The Integrity Challenge

Paul Ammann and Sushil Jajodia
Centre for Secure Information Systems
George Mason University
Fairfax, VA 22030-4444
USA
{pammann, jajodia}@gmu.edu

1. INTRODUCTION

Traditionally, system designers view integrity as Boolean: a system either has it or it doesn't. In practice, of course, no real database enjoys complete integrity: invariably, a variety of factors conspire to make some information stale, missing, or just plain wrong. System design and analysis should recognize that real systems lack integrity, to some degree, most, if not all, of the time. Significant benefits flow from such recognition. Risk management techniques can identify the severity of different integrity loss scenarios, thereby focusing scarce resources on critical areas. A designer can deliberately sacrifice nonessential integrity under carefully controlled conditions to achieve other design objectives, such as performance, autonomy, availability, or security. Designers can achieve these objectives and still assure the preservation of essential aspects of integrity. Advanced applications can use explicit integrity markers on data to make integrity-dependent decisions. Powerful techniques can recognize and avoid impending integrity losses, or, failing that, recognize and contain damage and subsequently help restore integrity. The Internet revolution demands a new perspective on integrity, since the closed world assumption of traditional database design simply does not apply. Instead, data flows from a variety of new and evolving sources, some more trustworthy than others. Indeed, making sense of this torrent of information is one of the fundamental challenges of the present computing era. In this position paper, we outline some recent developments in integrity, and then outline some yet to be solved challenges.

2. FACTORS THAT UNDERMINE INTEGRITY

Inadvertent error ranks high on the list of factors that undermine data integrity. If both the operator and the underlying system fail to recognize a problem, a transaction that shouldn't commit does anyway. Erroneous data, once recorded, often defies repair efforts. For example, many organizations - for sound audit and accountability reasons - have policies against the deletion of data; instead an

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35501-6_14

M. E. van Biene-Hershey et al. (eds.), *Integrity and Internal Control in Information Systems*

attachment to the erroneous record explains and corrects the error. A change in organizational policy, a transfer of data to a different organization, or a software upgrade then brings the error back to life. Designers need effective methods to undo transactions long after they have committed.

Even in cases amenable to a formal capture of integrity constraints, a system designer may choose deliberately to avoid enforcing them during execution, thereby opening the door to integrity violations. For example, strict enforcement in the form of serializability of execution histories reduces performance for long duration transactions. Strict enforcement in the form of atomicity of commit protocols diminishes availability in distributed databases. Strict enforcement in the form of maintenance of global integrity constraints blocks local autonomy in federated databases. Strict enforcement in the form of atomicity of transaction execution degrades security in multilevel databases. Designers need models that formalize explicitly the tradeoff of integrity for other desirable system properties so as to enable an engineering analysis of the decisions and their consequences.

Systems model the real world - imperfectly, of course. A database that satisfies formally specified integrity constraints nonetheless diverges, sometimes seriously, from reality. Many factors inhibit synchronization with the external world. Aside from the initial modeling problem, both the real world and the system evolve over time. Experience has shown keeping up with the evolution time-consuming and error-prone. Even worse, some systems evolve due to malicious activity. Attacker goals range from personal, such as fraudulently acquiring goods or services, to global, such as bringing down parts of the national information infrastructure. Designers need new ways to address the incompleteness and inconsistency that crops up in real systems.

In the Internet, the integrity problem is compounded by the distributed and decentralized nature of the data. Data sources are typically owned and maintained by parties outside the control of the designers for some particular system. As these data sources evolve, designers and users must continuously update their interpretation of the data sources or face a decay in integrity.

In summary, maintaining absolute integrity defies system designers best efforts, and so approaches based on a black and white view of integrity necessarily invite problems. Fortunately, researchers are developing techniques to recognize and even exploit deviations in integrity. For example, a radical approach turns a thorny problem - the inference problem [6] - into a solution. In the inference problem, attackers glean information that the designers intended to hide. For example, by aggregating multiple tables, making repeated queries, or populating a statistical model, an insurer may infer that a prospective client has a certain expensive medical condition, and subsequently deny coverage. Inference attacks succeed because even though the designer can hide explicit data, the designer often cannot hide fundamental patterns and properties of the hidden data. This relatively poor record

of defeating inference attacks makes them excellent candidates for preserving and/or reestablishing integrity. For example, summary data and outlier information can detect anomalies in data subject to degradation or attack. Further, the same summary data and outlier information can increase availability by supplying approximate values during the repair phase.

In succeeding sections, we look at several approaches to integrity in more detail, and then address the challenges posed by integrity for the Internet.

