W. Cellary, K. Vidyasankar ,
G. Vossen (editors):

Versioning in Database Management Systems

Dagstuhl-Seminar-Report; 55
01.02.-05.02.93 (9305)

ISSN 0940-1121

Versioning in Database Management Systems

Organizers: W. Cellary, K. Vidyasankar, G. Vossen

1 Introduction

Versioning of data refers to the ability of a database management system to create, organize, manage and maintain distinct *versions* of individual data items or objects over certain periods of time. In traditional database systems, versioning can be employed to improve the efficiency of transaction synchronization through the provision of multi-version concurrency control algorithms. While traditional systems emphasize the view of keeping versioning *transparent* from a user's point of view, i.e., having a user ignore that data objects may exist in multiple versions in the database, more recent applications of database systems have among their requirements that the versioning mechanism of the system has to be *apparent* to users. For example, *temporal* databases, which organize data across time, need to keep track of how data objects change over time (e.g., a person's employment history) and to answer historical queries (e.g., "what was a person's position ten years ago"). Moreover, in *design environments* as given in a computer-aided design (CAD) or software engineering (CASE) context, users need to experiment or simply work with different versions of the same object, in order to explore distinct evolution paths of the object under design, or to accommodate multiple composition requirements. Here, the situation is further complicated by the fact that design objects are typically of complex structure, with references to many subobjects each of which can itself be multi-versioned. In addition, these versions typically even represent objects which are not yet completed, and which are used by a group of cooperating designers. To support such a cooperation, versioning must be complemented by mechanisms to create configurations of potential final design objects.

Since especially in non-traditional applications of databases a versioning facility has been recognized as important and must be provided at the user's level of abstraction, a number of different aspects of versioning has been investigated over the past ten years. In particular, the questions that have been addressed include

1. *version models* which provide data structuring concepts for organizing versions into derivation histories, for composing complex objects from versions of their component objects (i.e., for forming configurations), and for tracking equivalent versions across multiple representations of the underlying object,

2. *operational issues* including inheritance and derivation, i.e., mechanisms through which new versions can be obtained from existing ones, change notification and propagation describing how complex interconnected design structures respond to changes, and workspace models, i.e., mechanisms through which new versions become visible to and sharable by a designer community,

3. *organizational frameworks for group cooperation*, which add an appropriate layer of abstraction to a versioned database,

4. *language concepts* through which users can conveniently and adequately employ an apparent versioning mechanism,

5. *implementation aspects* including storage structures for versions and their histories, transaction processing in the presence of versions and/or workspaces (and distributed system architectures), consistency preservation for configurations and multiple representations.

The work that has been done on versioning in databases in recent years shows that a number of issues is still not well-understood, although commercial products supporting versioning are already available in the marketplace. For example, the perception that a versioning facility should be orthogonal to both object composition and concurrency control has only recently been advocated. By the same token, a unification or even standardization of approaches is rarely in sight. Finally, the proper meaning, implications of and requirements to versioning in distinct application domains (including, CAD, CASE, office information systems, or multimedia systems) are neither fully understood nor exhaustively explored.

Given this situation, it was the goal of this workshop to bring together, to the organizers' knowledge for the first time separate from a major database conference or workshop, people from the international research community who have made important contributions to versioning and/or are still actively working on this subject. Furthermore, it was our intention to identify commonalities among the variety of proposals that have been made in the past, and to isolate demanding open problems in this field.

The meeting brought together 28 scientists from the eight countries of Brazil, Canada, Croatia, France, Germany, Italy, Poland, and USA (see last section). During the week, 20 presentations spanning a wide range of issues were given in a dynamically composed program, with plenty of discussion time after each; in addition, an open discussion was held on Thursday afternoon and evening.

We felt that all participants enjoyed the workshop, and we wish to thank the Dagstuhl staff for ensuring that everything ran so smoothly. The Dagstuhl office in particular provided financial support for the participants from the Central European countries, thereby enabling them to travel to the castle and to attend the workshop, which is gratefully acknowledged. Special thanks go to the kitchen staff for keeping us stuffed with good food all the time, and to Melanie Spang for her patience in handling everything that came up during the week.

2 Abstracts of Presentations

The following abstracts of presentations appear in the order the presentations were given; the first four were given on Monday, the next five on Tuesday, the next three on Wednesday, the next five on Thursday, and the last three on Friday.

Database Version Approach

W. Cellary, Franco-Polish School of New Information and Communication Technologies, Poznan, Poland[1]

To maintain consistency, a multiversion database is organized as a set of database versions, each database version containing one version of each object. A database version is a unit of consistency, i.e., the version of an object contained in a database version is consistent with the versions of all other objects contained in the same database version. A database version is also a unit of versioning, i.e., to create a new version of an object, a new database version must be created, where the new object version appears in the context of other object versions, and has to respect consistency constraints. There are two kinds of transactions: database version transactions and object transactions. A database version transaction derives a new database version, called a "child" from a given one, called the "parent". When derived, a child database version is a logical copy of its parent. Then it may evolve independently of its parent. An object transaction is an ordinary transaction known from the monoversion databases. It is addressed to one or several database versions. It queries and updates object versions contained in a database version in such a way that the database version remains consistent when the transaction commits. To avoid redundancy, object versions may be shared by several database versions. To obtain high performance of database management a special addressing is used based on "version stamps". Version stamps permit to deduce which database versions share a given object version without storing explicit links among them. As a consequence, creation of a new database version does not require any modification of control data storing information on object version association with database versions.

[1]Joint work with G. Jomier, University Paris-Dauphine, France.

