FAST MAINTENANCE OF SEMANTIC INTEGRITY ASSERTIONS USING REDUNDANT AGGREGATE DATA*

Philip A. Bernstein
Barbara T. Blaustein
Edmund M. Clarke

Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138

Abstract

Semantic integrity assertions are predicates that define consistent database states. To enforce such assertions, a database system must prevent any update from mapping a consistent state to an inconsistent one. In this paper, we describe an enforcement method that is efficient for a large class of relational calculus assertions. The method automatically selects minima and maxima of certain sets to maintain as redundant data in the database. This redundant data is sufficient for enforcing all of the assertions in the class, yet it can be easily maintained. Correctness proofs are expressed in Hoare's program logic.

1. Introduction

Accuracy is an important property of any database. One way to prevent inaccurate data from being stored in a database is to use *semantic integrity assertions*. These assertions are predicates on database states; a database state is *consistent* with these assertions if all assertions hold in that state. By defining a collection of semantic integrity assertions, a user specifies consistent states. The database system is responsible for ensuring database consistency by rejecting updates that produce inconsistent states.

The main components of an implementation of semantic integrity assertions are a *specification language* for defining assertions and *enforcement algorithms* for guaranteeing database consistency relative to those assertions. Expressive power is an asset for such a language, since it allows many types of constraints to be stated; but it is also a liability, since complicated assertions are often expensive to enforce. One language that is richly expressive is relational calculus [Codd 72]. However, since many applications do not need the full power of relational calculus to express semantic integrity assertions, and since arbitrary relational calculus assertions can be quite expensive to enforce, we focus on a restricted class of assertions. Our restricted class is sufficiently general to express many common assertions, yet simple enough to be enforced efficiently.

*This work was supported by the National Science Foundation under grants MCS-77-05314, MCS-79-07762, and MCS-79-08365,

Efficient enforcement depends not only on the complexity of the assertions, but also on the structure of the database. One method for improving the efficiency of enforcement algorithms is to augment the database, D, with stored redundant information, D', that summarizes the contents of D. If D' is cleverly designed, it will contain sufficient information for testing the consistency of most assertions during updates. However, D' itself must be kept consistent relative to the database, D, it is intended to describe. So, there is a trade-off between the work saved during consistency testing by exploiting D' and the extra effort required to keep D' consistent with respect to D. For D' to be effective, its benefit for consistency testing must exceed the cost of maintaining it.

We have adopted the use of redundant data to reduce the cost of testing consistency. The redundant data that we typically add to the database is aggregate information that characterizes a set of values in the database, such as the greatest lower bound of a set. We test consistency using the stored aggregate data rather than all the individual values in the set. The aggregate information is designed to be quickly accessed and easily maintained.

The enforcement method that is the subject of this paper includes: A formal definition of the class of assertions it can enforce; a procedure that selects the appropriate aggregate information to store for each assertion in the class; a procedure that determines the proper run-time test for each type of update and assertion; and a procedure that generates an efficient program for maintaining the correctness of the redundant aggregate information during database updates. Each of these procedures requires little more than a table look-up. The method requires no mechanical theorem proving, and can exploit the full capabilities of the database system's query processor (as in [Stonebraker 75]).

This method represents a qualitatively different approach to integrity enforcement than other published methods. We do not simply incorporate heuristics in a general purpsoe integrity enforcement mechanism and apply the heuristics whenever they seem cost effective. Rather, we define a class of assertions for which the heuristic--maintaining aggregate data--is virtually guaranteed to be cost effective. We can then conclude that any assertion in our class will be enforced efficiently by out method.

CH1534-7/80/0000-0126$00.75

Section 2 defines the database model and the restricted class of assertions we consider in this paper. Section 3 presents algorithms to generate fast consistency tests. We use Hoare's program logic [Hoare 69; Hoare and Wirth 73] to prove that these tests are sufficient to guarantee consistency. Implementation issues of accessing and maintaining aggregate data are discussed in Section 4. Finally, in Section 5, we compare our approach with previous work and argue that our approach has low cost.

2. Modelling Databases, Assertions and Updates

2.1 Relational Data Model

We use relations as our underlying data model. A database is described by a *database schema*, which consists of a set of relation schemas. Each *relation schema* consists of a relation name, say $\underline{R}$, and a set of attributes, say $\{A_1,\ldots,A_n\}$, and is denoted by $\underline{R}(A_1,\ldots,A_n)$. An example database schema that we use throughout this paper appears in Fig. 1.

