ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13 September 1996

Database Technology Beyond Object-Relational

Won Kim, Jorge F. Garza and Bruce Graham

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Database Technology Beyond Object-Relational

Won Kim, Jorge F. Garza, and Bruce Graham

UniSQL, Inc.

8911 Capital of Texas Highway, suite 2300

Austin, Texas 78759

Abstract

Currently, relational database technology, in terms of product installations, is the dominant database technology. During the past several years, object-oriented database technology has been introduced, in part to address various deficiencies of the relational model of data. Although object-oriented database products have not had a significant success in the market, key aspects of it are certain to find their way into relational database products. The basis of the next-generation database technology is the combination of relations and objects. On that basis, database architecture will still need to be augmented to address at least four major problems. These include management of multimedia data, long-duration transactions, spatial data, and temporal data. These four problems have been subjects of extensive research during the past 10 to 20 years, and many worthy results have been reported. However, today very few commercial database products include solutions to any of these problems. The objective of this paper is to provide directions for short-term research and development with the view to establishing solutions to these four problems that can be implemented in commercial database systems. We will review the fundamental objectives of each of the four problems, and either offer a concrete proposal as the basis of implementation in commercial products or at least provide a short list of concrete short-term research issues.

1 Introduction

During the past several years, object-oriented database systems and associated tools have been introduced to the market, partly in response to the anticipated growth in the use of object-oriented programming languages, and partly as an attempt to address some of the key deficiencies of relational database systems that are due to the inherent restrictions in the relational model of data. Although these products have not enjoyed a significant market success to date, they have significantly contributed to a better understanding of the benefits of object-oriented approach to database design and management. As a result, relational database vendors are now planning to augment their database systems and tools with object-oriented database design and management facilities. At the same time, object-oriented database vendors have recognized that one of the major impediments to their market success is the fact that their database systems do not support a full-blown non-procedural query language which is compatible with the ANSI SQL relational query language, and are now planning to augment their systems with such a query language. In the meantime, UniSQL, Inc. has developed a database system and associated application development tools much like those into which relational database vendors and object-oriented database vendors are both attempting to evolve their products.

It is clear that the foundation of the next-generation database technology is a unified relational and object-oriented database system; that is, a relational database system extended with the concepts of encapsulation (methods), arbitrary data types, nested objects, and inheritance; or an object-oriented database system extended with ANSI-SQL-compatible non-procedural query language and all major database features found in today's relational database systems (such as automatic query optimization, views, constraints, meta data management, authorization, triggers, transaction management, two-phase commit, parameterized performance tuning, etc.).

Database systems are tools that application developers use to develop applications, and database run-times are embedded in such applications. As such, if a database system lacks an important facility, application developers must implement the facility within the application. Today's database systems, both relational and object-oriented systems,

lack some major facilities that are often required in application development or in supporting application executions. There are four such facilities, including multimedia data management, long-duration transaction management, spatial data management, and temporal data management (history data and versioning). These have been subjects of extensive research for 10 to 20 years. But there are very few commercial database systems that support any of these facilities. UniSQL and ORION (ITASCA) [49] support a framework for multimedia data management. ORION [32] and Versant support an aspect of long-duration transaction management, namely the checkout/checkin mechanism between a public database and private databases. ORION [10] and Versant support versioning of simple objects (single records).

The research and development plans of relational database vendors and object-oriented database vendors, and the work of the ANSI SQL-3 standards committee, among other indicators, make it clear that the basis of the next-generation database system will be a unified relational and object-oriented data model. We believe that next-generation database systems will be built on this basis with support for management of multimedia data, long-duration transactions, spatial data, and temporal data. The UniSQL database system is the first commercial unified relational and object-oriented database system. It already supports a powerful framework for managing multimedia data. New releases of UniSQL between 1994 and 1995 will support long-duration transactions, spatial data management, and temporal data management. As such, this paper is in essence a statement of the research and development agenda for UniSQL for the next few years.

We believe that the accumulated results of research form a substantial basis for implementing each of these facilities in commercial database systems. However, additional, short-term, research is needed to realize the next-generation database system with these facilities. The research must proceed in two dimensions simultaneously. One is a critical assessment of the best known results with practical usefulness and implementability as the primary criteria; this will result in possible changes to (and even the discarding of) some of the current results. Another is to solve a reasonably short list of short-term research issues.

In this paper, we provide directions for short-term research in each of the four areas mentioned above with the view to their implementation in commercial database systems. For each area, we will review the objectives of the area, and either offer an outline of a proposal for meeting the objective or at least provide a list of short-term research issues to be addressed.

2 Multimedia Data Management

There is a growing trend in the computer industry to provide support for non-numeric data. The availability of low-cost data capture devices (digital cameras, digital scanners, digital audio, etc.), combined with low-cost mass storage (digital audio tape, optical disk media) and high-resolution display devices, has ushered in new classes of computer applications that manage multimedia data. Multimedia applications extend and apply the hypertext concept to a wide variety of data types beyond textual data to audio data, image data, video data, and animation.

Most of the current commercial database systems suffer from inability to manage arbitrary types of data, arbitrarily large data, and data stored on devices other than magnetic disks. They understand a relatively limited set of data types, such as integer, real, date, monetary unit, short strings, and BLOBs (binary large objects). Further, they are not designed to manage data stored on such increasingly important storage devices as CD-ROMs and video disks.

In this section, we will first review the objectives of multimedia data management, then outline solutions supported in UniSQL database management system, and provide a list of specific research and development issues.

2.1 Definition and Requirements

Broadly, multimedia data means arbitrary data types and data from arbitrary data sources. Arbitrary data types include the numeric data and short string data supported in conventional database systems; large unstructured data such as text, image, audio, graphics, video, animation, etc.; complex structured data, such as charts, graphs, tables, arrays, etc.; and compound documents that comprise of such data. Arbitrary data sources include a native database; external (remote) databases; host file base; data input, storage, and presentation (output) devices; and even data-generating and data-consuming programs (such as a text processing system).

The above definition of multimedia data leads to an enumeration of the requirements for multimedia data management, as follows.

1. the ability to represent arbitrary data types (including compound documents) and specification of procedures (programs) that interact with arbitrary data sources.
2. the ability to query, update, insert, and delete a multimedia data (including retrieval of multimedia data via associative search within multimedia data, minimally, text).
3. the ability to specify and execute abstract operations on multimedia data; for example, to play, fast forward, pause, rewind such one-dimensional data as audio and text; to display, expand, and condense such two-dimensional data as a bit-mapped image.
4. the ability to deal with heterogeneous data sources in a uniform manner; this includes access to data in these sources, and migration of data from one data source to another.