The Constellation Approach for Managing Multi–Version Objects in CAD Databases

G. Vossen, University of Giessen, Germany

One of the key issues in integrating database technology into design environments and processes is finding the appropriate logical organization of databases at the application-interface level, since this has to account for requirements like versioning, consistency, and design transactions. This talk described a novel approach to the organization, manipulation and management of (versioned) design objects in a design database and investigated its use in design processes. This approach is based on the fundamental perception that valuable database support for applications like *CAD* must be based on the specific way in which designers carry out projects, and that a database can only play the role of a tool which is easy to use. To this end, it must be reflected that design objects are composite and exist in multiple versions, a design process is evolutionary and cooperative, with many people of distinct levels of expertise and competence involved, and there is no completely predefined notion of "database consistency".

To account for these requirements, we proposed a new organizational framework for design databases based on *constellations*. Informally, a constellation is a set of multiversion design objects together with all their subobjects; it abstracts from physically existing object versions and just considers composition. Constellations can be versioned into *configurations*, where a configuration — as a "snapshot" of some part of an artifact under design — is comprised of one version of each object of the underlying constellation; thus, configurations state which versions of distinct objects make sense together. The constellation approach has several advantages: First, it can provide a natural reflection of a design process: since constellations are considered the units of allocation to designers, they account for dependencies between designers as well as parallel work of designers. Second, versioning of objects is orthogonal to object composition, which renders efficient manipulation and management of design objects possible. Third, it provides an adequate notion of consistency for a design database. This is due to the fact that only configurations are units of consistency in our approach, so that consistency becomes context-dependent.

The clear logical decomposition of a design database into well-defined units of consistency also has an important impact on transaction processing. Indeed, a flexible support of design processes becomes possible by using specific types of transactions whose scope is limited to well-defined portions of the database. Transactions can either operate on object versions, on configurations, or on constellations. Since consistency is confined to configurations, transactions operating on object versions within the context of a configuration are transactions in the traditional sense. They are, in a sense, orthogonal to transactions manipulating configurations and constellations. Finally, the approach is object-oriented: Design objects have an identity and a value, where the latter may contain references to other objects. Similarly, constellations and configurations have unique identities, and their "values" have to satisfy a number of conditions. However, since no specific assumptions are made on the object model used, our approach can be adopted by an arbitrary object management system.

A full version of the paper on which this talk was based appeared as Technical Report No. 9105, University of Giessen; it was written together with W. Cellary and G. Jomier, and is available from the author.

Version Consistency and Serializability in Design Databases

K. Vidyasankar, Memorial University of Newfoundland, Canada[2]

A database used by engineering or manufacturing applications for analysis and design purposes is called a *design database* . Here data items represent the actual physical structure of engineering parts and components. A design process can be thought of as starting with certain *base* data items or design objects, and deriving others from them. Several alternate designs may be tried from the same set of base objects. Designs with several sets of base objects may also be tried. Hence several versions of various data items may exist in the system simultaneously. Then "consistent" versions must be used in each derivation. Also a concurrency control mechanism for design database systems must allow keeping various versions until the end of the design process.

In this paper we propose a formal definition of version consistency in terms of transaction histories. This allows checking version consistency by constructing a "history" graph. We also propose that a notion called τ_*-serializability be used as the correctness criterion for concurrent executions in design databases. (For business and administrative databases, the commonly accepted criterion is view-serializability.) We then show that τ_*-serializability of a version consistent history can be checked in polynomial time. (With general histories the problem is NP-complete.) For this also a history graph needs to be constructed. Both version consistency and τ_*-serializability can be checked using the same graph.

[2]Joint work with C.N.G. Dampney, School of Mathematics, Physics, Computing and Electronics, Macquarie University, North Ryde, N.S.W. 2113, Australia. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada Individual Operating Grant A-3182 of K. Vidyasankar. This work was presented in ICDT '88, 2nd International Conference on Database Theory, Bruges, Belgium, August/September 1988, Proceedings LNCS 326, Springer-Verlag, pp. 368–382.

Evolution and Versioning of OODB Schemas Through a View Mechanism

E. Bertino, University of Genova, Italy

In this talk, we present a view model for object-oriented databases that extends in various directions view models typical of relational databases. In particular, the definition language allows views to be defined that augment class definitions (by adding attributes and methods), and that support a wide spectrum of schema modifications. Therefore, views represent a unified mechanism able to provide several functions, such as query shorthand, definitions of dynamic sets and partitions of classes, authorizations, schema changes and versions, object perspectives.

Managing the Interaction between Versions and Time in Temporal Object-Oriented Databases

C. Bauzer Medeiros, Universidade Estadual de Campinas, Brazil

Temporal databases allow maintaining information about the evolution of data along time. This evolution is usually considered along two independent time axes: *valid time* – the periods in which the data was valid; and *transaction time* – system-generated time stamps, which indicate when the data was stored. Users can therefore keep track of different "histories" of data, for the same set of data, and at the same time indicate when these were believed to be valid.

Several mechanisms have been proposed to support time in relational databases, but there are still many problems that remain to be solved. The support of time in object oriented databases presents yet further difficulties, e.g., the composition of objects which have different valid time spans, or the possibility of inheritance through time.

These issues can be yet further complicated when one considers applications that require maintenance of versions as well as their temporal evolution. In this case, the DBMS has to provide support for four different dimensions of data evolution: along two time axes (valid and transaction times) and according to two different version generation operations (revisions and variants).

