Assuring Database Integrity

David Stemple
University of Massachusetts

Eric Simon
INRIA, France

Subhasish Mazumdar
University of Massachusetts

Matthias Jarke
University of Passau, West Germany

Database integrity has many facets. Both **consistency**, the property of a database modeling a possible state of the world, and **correctness**, the exact correspondence of the data to the state of the world being modeled, are important aspects of integrity. Integrity also includes guaranteeing that meaningful views and changes result from interleaved access to the database occurring in typical transaction processing, a property known as **atomicity** of transactions. Assuring database integrity requires several mechanisms, including language capabilities, runtime processing, and operational controls. Here we concentrate on the maintenance of consistency, both the means of stating database consistency properties and techniques of efficiently assuring that a database can only change from one consistent state to another. We present a theorem-based method of simplifying integrity constraints at the heart of most techniques for achieving efficient integrity assurance. We argue that these techniques are currently practical enough to be included in database system products in the near future, but that languages used to specify constraints and transactions will have to be of a high level of abstraction, reasonably formal, and uniform in order for this to happen.

In this paper we will concentrate on the problem of assuring database consistency. We will present the various aspects of the problem of expressing integrity constraints, which will be the linguistic means of specifying the consistency of a database. We will not deal with either the problem of building controls for assuring the factual nature of a database or the issues of concurrency control, the means of effecting meaningful changes and views of a database subjected to interleaved access and update.

We will first outline the varieties of integrity and relate the consistency issues with those of “good” database design, such as those incorporated in relational
normalization theory. We find it useful to separate the consistency problem into two important subcases, static and dynamic. Transaction atomicity will be discussed briefly to explain why it is a requirement for maintaining consistency of a database, but is unrelated to the details of a particular database’s integrity constraints. Means of specifying integrity constraints and transactions are discussed in detail. We then survey the ways in which the efficiency of maintaining integrity constraints can be improved by using theorems that relate integrity predicates with update primitives. The main problem is to attain efficiency in processing transactions while assuring that the integrity of the database is maintained.

**Varieties of Integrity**

Database integrity addresses a variety of concerns, all subsumed under the desire that the data in a database be correct, that is, represent the proper state of the modeled world. However, consistency, the property of a database modeling a possible state of the world, is also an important concept since, as explained above, it is a precondition for correctness and has the possibility of system-maintenance, an option that does not exist for full correctness. Integrity also includes guaranteeing that meaningful views and changes result from interleaved access to the database occurring in typical transaction processing, a property known as *atomicity* of transactions. In this section we outline the varieties of integrity that are included under the notion of consistency, and discuss the issue of atomicity only briefly.

**State Predicates and Static Integrity.** Many of the properties that define the consistency of a database can be represented by predicates on the database state (Eswaran & Chamberlain, 1975), i.e., statements containing variables which, when the variables are assigned values from the database state, can be evaluated as true or false. If all of a set of such predicates defining the integrity of a database are evaluated as true using the values from a database state, then the state is consistent. Such predicates define what we will call state-based or static integrity.

In order to provide a concrete basis for talking about consistency, we will use an example database. The entity-relationship diagram in Figure 1 will provide the basis for our examples. The world modeled by the diagram consists of employees, projects, and skills. Employees possess skills and projects require skills. Projects depend on each other, which means that a dependent project cannot start until the project(s) on which it depends is (are) completed. Employees are managed by employees and are assigned for a certain percentage of their time to projects.

Examples of state-based integrity constraints in a database representing employees, skills, and projects include:

1. An employee number is less than 200,000.
2. An employee who is a manager has a salary greater than $40,000.
3. Each employee has a unique employee number.
4. All employee numbers in the database are employee numbers for employees described in the database.
5. The set of skills possessed by the employees working on a project contains the skills required by the project.
6. The starting date of a project must not be earlier than the end date of any project on which it depends.
7. The way in which projects depend upon each other cannot be cyclic.
8. The salary budget of a project will be the weighted sum of the assigned employees’ salaries, where the weighting is the percentage of time the employee is assigned to the project.

The variables in these constraints are *employee(s)*, *skills* and *project(s)*. It should be clear that if we understand the meaning of the data in the database, we could evaluate these predicates to determine whether the database was consistent or not.

The amount of consistency that can be expressed by static integrity constraints clearly depends on the amount of facts that are represented in the database. This is most obvious when the way in which the state of the
database was reached is an issue. For example, if it is corporate policy that a person cannot be a manager and have his or her salary reduced from any previous salary, then a database that only maintains records on current positions and salaries cannot have this requirement enforced on the database using state-based integrity constraints. However, if the database design includes position and salary histories, then such consistency can be expressed as static predicates.

**Dynamic Constraints and Dynamic Integrity.**
A *dynamic* constraint specifies a restriction on the possible sequences of states through which a database may be taken. Dynamic constraints are expressed through predicates on state sequences. To be clear about what a transition from one state of a database to another means, we must discuss transactions and atomicity.

Databases are changed by actions that read, change, and write data residing on some *stable* storage such as disks. Stable refers to the ability of a storage to hold its data unchanged without a process or computer being active. Here, we will not worry about the details of maintaining stability in faulty physical devices, the domain of concurrency control and recovery mechanisms. Nor will we normally be concerned with the details of reading and writing stable storage that constitute the implementation of database updates. Changes to a database at the level of granularity of reading and writing parts of stable storage are not the changes on which a database administrator should express dynamic constraints. Instead, a database administrator should express constraints on transitions between states representing successive states of the world being modeled, i.e., states before and after the real world events. Of course, this is the level of change effected by transactions. Transactions are sequences of actions that model a change from one state of the world to another. They are *atomic* in the sense that the intermediate states of the database during a transaction’s execution should not be interpreted as representing a state of the modeled world. Thus, dynamic constraints are expressed on sequences of *final* states of transaction executions.

It is reasonable to ask whether, in the chaos of interleaved executions of many transactions, output states of individual transactions are ever realized in the physical database. It is the job of concurrency control mechanisms to assure that any view of the database is *as if* the database were in a state produced by the complete executions of some sequence of transaction invocations. If the concurrency control mechanism is not sufficient to this task (the mechanism does not assure that read and write schedules have the technical property of *serializability*), then the performance of integrity control mechanisms (in the sense of consistency) is seriously compromised. The details of this contention are beyond the scope of this paper, and in the following we will assume that all transactions execute atomically.