A *state* of a relation schema $\underline{R}(A_1\ldots,A_n)$ is a relation, R, which is a subset of $dom(A_1)\times\ldots\times dom(A_1)$, where $dom(A_i)$ is the *domain* of values for A_i. A *database state* D of database schema $\underline{D}=\{\underline{R}_1,\ldots\underline{R}_n\}$ is a set of relations $\{R_1,\ldots,R_n\}$ where R_i is a state of $\underline{R}_i$, i = 1,...n.

2.2 The Assertion Language

We express assertions in a language much like relational calculus [Codd 72]. The symbols of our language include

variables
- -relation symbols (e.g., R,S);
- -tuple variable symbols (e.g., r, which denotes a tuple of a relation);
- -indexed tuple variable symbols (e.g., r.B, which denotes the B attribute of tuple r);

parameters
- -constant symbols (including "true", "false", and the rational numbers);
- - function symbols, including arithmetic functions (e.g., +,×,-);
- -predicate symbols, including arithmetic relations (e.g., =,≤ , etc.);
- -the quantifiers ∀ and ∃ ;
- -boolean operators (e.g.,¬ , ∧,∨ , ⇒).

Assertions are well-formed formulas (abbr. wffs) as in relational calculus, where terms are indexed tuple variables and constants and clauses are formed in the usual way. (Unlike relational calculus, the range of a quantifier can only be a single relation.) A[x/y] denotes a wff A with x substituted for all occurrences of y,

A *structure* for our language interprets the parameters and assigns a universe to the variables. It assigns a value to each constant symbol, a function to each function symbol (with the standard interpretation of arithmetic function symbols), a relation to each predicate symbol (with the standard interpretation of arithmetic relations), and a set of relations to each relation symbol (the set of possible states of each relation schema). An *interpretation* of our language includes a structure and a database state. In what follows, we assume a fixed structure; only the database state can change as a result of program execution.

Example 1 - Assertions

(a) English assertion: No item may be sold at a loss.
Assertion:* ∀buys∈BUYS ∀sells∈SELLS
(buys.ITEM=sells.ITEM ⇒ buys.COST≤sells.PRICE)

(b) English assertion:Items can only be bought by cases.
Assertion: ∀buys∈BUYS ∃packs∈PACKS
(buys.ITEM=packs.ITEM∧buys.QUANTITY÷packs.#PER-CASE)
where x ÷ y ≡ x is an integer multiple of y. □

(We will use ≡ to abbreviate "is defined as")

Figure 1
An Example Database Schema

DATABASE SCHEMA: $\underline{D}$ = {BUYS,SELLS,PACKS}
RELATION SCHEMAS:
BUYS(INVOICE#,DEPT,ITEM,QUANTITY,COST)
An invoice entry records a department buying a quantity of an item at a certain cost per item.

SELLS(INVOICE#,DEPT,ITEM,QUANTITY,PRICE)
An invoice entry records a department selling a quantity of an item at a certain price per item.

PACKS(ITEM,CASE-TYPE,#-PER-CASE)
A certain number of items are packed in each type of case (e.g., 'economy', 'jumbo', etc.)

ATTRIBUTES:

Attribute name	Domain
DEPT,ITEM,CASE-TYPE	Alphanumeric strings
INVOICE#,QUANTITY,#-PER-CASE	Nonnegative integers
COST,PRICE	Positive real numbers with two decimal places

2.3 A New Class of Assertions

Our assertion language is very powerful, making it potentially quite expensive to preserve the consistency of an arbitrary assertion. The purpose of this paper is to demonstrate methods for preserving the consistency of a restricted class of assertions, called two-free assertions.

An assertion is *two-free* if it is of the form: 1)∀r∈R∀s∈S(P(r,s) ⇒ r.A≤s.B), or 2) ∀r∈R∃s(P(r,s) ∧r.A<s.B), or 3) ∃r∈R∀s∈S(P(r,s) ∧ r.A≤s.B), where r.A and s.B have the same underlying domain, P is a wff, r and s are the only free tuple variables in P,

* We use ∀buys ∈ BUYS(...) to abbreviate the more complex but formally correct (∀buys)(BUYS(buys) ⇒(...)).

and no bound tuple variable in P has the same range as r or s.* □

We will only consider updates to relations whose tuple variables are free in P. In our examples, P is a function of r and s only. If R and S are the same relation, we perform tests for an update to R and for an update to S.