Two solutions were presented to allow simultaneous support of these four dimensions. Both solutions are based on the Multiversion Database Version Model, where the creation of an entity version entails the creation of a logical database version containing this new entity version. If one adopts this model, one can treat a temporal object-oriented database as a set of slices, where each slice is a "snapshot" of the database at a given transaction time. One solution entails considering each slice as a valid-time database with versions. The other solution sees each slice as containing a set of snapshot databases with versions. In the latter case, the slice is again partitioned in valid-time slices (the snapshot databases), where all elements have the same validity properties.

CLOOD: A Class-less Model for Object-Oriented Design Databases

M. Gross-Hardt, University of Giessen, Germany

Recently, it has been argued that traditional models for object-oriented databases are not adequate in many of those applications which originally triggered the development of this novel databases technology. In order for a data model to be useful in design applications like CAD or CASE, it must reflect and support the specific database requirements of these domains. In this talk, a specialized model for object-oriented databases was presented, which overcomes the limitations of previous models especially for technical applications. Its major features include that it is not based on the notion of class, and that a versioning mechanism is directly built into the model. No claim is made that this model can serve as a replacement for other object-oriented models, but it seems better suited for design applications than previous "general-purpose" models. The basic idea underlying this model is to have objects freely exist in a database. Objects still have a structure and a behavior, but can additionally be versioned. Thus, each object in our model has an identifier, an associated set of versions, and a set of messages it understands. Versions in turn have a number and a value, where values are elements of domains that are described by types. Objects can be composed of other objects, and can interact.

Clearly, a declarative way of querying and updating a database of objects is desirable. To this end, the general notion of a *collection* was introduced. Basically, collections are arbitrary sets of objects and versions; however, it seems useful to identify a number of collections which have specific properties. For example, *configurations* are consistent sets of versions, or *classes* are collections in which each object has the same structure and behavior. Thus, a database in our model can be grouped into a variety of application-dependent collections, which can then be subject to user operations.

A full version of the paper on which this talk was based appeared as Technical Report No. 9208, University of Giessen; it was written together with Gottfried Vossen, and is available from the author.

Environment to Develop and Maintain Large Knowledge Bases

S. Lanka, Penn State University, USA

We describe an environment that facilitates concurrent development of a large knowledge base in a collaborative manner. The updates performed by a knowledge engineer can be thought of as a long duration transaction. A version-oriented concurrency control mechanism is chosen to alleviate problems that arise due to a long duration transaction. An object store is designed to efficiently manage multiple versions of a knowledge base. We show that the cost of maintaining multiple versions is within a small constant factor of maintaining a single version. The object store is implemented using Symbolics' Statice object-oriented DBMS.

The multiple versions of a knowledge base thus created are merged to form a single unified version of the knowledge base that is consistent. To compute a merged version that is consistent is NP-complete. We then formulate the computation of a consistent merged version as an integer programming problem.

Supporting Engineering Applications by PRIMA — Versioning Concepts and Architectural Issues

W. Käfer, IBM Almaden, USA[3]

Engineering applications are characterized by a set of interrelated tools that perform complex transformations on highly interconnected data. So-called CAD frameworks, aimed to support engineering applications, focus on methods for organizing shared data repositories, organizing the design tools, their executions, the transfer of data among them, and the way they access data. Obviously, data organization and data processing techniques play a key role in such an environment.

In this talk, we present an architecture based on a complex-object database management system, such as PRIMA, suiting these requirements. We briefly discuss the overall architecture before we focus on the data management components. PRIMA supports the needs of engineering applications with respect to data organization and data management by a powerful data model which provides

- versioned complex objects (COs)

The COs offered by PRIMA are able to represent heterogeneous, network-like sets of elementary objects (EOs). EOs are similar to tuples known from the area of

[3]Joint work with H. Schöning, Univ. Kaiserslautern, FRG.

relational database systems. However, special structural relationships (or links) are used to set up nets of EOs, the so-called COs. The COs may be versioned, each version representing a different net of EOs. The versions of one CO are organized in a version derivation graph. COs and versions of different COs may be linked together by version links or object links, respectively.

- a descriptive SQL-like query language

The query language handles sets of heterogeneous COs linked together via object or version links, respectively. The result set of such a query consists of COs or versions of COs and their constituting EOs along with their structural links.

- a tailored processing model

Tools used in engineering applications expose a so-called load-work-store behavior. Using the descriptive query language a set of COs (along with their constituting nets of EOs) is extracted from the data base and loaded into an object buffer. There, the data can be processed using a main-memory pointer-based hierarchical cursor interface and new versions of the data may be derived. After this work phase, the data is written to the database.

Obviously, efficient extraction of COs and their components as well as query optimization are critical tasks. Instead of implementing the described data model from scratch we have chosen to implement it on top of an existing complex-object data model, which does not provide version support. We demonstrate that this approach is beneficial in contrast to similar approaches based on relational database systems. The key to success is the ability of the underlying complex-object data model to handle overlapping nets of objects efficiently.

Supporting Cooperative Design in a DBMS-based Design Environment

N. Ritter, University of Kaiserslautern, FRG

Facing the growing complexity of technical products, the process of design is typically carried out by a team of cooperating designers rather than by a single person. Several methodologies have been developed in order to structure the overall design process and to support designers working on partial design problems and cooperating with each other, e.g., by negotiating their individual design goals or by exchanging their partial results. However, today's CAD systems typically do not support cooperative work in a satisfactory manner. Exchange of preliminary results is usually done without system control and, therefore, also without system support. In larger designer teams this causes inconsistencies of design objects which must be hand-resolved with a considerable overhead. For that reason a processing model is needed that allows to control the inherent dynamics of design. It should be rich enough to reflect the major characteristics of a design process, e.g., goal orientation, hierarchical decomposition, stepwise refinement as well as team-orientation and cooperation. We propose a model of design process dynamics which distinguishes different levels of abstraction. At a higher level, the administrative part of design work has to focus on the description and ordering of design tasks and a controlled communication of (preliminary) design states. On the subordinated level the organization of the actions performed in order to fulfill a certain design task or sub-task is the subject of consideration. Different design states are captured by means of a version model and caused by tool executions. The derivation of a design object version is supported by the concepts of a third level of abstraction. This conceptualization provides several types of operational units serving for the structuring of the design process. The operational units can be considered as spheres of control, similar to transactions. In our contribution the different types of operational units are detailed. The discussion mainly focuses on their properties and assertions which allow to control the design process dynamics without loosing the flexibility of applying different design methodologies.