**Two-state transition constraints.** Many important dynamic integrity constraints can be expressed on the input and output states of single transactions. For example, if salary reductions do not occur in the world being modeled, then *transition integrity* can be maintained by assuring the perpetual truth of a predicate stating that the output salaries of each transaction be individually equal or larger than their corresponding input state values. Such a predicate can be termed a two-state *predicate* or simply a transition *constraint*.

Two-state predicates generally use a standard way of distinguishing the input state of a transaction from the output state. Typically languages which allow two-state integrity constraints refer to the input state values as *old* and the output values as *new*. For example, the requirement that salaries do not decrease can be expressed in SQL-like syntax by:

```
ASSERT SALINFLATES ON EMPLOYEES: OLD SALARY <= NEW SALARY
```

Note that this constraint cannot be expressed by a state-based predicate if the database only holds current salaries. Note also that a two-state predicate cannot express the previously discussed requirement on manager’s salary in a world in which salaries may be decreased for non-manager’s salaries. For this, we need predicates that can be termed *temporal*.

There are certain assumptions made in the use of transition constraints, mainly having to do with the identification and persistence of entities. There is a tacit assumption that the representation of a modeled individual will have the same continuity as the individual. For example, if an employee description (record or tuple) is removed from the database it is assumed that the employee is no longer an employee, and more importantly, that if he or she is rehired, it is as a new employee from the point of view of the transition constraints. If more sophistication is desired, either the employee description should not be removed, or some way of providing continuity of the employee description should be devised.

**Temporal constraints.** Temporal integrity constraints are predicates expressed on arbitrarily long sequences of states of a database. The following are examples of temporal constraints:

1. An employee that has been a manager three separate times may not be fired.
2. Once an employee has been a manager, his or her salary may not be reduced.
3. Every project will eventually be independent of all others.
4. Once an employee has managed five projects that depended on each other, he or she will always remain a manager during the remaining employment period.

5. Each employee will eventually either be a manager or reach the minimum salary of all the managers he or she has worked for.

These constraints are exceedingly difficult to maintain without reducing them to two-state or static constraints. The problem is that an arbitrarily long sequence of states may separate the occurrence of the precondition, being a manager for example, from a violation of a requirement. There are basically two ways to handle such constraints: maintain data in a shadow database that allows testing for past status, for example, having been a manager at least three times, or analysis of transactions to assure that there is no way the precluded action could take place because of transactions’ preconditions and internal structure. The normal way for such consistency to be maintained is to include enough history in the database to reduce general dynamic constraints to either state or two-state constraints. We will now outline the techniques for representing temporal constraints.

The temporal constraints described earlier are not expressible in classical first order logic, and hence beyond the scope of languages such as SQL. Temporal Logic extends first order logic by incorporating three modal operators ALWAYS, EVENTUALLY, and NEXT, and avoids explicit references to time. Temporal logic is thus a useful language for specification of temporal constraints.

- ALWAYS \( p \) is interpreted to mean that from the current state onwards, the predicate \( p \) will hold in every future state visited by the database.
- EVENTUALLY \( p \) means that in some state obtained from the current state through a finite but unknown number of applications of actions, the predicate \( p \) will hold.
- NEXT \( p \) means that \( p \) will hold in the state that the database will visit next.

Using temporal logic, our second example constraint would be written as:

\[
\forall x: (\text{manager}(x) \land \text{salary}(x, \text{dollar}1)) \rightarrow \\
\text{ALWAYS} [\text{salary}(x, \text{dollar}2) \rightarrow (\text{dollar}2 \geq \text{dollar}1)].
\]

Using the inference rules of temporal logic, if it can be shown that a temporal formula describing a constraint is equivalent to another, in which the only modal operator is NEXT, then the second formula represents an equivalent two-state (transition) constraint that can be more easily maintained. For example, the above formula can be rewritten to:

\[
\forall x: (\text{manager}(x) \land \text{salary}(x, \text{dollar}1)) \rightarrow \\
\text{NEXT} [\text{salary}(x, \text{dollar}2) \rightarrow (\text{dollar}2 \geq \text{dollar}1)].
\]

It is clear that the last formula is a 2-state (transition) constraint and is much simpler to enforce.

Thus, temporal logic provides a formal mechanism for specifying, simplifying, and reasoning about dynamic constraints.

However, some temporal logic formulas cannot even be evaluated given a database history up to the present state. Consider a formula of the form “EVENTUALLY \( p \)”, where \( p \) did not hold true in any of the database states prior to and including the current one. We can neither conclude that the constraint is false, since \( p \) may yet hold in some future state, nor that it is true, because \( p \) may never hold in the future. For this reason, the notion of partial validity has been introduced. The subsequence of database states up to the current state is said to be strictly valid if it satisfies the constraints regardless of the future states, and potentially valid if there provably exist some completions of the subhistory such that the constraints hold.

Temporal logic cannot capture some constraints. For example, with only the three modal operators, our last example constraint cannot be expressed. In general, temporal logic is too weak to express certain temporal patterns, e.g., multiple similar states.

**Operation-Based Constraints.** There are some constraints that are naturally associated with operations. For example, in a bank, it is reasonable to require that an account have a zero balance before it has its data removed from the database. It is not possible to state this constraint simply as a predicate on the state of the database. If an account is represented as a tuple in an accounts relation then we can state this constraint by a statement of the form: BEFORE DELETE OF A FROM ACCOUNTS CHECK A.BALANCE = 0.

While this does not have the simple form of a predicate on two successive states, it certainly expresses a two-state predicate. The meaning of this can be paraphrased by: *If a state contains account A, and the successive state does not contain A, then A has a balance of 0 in the first state.*

Similarly new data can be constrained by associating a predicate with an operation as in the following: BEFORE INSERT OF E FROM EMPLOYEES CHECK EMPLOYEES.WEIGHT < 300.

Clearly this is a transition constraint expressible as in the previous example. Note that this does not mean that an employee cannot have a weight of 300 or more, only that upon insertion of the employee’s tuple into the EMPLOYEES relation, his or her weight must be less than 300. Subsequent changes to a person’s weight to above 300 are not precluded by this constraint.