The assertions in Ex. 1 are two-free. In Ex. 1b,

∀buys ∈BUYS ∃packs ∈PACKS(buys.ITEM=packs.ITEM
∧ buys.QUANTITY ÷ packs.#-PER-CASE),

integer division defines a partial order. While we could denote the partial order by the symbol $\leq$, we use ÷ to avoid confusion with the standard arithmetic ordering.

In this paper, "assertion" refers only to two-free assertions. Assertions of forms 1, 2, and 3 are called ∀∀-assertions, ∀∃-assertions, and ∃∀-assertions, respectively. We fix the order of the quantifiers and require that the r quantifier precede the s quantifier.

2.4 Simple Updates

In this paper we consider only *simple* updates: single-tuple insertions and single-tuple deletions. An in-place modification of an existing tuple is modelled currently by a deletion followed by an insertion; techniques for handling these updates directly will appear in a future paper. We model updates with assignment statements. Given a tuple r_0 and a relation R, "R:=R ∪ r_0" denotes an insertion of r_0 into R, and "R:=R- r_0" denotes a deletion of r_0 from R. (We have dropped the usual set brackets, "{}", from enclsing r_0 to avoid confusion with the notation of Hoare's logic that follows.) Assignments to R have no effect on other relations in the database.

2.5 Hoare's Logic

We use Hoare's program logic [Hoare 69; Hoare and Wirth 73] to analyze the effects of updates on assertions. Formulas in the logic include formulas of the assertion language and formulas of the form P{u}Q, where P and Q are formulas and u is a program. In our case, u will always be a database update. A *formula* P{u}Q *is true in an interpreation* I=($\mathscr{S}$,D), where $\mathscr{S}$ is a structure and D is a database state, denoted $\models_I$P{u}Q, if whenever *precondition* P is true in I before the update then *postcondition* Q is true in ($\mathscr{S}$,u(D)), where u(D) is the database state after u executes. The logic is a set of axioms and inference rules that permits us to determine, whenever provable, if a formula is true in all database states (see Fig.2). We use $\vdash$P{u}Q to denote that the formula P{u}Q is provable in the logic.

Figure 2
Axioms of Hoare's Program Logic

General form: $\dfrac{E_1 E_2 \ldots E_n}{E}$ if $E_1 \wedge E_2 \wedge \ldots \wedge E_n$ then E

Assignment Axiom: $\vdash P[y/x]\{x:=y\}P$

Composition Axiom: $\dfrac{\vdash P\{Q_1\}R_1 \quad \vdash R_1\{Q_2\}R}{\vdash P\{Q_1;Q_2\}R}$

Conditional Axiom: $\dfrac{\vdash P \wedge B\{S\}Q \quad \vdash P \wedge \neg B \Rightarrow Q}{\vdash P\{\text{if } B \text{ then } S\}Q}$

Alternative Axiom: $\dfrac{\vdash P \wedge B\{S_1\}Q \quad \vdash P \wedge \neg B\{S_2\}Q}{\vdash P\{\text{if } B \text{ then } S_1 \text{ else } S_2\}Q}$

Consequence Rule: $\dfrac{\vdash P\{Q\}R \quad \vdash S \Rightarrow P \quad \vdash R \Rightarrow T}{\vdash S\{Q\}T}$

It follows from the soundness of the logic that if $\vdash$P{u}Q , then for all database states D, $\models_{(\mathscr{S},D)}$ P{u}Q [Clarke 79].

A database state D is *consistent* with an assertion A iff $\models_{(\mathscr{S},D)}$A. An update u *preserves the consistency* of D with respect to (abbr. w.r.t.) A iff $\models_{(\mathscr{S},D)}$A{u}A. We say u *preserves* A if, for all database states D, u preserves the consistency of D w.r.t. A. Note that if $\vdash$ A{u}A, then u preserves A.

We assume that the database state is consistent prior to the update. ∃∀-assertions are the only ones for which the empty database state is inconsistent. In this case only, we assume that consistency tests for each update are suppressed until an initial consistent state is reached

3. Determining Whether Updates Preserve Consistency

3.1 General Strategy

One way to test that an update, u, preserves the consistency of an assertion A, in a particular state is to perform u and then evaluate A in the new

* Assertions of the above form prefixed by ∃r∈R∃s∈S can also be handled by our method. However, extra technical machinery is required to do so. Since examples of such assertions are few and, for the most part, contrived, we choose not to discuss them in this paper.

state. If the new state is consistent, then the update is backed-out, thereby undoing its effects.