Graph Modeling in CASE-Applications

J. Ebert, University of Koblenz-Landau, FRG

Graphs are simultaneously expressive pictures, formal models, and efficient data structures. TGraphs, i.e., typed, attributed, and ordered directed graphs, are used as the data model for documents inside the KOGGE-CASE-Environments.

Classes of TGraphs are defined by entity-relationship-diagrams extended by additional integrity constraints in the graph specification language GRAL. These definitions directly correspond to Z-schemata. The E-R-diagrams describe the type structure and the attributes. GRAL is a Z-like language, which allows to formulate restrictions on TGraphs in such a way, that efficient procedures for testing these restrictions may generated.

In this talk, TGraphs are used on a higher level of granularity. A graph class is given which models whole documents as vertices and their relationships as arcs. Let the system graph be the subgraph induced by the set of `dependsOn`- and `hasAsVersion`-arcs. Then, a formal definition delivers *versions* as two-vertex subgraphs which are contained in *configurations* modeled as and/or-subgraphs. These again are contained in *constellations* which are progressively closed subgraphs of the system graph.

UNDO in CAD Systems

M. Rosendahl, University of Koblenz-Landau, FRG

The user of a CAD system wants the possibility to undo the last operation, for instance if the result of a operation is not what the user expected. The operation undo has to change the database to the status it had before the operation. Because a mistake may be seen not before more than one operation is finished, the undo operation should be evaluated for more than one operation. Also a redo feature should be possible. If after an undo operation a new different operation than before is performed, the history of the system states becomes a tree. The user should be able to navigate in this tree.

To achieve this, the difference between two states is held in a log file. After moving from state S_i to S_{i+1} the information $\Delta(i, i+1)$ is written. Then there is the function $\mathrm{UNDO}(S_{i+1}, \Delta(i, i+1)) \rightarrow (S_i, \Delta(i+1, i))$. $\Delta(i+1, i)$ is the information to compute state S_i from S_{i+1} and is needed by a later redo operation. For system performance two goals have to be achieved:

- Minimize the space for $\Delta(i, j)$,
- minimize the time to compute $\Delta(i, j)$.

The time for the UNDO operation is less critical, because this operation has not to be performed after each operation.

The difficulty with these goals is that in CAD systems one operation can change the values in nearly all entities of the database, e.g., a move of the total model. So as far as possible the performed operation is held in the delta information. Only if the operation cannot be inverted, then old data or the data differences are recorded. If the delete operation will only set a tag in the data, delete and undelete operations must only record the items to be (un-)deleted. User-defined operations are either broken down into the basic operation or the data changes are recorded. Additional problems arrive on systems where the data structure of the system is an arbitrary graph and not only a collection, because then the structure of the graph can be changed by an operation, e.g., building a segment or resolving a segment into its elements.

A step further is a solution to the problem to insert a operation after some undo operations and then redo the original operations from the state after the inserted operation. There are situation when this can only be done with user interaction.

Representing and Retrieving Information in a Multiversion Database (MVDB)

S. Gançarski, Universite Paris Dauphine, France

Most new database applications, nowadays, need tools that will help identifying and managing data versions. We discuss the issues involved in representing and retrieving versioned data in Multiversion Databases, particularly when using the Database Versions Approach [CJ90]. In this approach, a Multiversion Database is seen as a set of Database Versions (DBV). Each DBV contains exactly one version per object, and represents a state of the real world modelled by the Database. On a logical point of view, DBV are independent with each other and can be updated without any side-effect on other DBV. However, on a physical point of view, several DBV can share the same object version, to avoid redundancy, .

At the interface level (e.g., CAD application), the user may want to work within one DBV, or he may want to work on a part of the MVDB. The first case is quite similar to working in a monoversion Database. The second case is more specific. Indeed, each DBV has its particular meaning, which is related to two kinds of information: information about DBV and links between DBV. The first kind of information gathers both DBV content (object versions that are contained in the DBV) and DBV characteristics (e.g., the DBV status). The second kind of information concerns application semantics (e.g., derived_from link) and must be stored additionally to information about DBV. Usually, those links partially order the configurations set, and the MVDB can be seen as a multigraph, whose nodes are the configurations. In order to allow information retrieval in the MVDB, our first attempt will be to provide the user with new functionalities for retrieving interesting DBV in the MVDB multigraph. In a first step, we consider a simple, particular case: the MVDB multigraph is reduced to a DAG with one links type, called the "DBV users Graph".

The main problem is that the DBV users Graph can be very large. Therefore, in most cases, it is impossible for a system to present it to the user as a whole. Thus, tools must be developed to provide the user a way to define "views" on it. One such tool would be a Multiversion Database Graph Query Language. This language should enable graph view definition, using every kind of information contained in the MVDB. Graph views may be synthesized, by reducing the number of represented DBVs, and by directly representing transitive relationships instead of successions of direct relationships. Furthermore, this language should allow the user to work in an incremental way, navigating through different parts of a graph.