2.2 Multimedia Framework in UniSQL Database System

Figure 1. ESM as global controller of storage managers

The UniSQL database system is designed to meet all the requirements outlined above. ORION provided a preliminary framework for satisfying some of the requirements, in particular, abstract data types, query and manipulation on multimedia data, and abstract operations on multimedia data [49]. The object-oriented basis of UniSQL supports arbitrary user-defined data types, in addition to the data types supported by current database systems. UniSQL is also designed to address the problems of supporting arbitrarily large data in an arbitrary storage device with the extended storage manager (ESM). The ESM is essentially a global controller for arbitrary storage managers. As shown in Figure 1, the ESM consists of three types of "storage managers": the large object manager (LOM) which manages large unstructured objects in a native database (on a magnetic disk), a host file system interface manager which manages file based objects (FBOM), and the agent manager (AM) which controls arbitrary user programs that access or generate data.

From the perspective of the ESM, each of the storage managers it controls is merely a driver for a device in which objects of arbitrary sizes, structures, and types are stored. In particular, the ESM extends the database system functionality to include storage and retrieval of large objects (textual documents, images, and audio segments) that have been created and stored using file systems. This will obviate the need to first load files into a database before the data may be managed in a coordinated way with formatted records in the database. The LOM of UniSQL is designed to enhance flexibility and efficiency of storing and retrieving large objects, by providing specialized data structures and algorithms for that purpose. The FBOM may be viewed as an alternative implementation of the LOM which is to

deal with large objects that have been created and that will continue to be used under a different paradigm (i.e., file paradigm) from the database paradigm.

2.2.1 Generalized Large Objects

We combine the large objects (LOs) and the file based objects (FBOs) into a single construct, the generalized large object (GLO). The GLO is a system-defined class that has associated operations (methods) which can be used to access or modify the GLO. By generalizing the LO and FBO, we can simplify the user interface while also factoring out a common set of functions for both LOs and FBOs. GLO operations include read, write, delete, truncate, insert, append, etc.

It is simple for the user to build nested objects that include GLOs. An example of a nested object is an architecture design that includes its description and an architectural image. The object may have attributes `type_of_image`, `type_of_structure`, `architect`, and `construction_date` to be associated with the image of the building saved in a GLO. Users may query against non-GLO attributes.

Like ordinary data, the GLO is locked before it is read or updated. The application has the option of committing or aborting the transaction in which updates to the GLO have been made. In particular, any changes made to the LOs or FBOs may be backed out upon user-initiated transaction abort or recovery from a system crash.

2.2.2 Program Control via Agents

As mentioned earlier, agents are used to represent interfaces to arbitrary programs that can access or generate data. One motivation for the agent concept is to capture data that is generated by a program. Another motivation is to allow external processes to be triggered by a database access.

UniSQL provides an agent class with system-defined methods. The user may create instances of the agent class, just like any user-defined object. The user may also inherit from the agent class to create new agent classes that are customized for each application. There are system-defined methods to define, launch, suspend, resume, and terminate an agent.

The agent is made up primarily of three attributes: the shell string, the GLO list, and the host attribute. The shell string attribute is a specially formatted string that contains the UNIX-shell command (or program name) that will be run. The format of this string is similar to the C "printf" string format; it defines placeholders that are used by the agent to interact with GLOs. The GLO list attribute is a list of GLOs that are used in conjunction with the shell string to produce a string that may be executed by the shell processor. The third attribute is the name of the host where the shell command must be executed.

The GLO functionality and the agent functionality are implemented in different parts of the UniSQL database system. Figure 2 contains a broad overview of the different ESM components and their relationship in a typical client/server configuration. The large object manager (LOM) is implemented on the server side of the system, while the agent manager (AM) and the file-based object manager (FBOM) are both implemented on the client side of the system.

2.2.3 Object Migration

In a multimedia environment, three types of data migration are possible: data capture, data presentation, and data move. Data capture moves an object from a capture device to a storage or even directly to a presentation device (e.g., audio data is moved from a microphone to a speaker). Data presentation moves data from a storage or capture device to a presentation device. Data move transfers a data from one storage device to another storage device (possibly the same device) (e.g., move a file object from a file system into the database, move a database object to a file, move an object filtered through a process to a file, etc.).

data capture Data capture is simply the act of adding data to a GLO. This could be done at different times during the life of the GLO. After the GLO has been created, we can add data to it during the initialization process. When first creating a GLO, data can be copied to the GLO by an initialization routine. The initialization routine may provide a

default data set, or it may simply be used to specify that the GLO is an FBO or an LO. In the case of an FBO, the initial data may already exist in the file, and is available for access immediately. In the case of an LO, any initial data must be copied to the LO by the initialization routine. After the GLO has been created (and perhaps initialized), the user may add, delete or replace the data.

Data may be captured from one of many different types of data sources by executing an agent which has been "tuned" for it. For example, an agent may be a process that controls a FAX machine, or a process that controls audio input from a microphone.

data presentation Using the UniSQL built-in multimedia class hierarchy, one can build up new presentation classes for various types of data as subclasses. For example, UniSQL provides an audio interface through an audio class that presents audio data through a speaker. The user can further specialize the audio interface by defining the audio hardware requirements for presenting the data as additional attributes in a subclass.

data move There are many meaningful combinations of data source and data destination for data migration within a rich multimedia environment which supports interoperable access to data from different data sources. Some examples data move that are of obvious interest are the following:

- LO to FBO
- FBO to LO
- LO to LO
- FBO to FBO

To implement the migration of objects, we can combine the functionality provided by the agent with a data source and a data destination. An agent with a shell string that contains the special value "MIGRATE" will notify the Agent Manager to copy the data from the first GLO in the GLO attribute list to the second GLO in the attribute list.

2.3 R&D Directions

The fact that multimedia data management requires dealing with large unstructured data, besides the conventional small data, implies changes in the unit of concurrency control and authorization. The unit of concurrency control in conventional database systems is often a page or even a table, rather than a single record. The unit of authorization in conventional database systems is usually an entire table or a column of a table, but never a single record. In a multimedia environment, a single object, such as a single compound document, is the logical unit of access, and as such the unit of concurrency control and authorization in database systems that support multimedia data should really be a single object (comparable to a single record in a relational database).

Multimedia data sources, such as a satellite feed, often produce data at a rate that a database system cannot process in real-time. The relationship between a database system and multimedia data sources needs a closer examination, so that perhaps to some extent real-time data input and output may be combined with database processing (queries and updates).

There are situations when it becomes desirable to synchronize the presentation of data to the user. One example of this is the synchronization of audio data with visual data in a multimedia display. The need for synchronization creates a time-line dependency between the data and the actual presentation of the data. This implies a level of cooperation between the different presentation mechanisms for the different types of data. To some degree, this is an application-specific problem, because each multimedia application will have different synchronization requirements.