3. TRADING INTEGRITY AND OTHER DESIGN GOALS

One approach relaxes integrity requirements, thereby defining more states as consistent (e.g., see [5,10,12]). If transactions produce outputs within certain tolerances of the outputs from corresponding serial schedules, then execution proceeds, even if it produces a nonserializable schedule. For example, a withdrawal transaction may proceed if at most one other transaction concurrently accesses the same account. Such an approach can yield useful approximate answers in other circumstances. For example, in a stock transaction system, a buy or sell decision may proceed with the information that a stock's value lies in a certain range even if the concurrently executing transactions make the exact value unavailable.

A different approach replaces the syntactic criteria used to maintain integrity - transactions are atomic, map consistent states to consistent states, and proceed in isolation - with semantic criteria tailored to a particular application domain. The resulting property-oriented approach allows limited inconsistency in intermediate states, but ensures consistency in final states and also shields users from the inconsistencies [3]. From the integrity viewpoint, the approach guarantees just those integrity constraints needed at each particular point in a transaction's execution. The designer uses a mathematical formulation to capture and reason about the degree of inconsistency tolerable. The resulting increase in concurrency among transactions translates into improved performance for the user. Related applications of the method result in improved autonomy, availability, or security.

In the model, a database is specified as a collection of objects, along with some invariants or integrity constraints on these objects. At any given time, the state is determined by the values of the objects in the database. A change in the value of a database object changes the state. The invariants are predicates defined over the objects. A database state is said to be consistent if the values of the objects satisfy the given invariants.

A transaction is an operation that transforms one database state to another. Associated with each transaction is a set of preconditions and a set of postconditions. A precondition limits the database states to which a transaction can be applied. For example, a *Reserve* transaction has a precondition that a hotel have

at least one room available. A postcondition constrains the possible database states after a transaction completes. For example, a *Reserve* transaction has a postcondition that there be some room available before the reservation that is taken after the reservation. Postconditions also constrain outputs. Together, preconditions and postconditions must ensure that if a transaction executes on a consistent state, the result is again a consistent state.

For performance reasons, instead of executing a transaction as an atomic unit, it is common to break a transaction into steps and execute each of these steps as an atomic unit. A decomposition of a transaction is a sequence of steps. In place of the transaction, the steps execute atomically in order. A transaction that has been decomposed into two or more steps is referred to as a multistep transaction.

One possible approach to decomposition is to treat the steps as transactions. In particular, one could insist that the integrity constraints hold after each step, which is the decision taken in the Saga model [9]. But such a requirement is too strong for some applications, which need a formal model that can accommodate the notion that some - but not all - violations of the invariants are acceptable.

Figure 1 illustrates the standard model. Figure 2 illustrates a model that allows inconsistent states - as defined by the invariants - that are nonetheless acceptable. The temporary inconsistency introduced by a transaction that enters the 'doughnut' region of the figure is allowed, and steps of other transactions that can tolerate the are allowed to proceed. One general approach is to modify the original set of invariants and decompose transactions such that each step satisfies the new set of invariants. The model in figure 2 has many advantages, including greater concurrency among steps.

Let I denote the original invariants, and let **ST** denote the set consisting of all consistent states; that is, **ST** = $\{ST : ST$ satisfies $I\}$. In the standard model, a transaction T_i always accesses a consistent ST in **ST**. If ST_i denotes the state after the execution of T_i, then ST_i is also in **ST**. When T_i is broken up into steps S_{i1}, ..., S_{in}, each step S_{ij} executes atomically. If ST_{ij} represents the state resulting from the partial execution of T_i through step S_{ij}, it is possible that ST_{ij} no longer satisfies the invariants I and so lies outside **ST**.

A new set of invariants I relaxes the original invariants I. Now each transaction is decomposed such that execution of any step results in a state that satisfies I. Let ***ST*** = { ST : ST satisfies I }. The relationship between **ST** and ***ST*** is shown in figure 2. The inner circle denotes **ST** and the outer circle denotes ***ST*** (signifying that **ST** is a subset of ***ST***. The 'doughnut' denotes the set of all states that satisfy I but not I. The important part about figure 2 is that the set of inconsistent but acceptable states is formally identified and distinguished from the states that are unacceptable. The advantage is that formal analysis can be used to investigate activities in ST.

To reason about decomposing transactions into steps and to avoid the problems of a naive decomposition, we use auxiliary variables to generalize the invariants. Auxiliary variables are a standard method of reasoning about concurrent executions [11] and, in particular, have been applied to the problem of semantic-based concurrency control [8]. The auxiliary variables are introduced for purposes of analysis; the goal is to eliminate such variables from an implementation.