In view of this potential back-out, it may be preferable to test that u preserves A before u is acutally executed. To accomplish this, we construct a *consistency test*, t, that, for each database state, D, determines whether u preserves D w.r.t. A. We can check that we correctly constructed t by proving the theorem: $\vdash A\{\underline{\text{if}}\ t\ \underline{\text{then}}\ u\}A$. This theorem verifies that t is a correct test for all database states. (If t(D) = false, then u is not executed and the database state is unchanged.) We adopt this strategy of testing consistency before permitting the update. We note that this strategy is essentially the one used in the query modification method proposed by Stonebraker [75].

To *enforce* an assertion A, a database system must provide a consistency test for each update. Assuming the enforcement method is a compile-time algorithm that cannot access the database state, then enforcement amounts to an algorithms that maps each assertion. A and update u into a test t, such that $\vdash A\{\underline{\text{if}}\ t\ \underline{\text{then}}\ u\}A$ and $\vdash A\{\underline{\text{if}}\ \neg t\ \underline{\text{then}}\ u\}\neg A$. For the tests in this paper, the proof of $\vdash A\{\underline{\text{if}}\ t\ \text{then}\ u\}\neg A$ should be clear from the proof of $\vdash A\{\underline{\text{if}}\ t\ \underline{\text{then}}\ u\}A$.

To determine a test t for A and u, we could begin by finding the weakest precondition sufficient to ensure the truth of A after u executes, denoted wp(A,u) [Dijkstra 76]; so, $\vdash wp(A,u)\{u\}A$. However, wp(A,u) assumes we know nothing about the database state before u executes. In fact, we *do* know that A holds in that state. So, we can substitute any test t for wp(A,u) such that $\vdash (A \wedge t) \Rightarrow wp(A,u)$. One method for determining t is to substitute the Boolean constant true in each clause of wp(A,u) that A implies; the resulting formula is a correct test (although not necessarily a "minimal" one).

3.2 Trivial Tests

For some combination of two-free assertions and updates, the assertion implies the weakest precondition. That is, $\vdash A \Rightarrow wp(A,u)$. In this case, the consistency test is trivial-- it is simply true, because $\vdash A\{\underline{\text{if}}\ \text{true}\ \underline{\text{then}}\ u\}A$. A trivial consistency test for a particular assertion and update means that the update preserves the assertion. For such updates, the database system does not need to do any work to enforce the assertion.

Example 2 - A Trivial Test

Assertion: (as in Ex. 1a)
Update: SELLS:=SELLS-$sells_0$, where $sells_0$ is an arbitrary tuple in SELLS
Claim: The update preserves A, so no consistency test is required. Formally stated,
$\vdash A\{SELLS:=SELLS-sells_0\}A$

Proof.

1. $\forall buys \in BUYS \forall sells \in SELLS\ P(buys,sells) \Rightarrow \forall buys \in BUYS \forall sells \in (SELLS-sells_0) P(buys,sells)$
   ; by def. of two-free, there are no variables other than buys and sells bound to BUYS and SELLS in P.
2. $\vdash A \Rightarrow A[SELLS-sells_0/SELLS]$
   ; 1 and def. of substitution
3. $\vdash A[SELLS-sells_0/SELLS]\{SELLS:=SELLS-sells_0\}A$
   ; Assignment axion.
4. $\vdash A\{SELLS:=SELLS-sells_0\}A$
   ; 1,2, and Consequence Rule. □

3.3 Using Stored Aggregates to Simplify Consistency Tests

We can simplify all nontrivial consistency tests further, provided certain aggregate values--minima and maxima of certain domains--are maintained.

Let V be a set whose domain is partially ordered by $\leq$. We define $MIN(V,\leq) \equiv \{v \in V \mid \neg(\exists v' \in V)(v'<v)\}$, where (v'<v) abbreviates $((v' \leq v) \wedge (v' \neq v))$. Similarly, $MAX(V,\leq) \equiv \{v \in V \mid \neg(\exists v' \in V)(v<v')\}$. Note that MIN and MAX are sets, not necessarily singletons. We assume MIN and MAX are non-empty. When $|MIN(V,\leq)|=1$, we use $MIN(V,\leq)$ to abbreviate the unique element in the set (similarly for MAX). When the relevant partial order is clear in context, we drop $\leq$ as a parameter to MIN and MAX.