3 Long-Duration Transactions

The objective of long-duration transactions is to model long-duration, interactive database access sessions in application environments. The fundamental assumption about short-duration of transactions that underlies the traditional model of transactions is inappropriate for long-duration transactions. The implementation of the traditional model of transactions may cause intolerably long waits when transactions attempt to acquire locks before accessing data, and may also cause a large amount of work to be lost when transactions are backed out in response to user-initiated aborts or system failure situations.

The literature is fairly rich with proposals for long-duration transaction management. However, there exists a general confusion about long-duration transactions. There are two reasons for this. First, there are a variety of flavors of "long-duration" transactions, including transactions by interactive users, transactions among cooperating users, just plain long transactions, transactions that are really sequences of well-defined subtransactions, transactions that correspond to a hierarchical organization of users in terms of their roles, etc. Second, proposed models of long-duration transactions have largely been developed to allow the flexibility of interaction among users, without consideration of consistency criterion that it may have to satisfy.

The objective of a transaction model is to provide a rigorous basis for automatically enforcing a criterion for database consistency for a set of multiple concurrent read and write accesses to the database in the presence of potential system failure situations [21, 8]. The consistency criterion adopted for traditional transactions is the notion of serializability. Serializability is enforced in conventional database systems through the use of locking for automatic concurrency control, and logging for automatic recovery from system failure situations. A "transaction" that does not provide a basis for automatically enforcing database consistency is not really a transaction. To be sure, a long-duration transaction need not adopt serializability [8] as its consistency criterion; however, there must be some consistency criterion.

There are two fundamentally different models of long-duration, interactive database access sessions: single-database model, and multiple-database model. The single-database model captures the situations where multiple transactions should share a single common database. The multiple-database model addresses situations where multiple transactions should operate with multiple databases.

The single-database model has received considerably more attention than the multiple-database model. The focus of research into the single-database model has been on addressing the twin problem of long waits and loss of work that the long duration of transactions brings about [1, 19, 35, 36, 13, 15, 46, 12, 26].

The multiple-database model is represented by proposals for shared and private database architectures, checkout and checkin of data to and from shared and private databases [28, 30, 34, 32]. Each user may populate his/her private database with data checked out of the shared database, perform updates against the data, and check them back into the shared database.

The single-database model requires the introduction of a notion of database consistency, and protocols for concurrency control and recovery, that are different from those supported in traditional database systems. If a "reasonable" notion of database consistency is to be supported (i.e., the database system is to enforce it automatically), there are bound to be "conflict" situations where one transaction comes into an access conflict against some other transaction. If a wait is to be avoided, some means of a negotiated settlement of the conflict must be provided, thereby dragging the users into the details of concurrency control. The single-database model is more appropriate than the multiple-database model in an environment where it is difficult to determine in advance logical partitions of the database that correspond to work to be performed, and the users closely cooperate.

The multiple-database model can be used to "work around" the conflict situations inherent in long-duration database sessions. Since each user may copy data from the shared database and work against his/her private database, "disconnected" from the shared database (at least on the surface), the users can avoid the conflict situations. In particular, multiple users may be simultaneously updating the "same" data, without having to wait for other users to complete their updates. However, when updated data is to be checked into the shared database, it may have to be checked in as a new version, necessitating version management [10, 11, 29]. Further, when data in a private database references data in the shared data, or vice versa, a private database is not really disconnected from the shared database. For example, the evaluation of a query in general will require the database system to access both a private database and the shared database, even if the query may have been formulated against a private database [32]. The multiple-database model is more appropriate than the single-database model in an environment where it is easy to determine in advance logical partitions of the database that correspond to work to be performed, and the users do not or cannot not cooperate.

The two models of long-duration transactions are both valid; one is more appropriate than the other in different situations. As such, in a general-purpose database system, both models should be supported. The single-database and multiple-database dichotomy is analogous to the "single-version" and multi-version approaches in database concurrency control [7]. In the remainder of this section, we outline our proposals for both models. One is the proposal for cooperating transactions to support the single-database model. Another is the shared/private database architecture to support the multiple-database model.

There have been various proposals for nested transactions [30, 1, 37, 5]. A nested transaction is the transaction-equivalent of a nested complex object in data modeling. A transaction may recursively spawn (fork) one or more subtransactions, and the subtransactions may execute in parallel and commit/abort independently of one another. Nested transactions are primarily a mechanism for exploiting parallelism inherent in computations; they were not originally proposed as a means of addressing the requirements of long-duration transactions. However, the possibility of mapping the hierarchical structure of a group of interactive database users (e.g., the organizational structure of a company, contractor and subcontractors, etc.) to a nested transaction motivated various proposals of nested transactions in the context of long-duration, interactive database sessions [30, 1, 26]. These proposals all presume a single database, and may be regarded as a variation of the single-database model that maps individual transactions to users in a hierarchical structure in terms of the roles they play within a cooperative project involving database accesses. The nesting of transactions, although clearly useful, adds another dimension of difficulty, both in terms of semantics and implementation.

3.1 Cooperating Transactions

Cooperating transactions are transactions against a single shared database in a cooperating and interactive work environment. Because of their interactive nature, cooperating transactions tend to last much longer than traditional transactions. Further, unlike traditional transactions which must be isolated from all other concurrently executing transactions, cooperating transactions should (need) not be completely isolated.

A fundamental problem with cooperating transactions is that no consistency criterion has been established for them. One reasonable basis for a consistency criterion for cooperating transactions is to retain serializability for a collection of cooperating transactions, but not for each transaction within the collection. This makes a collection of transactions behave like a single traditional transaction, with respect to any other collection of cooperating transactions. We will call such a collection of cooperating transactions a team transaction. Within a team transaction, the strict isolation enforced in traditional transactions is relaxed so that each constituent cooperating transaction will be able to access the shared database in cooperation.

The issue then is reduced to establishing an appropriate notion of database consistency for cooperating transactions within a team transaction. One possibility is to use the level-1 (uncommitted) or level-2 (cursor stability) consistency notion [21]. Serializability is equivalent to level-3 consistency. Level-2 consistency prevents dirty reads, but does not guarantee repeatable reads. Level-2 consistency is obtained by requiring write locks to be retained until the end of transactions, while holding read locks only for the duration of reads. Level-1 consistency prevents lost updates, but can result in dirty reads and non-repeatable reads. Level-1 consistency is obtained by requiring write locks to be retained until the end of transactions, while not requiring read locks. It is said that level-2 consistency does not in practice increase concurrency significantly; so level-1 consistency makes more sense for consideration for cooperating transactions.

One obvious problem with level-1 (and levels 2 and 3) consistency is that write locks are held until the end of transactions, thereby preventing all other transactions from accessing locked data. In a long-duration, interactive work environment, the blocking forced on cooperating users is not desirable. One possibility is to allow locks held by any cooperating transactions to be broken by other cooperating transactions under certain circumstances [25]. The notions of level-1 (or level-2) consistency may be combined with this notion of soft (breakable) locks.