Related work can be divided into two main parts: Multiversion Query Languages and Graph Query Languages (GQL). Multiversion Query Languages are, in most cases, extensions of object-oriented query languages. In some cases [Kaf92], the only extension is that selection criteria can concern object version's values or ranking number. [Bjo89] proposes primitives that allow navigation through version graphs. The main restriction with such languages resides in that queries only return either object versions or configurations, i.e., it is not possible to select particular links between configurations. Graph Query Languages are better adapted to define Graph Views. For example, in Graphlog [Con89], it is possible to define "virtual links", which are computed but not stored. Actually, virtual links are a restricted form of Graph Views, but the logical approach of such a language implies efficiency problems.

We formalize the notion of Graph View, and present our first attempt to define an algebraïc Multiversion Graph Query Language. We present primitive functions and function constructors that allow the definition of more sophisticated functions or queries. In future work, we will implement this language on top of the Database Version Manager [CKW90], and we will extend it to generalize the notion of view on the MVDB.

References:

[Bjo89] Björnerstedt A., Hultén C. *Version Control in an Object-Oriented Architecture. Section 3 : Object versions at the application level* Object-Oriented Concepts, Databases, and Applications (Kim W., Lochovsky F. Editors), chapter 18 (p. 451).

[CKW90] Cellary W., Koszlajda T., Wieczerzycki W. *Database Version Manager: Prototype* Technical University of Poznań, July 1990.

[CJ90] Cellary W., Jomier G. *Consistency of Versions in Object-Oriented Databases.* VLDB 90, Brisbane (Australia).

[Con89] Consens M., Mendelzon A. *Expressing Structural Hypertext Queries in GraphLog* Hypertext '89, Tempe (Arizona), November 89.

[Kaf92] Käfer W., Schöning H. *Mapping a Version Model to a Complex-Object Data Model.* IEEE Data Engineering , Tempe (Arizona), February 92.

Configuration Management for the Integration, the Control, the Maintenance and the Installation of Hardware/Software Systems[4]

M.J. Blin, Universite Paris Dauphine, France

To be adapted to specific environments, the different elements — entities — of systems are developed in several versions. Thus, systems exist in several versions. When the development is finished, different users are concerned with a developed system: the integration team which will verify that a version of the system is well integrated with the operating environment, the certification team which will submit the versions of the system to certification tests like performance or ergonomic tests, the installation team which will install the versions of the system on sites, the maintenance team which will create new versions of the system. Entities are, the most time, of different nature: programs, manuals, test data, etc., and may be complex. References between entities may have different meanings like: "tested by" or "may be replaced by"... All the users are not necessarily interested by all the entities composing the system. A delivery configuration manager has to supply users with consistent sets of entities.

We propose a new model of configurations based on the Database Version Model. Our solution provides efficient solutions to the raised problems and allows operations on entity versions and on configurations like finding all configurations containing specific entity versions, creating a new configuration containing new entity versions from each configuration specified by a rule, comparing or merging configurations.

Versions in the Context of Object-Oriented Databases

L. Goldstein Golendziner, Universidade Federal do Rio Grande do Sul, Brazil

Object-Oriented Database Systems emerged as a consequence of the requirements imposed by the new computer applications like CAD, CASE and Office Automation. The principles on which the object-oriented paradigm are based are: objects with unique identification, encapsulation, types or classes, methods, and class hierarchies, with the inheritance mechanism.

Versions represent design steps or alternatives of primitive or structured objects and there have been several works trying to understand and integrate the version concept with object-oriented models.

The talk focuses on three aspects that have to be considered when trying to integrate versions and object-oriented models. The first is the concept of generic object, similar to that defined in the ORION and Iris systems. A generic

[4]Joint research with G. Jomier Lamsade/Paris-Dauphine University

object gathers several versions of one object and it partitions the set of versions of a class.. The second aspect refers to how objects (and versions) are related in a class hierarchy, taking inheritance into consideration. Versions of objects can exist in more than one class of the hierarchy and a correspondence must be defined among them. The third aspect discussed is a discipline for version creation. The versioning process can be constrained, based on the structure of the versions. Rules can be defined to restrict the changes allowable to versions.

Concurrency Control in the Database Version Approach

W. Cellary, Franco-Polish School of New Information and Communication Technologies, Poznan, Poland[5]

The problem of consistency of object-oriented databases is considered, when objects contained in the database are multiversioned, and they are accessed by concurrent transactions. In such databases there are two aspects of the consistency problem, one related to concurrency, as in monoversion databases, and the other related to versioning. In this paper the problems of mutual dependencies of version management upon concurrency control are studied. It is shown that in different approaches developed up to date in different application areas, these aspects of consistency are not separated. As a result, a trade off is assumed between versioning functionalities provided and concurrency degree allowed. It is shown that in the database version approach object version management is orthogonal to concurrency control. As a result, there is no more need for the trade off: versioning functionalities provided are rich, and concurrency degree is high. In the paper the version manager and concurrency controller for a multiversion object-oriented database are described in detail.

Database Versions and Long Duration Transactions

G. Jomier, Universite Paris Dauphine, France

Long duration transactions appear in design applications. In those applications there is also a need for people to work in cooperation and to use versions to be able to follow the history of the project, to try options etc. Propositions have been made to solve the problem of concurrent work in that context. They include schemas of long duration transactions (trees or sets of sub-transactions, nested transactions, cooperative transactions etc.), intermediate save-points or recovery points, classical mechanisms of concurrency control with long and short

[5]Joint Work with G. Jomier, University Paris-Dauphine, France

duration locks, check-out and check-in protocols, management of versions and configurations.