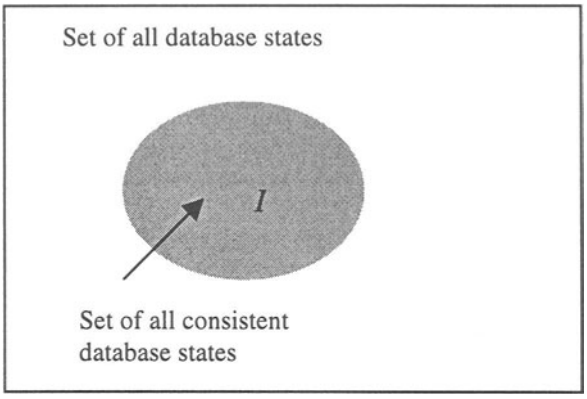

Figure 1. Standard Classification of the Database States

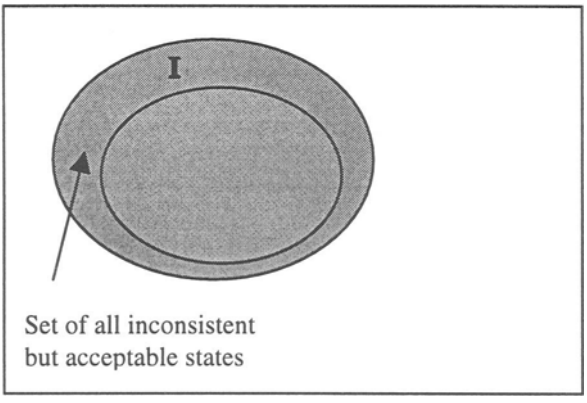

Figure 2. Database States as Classified in our Model

To reason about decomposing transactions into steps and to avoid the problems of a naive decomposition, we use auxiliary variables to generalize the invariants. Auxiliary variables are a standard method of reasoning about concurrent executions [11] and, in particular, have been applied to the problem of semantic-based concurrency control [8]. The auxiliary variables are introduced for purposes of analysis; the goal is to eliminate such variables from an implementation.

The model described above discards three of the four properties from the traditional ACID model for databases, namely atomicity, consistency, and isolation. Only durability is unaffected. These three syntactic properties are replaced by three semantic ones. Atomicity is replaced by semantic atomicity, which states that all of the steps of a transaction eventually execute, or none appear in the final history. Typically, this is achieved by compensation. Consistency is replaced by semantic consistency, which states that after all partial transactions have their final steps execute, the integrity constraints I again hold. Finally, isolation is replaced by semantic isolation, which keeps sensitive transactions from viewing any inconsistency with respect to the original invariants.

The approach outlined above shows that intentional integrity degradation can allow other goals, such as performance, autonomy, or security, to be achieved. The accompanying formalization allows these tradeoffs to be made with more confidence and less likelihood of error.

4. RESTORING INTEGRITY

Compensation accommodates a loss of integrity by directly repairing damage [8]. Because other transactions may access or update the state after a transaction commits, syntactic undo methods do not apply to committed transactions. Instead a compensation transaction attempts to repair semantically the effects of an erroneously committed transaction, and possibly the effects of transactions that read from it. Some actions, such as dispensing cash or firing a missile, complicate or defy satisfactory compensation within a system. In the former case, a bank can compensate erroneously dispensed cash by establishing a loan, as in overdraft protection. In the latter case, compensation occurs outside the system, for example by courts or diplomatic means.

The semantic approach also applies to the problem of restoring a database after the identification of a set of undesirable, but committed, transactions. The key transformation - which uses limited deviations from the specified integrity constraints - rewrites the execution history so that undesirable transactions appear as near to the end of the history as possible, thereby minimizing rework.

For a serial history we can augment the history with explicit database states so that the result is a sequence of interleaved transactions and database states. In rewriting histories, the general goal is either to move bad transactions towards the end of a history or to move good transactions towards the beginning of a history. It turns out that the transformations do not necessarily result in a serializable history which is conflict-equivalent or view-equivalent to the original history [4]. The lack of serializability is justified by the observation that bad transactions ultimately must be backed out anyway along with some or all of the affected transactions. Hence the serializability of such transactions is not a requirement.

The model above shows a novel approach to integrity restoration. The model has the flavor of fault tolerance, because it assumes that faults will cause damage. The goal is to minimize consequences and restore integrity. The model has unexpected consequences. For example, serializability is not necessarily required for integrity restoration.