The essence of soft locks is as follows. The user may request the system to set soft locks on certain data, while continuing to request standard (hard unbreakable) locks on other data. On soft locks, the user may attach certain operating parameters, including who may break the locks, whether the locks may be broken only with the user's consent, and whether the user should be notified if the locks are broken. Once the locks are broken, the user who breaks the locks may obtain the locks.

Soft locks remain soft locks; that is, a cooperating transaction that acquires a soft lock by taking it away from another cooperating transaction may also give it up to another cooperating transaction. A soft lock cannot be converted to a hard lock, just as a hard lock cannot be demoted to a soft lock.

A cooperating transaction T1 may have its soft locks broken by another cooperating transaction T2, and may also be able to later brake it back from T2. From the perspective of traditional transactions, a cooperating transaction may suffer from non-repeatable reads, dirty reads, and even lost updates. However, soft locks are broken with consent within a team of cooperating users, and that, when a soft lock is broken, the lock ownership (and therefore responsibility for integrity enforcement) is transferred from one cooperating transaction to another. So, when a cooperating transaction has its soft lock broken, there is no loss of database integrity as long as the memory of the data on which it held the soft lock is also erased from the application/user and from the database system.

The memory of the data with a broken soft lock is completely lost if two conditions are satisfied. First, the cooperating transaction does not henceforth make use of any knowledge of the data. Second, any insert (creation) and updates that may have been made to the data by the cooperating transaction until the soft lock was broken are not backed out by the database system during crash recovery or transaction abort. If these two conditions are satisfied, and if all cooperating transactions that exchanged locks are aborted or committed together, cooperating transactions may be said to satisfy even the serializability criterion. In practice, it may be impossible to force the application or user to not make use of the knowledge of data after a soft lock on it is broken. Even so, as long as changes to the data can be excluded from consideration during recovery, cooperating transactions may be said to satisfy at least the level-1 consistency criterion of traditional transactions.

3.2 Shared and Private Databases

The shared and private database architecture, augmented with checkout/checkin facilities, and versioning and triggering to in turn augment the checkout/check in facilities, is a valid means of addressing the requirements of some types of long-duration, interactive database environments. Of course, the checkout and checkin facilities themselves read

and write data, and as such checkout and checkin are database activities regulated by the transaction management facilities of a database system. The long-duration nature of interactive database sessions do not cause any problem for transactions against a private database that do not involve access to the shared database, since such transactions do not cause access conflicts. For transactions that involve access to the shared database, either traditional transactions or cooperating transactions may be used.

Despite some technical problems that it failed to properly address, the definitive work thus far on a shared and private database architecture is that reported on the ORION object-oriented database prototype [32]. The semantics of checkout and checkin described there may be the best basis for adaptation in commercial database systems.

An authorized user may create one or more private databases. For a user who creates a private database, the private database is simply a logical extension of the shared database. The shared database may be accessed by all authorized users; however, a private database may be accessed only by the user who created it. The schema of a private database is a logical extension of the global schema of the shared database.

The owner of a private database may populate the private database by directly creating objects in it or by checking objects out of the shared database. A checkout means copying data from the shared database and inserting the data into a private database. A checkin is the converse of a checkout; however, it moves (not copies) data from a private database into the shared database.

A user may check out or check in one object or a set of objects (identified by object identifiers), or a set of objects via a query expression.

An object in a private database may reference objects in the shared database, and vice versa. This situation arises as long as a checkout will not be for a closure of objects, that is, all objects recursively referenced by an object being checked out are also included in the checkout; and, similarly, a checkin will not be for a closure of objects.

There are some semantic issues that remain to be thought through. One is whether checkouts and checkins should be for closures of objects. Allowing checkouts and checkins of closures of objects will simplify query processing, and strengthen to some extent the firewall between the shared database and private databases. However, objects that are not needed by the user may be transferred into private databases.

Another issue is whether there should be a one-to-one correspondence between a checkout and a checkin; that is, should all objects that are checked out for update be required to be checked in together, or can a private database retain some of the checked-out objects after other objects have been checked in.

Next, if the users are completely free to name classes in private databases, they may come in conflict when they are checked into the shared database. The ORION approach is to prevent this by registering each class with a global name server (even if the class is being created in a private database), rejecting it if it conflicts with another name that has already been registered. This is somewhat undesirable, since it compromises "autonomy" of private databases.

A fourth issue is integrating multimedia, spatial, and temporal data management frameworks into the shared and private database architecture as well.

3.3 R&D Directions

The most urgent research issue in long-duration transactions is to establish a taxonomy of cooperating transactions (including hierarchical organizations of transactions). The resulting taxonomy, along with the models of database consistency, can then be used to guide the design and implementation of one or more worthy models of cooperating transactions. The taxonomy should be based on a careful consideration of the type and importance of the application environment for which a particular proposal has been designed, and the model of database consistency the proposal implies. If a particular proposal implies no consistency, the proposal should be discarded. If a proposal aims at a narrow area of application, it too should be discarded. A good starting point of literature search for this purpose may be [2], which even offers a preliminary taxonomy of existing proposals; but in our view, a better taxonomy based on a more critical assessment of the current proposals is needed.

4 Spatial Data Management

Spatial data management has been an active area of research in the database field for two decades, with much of the research being focused on developing data structures for storing and indexing spatial data: spatial data structures that have been proposed include Grid File [38] (and Bang File [17], which is a variation of the Grid File); tree structures extended for multidimensional point data, such as quad trees [16, 6, 42], K-D trees [6] (also K-D B trees [40], which extends the K-D trees with secondary storage considerations; and spatial K-D trees [39]); and tree structures extended for rectangular objects, such as R-trees [23] (also R+ trees [14, 45, 22], R* trees [4], and Parallel R-trees [27], which are optimized/extended versions of the R-trees). However, no commercial database system provides facilities for directly defining and storing spatial data, and formulating queries based on search conditions on spatial data. We believe the following are the relevant issues on which near-term research should be focused (in the order of decreasing importance and urgency).

First, relational query optimization techniques need to be extended to deal with spatial queries, that is, queries that contain search conditions on spatial data. In particular, a reasonable cost model for computing the selectivity of spatial predicates needs to be developed.

Second, more work needs to be done on experimental validation of the relative performance of some of the more promising data structures and indexing structures proposed thus far, with consideration of a much broader set of operations than just a few operations that have typically been used in the limited performance studies conducted thus far.

Third, it is difficult to build into a single database system multiple data structures for spatial indexing, and all spatial operators that are useful for a wide variety of spatial applications; as such, it is desirable to build a database system so that it will be as easy as possible to extend the system with additional data structures and spatial operators.