Example 3 - Using a Stored Aggregate to Simplify a Consistency Test

Assertion: same as Ex.1a.
Update: BUYS:=BUYS$\cup buys_0$,
where $buys_0$=(494,'toy','whistle',100,.20)
Claim: If A is true before the update then
TEST≡$(\forall m \in MIN(\{sells.PRICE \mid sells \in SELLS \wedge sells.ITEM=buys.ITEM\})(buys.COST \leq m))$
is sufficient to ensure consistency. Formally stated
$\vdash A\{\underline{\text{if}}\ TEST\ \underline{\text{then}}\ BUYS:=BUYS \cup buys_0\}A$.

Proof.

1. $A \wedge A[buys_0/BUYS] \Rightarrow A[BUYS \cup buys_0/BUYS]$
   ; defs. of A and U
2. $TEST \Rightarrow A[buys_0/BUYS]$
   ; defs. of A, TEST, and MIN
3. $\vdash A \wedge TEST \Rightarrow A[BUYS \cup buys_0/BUYS]$
   ; 1. and 2
4. $\vdash A[BUYS \cup buys_0/BUYS]\{BUYS:=BUYS \cup buys_0\}A$
   ; Assignment axiom
5. $\vdash A \wedge TEST\{BUYS:=BUYS \cup buys_0\}A$
   ; 3, 4, and Consequence rule
6. $\vdash A\{\underline{\text{if}}\ TEST\ \underline{\text{then}}\ BUYS:=BUYS \cup buys_0\}A$
   ; 5 and Conditional axiom

Since we design our tests to promote efficiency, let us briefly discuss here the cost of this method (a fuller discussion is in Section 5.2). If the minimum PRICE of all 'whistle' tuples in SELLS is available, we only need one comparison to evaluate

TEXT. By constrast, note that query modification [Stonebraker 75] sets out to prove $\vdash A\{BUYS:=BUYS \cup buys_0\}A$ and uses the Assignment Axiom to produce the precondition $\forall sells \in SELLS$ ('whistle'=sells.ITEM$\Rightarrow$.20$\leq$sells.PRICE). Assuming no inverted files, this formula entails searching the entire SELLS relation, checking the ITEM values, and comparing .20 to the PRICE value for every tuple with ITEM='whistle'. If SELLS is inverted on ITEM, then the test must still be made on all 'whistle' tuples in SELLS.

In general, for any two-free assertion A and any simple update u, there is an efficient test t such that $\vdash A\{$if t then u$\}A$. Figure 3 shows the test t for each type of assertion and update. In all cases where t is nontrivial, t relies on a MIN or MAX value that must be maintained as redundant information in the database. Efficient methods for locating and maintaining these MIN and MAX values are discussed in Section 4.

The proof of $\vdash A\{$if t then u$\}A$ for each case included in Fig. 3 is similar to those in Examples 2 and 3; proofs appear in [Bernstein and Blaustein 80].

Figure 3
Consistency test t for assertion A and update u such that $\vdash A\{$if t then u$\}A$

Assertion: $\forall r \in R \forall s \in S(P(r,s) \Rightarrow r.A \leq s.B)$
Update:

$R:=R \cup r_0$	$\forall m \in MIN(\{s.B \mid s \in S \wedge P(r_0,s)\}) r_0.A \leq M$
$S:=S \cup s_0$	$\forall m \in MAX(\{r.A \mid r \in R \wedge P(r,s_0)\}) m \leq s_0.B$
$R:=R-r_0$	TRUE
$S:=S-s_0$	TRUE

Assertion: $\forall r \in R \exists s \in S(P(r,s) \wedge r.A \leq s.B)$
Update:

$R:=R \cup r_0$	$\exists m \in MAX(\{s.B \mid s \in S \wedge P(r_0,s)\}) r_0.A \leq m$
$S:=S \cup s_0$	TRUE
$R:=R-r_0$	TRUE
$S:=S-s_0$	$\forall r \in \{r \in R \mid P(r,s_0)\}(\exists m \in MAX(\{s.B \mid s \in S-s_0 \wedge P(r,s)\}) r.A \leq m)$

Assertion: $\exists r \in R \forall s \in S(P(r,s) \wedge r.A \leq s.B)$
Update:

$R:=R \cup r_0$	TRUE
$S:=S \cup s_0$	$\exists m \in MIN(\{r.A \mid r \in R \wedge P(r,s_0) \wedge \forall s \in S(P(r,s) \wedge r.A \leq s.B)\}) m \leq s_0.B$
$R:=R-r_0$	$\exists m \in MIN(\{r.A \mid r \in R-r_0 \wedge \forall s(P(r,s) \wedge r.A \leq s.B) \wedge \forall n \in MIN(\{s.B \mid s \in S\}) \; m \leq n$
$S:=S-s_0$	TRUE

3.4 Special Cases

The tests in Fig. 3 are sufficiently general to handle MIN and MAX as sets. The cost of performing a test, then, depends principally on the size of the MIN or MAX set. However, in most common cases MIN and MAX each consist of a single value. Consistency tests for these cases require at most on comparison per update.