Such constraints are known as assertion triggers. Eswaran (1976) defines a trigger as a block of statements to be executed whenever some triggering event
takes place. In case the triggering event is a condition, monitoring for its occurrence is a similar problem to integrity constraint checking. An assertion trigger provides a pragmatic hint regarding when the triggering event need be checked. The above examples show that an operation-based constraint can be specified with an assertion trigger by defining the operation as the triggering event; the implicit triggering action is a backout from the operation in case the query following the keyword CHECK returns false.

Triggers are certainly powerful enough to be used to specify all static constraints. Indeed, some believe this may be the preferred method of specifying integrity, including static integrity (Date, 1983).

Are triggers preferable to static constraints? Certainly, there are some cases where it is cheaper to enforce a constraint by doing ad-hoc compensating updates to the database instead of checking a predicate and then deciding whether to reject the transaction or not. Take for example a referential integrity constraint, where a deletion of the primary entity needs to be compensated by a deletion of all tuples having the same value for foreign key. Such reliance on triggers, however, can lead to undesirable side-effects as an error in a transaction may cause the destruction of parts of the database.

It is difficult to replace arbitrary constraints with triggers. Of course, one can declare the commit of a transaction to be a triggering event and check the constraint as a triggering action, but that, in general, is a very inefficient solution. Moreover, for multi-update transactions where a constraint can be temporarily false during the execution of the transaction, it is not always easy to design a cheap trigger. When it is possible to design efficient triggers, they offer the advantage of ease of modifiability as well as improvement over leaving constraint maintenance entirely as the responsibility of transaction programmers.

The strong procedural nature of triggers together with the interaction among many applicable triggers (leading to an unpredictable flow of control), often make it difficult, sometimes impossible, to determine what constraint is being maintained by a set of triggers. This problem generalizes to transactions in the following sense. Given a set of transactions, it is in general undecidable whether a given predicate is an invariant of all combinations of the transactions. Abiteboul and Vianu (1985, 1989) address the possibility of describing valid states of a relational database using a set of admissible transactions, called a transactional schema. They show that the two approaches (i.e., expressing integrity as constraints or as valid transactions) are incomparable in that there are transactional schemas with no equivalent constraint schema and conversely. They advocate that both should co-exist.

The action of a trigger could be something beyond the prevention of the operation execution; for example, it could cause a message to be sent to some user. Buneman and Clemons (1979) describe alerters as programs which monitor a database and report when a specific condition occurs. For example, an alerter may report when there is a substantial overlap among the leave periods scheduled by the members of a project. When appropriate, some operation-based constraints may therefore be better expressed as assertion alerters.

**Normal Forms.** Relational normal forms might be thought to express integrity constraints. Indeed, the motivation for normal forms is to simplify the maintenance of integrity. Normal forms are defined in terms of functional and other dependencies, and dependencies are predicates on relations. But a normal form requires that all dependencies fit a prescribed pattern of dependencies, one in which certain dependencies do not exist. The non-existence of dependencies is not a property of relation instances (or of database states) that can be tested in the same manner as the existence of a dependency.

**Formal Models of Integrity.** There are several formal models of integrity, based on formal models of database semantics. The predominant approach in dealing with the formal issues of database management, as opposed to logic programming, knowledge bases and deductive databases, is the model-theoretic approach. The model-theoretic view considers a database state to be a model of a theory defined by the data model (e.g., relational), a schema, and the integrity constraints, which constitute axioms beyond those of the data model. A model of a theory is an assignment of concrete functions and relations (instances) to the symbols of the theory. To be a consistent database state is simply to be a model of the theory defined by the data model, schema and integrity constraints. In this approach, database updates are changes from one model to another. The typical database model is also simplified by the assumption that the database is complete, i.e., contains all facts (of the kind kept in the database) about the modeled world. This assumption was termed the closed-world assumption by Reiter (1978).

We will use the term deductive databases to refer to logic programming, knowledge based systems, and other approaches that include rules in the database or accommodate incomplete information. In these approaches integrity and update semantics are not so simple as in the model-theoretic approach. Deductive databases have the possibility of containing rules for generating facts beyond those contained in “bare facts” — statements containing no variables (tuples in relational databases). They may also be incomplete, i.e., not obey the closed-world assumption.

Problems stemming from rules in the data
employee salaries. This could be defined in a view, of course. In our style of specification, we see the conceptual schema, which is our primary concern here, as having the possibility of implementations (physical designs) that have structure significantly different from that of the conceptual schema. The implementation of SALBUDGET by a function that computes SALBUDGET on demand and ignores updates (after checking them for correctness) is a possible implementation of this schema. PLANSTART and PLANEND are dates in YYYYMMDD format that we intend to be in concert with the dependency relationship recorded in DEPEND. A DEPEND tuple records that the project identified by BEFPROJ must be completed before (or at the same time as) the project identified by AFTPROJ is started.

We now express the static integrity constraints enumerated earlier in a language close to SQL.

1. An employee number is less than 200,000.

\[
\text{ASSERT CONSTRAINT1 ON EMPLOYEES: EMPNUM < 200000}
\]

Or, in the CREATE statement of standard SQL (Date, 1989):

\[
\text{CHECK EMPLOYEES.EMPNUM < 200000}
\]

2. An employee who is a manager has a salary greater than $40,000.

\[
\text{ASSERT CONSTRAINT2 ON EMPLOYEES: IF EMPTITLE="MANAGER" THEN SALARY > 40000}
\]

Or, in the CREATE statement of standard SQL:

\[
\text{CHECK EMPLOYEES.EMPTITLE <> "MANAGER" OR EMPLOYEES.SALARY > 40000}
\]

3. Each employee has a unique employee number.

\[
\text{ASSERT CONSTRAINT3 ON EMPLOYEES: KEY(EMPLOYEES, EMPNUM).}
\]