In the following subsections, we will discuss these issues in some detail. (The materials in the following subsections have been taken from [33].)

4.1 Query Optimization and Processing

Spatial operators manifest themselves as comparison operators in search predicates in queries, or as postprocessing functions against the results of queries. Each of these operators allows comparison of a user-supplied spatial object with the value of a spatial attribute (i.e. the attribute whose domain is a spatial data type) in each record in a database.

To support spatial operators, the following changes/extensions need to be made to the query processor and query optimizer.

- The query processor should be able to evaluate spatial predicates. A spatial predicate is a predicate which includes any spatial comparison operator.
- The cost model for computing the selectivity of a predicate needs to be extended to take spatial predicates into account.

The first point above is important, regardless of whether spatial indexes are used for query processing. Since minimum bounding rectangles are used to approximate arbitrary shapes of interest to applications, the objects identified through a spatial index on rectangular regions need in general to go through a postprocessing phase.

The query optimizer should determine when a spatial index can profitably be used in place of a sequential search. It is not desirable or even necessary to define a very detailed and precise cost model. The cost model used in relational query optimizers is based on a few not-exactly-realistic assumptions (e.g., uniform distribution of indexed-key values, independence of values in different columns) and existence of statistics about the database (e.g. cardinality of a table, minimum and maximum value of an indexed key, etc.) where the statistics are not in general up to date [44]. An approximate cost model comparable in precision to that used in relational query optimization is essentially what is needed for optimizing spatial queries.

Figure 3. 2-dimensional Scales

4.1.1 cost model for spatial predicates

We propose the following as the basis of the cost model for computing the selectivity of spatial predicates. To support it, the database system needs to maintain the following information (assuming rectangular regions only, for expository simplicity – extensions for k-dimensional data are straightforward).

N: cardinality - total number of spatial objects

scale for each dimension (i.e. the maximum value and minimum value for each dimension)

The selectivity for the exact match operator is $1/N$; this is equivalent to the selectivity of the standard equality predicate. The lowest selectivity is 1, when all records satisfy the predicate. The selectivity of left-of, right-of, above, or below, a given coordinate value is computed as that for standard predicates involving $<$ or $>$:

$1 / (\text{maximum value of the scale of the } i\text{-th dimension} - \text{the given coordinate})$

For example, for the predicate "left of a region $(x_1, y_1), (x_2, y_2)$ ", where $x_2 < x_1$, and the maximum value of the x coordinate is x-max and the minimum value is 0, the selectivity of the predicate would be

$1 / (x\text{-max} - x_1)$.

Note that this selectivity can be easily fine-tuned by maintaining more precise information about each scale. For example, if each scale is partitioned into a number of unequal-length partitions, the precise scale information may be maintained and used in selectivity computation.

The selectivity computation for the contained-in, contains, within-distance-of, nearest and farthest may straightforwardly be reduced to a union and/or intersection of selectivities along each dimension. For example, the selectivity for the predicate "intersects a region $(x_1, y_1), (x_2, y_2)$ ", shown in Figure 3, where $x_2 < x_1$, and the maximum values of the x-dimension is x-max, and the maximum value of the y-dimension is y-max, with their respective minimum values being 0, would be computed by first computing the selectivities for the predicates "above y_2 ", "below y_1 ", "left of x_1 ", and "right of x_2 "; and then computing the union of the four selectivities.

4.1.2 processing spatial "join"s

In relational database systems, join is used to correlate tuples of two or more tables on the basis of the values of user-specified columns of the tables. Relational database systems typically support two algorithms to evaluate join queries: nested-loop algorithm, and sort merge algorithm [44]. The spatial intersect operator involving two or more sets of spatial objects requires spatial joins. A spatial join correlates spatial tuples (i.e. tuples containing spatial data) belonging to two or more tables on the basis of user-specified spatial attributes of the tables.

The literature on spatial query processing has assumed a spatial equivalent of the nested-loop algorithm for spatial joins. Spatial equivalents of the sort merge algorithm need to be developed. Since spatial data is multidimensional, it is not obvious what "sorting" means. In sort merge join in relational database systems, sorting is done to prepare tuples of the tables being joined so that join may be performed with a single scan of each of the tables.

4.2 Spatial Data Structures

A major focus of research into spatial data management has been on data structures for storing and retrieving spatial data. The results of this line of research have found their way into various commercial geographical information systems (GIS). Of the known data structures, R-trees seem to be the most popular for storing non-point objects (in particular, rectangular regions). One serious problem with R-trees is that it allows bounding rectangles to overlap. Since objects within an overlapped area belong to only one of the bounding rectangles, in general multiple branches of the index tree need to be traversed. Some optimization techniques have been suggested in the literature for R-trees [41, 45, 4].

Although some work has been done on the relative performance of the R-tree variants and some other data structures [41, 45, 14, 22, 4, 24], it is not clear if the results are conclusive. There are a few problems with the current results. First, most of the current performance studies are not based on actual implementations of the data structures within a full-blown database system; [22] is more realistic than others. It is not clear to what extent the simplifying assumptions made in analytic studies invalidate the results.

Second, most performance studies consider only a few types of spatial operators (e.g., find all objects that contain a given point, find all regions that intersect a given region); the set of spatial operators needs to be expanded. For example, for applications that deal with only points, lines, and regions, there are six classes of spatial operators: region vs. region, line vs. region, point vs. region, line vs. line, point vs. line, and point vs. point. Of course, within each class, there are several meaningful operators. For example, within the region vs. region class, useful operators include, besides the obvious "exact match", intersect, containment, one-side-of (left of, right of, above, below), within-distance-of, nearest (farthest), etc. A data structure that performs well for the region vs. region class of spatial operators may not be suitable at all for, say the line vs. line class of spatial operators.

4.3 Extensible Architecture

There have been a few proposals for "extensible" database systems, ranging from proposals for designing an elaborate extensible query optimizer [18], even an optimizer generator to accommodate different data models [20], and even a database system configurator that constructs a database system from a set of database system building-block modules [3]. There is a reasonable case for an extensible architecture to support spatial objects. First, the index data structure supported in a database system is likely to be unsuitable for some of the applications. Second, it may not be practical to support all possible spatial operators; that is, some applications may require spatial operators beyond those that are built into a database system for a wide variety of applications.

If a database system is to allow installations to add their own index data structures and/or spatial operators, it must be designed for extensibility in the first place. The following discussion outlines how the design-for-extensibility affects the database system architecture.

First, let us see what it will take to allow a new index data structure to be added to a database system. The new data structure will co-exist with other index data structures. For each data structure, there needs to be a separate index manager that supports basic indexing functions.