Lemma 1. *If $\leq$ defines a lattice and X is a finite set, then there is a single value v which is the greatest lower bound of* MIN(X,$\leq$). *Similarly, there is a single value v' which is the least upper bound of* MAX(X,$\leq$). □

Note that Lemma 1 is only useful for $\forall\forall$-assertions.

Lemma 2. *If $\leq$ defines a total ordering, then* $|MIN(X,\leq)| = |MAX(X,\leq)| = 1$. □

Examples 1-3 use a partial ordering that is also a total ordering. The following example applies our strategy to a different partial order, integer division, and illustrates a case where the MAX values are sets.

Example 4 - A Different Type of Partial Order

Assertion: same as Ex.1b.
Update: PACKS:=PACKS-$packs_0$,
where $packs_0$ = ('whistle','economy',100)

Application of Figure 3:
For a $\forall\exists$-assertion and a deletion from S, Fig. 3 gives TEST $\equiv \forall r \in \{r \in R \mid P(r,s_0)\}(\exists m \in MAX(|s.B| s \in S-s_0 \wedge P(r,s)\}) r.A \leq m)$ where $\vdash A\{$if TEST then S:=S-$s_0\}A$.

Substituting BUYS, PACKS and $packs_0$ for R, S, and s_0 respectively, we obtain:

TEST=$\forall buys \in \{buys \in BUYS \mid buys.ITEM='whistle'$
$(\exists m \in MAX(\{packs.\#\text{-}PER\text{-}CASE \mid packs \in PACKS-packs_0$
$\wedge packs.ITEM='whistle'\}) buys.QUANTITY \leq m)$

This simplifies to

$MAX(\{buys.QUANTITY \mid buys \in BUYS \wedge buys.ITEM='whistle'\})$
$< MAX(\{packs.\#\text{-}PER\text{-}CASE$
$\mid packs \in PACKS-packs_0 \wedge packs.ITEM='whistle'\})$

Integer division defines our partial order ($m \div n$ means that n divides m), so MAX contains the least common divisors in the set. We take the MAX (or least common divisors) of QUANTITY values of tuples in BUYS with ITEM='whistle' and try to find an integer divisor for each such QUANTITY value from the set of least common divisors of #-PER-CASE values of PACKS tuples with ITEM='whistle'.

4. Implementation

Having discussed our general strategy, we now focus on implementing a system based on this strategy. For any assertion and any update, Fig.3 gives a consistency test. For trivial tests, the update preserves the assertion, so there is nothing to implement. For nontrivial tests, MIN and MAX values are needed. So, to support nontrivial tests, the system must create MIN and MAX values in the database when an assertion is defined, and must maintain these values during updating.

When an assertion is defined and added to the system, the following steps must be taken.

A1. Augment the data description to include appropriate MIN or MAX sets needed for all nontrivial tests.

A2. Compute these MIN and MAX values.

A3. Test that the new assertion is true in the current database state. When an update is processed, the following steps must be taken for each assertion:

U1. Find the appriate test in Fig. 3.

U2. Locate the correct MIN or MAX value.

U3. Perform the test. If it fails, reject the update. Otherwise perform U4 and U5.

U4. Do any necessary bound maintenance.

U5. Execute the update.

We now explain how to perform each of the above steps.

4.1 Identifying Bounds

Step A1 uses Fig.3 to determine which bounds must be included in the data description for nontrivial tests. Suppose Fig. 3 specifies that a MAX of the set $\{s \in S \mid P(r,s)\}$ is needed to test consistency when some $r \in R$ is deleted. So, the MAX of this set must be incorporated in the database. It appears that each $r \in R$ has its own set $\{s \in S \mid P(r,s)\}$ and its own MAX. Fortunately, fewer sets and MAX's are usually sufficient. The smaller number of sets is obtained by grouping together R tuples that satisfy P for precisely the same S values, since each of these R tuples has the same associated set. For formalize this idea, we define the *equivalence set* of $r_0 \in R$ w.r.t. P in state D to be $P_{r_0}(D)$ $\equiv \{r \in R \mid$ in state D, $\forall s \in S(P(r_0,s) \Leftrightarrow P(r,s))\}$. We will drop D as a parameter when it is clear in context.