We propose a new approach of the long duration transactions problem using the Database Version (DBV) model. This model allows the storage and the manipulation of as many database versions as necessary for the users, each database version storing a state of the universe modelled by the database. These database versions are identified and may be qualified and associated to authorizations. Tools are provided to help users to navigate in the set of database versions, to compare the content of any two database versions and to merge two database versions. Moreover, when several users want to work simultaneously on the same database version D they may work concurrently on D using classical concurrency control mechanisms or they may work on different database versions derived from D; then, if necessary, they will merge their results later.

Long duration transactions may be seen as sets of subtransactions. They begin in one DBV and end in one DBV. Intermediate states of the database corresponding to the end of sub-transactions may be temporally stored in database versions and used as save points and visibility points by users, according to associated qualifications and authorizations. At the commitment of the long duration transaction these particular database versions disappear, except for the other simultaneous long duration transactions which would have used them as intermediate states. Using DBVs in that way, long duration transactions are generalizations of classical transactions taking into account the fact that the multiversion database stores as many states of the real world as necessary, and may generate new database versions at will.

On the Correctness of Concurrency Control for Multiversion Database Systems with Limited Number of Versions

T. Morzy, Technical University of Poznan, Poland

In the talk the concurrency control problem for multiversion databases with a system-imposed upper bound on the total number of data-item versions stored in a database is addressed. We call such systems *KV database systems*. The main practical reason for studying this problem is that in practice multiversion databases with an unlimited number of versions do not appear, because of the limited storage space available. On the other hand, KV database systems become more and more popular in practice (e.g., RDBMS ORACLE or DEC Rdb), but the behavior of these databases is not yet well-understood. KV databases add a new aspect to multiversion concurrency control, namely the overwriting of versions. Transactions issue write operations, and the system must decide whether the versions produced by these operations are added to the database or old versions stored in the database get overwritten. In the second case, the system must decide which stored versions to overwrite, in order to ensure the correctness of concurrency control and performance. In the talk, the extended multiversion concurrency control to account for the version overwriting aspect of KV databases is presented. Three correctness criteria for transaction schedules are presented, and formal tools for verifying these criteria are given. The hierarchy of schedules corresponding to the hierarchy of concurrency degrees achieved for particular correctness criteria is given.

Stamp Locking Method for Multiversion Composite Objects

W. Wieczerzycki, Franco-Polish School of New Information and Communication Technologies, Poznan, Poland

Multiversion objects require proper management. It is particularly hard in the case of composite objects in a multiuser environment. The main problem is how to maintain database consistency, which is related to two aspects: concurrency and versioning. The concurrency aspect follows from concurrent transaction processing, while the versioning aspect concerns the identification of versions of different objects which go together. In the paper the concurrency aspect of database consistency is considered.

In classical databases, to preserve database consistency during concurrent transaction execution the hierarchical locking method is commonly used. This method, however, may not be directly applied in the multiversion object-oriented databases (OODBs). It follows from the fact that it concerns a single hierarchy, namely the granularity hierarchy, which is usually composed of three levels only. On the contrary, in multiversion OODBs at least four hierarchies may be distinguished: granularity, inheritance, composition and version derivation. The latter three are usually composed of much more levels than the classical granularity hierarchy. According to the concept of hierarchical locking, all the hierarchies mentioned above should be taken into account by the locking method in a uniform way. Because of the complexity of OODBs hierarchies, any attempt to adapt the classical method leads to a great number of intentional locks and complex locking strategy. The reason is that intentional locks have to be independently set in all the hierarchies, on all the nodes which are predecessors of the accessed nodes.

In the paper a stamp locking approach was proposed, which offers simple and efficient locking strategy for all the hierarchies of multiversion OODBs. In this approach, so called "stamp locks" are used, instead of intentional locks. A stamp lock is defined as an extension of a classical lock in such a way that it contains the information about the position of nodes concerned in all the hierarchies. Thus, to lock related nodes or subtrees of nodes of all or some of the hierarchies it is sufficient to set a single stamp lock only. The stamp locking approach may be easily applied to different versioning models and different object composition models. It also may be extended for directed acyclic graphs.

Versioned B^+-Trees

S. Lanka, Penn State University, USA

In this talk we will describe a versioned B^+-Tree index mechanism. In a versioned B^+-Tree the effects of an update from a transaction does not obliterate the effects of updates from previous transactions. The execution of a transaction creates a new version of the B^+-Tree. In addition, the other versions of the B^+-Tree created from the previous transactions also persist. A versioned B^+-Tree index mechanism is useful in databases that support – design activity and large software development projects. A versioned B^+-Tree is implemented on top of the B^+-Tree implementation from UC Berkeley. Our experimental results demonstrate the viability of a versioned B^+-Tree. We constructed a versioned B^+-Tree with up to 700 versions and a million keys, thus exhibiting the scalability of the versioned B^+-Tree implementation.

Handling Versions of Different Quality in Real-Time Databases

A. Buchmann, University of Darmstadt, FRG

Real-time databases include timely execution of transactions as part of the correctness criterion. When overload situations occur it may be preferable to execute a transaction that produces results of lower quality rather than missing the deadline. We call this substitution of cheaper but lower quality tasks a *contingency plan*. From an analysis of applications we have identified four possible trade-offs between precision, currency, completeness or consistency vs. timeliness. When transactions write data of lower quality back into the database during an overload situation we obtain versions of lower quality. Special mechanisms are required for masking data of lower quality and for eventually repairing the database after the overload situation is over. This process is a variant of the version merging problem. We discussed the kind of masks required and some criteria under which merging of the various versions is possible.