Between the index managers and the query processor, there needs to be a software module, called a generalized index manager (GIM). The GIM is a thin layer on top of the individual index managers. It is a collection of modules that implement higher-level index operations than are supported directly within each individual index managers; for example, the closed-area search operation, if not supported directly as a basic indexing function, may be implemented in the GIM in terms of a set of index search operations; or insertion of a set of index entries may be implemented on top of an index manager by repeatedly inserting one index entry at a time. Each of such index operations may be implemented as a module in terms of the basic functions of the underlying index managers.

The query optimizer needs to be designed so that the cost models for any spatial operators that the installations may choose to include may be added without requiring changes to the existing query optimizer code. The part of the query optimizer that computes the selectivities of predicates should be structured so that new types of predicates and their selectivities may be added as query optimizer parameters. For example, a data structure may be designed to maintain the selectivity model for each type of the predicates supported, and a set of commands may be defined to add/delete/update the cost model of any predicate.

The query processor also needs to be designed to invoke installation-supplied algorithms, i.e., modules, that evaluate predicates that involve installation-defined spatial operators.

Figure 4 summarizes the essence of the architectural changes to the query processor, query optimizer, and index manager.

5 Temporal Data Management

There are two components to temporal data management: history data management and version management. Both have been the subjects of research for over a decade. The troublesome aspect of temporal data management is that the boundary between applications and database systems has not been clearly drawn. Specifically, it is not clear how much of the typical semantics and facilities of temporal data management can and should be directly incorporated in a database system, and how much should be left to applications and users. In this section, we will provide a list of short-term research issues that should be examined to shed light on this fundamental question.

5.1 History

The focus of research into history data management has been on defining the semantics of time and time interval, and issues related to understanding the semantics of queries and updates against history data stored in an attribute of a record [47, 48]. Typically, in the context of relational databases, a temporal attribute is defined to hold a sequence of history data for the attribute. A history data consists of a data item and a time interval for which the data item is valid. A query may then be issued to retrieve history data for a specified time interval for the temporal attribute. The mechanism for supporting temporal attributes is similar to that for supporting set-valued attributes in a database system, such as UniSQL.

In the absence of a support for temporal attributes, application developers who need to model and manage history data have simply simulated temporal attributes by creating attribute(s) for the time interval, along with the "temporal" attribute. This of course may result in duplication of records in a table, and more complicated search predicates in queries. The one necessary topic of research in history data management is to quantitatively establish the performance (and even productivity) differences between using a database system that directly supports temporal attributes and using a conventional database system that does not support either the set-valued attributes or temporal attributes. If it

can be convincingly established that the benefits of a database system that supports temporal attributes are substantial, database vendors will be strongly motivated to augment their systems with history data management.

5.2 Versioning

Despite a large number of proposals on version support in the context of computer-aided design and software engineering [10, 29], the absence of a consensus on version semantics has been a key impediment to version support in database systems. Because of the differences between files and databases, it is intuitively clear that the model of versions in database systems cannot be as simple as that adopted in file systems to support software engineering. For databases, it may be necessary to manage not only versions of single objects (e.g., a software module, a document, etc.), but also versions of a collection of objects (e.g., a compound document, a user manual, etc.) and perhaps even versions of the schema of a database (e.g., a table or a class, a collection of tables or classes).

Broadly, there are three directions of research and development in versioning. First is the notion of a "parameterized versioning", that is, designing and implementing a versioning system whose behavior may be tailored by adjusting system parameters. This may be the only viable approach, in view of the fact that there are various plausible choices for virtually every single aspect of versioning (e.g., updatability and deletability of versions, version identifier, version timestamp, version-derivation hierarchy). The second is to revisit these plausible choices for every aspect of versioning, with the view to discarding some of them as either impractical or flawed. The third is the investigation into the semantics and implementation of versioning collections of objects [9] and of versioning the database schema [31]. In the remainder of this section, we will discuss some of the plausible choices for some aspects of versioning, and then outline parameterized versioning.

5.2.1 Some Aspects of Versioning

updatability and deletability [10] proposes three types of versions in terms of updatability and deletability: transient versions, stable versions, and released versions. A transient version may be updated and deleted by its creator at any time. A stable version may be updated, but not deleted. A released version may not be updated or deleted at any time. Further, [10] proposed that multiple new versions may be derived from any version at any time, giving rise to a version-derivation hierarchy. If a new transient version is derived from a transient version, the original transient version automatically becomes a stable version.

Under this proposal, some questions arise for which there are differences of opinion. Should we allow non-leaf versions to be deleted? What should happen to child versions (derived versions) if their parent version is deleted? What does it mean if a non-leaf version is updated after versions have been derived from it, possibly copying its contents; should the changes be propagated to the derived versions?

Some also argue that updatability and deletability should be applied across a collection of versions, rather than on individual versions.

version identifier Proposals for versioning almost invariably associate a version identifier (version id) with a version. There are at least three alternatives for maintaining and accessing version ids.

One is to maintain the version id as an internal data that is not projected as an attribute. This means the version id is independent of any semantics of attributes. Because it is not an attribute, however, a new syntax must be provided in order to access it. For example, to find objects with "version id equal to 1" may be expressed in the following SQL-like syntax.

```
SELECT x FROM verclass x
WHERE version is 1;
```

The expression "version is 1" is a language extension for accessing the version identifier. The advantage of this approach is that there is a clear distinction between what a version id is and how it is accessed. The disadvantage is that a new syntax must be defined to access this value.

Another alternative is to maintain the version id as a system-defined attribute (but the attribute is defined automatically by the system and has some additional properties.) This has the advantage of requiring no additional syntax for

accessing the version id. For example, suppose that the system defines an attribute called `version_id` if versioning is requested. The previous example now becomes:

```
SELECT x FROM verclass x
WHERE version_id = 1;
```

The disadvantage is that it may cause complications for class hierarchies (inheritance hierarchies) in object-oriented systems. In particular, we must decide if the version id attribute should be automatically inherited by subclasses. If we decide that the version id attribute should be automatically inherited, it will force versioning on all classes whether or not the creators and users of the classes want. To avoid this, we may form two separate class hierarchies: one for classes that need versioning, and one for those that do not. This, however, may force duplicate definition of classes and their attributes along two separate class hierarchies. (The question arises for the first alternative discussed above. What is the behavior for subclasses of `verclass` in the example query that are not versioned.? Are they silently excluded from the query?)

The third alternative is to define the version id by having the user define a normal attribute for it. Of course, the user-defined version id attribute must be made non-updatable, and the users will enter the version ids as they create new versions. All other attribute semantics are retained.