5. INTEGRITY MARKERS

The self stabilization model [1,7] makes a good starting point for another approach that revisits the notion of consistency and tags data with explicit consistency markers. Informally, a self-stabilizing system is guaranteed to return safely to normal operation following some abnormal interference. We illustrate the approach in the database context with a damage marking approach to delineate degrees of damage and repair, a notion of consistency built on the damage markings, a formalization of system behavior organized similar to self-stabilization, and the requirements for consistency maintenance that this approach places on transactions.

In this approach explicit integrity grades - correct, acceptable, wrong but usable, unacceptable, and so forth - mark data. Which integrity constraints apply depends on the markings of the referenced data. Correct data should conform to the full set of integrity constraints; progressively more damaged data should conform to progressively weaker integrity constraints. Sophisticated algorithms identify integrity deviations, mark inconsistent data, track and contain the spread of inconsistency, inform users as to the reliability of query results, manage repair, and oversee a return to normal service. This fault-tolerance approach to integrity enables systems to survive malicious information-warfare attacks [2].

Data markers maintain precise information about detected integrity losses. Below, we assume that the losses are due to malicious behavior, although the model applies in other situations as well. We show a four part classification; more complex schemes are certainly possible. The four markings are:

1. Red - damaged data that is unsafe to use
2. Yellow - damaged data whose use is nonetheless essential
3. Green - approximate data
4. Blue - normal data

Damaged data is untrustworthy; the value of damaged data is determined by the attacker. We assume that damaged data is incorrect and thus in need of repair. The damage in Red data is so serious, and the consequences of using Red data so severe, that the goal is to prohibit access to Red data.

Yellow data is damaged, but the cost of prohibiting access to such data outweighs the cost of allowing access, auditing the spread of the damage, and subsequently recovering. One motivation for the Yellow marking is if the damage is not mission

critical. Allowing use of the Yellow value permits transactions that read the damaged field to proceed. At the same time, use of the altered value clearly requires audit. One motivation for the Yellow marking is a honeypot, where an attacker is deliberately allowed access so that information about the attacker may be collected.

Green data is not necessarily correct. For example, an approximate value may violate an integrity constraint. The goal of approximate data is to allow on-the-fly repairs where the correct value is not known or available, but an approximate value satisfactory to the application is known. Green values are never those supplied by the attacker, but rather replacement values for data damaged by the attacker. Backup versions are an example of approximate data.

Blue is the marking for data where no damage has been detected. Of course, entries with undetected damage may be marked Green or Blue as well; there is no way in general of avoiding such a possibility. As in ordinary databases, data may also be incorrect for reasons unrelated to any attack, even if the data is marked Blue.

Markings are possible at different granularities. At a very fine granularity level, the value for a single attribute in a tuple may be marked. Markings are sometimes more appropriate at coarser granularities, specifically tuples, attributes for tables, and entire tables. The integrity constraints can influence the choice of granularity.

We wish to provide an alternative to the standard all or nothing view of integrity constraints, namely, that a consistent database satisfies its constraints and a corrupted database does not. The starting point for such an alternative is to revisit of the standard notion of database consistency. We revise the definition for consistency to accommodate the color markings. The main formal result is a syntactic protocol for transactions that preserves the revised notion of consistency [2].

The model redefines the notion of consistency with respect to integrity constraints. Here there are two sets of integrity constraints. The first set, denoted I, is the 'normal' set of integrity constraints. If all data referenced by an integrity constraint in I is Blue, then we require the integrity constraint to hold. The second set, denoted I, is the set of integrity constraints describing values that are acceptable, although not necessarily consistent with respect to I. Integrity constraints in I are required to hold for all data that is either Green or Blue. Since it is unreasonable to expect integrity constraints to hold for data supplied by the attacker, integrity constraints do not apply to Red or Yellow data. Again, the relationship between I and I is shown in figure 2.

Definitions can be constructed so that the integrity constraints in I are (possibly) weakened versions of the integrity constraints in I. That is, if the integrity constraints in I are satisfied for some state of the database (regardless of the markings on the data), then the integrity constraints in I must also be satisfied for that state. Put succinctly, I implies I. Note that the closure property of self-

stabilization [1], namely that activity stays within I, is implicit in the model described here. We view the convergence property, namely that activity must return to *I* upon cessation of hostile activity, as too strong in general in the present context, and so do not include it. Clearly, for some systems, convergence may be possible.

In standard databases, normal transactions need only map consistent states to consistent states, where consistency has the usual definition. For the approximate values of Green data to be useful, a normal transaction that accesses such a value must be prepared to maintain the extended notion of consistency. Consistency preservation for transactions only forces the developer to consider the first two parts of the definition of consistency. Specifically, although the developer must consider the integrity constraints in I as well as those in *I*, the developer is not obligated to consider the large number of possible damage scenarios. This simplifies the developer's job.

