An approach to automatic maintenance of semantic integrity in large design data bases

by GILLES M. E. LAFUE
Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

One way a data base can cease retaining a meaningful relationship with the "real world" situation that it models is through violations of its semantic integrity, i.e., transgressions of the defining constraints of its data. These constraints state the legality of the data values and are defined by the creators of the data, e.g., a checking account cannot be negative, or the salary of an assistant professor cannot be greater than that of a full professor.

The variety of integrity constraints to be maintained can be extremely wide. It can range from general low-level management such as dangling references to complex knowledge specific to the "real world" situation, e.g., in the case of a building, detection of spatial conflicts or even statics and mechanics.

Many of the debates about the different data models proposed in the past few years center around the extent of the semantics that these various models hold. Whatever data model one can use nowadays, there will always be semantics which escape it. In daily practice, these semantics are often maintained by the application programs or the manual users. This forces them to maintain information about the data base in a form external to it (a data base about the data base) when in fact it is conceptually part of the data base itself. Incorporating this external information in the data base would be an extension of the effort which led in the first place to modeling a "real world" situation with a computer data base. Moreover, interaction with the data base would be improved by automating the maintenance of these semantics.

This need exists for any kind of data base but it is especially pressing for what I call design data bases. A design data base not only stores a model of some artifact and provides primitive accesses to it (e.g., query processing) but also supports the design of this model with the hesitations and backtracking usually involved, e.g., the schema (the set of data types) is continuously modified. The design of, and the interaction with, such a system are greatly helped by the data and control abstraction features of a high-level programming language. GLIDE is an example of a design data base. The integration of language and data base concepts for design applications and its relationships with the principles presented here are discussed in Reference 4.

Unary integrity constraints apply to a single record and n-ary constraints apply to several records. The proposed approach starts by considering an n-ary integrity constraint as composed of one set of dependent variables and one set of independent variables. Variables can be both dependent and independent. The value of a dependent variable is partially or entirely determined by the values of the independent ones. Maintaining an n-ary integrity constraint consists in recomputing or checking the values of its dependent variables when the independent ones change.

The variables of an intra-record integrity constraint are attributes of the same record. A record is a collection of data logically related and stored together. The variables of an inter-record constraint are attributes of several records. These records can be instances of the same type or of different types.

The distinction of constraints according to the number of records they involve is important from two viewpoints. First, the record types and the operations performed on them determine the modularity of the data base schema and of the "contexts" in which the applications execute. Secondly, records belonging to the same integrity constraint are rarely in core at the same time and, in a traditional computer architecture, the cost of disk access is high. The emphasis of this approach is on inter-record constraints.

This paper presents three basic principles of an approach to automatic maintenance of semantic integrity in large design data bases. First, the maintenance of integrity is delayed until strictly necessary. Second, integrity violations are temporarily tolerated. Third, integrity constraints are procedures included in the record definitions and automatically activated by the system. These principles are generally not supported by the data base systems attempting elaborated automatic maintenance of integrity, in particular INGRES and System R. A mechanism based on these principles is currently under investigation.

DELAYED MAINTENANCE OF INTEGRITY

An integrity constraint is to be maintained when one or more of its independents has been updated.
Tolerance of Integrity Violations

A consequence of delayed maintenance of integrity is that the database must be able to tolerate violations of its integrity, at least temporarily. Between the time an independent record is updated and the time a record depending on this record is accessed, the constraint that links these two records is left unenforced. Furthermore, the independent update may imply a dependent value which is illegal because of another constraint applying on the dependent. Violations are tolerable only as long as they, or their causes, i.e., the updates, are recorded. In contrast, in System R and INGRES, updates leading to violations are immediately rejected.

Sometimes, tolerance of violations is not only an acceptable consequence but is desirable. Integrity constraints may apply at some moments and not at others. This is particularly true for design databases. One advantage of using a model to design a complex artifact is that the constraints for making the model are different, and hopefully more manageable, than those for making the artifact. Some constraints pertaining to the artifact may be ignored for convenience, until the end of a design phase. Such a phase defines an integrity transaction for the constraints temporarily dropped and the records they concern.

In the current approach, integrity transactions are delimited by (dependent) record openings. Opening a record ends the transaction for the record and the constraints in which it is a dependent. If it is desired to open a record without maintaining one of its constraints, this constraint can be associated with another record. This other record is opened when the constraint needs to be checked. The purpose of some records may be to implement the external integrity of other records. Such records are typically aggregation abstractions as defined by Smith and Smith. They are the dependents of the aggregations and the records whose integrity they implement are the independents. For instance, aggregations are useful for centralizing the management of circular dependencies.