Example 5 - Equivalence Sets of Tuples

Assertion: same as Ex.1b.
Update: (as in Example 4) PACKS:=PACKS-$packs_0$,

where $packs_0$= ('whistle','economy',100)

In Ex. 4 we need the MAX of the set {packs.#-PER-CASE $\mid$ packs$\in$PACKS-$packs_0$ $\wedge$ packs.ITEM='whistle'}.

This set is simply the projection of $P_{packs_0}(D)$ on #-PER-CASE, where P=(buys.ITEM=packs.ITEM) and D is the state after the deletion. In words, P_{packs_0} is the set of remaining PACKS tuples with ITEM='whistle'. □

Equivalence sets can be indexed by the attributes referenced in P, called P-*attributes*. Let $A_1,\ldots,A_m$ be all the P-attributes for R. Since P is a formula on indexed tuple variables and constants, each $A_1,\ldots,A_m$ value uniquely identifies an equivalence set of R tuples. Therefore, each equivalence set and its relevant bounds can be indexed by P-attribute values.

Example 6 - Identifying Aggegate Values

Assertion: same as Ex.1a.

All BUYS tuples with the same ITEM value are equivalent with respect to this assertion, as are SELLS tuples. We store bounds of COST values indexed by ITEM values for BUYS tuples and bounds of PRICE values indexed by ITEM values for SELLS tuples. □

Executing A1, then, involves identifying the P-attributes of the relations in the assertion and using these P-attribute values to identify stored bound values. A2 computes the MIN or MAX of all tuples in the relation having the same P-attributes. A3 then compares bound values to test the current state. We proceed in Section 4.2 to show how to decide which values must be compared with each other.

4.2 Locating the Correct MIN and MAX Values

The tests in Fig. 3 show which aggregates to store for assertions of each type. Steps A3 and U2 depend on accessing particular bound values. Once the appropriate values are accessed, A3 and U3 simply compare them. Using equivalence sets of tuples reduces the number of bounds stored, and these bounds are easy to locate because they are indexed by attribute values. The only remaining difficulty is to find pairs of equivalence sets from R and S that *simultaneously* satisfy P. That is, given an assertion and P-attribute values for one relation (the one being updated), we need to find the (set of) P-attribute values in the other relation that satisfy P. In this paper, we assume that each tuple has a unique associated equivalence set in the other relation*. Essentially, P is being interpreted as a query.

*Although our examples deal only with assertions where P is a single equality formula, methods handling general expressions have been developed and will appear in a later paper.

Example 7 - P as a Query

Assertion: same as Ex.1a.
Update: BUYS:=BUYS $buys_0$, where

$buys_0$=(324,'toy','whistle',100,.10]

Before we can compare .10 with the minimum PRICE value, we must evaluate P(BUYS.ITEM=sells.ITEM), with $buys_0$ substituted for buys, to find the ITEM-value in SELLS which indexes the correct equivalence set. Thus, P acts as a query which finds an ITEM-value in SELLS given a tuple in BUYS.

Even had we not used equivalence sets, it would have been necessary to compare .10 with PRICE values for all tuples satisfying ∀sells∈SELLS('whistle'=sells.ITEM→.10≤sells.PRICE). P would have had to be evaluated in exactly the same way. Consistency checking methods must all evaluate the query P and can all use the same mechanism to do so. □

Interpreting P is basic to all consistency testing methods. It is essentially a query optimization problem and can be abstracted from other aspects of consistency verification. We choose to treat it in this way and do not discuss it further in this paper.

4.3 Maintaining Bounds

If an update preserves consistency (the test in U3 succeeds), then we may need to change the bound value of the updated tuple's equivalence set (in step U4). It is not enough for the assertion A to be true after the update; the stored bound value must also be accurate relative to the new database state produced by the update. In effect, we are adding a new precondition and postcondition that describe the accuracy of our bounds. Formally, we must define a formula B that is true in a database state iff the stored bound is accurate in that state. We then augment the given update, u, by another update, u_b, that maintains the consistency of the bounds. That is,

$\vdash A \wedge B\{\text{if } t \text{ then } (u_b;u)\}\ A \wedge B$.

For our method to be cost effective, the cost of bound maintenance must not exceed the savings gained in using those bounds to test the consistency of assertions. So, bound maintenance must be efficient. This efficiency is obtained by combining bound maintenance with the consistency test. This combined activity helps when a tuple update does not affect the bound of its equivalence set. Since consistency only depends on bound values, if the bound is unchanged, then the database must be consistent and no consistency test is needed. In such cases, bound maintenance subsumes the consistency test.