Or, after the EMPNUM declaration in the CREATE statement by the keyword UNIQUE

4. All employee numbers in the database are employee numbers for employees described in the database.

\[
\text{ASSERT CONSTRAINT4 ON POSSESS, EMPLOYEES: PROJECT(EMPLOYEES,EMPNUM) CONTAINS PROJECT(POSSESS,EMPNUM)}
\]

\[
\text{ASSERT CONSTRAINT5 ON ASSIGN, EMPLOYEES: PROJECT(EMPLOYEES,EMPNUM) CONTAINS PROJECT(ASSIGN, EMPNUM)}
\]

\[
\text{ASSERT CONSTRAINT6 ON POSSESS, EMPLOYEES: PROJECT(EMPLOYEES,EMPNUM) CONTAINS PROJECT(SKILLS, EMPNUM)}
\]

\[
\text{ASSERT CONSTRAINT7 ON MANAGE, EMPLOYEES: PROJECT(EMPLOYEES,EMPNUM) CONTAINS PROJECT(MANAGE,EMPNUM)}
\]

These are referential integrity constraints and are invoked in standard SQL using the FOREIGN KEY clause, ignoring nulls. For example, in the CREATE TABLE POSSESS statement the following would be included:

\[
\text{FOREIGN KEY (EMPNUM) REFERENCES EMPLOYEES}
\]

5. The set of skills possessed by the employees working on a project contains the skills required by the project.
6. ease of maintenance over given data structures, e.g., easily maintained in hierarchical data structures.

The locality of data needed to check a constraint is clearly an indication of the difficulty of checking the constraint. For example, a constraint involving only a field of a tuple, e.g., CONSTRAINT1 above, often called a domain constraint, is clearly easy to check. It needs only to be checked when a tuple is inserted into a relation or modified. A constraint involving a complete tuple, for example comparing two fields as in CONSTRAINT2, is only marginally more difficult to check.

Constraints that express conditions that a relation must possess, for example, CONSTRAINT3 and CONSTRAINT11, can be more difficult to maintain than domain or tuple constraints even when supported by data structures such as indexes. Interrelational constraints, e.g., CONSTRAINT4 through CONSTRAINT10 and CONSTRAINT12, involving the comparison of data from more than one relation obviously have the potential of requiring more data to be accessed in checking them than either tuple or relational constraints, though CONSTRAINT11, involving only one relation, is inherently the most complex of our examples.

The complexity of computations needed to check constraints in a naive manner is an important feature of constraint classification, but does not determine the real cost of maintaining the constraints. The reasons for this will be given below when we discuss means of maintaining constraints.

The level of language needed to express the constraint is, as noted, intimately related to the complexity: there are known equivalences between the complexity classes of complexity theory and language levels (Aho & Ullman, 1979; Immermann, 1987; Qian, 1989). The most significant of the language levels from the point of view of current database technology are the classes equivalent to relational algebra and relational algebra with a transitive closure operator. A transitive closure operator takes a binary relation and repeatedly performs a join and a project, the first time with the input relation, subsequently with the result of the previous join/project operation. The join is on the predicate equating the second component of its first relation with the first component of its second relation; the project is on the first and last components of the join. One execution of this join/project would add the grandparent relation to a parent relation. The repetition is continued until no new tuples are generated. It is not possible to state the constraint that a given binary relation is to be acyclic (CONSTRAINT11) using only relational algebra, while it is expressible if a transitive closure operator, or its equivalent, is in the language.

CONSTRAINT12 is also inexpressible in SQL.
(except through use of updates and temporary relations) or relational algebra. This can be remedied by having an operation that takes the elements of a set (a relation in our example), applies a function to each, and accumulates the results using a second, binary accumulating function. Our example of weighted sum is a specific version of this in which the accumulating function is arithmetic plus and the application function applied to each tuple is the multiplication of the PERCENT value with the SALARY value. We could write this in functional form as:

\[ \text{set-reduce}(S, \lambda(x)(x\cdot\text{PERCENT} \times \text{x.SALARY}), \text{plus}, 0), \]

where set-reduce returns the value 0 when the input S is an empty set. Its second argument, \( \lambda(x)(x\cdot\text{PERCENT} \times \text{x.SALARY}) \), is the function of one argument (a tuple) that returns the SALARY of its input tuple multiplied by its PERCENT value. The higher order function set-reduce applies its second argument (a properly typed function) to each element of its first argument (a set) and accumulates the result of each application with the result of the accumulation so far. Since the accumulation is done after each application, this does not have the problem of a projection on a column followed by a summing operation (namely the loss of duplicates). (Nor does it need a concept of distinct values in order to avoid the problem of a too limited set of functions over sets.) Use of such a construct as set-reduce would provide a means of eliminating the lamentable lack of uniformity in SQL and simultaneously increase the language to include the ability of expressing constraints such as CONSTRAINT11. In particular, specific forms of set-reduce, similar to WEIGHTED-SUM, could be included in the language for succinctness. All of relational algebra operations, as well as all of the example integrity constraints, can be expressed using set-reduce.

Other ways of classifying constraints include the times at which the constraints should be checked. Times could be certain points in real time, commit times for all or specific transactions, or when certain operations are executed as in triggers. Another feature that separates constraints is the kind of action to be taken whenever a constraint is violated. Actions, which are often called exception routines, include aborting the transaction, issuing a warning message and taking corrective measures. See the work of McCarthy and Dayal (1989) for the architecture of an “active” system built around triggers as the basic primitive and the work of Stonebraker and others (Stonebraker et al., 1988) for a broad exploitation of “rules” in the POSTGRES system.

**Transaction Specification**

In order to have only meaningful views of the changing data in a database subjected to multiple interleaved access and update, it is necessary for the models of complex events in the world to be formed into transactions. A transaction is a series of accesses and updates of the database whose intermediate changes are to be made invisible to other accessors of the database. If a transaction successfully completes, all of its updates to the database are to be made visible in an atomic manner, i.e., as if simultaneously. If a transaction is unsuccessful or aborts, all of its actions are to be undone and no access to its temporary effects are to be allowed. If, for reasons of efficiency, provisional access to the aborted transaction’s updates was allowed, this must be undone, typically by causing the accessing(s) transaction also to abort, process called *cascading aborts*.