It is highly desirable to have the database system generate the version ids automatically. But even the automatic generation of version ids is not so straightforward. It may be difficult to define a single algorithm for generating version ids, since it is up to the application to define the semantics of version id values. The only thing the system may be able to enforce is that the values be unique. For example, the application may choose to have the version id be a string and the values follow the pattern: "1.0", "1.1", "1.2", etc. Alternately, they may want something like "1.a", "1.b", "2.c", etc. for version ids.

The question becomes how we may allow the users to "register" an algorithm for generating version ids. One promising approach would be through a method. Now the question is how this method is invoked and what the arguments of the method are. One possibility is to make this explicit in the class definition, for example, as follows:

```
CREATE CLASS person
  (name string,
   age integer,
   vid integer)
METHOD vid_generator() : integer
VERSION vid AUTOMATIC vid_generator;
```

Here the `VERSION` clause has an additional phrase `AUTOMATIC vid_generator` that requests automatic generation of the version id and specifies that the method `vid_generator` is to be called to perform the operation.

version-derivation hierarchy Some argue that the version-derivation structure should be a hierarchy (tree) and others argue that it should be a directed acyclic graph (DAG). A version-derivation hierarchy results when one or more versions may be derived only from one other version; while a DAG results if one or more versions may be derived from one or more other versions. For simplicity, in the remainder of this section, we will use the term hierarchy to mean either a tree or a DAG structure.

There is also some question as to whether the database system should maintain version-derivation hierarchies. Some argue that the maintenance or even existence of derivation hierarchies should be handled by the application. The argument is similar to that for the generation of version ids. The system does not know what the parent/child relationship is between two objects in a version cluster (collection of versions). The system may be able to determine this relationship at the instant that the new version is created but the application may wish to change this relationship later. If the application must have complete control over the structure of the derivation hierarchy, why then can it not be maintained by the application? Does the system provide enough functionality in this area to merit its inclusion?

A related question is what it means to derive a version from another. Does it mean that the attribute values of the new version are initialized to the current attribute values of the source object? If they are not, why is this object considered to be a derivation of the source object? Is it enough to simply add another version to the version cluster,

and let the application be responsible for initializing the values of the attributes? If the application wishes to establish a parent/child relationship between the new object and some other object in the version cluster, it can do this through the usual object update operations.

If creating a version means that the attribute values of the new object will be initialized to the values of the source object, what does it mean when there are more than one source objects? In this case, the system cannot predict how the user intends to merge the source attributes into the new version. For example, consider the following statement.

```
CREATE VERSION '0.9'
FROM '0.5', '0.8'
OF SELECT chip FROM chip WHERE designer = 'scottie';
```

What are the attributes of the new version? If they are undefined, then the application must go in and initialize the attribute values manually. If that is the case, then it is no harder to establish the derivation hierarchy using normal attributes at the same time the attribute values are assigned.

A reasonable proposal may be that the system only provide support for the most fundamental requirements for versioning. To this end, the system must be able to add a new object to a version cluster and assign it an unique version id. Any higher level semantics can be defined at the application level. With these simplified semantics, a simpler syntax would be possible.

```
INSERT INTO person (name, age)
VALUES ('jeff', 34)
AS VERSION OF
  SELECT person FROM person WHERE name = 'jeff';
```

This is an otherwise normal insert statement but the addition of the "as version of" clause declares that the new object is to be a version of an existing object. From the implementation level, it means that a new object is inserted and added to the version cluster of the object in the select list.

5.2.2 *Parameterized Versioning*

As illustrated thus far, there are various plausible choices for each of the various aspects of versioning. It may be unreasonable to pick only one choice for implementation in a database system for each aspect of versioning. A versioning system that does not admit greater flexibility may be doomed to failure, since the behavior of the system may be unacceptable to users who happen not to agree with the choices selected by the database system designers. This has led us to believe that the notion of parameterized versioning is perhaps the best way to go. The idea is to implement a versioning system with two components. One component is to provide an infrastructure for all the basic functions that are necessary to support most of the plausible choices in all aspects of versioning. The other component is a mechanism that allows the users to select one particular choice in each of the aspects of versioning. This component makes it possible for the versioning system to be tailored to a particular set of choices for all aspects of versioning.

For example, the infrastructure may be designed to support any version-derivati on structure: chain, hierarchy, or directed acyclic graph. Further, it may be designed to support different notions of deletability and updatability: a non-leaf version may or may not be deleted; a stable version may or may not be deleted; etc. The user (or system administrator) may pick the hierarchy as the version-derivation structure, specify that non-leaf versions not be deleted, and indicate that a stable version may not be deleted, etc.

6 Summary

The trend in database industry today clearly indicates that the basis of the post-relational database technology is a database technology that unifies the relational and object-oriented models of data, and implements a nonprocedural database language that extends ANSI SQL with object-oriented database design and manipulation facilities. There

are at least four problem areas that have been the subjects of long and extensive research and prototyping activities during the past 10 to 20 years. These include multimedia data management, long-duration transactions, spatial data management, and temporal data management. The research and prototyping activities in these areas have yielded many worthy results, and now concerted short-term research and development efforts should be made to bring the areas to fruition, that is, bring the results to commercial post-relational database systems. In this paper, we provided, for each of these four areas, a brief review of the objectives of the area, and then outlined a proposal for pushing the research and development activities to fruition or at least provided a list of short-term research topics.

References

- [1] Bancilhon, F., W. Kim, and H. Korth. "A Model of CAD Transactions," in Proc. Intl Conf. on Very Large Data Bases, August 1985, Stockholm, Sweden.
- [2] Barghouti, N., and G. Kaiser. "Concurrency Control in Advanced Database Applications," ACM Computing Surveys, vol. 23, no. 3, Sept. 1991, pp. 269-317.
- [3] Batory, D., et al. "GENESIS: An Extensible Database Management System," IEEE Trans. on Software Engineering, vol. 14, no. 11, Nov. 1988.
- [4] Beckman, N., H-P Kriegel, R. Schneider, and B. Seeger. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," in Proc. ACM SIGMOD Intl. Conf. on Management of Data, Atlantic City, New Jersey, May 1990, pp. 322-331.
- [5] Beer, C., P. Bernstein, and N. Goodman. "A Model for Concurrency Control in Nested Transaction Systems," JACM 36, pp. 230-269.
- [6] Bentley, J. "Multidimensional Binary Search Trees Used for Associative Searching," Comm. of ACM, vol. 18, no. 9, 1975.
- [7] Bernstein, P., and N. Goodman. "Multiversion Concurrency Control Theory and Algorithms," ACM Transactions on Database Systems, vol. 8, no. 4, December 1983, pp. 465-483.
- [8] Bernstein, P., V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, Mass. 1987.
- [9] Bertino, E., W. Kim, and J. Garza. "Composite Objects Revisited," in Proc. ACM SIGMOD Intl. Conf. on Management of Data, Portland, Oregon, June 1989.
- [10] Chou, H.T., and W. Kim. "A Unifying Framework for Versions in a CAD Environment," in Proc. Intl Conf. on Very Large Data Bases, August 1986, Kyoto, Japan.
- [11] Chou, H.T., and W. Kim. "Versions and Change Notification in an Object-Oriented Database System," in Proc. Design Automation Conference, June 1988.
- [12] Chrysanthis, P., and K. Ramamritham. "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior," in Proc. ACM SIGMOD Intl. Conf. on Management of Data, May, 1990, pp. 194-203.
- [13] A. El Abbadi, and S. Toueg. "The Group Paradigm for Concurrency Control Protocols," IEEE Trans. on Knowledge and Data Engineering, vol. 1, no. 3, Sept. 1989, pp. 376-386.
- [14] Faloutsos, C., T. Sellis, and N. Roussopoulos. "Analysis of Object-Oriented Spatial Access Methods," in Proc. ACM SIGMOD Intl. Conf. on Management of Data, San Francisco, Calif., May 1987, pp. 426-439.
- [15] Fernandez, M., and S. Zdonik. "Transaction Groups: A Model for Controlling Cooperative Work," in Proc. 3rd. Intl. Workshop on Persistent Object Systems: Their Design, Implementation, and Use, Jan. 1989, Morgan Kaufman, pp. 128-138.