3 Specific Discussion Topics

The organizers considered it important to get discussions between the participants started early on, and to provide enough opportunities to keep them going. Two things were done in this respect. First, every participant was asked to pose a question he or she considered most important for the week of the workshop. Second, on Thursday afternoon and evening, an open discussion was held during which people could express what they felt they had learned, or what they considered the most demanding open problems.

The opening session was started with an introduction by K. Vidyasankar, who gave a decomposition of the field of versioning according to the organizers' perception, and asked everybody to place his or her forthcoming contribution or interests into the appropriate location of that scheme. The decomposition looked as follows:

- Versioning
  - apparent (i.e., visible to end-users)
    - models
    - applications
    - implementations issues
    - other issues
      - cooperation
      - temporal aspects
      - object orientation
  - transparent (i.e., not visible to end-users)
    - concurrency control
    - recovery
    - other issues

The questions that were then raised by participants were the following (in the order they were posed):

1. Do we need conceptually distinct approaches for dealing with internal and external versioning, or are these just two facets of the same problem, and is the object-oriented paradigm powerful enough to cover versioning as just one issue (i.e., can versioning be captured completely by an object-oriented model)? (Vossen)

2. Which requirements to versioning can be identified from an application point of view? (Cellary)

3. How can versions be used in schema updates for object-oriented databases? (Ferrandina)

4. What is the relationship between version models for computer-supported cooperative work (CSCW), complex objects and non-ACID transactions? (Härder)

5. What techniques can be used for redo and undo in the presence of versioning in CAD systems, and what restrictions on data structures are needed for efficient implementations? (Rosendahl)

6. What can database people help CASE people for keeping large configuration graphs, and how does the underlying model compare to database models? (Ebert)

7. What requirements are needed for design applications and their integrity constraints? (Gross-Hardt)

8. Do we need an appropriate versioning for CSCW, and what is the relationship between versioning and extended transaction models for CSCW? (Ritter)

9. What is the relationship between a version model and a complex-object model, and how do they fit into a cooperation model? (Käfer)

10. How general are version models for CAD applications, in particular VLSI design, (or how general can they be) with respect to other areas than those for which they were developed? (Schöning)

11. What are application requirements to version models, and what impact do schema updates have on applications? (Bertino)

12. What are the database systems fitting CAD applications, what features need to be added to OODBS to satisfy CAD, and what should be left to the application level? (Goldstein Golendziner)

13. What are the fundamental questions with respect to serializability in multi-version systems? (Morzy)

14. What are the requirements for a specialized language for user-defined versioning concepts? (Brosda)

15. What are the implementation issues for OODBS which use versioning for schema updates? (Madec)

16. Is versioning really going to matter when we are looking at throughput from a concurrency control point of view? (Lanka)

17. Are there any efficient mechanisms to efficiently keep track of coherent configurations, and how can users be given nice views of their configurations? (Bauzer Medeiros)

18. How do changes to a schema (in the type hierarchy) of an OODBS influence the versions? (Even)

19. Is it possible to adapt classical concurrency control to versioning in an efficient way? (Wieczerzycki)

20. What are the specific issues of information retrieval in multi-version systems compared to mono-version systems? (Gançarski)

21. What are the differences in versioning between CAD and CASE? (Rykowski)

22. What are the language issues for long-duration transactions? (Jomier)

The following is a brief account of issues raised during the Thursday discussions: First, the notion of *consistency* was discussed in some detail, with the following points being raised:

- Database management systems are among the only true multi-user systems running on a computer, and if they offer versioning, they contribute to solving the consistency problem for databases (Cellary).
- However, this is not true for CASE, since there consistency, i.e., type consistency, is well-defined, there is a concept for this (Ebert).
- It seems unreasonable to look at consistency as a binary thing; more appropriate would be to try to define a layered notion of consistency, and to separate consistency from "integrity." Also, design is more structured than most people think; in particular, a schema can always be designed ("instantiate predefined parameters"). Since re-use is a big portion of any design activity, constraint specification and verification mechanisms are needed. (Buchmann)

Second, there was a major discussion on the operation of *merging*, in particular merging versions or configurations.

- The really crucial operation in design databases is the step of merging; therefore, this needs more much emphasis in people's considerations. The problem is how to *reduce* the number of versions in a database, not how to increase them. What is needed for merging, as for transactions, is some hierarchical structure from the objects operated upon. For merging, we also need to be very clear about consistency; in particular, we need to distinguish "completeness" from "value consistency" etc. (Buchmann)

- The main problem is that of merging, and to define conditions under which a merge can be performed. (Morzy)
- A definition of "merge" cannot be given without appropriate preconditions. (Schöning)
- A merge operation, which is semantic in nature, cannot be defined just via the syntactic notion of a transaction. (Cellary)
- Another option would be to decrease the granularity of objects to small things which can be combined (merged) by the user; this can be done, for instance, in software configuration management. (Ebert)
- One constraint imposed on merging by cooperation is to "merge carefully." (Bauzer Medeiros)

The discussion was concluded by a few statements about what people had learned, for example:

- Database systems can *contribute* to solving the problems of advanced applications, but cannot solve them. (Cellary)
- Dealing with versions is a complex problem, but nevertheless we would like to have a solution. What we have seen (also here) is that people restrict the problem; then it can be experimented with techniques etc. The result is a partial solution only, but it helps. (Vidyasankar)
- Database and CASE problems are not as distinct as anticipated. In CA-SE, concurrency is not a big issue. Database people deal with object-orientation, as do people in programming languages, operating systems, software engineering, and artificial intelligence. Removing the borders would be nice and is desirable. (Ebert)
- Encapsulation is in chips! Computer science has not yet seen another such revolution as chips; object-orientation might bring that along. (Cellary, Rosendahl)