The means of specifying transactions affects the difficulty of maintaining integrity. There are basically three means currently being used or proposed for specifying transactions:

1. bracketing an extent of a program in some programming language containing database access with begin and end transaction commands
2. specifying a transaction in a higher level database-oriented language such as SQL (or a fourth generation language)
3. specifying a transaction by pre and post-conditions in a predicate language

**Programming Language Specification of Transactions.** Unfortunately, the first method is the most common. Calls to the database or transaction management system signals that subsequent database manipulations, up to a transaction end command, are to be in a transaction. The database management system then coordinates the actual access and updating of the database in order to isolate other database use from this transaction’s intermediate results. At transaction end time, the database management system executes a commit which, if successful, makes all of the transaction’s changes visible to (logically) subsequent transactions and queries. If programs in current programming languages augmented by calls to low level update primitives or embedded query languages such as SQL are used as the highest level of transaction specification, leaving the assurance of integrity to programs written in the same languages is probably the best that can be attained. The reason for this is that programming languages are too powerful (and therefore complex) and have formal semantics that do not support verification of complex properties. If programming languages are used as the highest level of transaction specification, integrity maintenance is no simpler than the general problem of program verification. It is hoped that in the database setting the problem can be made simpler. One means of attaining this goal is high level, computationally limited, formal transaction specification languages.
High Level Languages for Transaction Specifications. A transaction specification language at the same level as relational query languages, proposed by Sheard and Stemple (ADABTPL) (Sheard & Stemple, 1989), appears to support the enforcement methods discussed in the next section. This language contains set-oriented update mechanisms and looping constructs. Cursor-based updates such as those in standard SQL need to have their semantics formalized, and probably limited, if they are to support sound techniques for assuring integrity maintenance. Qian (1989) discusses the expressiveness of the bounded loop primitive, which has the power to express updates beyond those expressible with only relational algebra. Language primitives with power beyond that of Qian's bounded loops are discussed in (Abiteboul & Vianu, 1988; Casanova & Bernstein, 1980; Chandra, 1981).

The example in ADABTPL given in Figure 3 illustrates the level and form of languages which seem to be tractable with current technology. We are not concerned here with specifying output messages and format or the way in which data from the users (the input parameters p1 and p2 in this case) is entered. These are important but have no direct impact on integrity maintenance.

Such a language can be quite formal and still support a reasonable style of transaction specification. See (Sheard & Stemple, 1989) for more on the formal underpinnings of ADABTPL. A feature of these languages is their lack of full computational power. They cannot compute all the functions that are definable in programming languages. Features (or limits) that contribute to tractable integrity maintenance methods include a uniform model of computations and predicates over sets, and limits on the use of assignments, cursors, and looping constructs. The exact nature of the relationship between language constructs and integrity constraint maintenance is still a subject of research. High level, limited languages will constitute the transaction specification languages that support verifiable integrity maintenance for some time, in our opinion. Our discussion of integrity enforcement methods below will clarify the reasons for this.

Transaction Specification Using Preconditions and Postconditions. Another way in which transactions could be specified is by using only a precondition and a postcondition. This method has its problems as we discuss below, but would appear to support the required formality quite naturally. The intended semantics is that if the transaction is executed in a state \( s_0 \), then resulting state \( s_1 \) must satisfy the following logical statement: IF \( \text{precondition}(s_0) \) THEN \( \text{postcondition}(s_1) \), where precondition and postcondition are boolean combination of predicates.

The advantage of this approach is that the intent of the transaction is specified in a declarative manner. But it suffers from the following problems. First, if the precondition is false in state \( s_0 \), then the above logical statement is vacuously true (a false premise makes an implication true), and hence, by the letter of the definition, the transaction may indulge in any operation whatever. This is the applicability problem. Second, updates that do not violate the postconditions are not disallowed. This is the frame problem once more. Third, integrity constraints are not considered — the correctness problem.

To address these problems, the transaction as specified must be strengthened. To take care of the first problem, the precondition can serve as an enabling condition (the ON statement in SQL), so that the transaction will not be scheduled for execution unless the precondition holds. The second problem can be formally addressed by adding the frame axioms. In practice, one has to ensure that the transaction effects the minimal change compatible with the pre- and post-conditions, and yet there may not be a (unique) minimum change. The third problem can be finessed by noting that the integrity constraints must have been true at the beginning of the transaction, and simply adding them to the postconditions. Of course, this may make the transaction prohibitively expensive, as we have mentioned earlier; therefore some form of transformation is advisable in order to move these integrity postconditions into simpler preconditions.

Pre- and post-conditions can be used as the first stage of specification in a layered development approach. The DAIDA project (Borgida et al., 1989) is an example of such an approach. This project has the goal of building a novel software engineering environment for developing software systems and maintaining information systems.
Transactions are specified using enabling conditions, pre- and post- conditions. They suggest an object-oriented approach where the data is first described as classes of objects related by attributes, and transactions specify operations that can be applied on the data. These descriptions are mapped into a language called DBPL, a successor of Pascal/R based on Modula-2.

**Constraint Enforcement Methods**

In this section we will examine the methods of enforcing integrity constraints. We will deal only with static constraints. Transition constraints can be handled by methods that are very similar to those useful for static constraints, while the maintenance of temporal constraints is still the domain of research efforts with practical solutions not to be expected soon. We examine the general approaches to maintaining constraints efficiently, which is the real problem, and place some of the research in this context. We will basically ignore constraints that can be maintained by checking individual tuples when they are inserted or deleted. These are handled adequately by current systems. We are more interested in those techniques for maintaining constraints of wider data scope that show promise of being incorporated into systems in the near future.

For succinctness, let us use the symbol IC to mean the composite predicate on the database that expresses (static) integrity for the complete database. Let T stand for a transaction modeled as a function from a database state, DB, and the transaction’s input, I, to a new database state, T(DB, I). The problem of constraint maintenance is to assure that for every transaction T the predicate IC is true of T(DB, I), or symbolically, IC(T(DB, I)).

The most straightforward way of maintaining integrity is to evaluate IC on the output state of every transaction before committing the state, and to abort the transaction if it evaluates to false. This is prohibitively expensive. A better way is to evaluate only those parts of IC that a transaction could possibly violate and abort the transaction if that produces false. This is more complex than it sounds but is feasible for certain integrity constraints and simple updates. The problem is determining the parts of the overall constraint that could be falsified. User-defined triggers address this problem by allowing the user to decide when to check individual constraints. For even moderately complex constraints and updates, this is prone to error, both checking when there is no need, and, more seriously, missing checks that should be made.