- [16] Finkel, R., and J. Bentley. "Quadtrees: A Data Structure for Retrieval on Composite Keys," *Acta Informatica*, vol. 4, no. 1, 1974.
- [17] Freeston, M. "The BANG File: A New Kind of Grid File," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, Calif., May 1987, pp. 260-269.
- [18] Freytag, J. "A Rule-Based View of Query Optimization," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 173-180, San Francisco, Calif., 1987.
- [19] Garcia-Molina, H., and K. Salem. "Sagas," in *Proc. SIGMOD Intl. Conf. on Management of Data*, May 1987, pp. 249-259.
- [20] Graefe, G., and D. DeWitt. "The EXODUS Optimizer Generator," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 160-172, San Francisco, Calif., 1987.
- [21] Gray, J.N. *Notes on Data Base Operating Systems*, IBM Research Report: RJ2188, IBM Research, San Jose, Calif. 1978.
- [22] Greene, D. "An Implementation and Performance Analysis of Spatial Data Access Methods," in *Proc. IEEE Intl. Conf. on Data Engineering*, Feb. 1989, Los Angeles, Calif., pp. 606-615.
- [23] Guttman, A. "R-Trees: A Dynamic Index Structure for Spatial Searching," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Mass., June 1984, pp. 47-57.
- [24] Hoel, E., and H. Samet. "A Qualitative Comparison of Data Structures for Large Line Segment Databases" in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, Calif. June 1992.
- [25] Hornick, M., and S. Zdonik. "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. on Office Information Systems*, vol. 5, no. 1, Jan. 1987, pp. 70-95.
- [26] *IEEE Data Engineering bulletin special issue on Unconventional Transaction Management* (ed. Ahmed Elmagarmid), March, 1991.
- [27] Kamel, I., and C. Faloutsos. "Parallel R-Trees," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, California, June 1992.
- [28] Katz, R., and S. Weiss. "Design Transaction Management," in *Proc. 21st Design Automation Conference*, Albuquerque, New Mexico, June 1984.
- [29] Katz, R. "Towards a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990, pp.375-408.
- [30] Kim, W., D. McNabb, R. Lorie, and W. Plouffe. "A Transaction Mechanism for Engineering Design Databases," in *Proc. Intl. Conf. on Very Large Databases*, 1984, Singapore.
- [31] Kim, W., and H. Chou. "Versions of Schema for Object-Oriented Databases," in *Proc. Intl. Conf. on Very Large Data Bases*, Long Beach, Calif., Sept. 1988.
- [32] Kim, W., N. Ballou, J. Garza, D. Woelk. "A Distributed Object-Oriented Database System Supporting Shared and Private Databases ," *ACM Trans. on Office Information Systems*, Jan. 1991, pp. 31-51.
- [33] Kim, W., J. Garza, A. Keskin. "Spatial Data Management: Research Directions," in *Proc. Intl. Symposium on Large Spatial Databases*, June 1993, Singapore.
- [34] Klahold, P. G. Schlageter, R. Unland, and W. Wilkes. "A Transaction Model Supporting Complex Applications in Integrated Information Systems ," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, May 1985, pp. 388-401.

- [35] Korth, H., W. Kim, and F. Bancilhon. "On Long-Duration CAD Transactions," *Information Science* in Oct. 1988.
- [36] Korth, H., and G. Speegle. "Formal Model of Correctness Without Serializability," in *Proc. SIGMOD Intl. Conf. on Management of Data*, June 1988, pp. 379-386.
- [37] Moss, E. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.
- [38] Nievergelt, J., H. Hinterberger, and K. Sevcik. "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. on Database Systems*, vol. 9, no. 1, March 1984.
- [39] Ooi, B-C. *Efficient Query Processing in Geographic Information Systems*, *Lecture Notes in Computer Science*, Springer-Verlag, 1990.
- [40] Robinson, J. "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Ann Arbor, Michigan, April 1981.
- [41] Roussopoulos, N., and D. Leifker. "Direct Spatial Search on Pictorial Databases Using Packed R-Trees," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Austin, Texas, May 1985.
- [42] Samet, H. "The Quadtree and Related Hierarchical Data Structures", *ACM Computing Surveys*, vol. 16, no. 2, June 1984.
- [43] Seeger, B., H-P Kriegel. "The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems," in *Proc. Intl Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, pp. 590-601.
- [44] Selinger, P., et. al. "Access Path Selection in a Relational Database Management System," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Mass., May 1979, pp. 23-34.
- [45] Sellis, T., N. Roussopoulos, and C. Faloutsos. "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," in *Proc. Intl Conf. on Very Large Data Bases*, Brighton, England, 1987, pp. 507-518.
- [46] Skarra, A., and S. Zdonik. "Concurrency Control and Object-Oriented Databases," in *Object-Oriented Concepts, Databases, and Applications* (W. Kim and F. Lochovsky, eds.), Addison-Wesley/ACM Press, 1989.
- [47] Snodgrass, R. and I. Ahn. "Temporal Databases," *IEEE Computer*, vol. 19, no. 9, Sept. 1986, pp. 35-42.
- [48] Snodgrass, R. "The Temporal Query Language TQuel," *ACM Trans. on Database Systems*, vol. 12, no. 2, June 1987, pp. 247-298.
- [49] Woelk, D., and W. Kim. "Multimedia Information Management in an Object-Oriented Database System," in *Proc. Intl. Conf. on Very Large Data Bases*, Brighton, England, Sept. 1987, pp. 319-329.