This means, for example, that the join of two moderate sized relations can easily result in thousands of calls to storage method and attachment routines It is imperative, therefore, that the linkage to storage method and attachment routines, and to the common service routines, be very efficient For this reason, certain common service routines must be recompiled and new extensions must be linked with the rest of the DBMS in order to add a new data management extension to the system Directly linking extensions to the rest of the DBMS allows extensions to call system services directly and facilitates efficient invocation of extension procedures This is not considered to be a significant restriction because our intention has been that data management extensions must be made "at the factory " The implementer of a data management extension must have knowledge of internal database interfaces, and data management extensions have access to critical database resources Thus, data management extensions must be written by sophisticated personnel at the factory, and not by casual database users

Because extensions are bound into the database system, performance is improved by simplifying the runtime invocation of extension procedures For each direct or indirect generic operation, there is a vector of addresses for the procedures that implement the corresponding operation More precisely, for each generic operation on stored relations, there is a vector of procedures with an entry for each relation storage method For generic operations on attachments, there is a vector of procedures with an entry for each attachment type Finally, for relation insert, update, and delete operations, there are vectors of attached procedures with entries for each attachment extension Storage method and attachment internal identifiers are small integers that serve as indexes into the vectors or procedures For example, the base database system has a storage method for implementing temporary relations and that storage method is assigned the internal identifier 1 The address of the insert routine for the temporary storage method can be found in entry 1 of the vector of storage method insert routine addresses This approach makes the activation of the appropriate extension quite efficient

Database management systems commonly maintain runtime *descriptors* for relation and access path instances These descriptors contain the meta-data needed to access the relation or access path instance Instead of requiring each relation storage or access path to store and access its own descriptor data, the common system will maintain and manage relation descriptors Each extension supplies and interprets the contents of its own descriptor data, but the common system manages the composite relation descriptor and presents extensions with their own meta-data during execution This strategy allows the common system to fetch the relation descriptors from the system catalogs at query compilation time and store them in the query access plan It eliminates the need to access the catalogs to obtain relation descriptors at run time

The relation descriptor is composed of a relation storage method descriptor and descriptors for any attachments defined on the relation instance The structure of the relation descriptor is a *record* whose header contains the storage method identifier and whose first field contains the storage method descriptor Each attachment has an assigned identifier, and the descriptor for the attachment with identifier N is found in field N of the relation descriptor If there are no instances of attachment type N defined on a particular relation, then field N of that relation's descriptor will be NULL This encoding of the relation descriptor gives storage method and attachment implementations wide latitude in designing their encodings for describing relation and attachment instances It also leads to efficient runtime selection of which storage method and attached procedures to invoke via the extension procedure vectors However, note that this method for representing relation descriptions effectively limits the number of different attachment types to a few dozen without beginning to incur significant storage overhead for the relation descriptor (since non-present attachments will require a few bytes in the record-oriented relation descriptor format)

For example, the storage method descriptor for a simple relation storage method would contain the identification of the file containing the records of the relation A B-tree index attachment descriptor would contain the identification of the file containing the B-trees indexes of the relation, the number of B-tree indexes currently defined, and a root page id plus a list of indexed fields for each index A simple integrity constraint extension descriptor would contain a (common service) encoding of the predicate to be tested when records of the relation are inserted or updated More elaborate extensions would have correspondingly more complex descriptors, including embedded references to descriptors for other relations whenever the extension involves multiple tables (e g , referential integrity constraints or join indexes)

The execution of relation modification operations proceeds in two steps The first step, using the storage method identifier from the relation descriptor, calls the appropriate storage method modification routine via the storage method operation vectors After completing the storage method operation, the extensions attached to the relation are invoked via the attached procedures vectors Again, the relation descriptor is consulted to determine which attachment types have instances on the relation and must, therefore, be notified of the relation modification The old record value is available to the extension routines on updates and deletes, and the new record value is available on updates and inserts The record key is available on all record modification operations As mentioned earlier, attached extensions test the validity of the relation modification, and update access path data structures to reflect the relation modification The storage method operation or the procedurally-attached extensions can abort the entire relation modification operation Common system facilities will be used to undo the effects of completed storage method and attachment modifications if the relation modification operation is aborted

It is also 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

For data access operations, parameters of the operation select the access path to be used Access path extensions are selected using their attachment identifier plus an instance number (e g , access via B-tree number 3) Access path zero is interpreted as an access to the storage method The internal interface for data access is uniform across relation storage and access path extensions All accesses take keys as input and return keys and data Recall that relation storage methods define "keys" for their stored records, and that access paths maintain mappings from access path keys to record keys Normally, access paths will return record keys that can then be used to access the stored record directly via its storage method implementation