**Constraint Simplification or Pre-Test Generation.** Two methods investigated by many researchers has been the generation and use of simplified tests either before (a pre-test) or after (a post-test) the execution of the transaction. These simplified tests need to have the properties of being more efficient than the complete integrity constraints and of implying the truth of the integrity constraint. It is only necessary that the implication of the integrity constraints’ truth be valid when the constraints are met in the input database. In other words, we are trying to verify that the following is true for post-test IC’:

\[ IC(DB) \rightarrow (IC'(T(DB, I)) \rightarrow IC(T(DB, I))). \]

This follows from the assumption that the database evolves by starting in a consistent state and only being updated by consistency-preserving transactions.

IC’ is a post-test since it is applied to T(DB, I). A pre-test is a test that can be applied to the unchanged database and the transaction input to guarantee the consistency of the transaction results given the input. In order for a pre-test to be valid the following must be true:

\[ IC(DB) \rightarrow (IC'(DB, I) \rightarrow IC(T(DB, I))). \]

Some pre-tests only look at the database and others look only at the input. The latter are normally preferable since the input to a transaction is usually small and does not require database access to test.

Let us look at an example, one of the simplest. Suppose a constraint specifies that the predicate P is true of every tuple in relation R. Suppose further that the transaction T consists simply of an insert of a tuple, denoted by the variable i, into R. The following is not true:

\[ P(i) \rightarrow [\forall t \in T(R, i): P(t)]. \]

Remember T(R, i) is the relation R with i inserted.

The following however is a theorem:

\[ [\forall t \in R: P(t)] \& [P(i)] \rightarrow [\forall t \in insert(i, R): P(t)]. \]

Since we know that \[ \forall t \in R: P(t) \] is true, by the assumption that the database is consistent before the transaction, the following is true:

\[ P(i) \rightarrow [\forall t \in R: P(insert(i, R))]. \]

Because of this, it is sufficient to test the predicate P on the input i for the constraint to hold on T’s output state. Thus a predicate applied to all the tuples in a relation can be applied to only a new input tuple and result in integrity maintenance. The power of such a ploy, even as simple as this, is its generality. Consider its use in maintaining unique or primary key constraints.

A primary key constraint stating that K (a column or set of columns) is a primary key of R can be expressed by:

\[ \text{key}(R, K) = \forall t \in R: \text{thekeyof}(t, K) \in \text{project}(\text{delete}(t, R), K), \]

where thekeyof is the function returning the value of the key column or columns and delete(t, R) is the relation R with t removed. The previous theorem allows us to avoid testing the predicate thekeyof(t, K) in project(R, K) on all the tuples of R after the transaction T (that simply inserts a tuple into R), and simply to test the inserted tuple to see if its key value is present in the projection of R on the key column or...
columns.

Note that this example depends on the form of the transaction, a single insert, and the form of the constraint, universally quantified over the tuples of a relation. The cost savings in using this simplification depends on the cost of testing $P$. It is possible that $P$ can be evaluated entirely on the input tuple, or it may be necessary to access multiple tuples from the database, potentially all of the relation as in the key example. Of course, the cost of testing for membership in the relation might be greatly facilitated by an index on the key value. The point to be made is that certain cases are amenable to effective optimization procedures. Much of the work in the area of generating pre-tests, or simplifying constraints as it is also called, (post-tests are not all that popular for obvious reasons), has concentrated on special cases.

We now turn to an even more general form for theorems that can be used to simplify integrity constraints for purposes of simplifying tests needed to maintain the constraints, namely: $\text{COND} \lor [\text{CONSTR}(\text{UPDATE}(D)) = \text{SIMPLER}(D) \land \text{CONSTR}(D)]$

The idea behind this form is that our obligation in maintaining constraints will be expressed by $\text{CONSTR}(\text{UPDATE}(D))$ which stands for the satisfaction of a constraint on some updated data. The updated data is represented by $\text{UPDATE}(D)$ where $D$ is from the state existing before the update. $\text{CONSTR}(D)$ is the constraint on the unchanged data and in the context of database updating will be assumed to be true, corresponding to the assumption that the database is consistent before the update. $\text{COND}$ stands for the conditions under which the equality holds. $\text{COND}$ sometimes needs to be shown to be a consequence of the integrity constraints in order for the equality to be used in generating a pre-test. In other cases, one of which will be presented below, $\text{COND}$ is not a predicate on the database state, but a higher order predicate on function variables. These function variables are given values when higher order functions, such as the relational update given below, are called with concrete tuple updating functions, such as adding a constant to a tuple.

Stonebraker (1975) first showed that a query modification technique exploiting the fact that the database is consistent prior to the update leads to simplification of constraint checking. See (Stonebraker et al., 1976) for a limited implementation and (Simon & Valduriez, 1984) for an improvement of this technique. For transactions that allow at most one update on any relation, Sarin (1977) did a perturbation analysis of the effect of such transaction on the constraint, converting perturbations that falsify the constraint into run-time tests; he offered no algorithms. Nicolas (1982), Bernstein and Blaustein (1981), and Kobayashi (1984), treat the same class of transactions, analyzing the syntactic pattern of the quantifiers in the first-order expression of the constraint; this approach was generalized by Hsu and Imielinski (1985), who treat multiple tuple updates as well as multiple quantifiers.

It is well known that redundant data improves the performance of certain queries. Since integrity constraints are special cases of queries, it is clear that the combination of redundant data (that must be kept cheaply up to date) and the fact that the database is consistent prior to the update (i.e., the result of the last query is known) can be exploited for better test generation. While Bernstein et al. (1980) and Hsu and Imielinski (1985) did take advantage of this idea, it was formalized by Koenig and Paige (1981) (Paige, 1984) through a technique called finite differencing, which replaces costly global re-calculations by more efficient incremental modifications. Their method can deal with aggregate constraints directly.