In the cases described so far, violations are tolerated until they are detected. Now, tolerating and recording a violation may be the action to take when the violation is detected, in order to postpone its resolution. For instance, in a transaction during which an integrity constraint does not apply, it may be useful to record the violations of this constraint on the fly, so that they can be readily taken care of at the end of the transaction, instead of being recomputed. Recording violations is also useful when a constraint possesses several independent variables and is violated by the update of one of them. This violation can be notified to the other independent variables. Often, at least one can change its value in order to accommodate the requested update. For instance, if several persons share a checking account, one person can overdraw the account and another one compensate the overdraft.

Simultaneous alternative values for variables can be assimilated to violations of the data base integrity. They indicate some indecision as to what the unique values of the variables are. They are often useful to tolerate temporarily, especially in design activities, due to the hesitant nature of these activities.

Alternatives may be generated by the same process or by different ones. Regarding the latter case, the principles of resource protection and concurrency control, traditionally used in operating systems, are necessary but not sufficient for data bases. Locking mechanisms serialize concurrent accesses. However, since writing in a data base is to leave a more or less permanent mark, the question of whether the authorized processes write in a record serially or in parallel,
is immaterial. What is important is that overwriting each other may lead the processes into conflicts which cannot necessarily be solved as soon as they occur.

**INTEGRITY CONSTRAINTS INCLUDED IN THE RECORDS AND AUTOMATICALLY MAINTAINED**

Since all the semantics usually desired in a database cannot be incorporated in the data structures, one has to resort to executable code. This code should be of a high level in order to maximize the power of integrity constraints.

The code implementing the integrity constraints of a record can be included in the record definition in the manner of abstract data types. It participates in defining the record semantics, particularly, its behavior, as much as data structures. This code should be pre-compiled in order to avoid recompiling it every time it is executed. Integrity constraints can then be implemented by pre-compiled procedures called **integrity procedures**.

Clearly, including integrity constraints in the records is acceptable for intra-record constraints since it confines them to the concerned records. It also holds for inter-record constraints. While the records should not know their uses, they must know the records on which they depend or, rather, abstractions of these records. Consequently, integrity procedures are included in the dependent records. This inclusion respects abstraction boundaries. Integrity procedures write in the dependent records or check their values, and they simply read the independent ones.

While the usefulness of the notion of abstract record type in data bases has been acknowledged by several authors, e.g. Reference 6, others have objected to it. In particular, Hammer rightly points out that the behavior and the uses of a record type evolve over time. The proposal to exclude the uses of a record from its definition reduces this evolution significantly.

This procedural approach is different from one which guarantees integrity by providing procedures for accessing the data base. Such procedures guarantee that the data base transits from one valid state to another. They implement **state transition** integrity. State transition procedures enforce immediate maintenance of integrity and do not tolerate violations.

The inclusion of integrity constraints not only in the data base but in the records, contrasts with other approaches. The separation of integrity constraints from the records, as in INGRES and System R, makes insertions and deletions of integrity constraints easier and it centralizes the detection of conflicts between them. The cost of this separation, however, is to spread the definition of the records.

The mode of activating integrity checks, i.e., integrity procedures, remains to be examined. There are two kinds of integrity checks. Some are implicit. The user does not want to have to activate them explicitly all the time. They can be activated automatically whenever necessary, i.e., when the records are accessed. The others are explicit. They take place at the end of a user-defined transaction which can span several record accesses. A unified scheme is proposed which satisfies both sorts of integrity checks.

Integrity procedures should be written by the user and compiled in relation with record declarations (either types or instances), but automatically invoked by the system upon well defined conditions. These conditions are data base operations, e.g., record reads and writes. System R possesses a similar triggering mechanism which activates the execution of statements of a query language.

For implicit checks, integrity procedures are automatically activated every time their records, supposedly the dependent records of the constraints, are accessed. As for explicit checks, opening a record which implements integrity constraints of other records marks the end of a user-programmed transaction.

**ACKNOWLEDGMENTS**

Thanks are due to Charles Eastman, Kevin Weiler, Molly Meigs and to the NCC referees for their helpful comments.

**REFERENCES**