Comparisons and Conclusions

There is currently considerable interest in the database research community in developing methods and mechanisms to support database "extensions" Projects involving database extension research include the EXODUS project at the University of Wisconsin [CAREY86], POSTGRES at the University of California at Berkeley [STONEBRAKER86], ENCOMPASS at Tandem Computers [TANDEM83], PROBE at the Computer Corporation of America [DAYAL85], GENESIS at the University of Texas at Austin [BATORY86], and our own project, STARBURST, at the IBM Almaden Research Center [SCHWARZ86] None of the other projects support alternative implementations of relation storage, although EXODUS supports alternative mappings of conceptual relations onto their JUPITER storage system The EXODUS and GENESIS systems are database generators or tool kits which support creation of DBMS instances tailored to specific applications In contrast, the alternative implementations of the data management extension architecture are designed to cohabit an integrated database supporting multiple applications and cross-application usage ENCOMPASS allows alternative implementations of relation storage with significant restrictions (e g , no updates, and only key-sequential accesses)

Several of the extensible database projects support access path extensions POSTGRES, GENESIS, and EXODUS allow access path extensions using common system (page-oriented!) recovery and concurrency control facilities The data management extension architecture gives both relation storage and access path implementers more responsibility and wider latitude in the selection and implementation of recovery and concurrency control techniques In particular, record-level recovery and concurrency control, and partial transaction rollback can and will be used by data management extension implementations The "specialized processors" of PROBE resemble the alternative implementations of the data management extension architecture more closely than do the extension facilities of POSTGRES, GENESIS, and EXODUS

The data management extension architecture, like POSTGRES, is targeted for use in a full-function, *relational* database system PROBE, EXODUS, and GENESIS do not assume a relational data model or attempt the "vertical" integration of extensions into all levels of the database system By concentrating on extensions for relational database constructs, the data management extension architecture has ignored issues related to important data model extensions in the areas of data version management and non-normalized data models While these extensions are important, they are beyond the scope of the data management extension architecture and would require non-trivial modifications to the user interface

The importance of extending traditional database management systems into new application and hardware domains requires that it be possible to extend the low level data management facilities of the database system with alternative relation storage methods and new access paths and integrity constraints We have described a data management extension architecture which facilitates the integration of new data management extensions into the database management system The key to supporting data management extensions is to define generic abstractions for relation storage and access, and to view extensions as alternative implementations of the generic abstractions The use of procedural attachment of extensions to relation modification operations allows access path and integrity constraint extensions to participate, as needed, in the relation modification operations Finally, careful definition of common system services allows extensions to participate in system events and to be properly coordinated during transaction recovery and rollback

The integration of data management functions into the database management system is simplified by the proposed extension architecture while, at the same time, avoiding excessive overhead due to the close coupling of the extensions to the operations they support and to the common service facilities they require The extended relation descriptor design gives extension implementations considerable flexibility in their representation of the meta-data they need to access and manage extension representations, and simplifies extension implementations by providing a common descriptor management facility

The data management architecture described in this article is currently being implemented as part of the Starburst database project [SCHWARZ86] at the IBM Almaden Research Center It is our intention to exploit the flexibility of the data management extension architecture to experiment with application-specific extensions and to test alternative implementations of "standard" storage and access paths

There are, of course, other directions in which it is important to be able to extend database management systems In particular, user-defined abstract data types need to be addressed Also, we will investigate mechanisms for extending the repertoire of query evaluation techniques available to the query planning and execution facilities By constructing an experimental database management system that can host multiple implementations of the basic relational database abstractions, we will have an opportunity to evaluate the performance and utility of alternative data storage and access techniques within the uniform context of the Starburst database

Acknowledgments

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 Hass, George Lapis, Guy Lohman, C Mohan, Kurt Rothermel, Peter Schwarz, Irv Traiger, Paul Wilms, and Bob Yost

Bibliography

[ASTRAHAN 76] M Astrahan, M Blasgen, D Chamberlin, K Eswaran, J Gray, P Griffiths, W King, R Lorie, P McJones, J Mehl, G Putzolu, I Traiger, B Wade, and V Watson, *System R Relational Approach to Database Management,* **ACM Transactions on Database Systems, Vol. 1,** No 2 (June 1976) pp 97-137

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

[BERNSTEIN 81] P Bernstein and N Goodman, *Concurrency Control in Distributed Database Systems,* **ACM Computing Surveys , Vol. 13,** No 2 (June 1981) pp 185-221

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

[CODD 79] E Codd, *Extending Database Relations to Capture More Meaning,* **ACM Transactions on Database Systems, Vol. 4,** No 4 (December 1979) pp 397-434

[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)

[GUTTMAN 84] T Guttman, *R-Trees A Dynamic Index Structure for Spatial Searching,* **Proceedings of ACM SIGMOD '84,** Boston, MA (May 1984) pp 47-57

[SCHWARZ 86] P Schwarz, W Chang, J C Freytag, G Lohman, J McPherson, C Mohan, and H Pirahesh, *Extensibility in the Starburst Database System,* **Proceedings 1986 International Workshop on Object-Oriented Database Systems,** Pacific Grove, CA (September 1986) pp 85-92

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

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

[TANDEM 83] *TANDEM ENFORM User's Guide,* **TANDEM Computers Inc** (1983)

[VALDURIEZ 85] P Valduriez, *Join Indices,* MCC Technical Report Number DB-052-85 (To appear in ACM Transactions on Database Systems) (1985)