Gardarin and Melkanoff (1979) introduced a powerful ALGOL-like language (with Hoare-style axiomatic semantics) for transactions that allowed multiple updates as well as updates within a loop construct. They performed manual theorem proving to demonstrate the consistency preservation of transactions, and did not deal with test generation. Henschen, McCune, and Naqvi (1984) first suggested a general approach to test generation through the use of a theorem prover. McCune and Henschen (1989) report an implementation that can produce necessary and sufficient tests. Stemple et al. (1989) have implemented a reasoning system for verifying the consistency preservation of transactions as well as for generation of sufficient and necessary pre-tests (Stemple et al., 1987). They use a high level language called ADABTPL in which all functions (and predicates) are certifiably terminating recursive functions.

Bunker (1986) adopts a symbolic execution approach towards the analysis of transactions during their compilation; his LISP system runs on an IBM PC. salary field. Often $\text{COND}$ is simply true indicating that the equality always holds. Under the assumption that the database is consistent and $\text{COND}$ is true in consistent databases, testing the predicate $\text{SIMPLER}$ on the initial database state is sufficient to assure the consistency of the updated $D$.

The previous theorem used in simplifying a primary key constraint under insertion of a new tuple can be written in this form as $\text{true} \rightarrow [\text{key}(\text{insert}(i, R), K) = (\text{thekeyof}(i, K) \notin \text{project}(R, K)) \land \text{key}(R, K)]$.

A $\text{COND}$ of true means that there are no conditions under which the equality does not hold.

Let us consider a more complex example. Suppose we have an update operation that selectively changes tuples of a relation by replacing each tuple $t$ in the relation with $f(t)$ if $t$ satisfies a predicate $p$. The modification function $f$ must return tuples of the type appropriate for the
relation. Tuples that do not satisfy p are left unchanged. We will write a use of this update as UPDATE(R,p,f).

A useful theorem is that if the application function f does not change the K value of a tuple, then a relation with K as its key will still have K as its key after being updated by f.

\( \forall t : \text{thekeyof}(t,K) = \text{thekeyof}(f(t),K) \implies \text{key}(\text{update}(R,p,f),K) = \text{true} & \text{key}(R,K) \)

The true & is included simply to match the general form above.

This is the best kind of simplification. The theorem validates the simplification of the key constraint to no pre-test at all, i.e., the constant predicate true. In order for this simplification to be valid, the function f must satisfy the precondition \( \text{thekeyof}(t,K) = \text{thekeyof}(f(t),K) \). This condition is a property of the call to the update rather than a property of the database state or input of a transaction. Thus the condition can be checked during the compilation of the transaction. Of course if f is too complicated, the compiler or its constraint system may not be able to verify the precondition. If so, the key constraint will have to be checked on the updated relation at run-time. Most often, f will be a modification of columns not involved in the key and thus will trivially satisfy the precondition.

**Enforcing Aggregate Constraints.** Aggregate constraints are constraints, such as CONSTRAINT12, that require a value in a tuple to be an aggregation of values in some other part of the database. Such constraints lead to the necessity of updating the aggregately constrained data whenever the data upon which it depends is updated. For example, any time a new employee is assigned to a project, his or her salary, times the percentage assignment, must be added to the project’s salary budget. If the transaction simply adds a new assignment without updating the salary budget of the project, the only pre-test that will validate the transaction is a check ensuring that either the salary or percentage of assignment is zero. This is clearly not adequate. The transaction needs to jointly update them and the pre-test generator should recognize that the transaction will maintain the integrity constraint.

Aggregate constraints are amenable to the same techniques as other constraints. The form of the theorems that are useful in this effort is the same form as used above in which no arithmetic was used. This argues strongly for a uniform approach to arithmetic and relational algebra, which is certainly not a feature of current relational languages and systems.

**Acyclicity and Constraints Involving Transitive Closure.** As previously noted, CONSTRAINT11 requiring that the DEPEND relation not be cyclic, is not expressible in relational algebra. It requires a more powerful language to state and therefore can also require more resources to compute. There is a strong temptation to avoid such constraints, even to avoid language features that allow the expression of the computation involved in validating such constraints. On the other hand, the recent efforts to optimize “recursive queries” discussed by Bancilhon and Ramakrishnan (1986) represent attempts to deal with such power in an efficient manner. We argue that such language features can be used in many cases without crippling inefficiencies if integrated into a framework of integrity maintenance such as we have outlined.

In order to show how this might be accomplished, we will use the example constraints, CONSTRAINT11 and CONSTRAINT10, and show that CONSTRAINT11, the constraint expressed in the expanded language, is provably maintained at no cost beyond that incurred by CONSTRAINT10.

Clearly, deletion of a tuple from DEPEND will not create a cycle. Insertion of a new tuple is however problematic. Checking the acyclicity of the relation after an insert by computing the transitive closure and checking for tuples that have the same BEFPROJ and AFTPROJ numbers is clearly prohibitive. A cheaper test can be generated from use of the following theorem:

\[ \text{acyclic}(\text{insert}(t,R)) = \text{nopath}(t.TO,t.FROM,R) & \text{acyclic}(R), \]

where R is a relation with columns FROM and TO, and \( \text{nopath}(a,b,R) \) is true if there is not a series of links from a to b through R.

This states that the only way an insertion could produce a cycle in an acyclic relation is for there to be a path from the destination (the TO component) of the new tuple back to the origin (the FROM component) of the new tuple in the unchanged relation. This allows the reduction of a search for any cycles in the new relation to a search for a single path in the old relation.

While the path search is a significant improvement (from an \( n^3 \) algorithm to linear if the proper access mechanisms are present), even it does not need to be done if a transaction that inserts a tuple performs the much simpler check that assures the maintenance of CONSTRAINT10. Since CONSTRAINT10 is a universally quantified predicate, we need only to check the input tuple when an insert is in question (by virtue of the previously discussed methods). Checking the input tuple involves access only to the two PROJECT tuples referenced by the DEPEND tuple, accesses that would need to be made in any case for referential integrity to be maintained. Such checks are often made quite efficiently for primary keys and so the acyclicity of DEPEND can be guaranteed very cheaply.