The discussion above illustrates the utility of making integrity explicit. Damage tags allow for decisions to explicitly consider damage when determining whether and how transactions should access different data sources.

6. INTEGRITY AND THE INTERNET

The Internet compounds the integrity issues discussed so far by virtue of its decentralized and distributed nature. It is unlikely for a designer to delineate a system boundary that includes part of the Internet and still retain control over system behavior. The reason is that the data sources from the Internet are largely autonomous. Services offered at a particular point are almost certain to evolve due to changing technology and demands of other users. Often these changes represent improvement, but even so, the changes can adversely affect integrity. For example, consider an application that assumes that its data sources are updated at most once a day. The application might rely on this assumption when querying the data source over an extended period of time. If the data source changes to more frequent updates, the multiple queries no longer reflect a single consistent state, and the application may well be in trouble.

Temporal aspects of data also require attention from the integrity perspective. It might seem that purely objective data such as satellite images might be safe from integrity degradation, but the real world evolves, for example, due to human development or natural processes such as hurricanes.

The dynamic nature of the Internet poses a different challenge. New web sites continuously appear, often obviating the role played by established web sites. The old web site may disappear or offer a degraded or different service. Also, the increase of information available on the web substantially changes the meaning of a

query over time. Consider the same search query submitted on dates one year apart. The responses are strikingly different.

One implication of these observations is that integrity management in the Internet must be a dynamic activity. The old model where a system designer defines integrity criteria at system conception and then verifies that a system's design enforces those criteria is not adequate. We do not argue that the old model should be discarded, but for integrity to be maintained, it must be complemented with a dynamic aspect. System integrity requires continuing attention throughout the system's operational life. Researchers should direct attention to how this might be accomplished.

The marking notion described earlier already exists in various forms on the Internet. Some organizations grade sites using their own criteria. Thus, if a user's needs are closely related to the grading organization's criteria, the grading mechanism serves a useful integrity function. For example, parents can look for the 'ABC's of Parenting' star rating when searching the Internet, thereby increasing their confidence in the resulting data.

The 'seal' notion attached to web pages reflects the 'neighborhood' notion of integrity common in the Internet. Top-down hierarchies, while they certainly exist, are often less useful than informally organized collections of web sites that cross reference each other. These collections often reflect a common set of interests or goals; clearly this is related to integrity, but there are as yet no formal criteria for analysis.

7. SUMMARY

Since no real system enjoys absolute integrity, corresponding models for these systems should explicitly address integrity losses. For example, damage markers make it easier to provide continued trustworthy service in systems under information warfare attacks, and inference methods can restore lost information. Also, differentiating essential from nonessential integrity frees designers to deliberately sacrifice the latter for improved performance, autonomy, availability, or security. Rethinking integrity promises significant returns on the investment, particularly in the Internet, where there is as yet no consensus on a basic definition for integrity, let alone on mechanisms for implementing it.

8. REFERENCES

1. Arora and M. Gouda. Closure and convergence: A foundation for fault-tolerant computing. *IEEE Trans. on Software Engineering, 19(11):1015-1027, Nov*ember 1993.

2. Paul Ammann, Sushil Jajodia, Catherine D. McCollum, and Barbara T. Blaustein. Surviving Information Warfare Attacks on Databases. *Proc. IEEE Symp. on Security and Privacy,* pages 164-174, Oakland, CA, 1997.
3. Paul Ammann, Sushil Jajodia, and Indrakshi Ray. Applying formal methods to semantics-based decomposition of transactions. *ACM Trans. on Database Systems.* 22(2):215-254, June 1997.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.
5. E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. of the ACM,* 17(11), November 1974.
6. Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, MA, 1983.
7. Weimin Du and Ahmed K. Elmagarmid. Quasi serializability: A correctness criterion for global concurrency control in Interbase. *Proc. Very Large Data Base Conf.*, pages 347-355, Amsterdam, Neatherlands, 1989.
8. H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems.* 8(2):186-213, June 1983.
9. H. Garcia-Molina and K. Salem. Sagas. *Proc. ACM SIGMOD Int'l. Conf. on Management of Data,* pages 249-259, San Francisco, CA, 1987.
10. Narayannan Krishnakumar and Arthur J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. on Database Systems.* 19(4):586-625, 1994.
11. S. Owicki and D. Gries. Verifying properties of parallel programs: An Axiomatic Approach. *Comm. of the ACM,* 19(5):279-285, May 1976.
12. Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. *Proc. ACM SIGMOD Int'l. Conf. on Management of Data,* pages 377-386, Denver, CO, May 1991.