4 List of Participants

1. Claudia Bauzer Medeiros
   Departamento de Ciencia
   da Computacao
   Instituto de Matematica,
   Estatistica e Ciencia
   da Computacao
   Universidade Estadual
   de Campinas
   Caixa Postal 6065
   13081-970 Campinas SP, Brazil
   cmbm@dcc.unicamp.br

2. Elisa Bertino
   Dipartimento di Informatica
   Universita di Genova
   Via L.B. Alberti, 4
   I-16132 Genova, Italy
   bertino@cisi.unige.it

3. Marie Jose Blin
   Universite Paris-Dauphine
   LAMSADE
   Place du Marechal de Lattre
   de Tassigny
   F-75775 Paris Cedex 16, France
   blin@dauphine.fr

4. Volkert Brosda
   Fachhochschule Hannover
   FB Bibliotheks-, Informations-
   und Dokumentationswesen
   Bernhard-Caspar-Straße 7
   W-3000 Hannover 91, Germany

5. Alex Buchmann
   TH Darmstadt
   Fachbereich Informatik
   Alexanderstr. 10
   W-6100 Darmstadt, Germany
   buchmann
   @informatik.th-darmstadt.de

6. W.S. Cellary
   Franco-Polish School of
   New Information and
   Communication Technologies
   ul. Mansfelda 4
   PL-60854 Poznan, Poland
   cellary@efp.poz.edu.pl

7. Branimir Dukic
   University of Osijek
   Ekonomik Fakultet
   Gajec TRG 7
   5400 Osijek, Croatia

8. Jürgen Ebert
   Fachbereich Informatik
   Universität Koblenz-Landau
   Rheinau 3–4
   W-5400 Koblenz, Germany
   ebert@informatik.uni-koblenz.de

9. Susan Even
   Fachbereich Informatik
   Universität Frankfurt
   Robert-Mayer-Str. 11-15
   W-6000 Frankfurt/Main 11
   Germany
   even@informatik.uni-frankfurt.de

10. Marie-Christine Fauvet
    IMAG–LGI
    Universite de Grenoble
    B.P. 53X
    F-38041 Grenoble Cedex, France
    fauvet@imag.fr

11. Fabrizio Ferrandina
Fachbereich Informatik
Universität Frankfurt
Robert-Mayer-Str. 11-15
W-6000 Frankfurt/Main 11
Germany
ferrandi
@informatik.uni.frankfurt.de

12. Stephane Gançarski
Universite Paris-Dauphine
LAMSADE
Place du Marechal de Lattre
de Tassigny
F-75775 Paris Cedex 16, France
gancarski@dauphine.fr

13. Lia Goldstein Golendziner
Universidade Federal
do Rio Grande do Sul
Instituto de Informatica
Dept. Informatica Aplicada
Caixa Postal 15064
91501-970 Porto Alegre
Rio Grande do Sul, Brazil
lia@inf.ufrgs.br

14. Margret Gross-Hardt
Arbeitsgruppe Informatik
Universität Giessen
Arndtstr. 2
W-6300 Giessen, Germany
gross-hardt@neumann.
informatik.uni-giessen.de

15. Theo Härder
Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
W-6750 Kaiserslautern, Germany
haerder@informatik.uni-kl.de

16. Genevieve Jomier
Universite Paris-Dauphine
LAMSADE
Place du Marechal de Lattre
de Tassigny
F-75775 Paris Cedex 16, France
jomier@dauphine.fr

17. Wolfgang Käfer
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099, USA
kaefer@almaden.ibm.com

18. Sitaram Lanka
Computer Science Department
Pennsylvania State University
University Park, PA 16802, USA
lanka@cs.psu.edu

19. Joelle Madec
O_2 Technology
7 Rue du Parc de Clagny
F-78035 Versailles, France
joelle@o2tech.fr

20. Tadeusz Morzy
Politechnika Poznanska
Instytut Informatyki
ul. Piotrowo 3a
PL-60965 Poznan, Poland
morzy@plpotu51.bitnet

21. Carolle Palisser
Universite de Nantes
2 Rue Houssiniere
F-44072 Nantes, France
palisser@narech.dnet.circe.fr

22. Norbert Ritter
Universität Kaiserslautern
Fachbereich Informatik

Erwin-Schrödinger-Str.
W-6750 Kaiserslautern, Germany
ritter@informatik.uni-kl.de

23. Manfred Rosendahl
Fachbereich Informatik
Universität Koblenz-Landau
Rheinau 3–4
W-5400 Koblenz, Germany
ros@informatik.uni-koblenz.de

24. Jarogniew Rykowski
Franco-Polish School of
New Information and
Communication Technologies
ul. Mansfelda 4
PL-60854 Poznan, Poland
rykowski@efp.poz.edu.pl

25. Harald Schöning
Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
W-6750 Kaiserslautern, Germany
schoenin@informatik.uni-kl.de

26. K. Vidyasankar
Department of Computer Science
Memorial University of Newfound-land
St. John's, Newfoundland
Canada A1C 5S7
vidya@garfield.cs.mun.ca

27. Gottfried Vossen
Arbeitsgruppe Informatik
Universität Giessen
Arndtstr. 2
W-6300 Giessen, Germany
vossen@informatik.rwth-aachen.de

28. Waldemar Wieczerzycki
Franco-Polish School of
New Information and
Communication Technologies
ul. Mansfelda 4
PL-60854 Poznan, Poland
wieczerzycki@efp.poz.edu.pl