It is still a challenge to mechanically verify that a transaction with a check for CONSTRAINT10 also assures the maintenance of CONSTRAINT11. There seems to be no inherent reason why the theorem-based
approach that we have presented, and which works in situations similar to the previous examples, should not be effective for such cases. Developing and proving the appropriate theorems is however more difficult due to the complexity of transitive closure and its related computations. Giving feedback to designers that lead them to discover missing integrity constraints is also a challenge. For example, the transaction given in Figure 3 can produce a planned starting date for a project that is later than the project’s planned end date. This would be missed if the requirement for starts coming before ends was not entered as a constraint. If it is stated as a constraint, the transaction could be changed to have as a precondition a predicate that keeps the constraint from being violated. Another change that would make the transaction preserve consistency is to have it update the end date by an appropriate amount, say the amount of time between the unchanged start and end dates. This would have a ripple effect in that projects dependent on the adjusted project would have to be checked and potentially adjusted. Expressing this open-ended, though finite computation requires the same power in the update language as in a predicate language that allows the acyclic predicate. Clearly more research in this area is called for.

Use of Pre-Tests. The pre-tests discussed in the previous section could be associated with transactions and executed automatically by the run-time control system when a transaction execution is requested. Another option for their use is in the transaction design phase. We could take the stance that a transaction specification should be complete in the sense of being runnable without any possibility of disobeying integrity constraints. All tests would have to be part of the transaction specification. This is in fact the current state of the art for all constraints of any complexity since database management systems handle only a simple set of constraints. The difference in what we mean is that there be specifications of all integrity in a predicate language instead of solely in the procedures of transaction programs. In this setting pre-tests would form part of the feedback to transaction designers and would be included in the corrected transaction specification.

The reason that suggesting the correction of pre-tests is better than their automatic invocation is that the necessity for tests beyond those specified in the transaction may indicate errors in the specification that are not corrected by the addition of the pretests. An example is given by a database in which employees and dependents are described in the database but a dependent must be associated with an employee. A transaction that simply deletes an employee will cause the generation of a pre-test for the emptiness of the set of the employee’s dependents. The designer may have simply forgotten to include the deletion of the employee’s dependents. The suggestion of the pre-test to the designer may cause him or her to add the deletion of the dependents to the transaction specification. The new specification will not have the pre-test generated: it will obey the integrity constraint by the nature of its structure with no extra tests.

This brings us to another means of enforcing integrity—so-called update propagation, the subject of the next section.

Update Propagation and the Design-Time Option. As just discussed, adding tests may not be the right thing to do to a transaction. Missing tests may simply be a symptom of another problem. For example, the transaction writer may have forgotten updates. Integrity constraints often require that updates be grouped in order to rationally update a database to model a real world event.

In order to suggest that certain updates be added, a limited kind of functional unification may be employed. The basic idea (Stemple et al., 1987) is as follows. Given an integrity constraint of the form: IC(r1,r2,...), and a transaction that deletes a from r2 (say), we need to prove that the constraint holds after the transaction, i.e., that the expression: IC(r1,delete(a,r2), ...) is provably true. If we cannot demonstrate that, we pose the question, “for what value of the function variable !x, is the expression: IC(!x(r1),delete(a,r2), ...) true?” The variable is pattern matched against available theorems to find a suitable alternative.

Conclusions

Much research has been done on the problem of integrity maintenance. There are still intense efforts being made in the area, though attention has shifted largely from traditional databases to deductive databases. It is not clear how much pressure there is in the marketplace for the development of products that offer effective means of assuring integrity. Development of an effective, practical integrity constraint system that can assure the maintenance of constraints specified as a part of a conceptual schema in a high level language requires both a high level, reasonably formal, and uniform schema and transaction language. Uniformity is required to keep the mechanical reasoning tractable. It will not be possible to assure with a high degree of reliability that a system maintains integrity of complex constraints if the system is specified using languages with the lack of uniformity evidenced by SQL (Date, 1989), or when mixed language approaches, such as embedding query languages in programming languages, are the primary means of specifying transactions. In order to produce efficient transaction execution, it may be required to manually transform some parts of these high level transaction specifications into programs writ-
ten in some programming language; other parts may be compilable and optimizable using transformational techniques beyond those needed for programming languages. A layered approach to the generation of the programming language parts of a database system specification (Borgida et al., 1989; Lingat & Rolland, 1989; Stemple, 1989) could be very effective.

Acknowledgements

This paper is based in part on work supported by National Science Foundation under grants DCR-8503613, IRI-8606424 and IRI-8822181, and by the Office of Naval Research University Research Initiative contract number N00014-86-K-0764.

REFERENCES


David Stemple received his PhD in Computer and Information Science from the University of Massachusetts in 1977. He is currently an associate professor of Computer and Information Science at the University of Massachusetts at Amherst. His research interests are in database theory, database programming languages, and distributed operating systems.

Eric Simon is currently a researcher at INRIA, Rocquencourt, France, where he heads a team on the design and implementation of deductive database systems. His major areas of research include integrity control and query optimization in database systems.

Subhasish Mazumdar received his Bachelor of Technology degree in Electronics and Electrical Communication Engineering from the Indian Statistical Institute. He received his Master of Science degree from the University of Massachusetts in Computer Science and is currently completing his PhD there. His dissertation explores the analysis of database transactions for enhancement of database integrity and security.

Matthias Jarke received diplomas in business administration and computer science and a PhD in economical sciences from Hamburg University. He is currently on the faculty of mathematics and computer science of the University of Passau and a member of the ESPRIT DAIDA team. His interests are in the design and optimization of database interfaces and in the design and implementation of data-intensive applications.
Related Content

Database Pointers in Navigational and Object–Oriented Database Management Systems: A Comparison
www.irma-international.org/article/database-pointers-navigational-object-oriented/51155/

Visualization Tools for Big Data Analytics in Quantitative Chemical Analysis: A Tutorial in Chemometrics
www.irma-international.org/chapter/visualization-tools-for-big-data-analytics-in-quantitative-chemical-analysis/198789/

Information Analysis in UML and ORM: A Comparison
www.irma-international.org/chapter/information-analysis-uml-orm/4334/

Dataveillance and Panoptic Marketspaces
www.irma-international.org/chapter/dataveillance-panoptic-marketspaces/11141/

Object-Process Methodology Applied to Modeling Credit Card Transactions
Dov Dori (2001). *Journal of Database Management* (pp. 4-14).
www.irma-international.org/article/object-process-methodology-applied-modeling/3257/