Example 9 - Combining Bound Maintenance and Consistency Tests

Assertion: same as Ex.1a.
Update: BUYS:=BUYSUbuys$_0$, where

$buys_0$ = (434,'toy','whistle',500,.05).

Claim: Let MX be the BUYS aggregate used to test consistency of insertions into BUYS. Let B ≡ (MX=MAX({buys.COST | buys∈BUYS∧buys.ITEM='whistle'})). B is an assertion that describes states in which MX has the intended value . We claim that if A ∧ B hold before the update and the update does not force a recalculation of MX, then no consistency test is required. Formally stated

$\vdash A \wedge B\{$if .05≤MX then BUYS:=BUYSUbuys$_0$
else if .05≤MIN({sells.PRICE | sells∈SELLS∧sells.ITEM='whistle'})
then begin MX:=.05; BUYS:=BUYSUbuys$_0$ end$\}A \wedge B$.

In the above program, if .05≤MX, then the assertion is satisfied, the existing bound (MX) is still correct, and no SELLS tuples need be accessed. If not, a consistency test is performed and, if it yields true, then the stored bound is changed and the update is executed.

Sketch of Proof. Let T1 ≡ (.05≤MAX({buys.COST | buys∈BUYS∧buys.ITEM='whistle'})) and
T2 ≡ (.05≤MIN({sells.PRICE | sells∈SELLS∧sells.ITEM='whistle'})).
The proof follows immediately from
$\vdash A \wedge B \wedge T1\{BUYS:=BUYSUbuys_0\}A \wedge B$ and
$\vdash A \wedge B \wedge \neg T1 \wedge T2\{MX:=.05\}A[BUYSUbuys_0/BUYS]$
$\wedge B[BUYSUbuys_0/BUYS]$, using the axioms in Fig. 2.□

Using techniques such as that of Ex. 8, we have produced algorithms that combine consistency testing and bound maintenance for each type of assertion and update. The algorithms are defined by two procedures: CHECK to test consistency and maintain bounds, and MAINT to maintain bounds only (used for trivial tests). These procedures use two consistency tests, called TEST1 and TEST2, and a recalculation of bounds, called BOUND, which are defined in Fig. 4.

```
CHECK(A,U,TEST1,TEST2,BOUND) ≡
begin
/*first compare tuple with its own equivalence set*/
if TEST1 then U;
/*test against other relations and maintain bound*/
else if TEST2 then (call BOUND;U;)
end

MAINT(A,U,TEST1,BOUND) ≡
begin
/*do maintenance if necessary*/
if TEST1 then call BOUND;
U;
end
```

$MX_{P_{r_0}}$ is defined MAX({r.A | r∈P_{r_0}}); similarly for $MN_{P_{r_0}}$, $MX_{P_{S_0}}$ and $MN_{P_{S_0}}$, and $MN_{P_{S_0}}$. $MN'_{P_{r_0}}$ is defined MIN({r.A | r∈P_{r_0} ∧∀s∈S(P(r,s)∧r.A≤s.B)}).

Each combined consistency check and bound maintenance algorithm given by Fig. 4 and the above procedure definitions ensures that A is true and the stored bound is correct after the algorithm is executed.

Figure 4
Comined Integrity Checking and Bound Maintenance

Assertion type: $\forall r \forall s$

UPDATE: $R:=R \cup r_0$; PROCEDURE: CHECK

TEST1 $\equiv (\exists m \in MX_{P_{r_0}})(r_0.A \leq m)$

TEST2 $\equiv (\forall m \in MIN(\{s.B \mid s \in S \wedge P(r_0,s)\}))(r_0.A \leq m)$

BOUND $\equiv MX_{P_{r_0}} := (MX_{P_{r_0}} - \{m \in MX_{P_{r_0}} \mid m < r_0.A\}) \cup \{r_0.A\}$

UPDATE: $S:=S \cup s_0$; PROCEDURE: CHECK

TEST1 $\equiv (\exists m \in MN_{P_{s_0}})(m \leq s_0.B)$

TEST2 $\equiv (\forall m \in MAX(\{r.A \mid r \in R \wedge P(r,s_0)\}))(m \leq s_0.B)$

BOUND $\equiv MN_{P_{s_0}} := (MN_{P_{s_0}} - \{m \in MN_{P_{s_0}} \mid s_0.B < m\}) \cup \{s_0.B\}$

UPDATE: $R:=R-r_0$; PROCEDURE: MAINT

TEST1 $\equiv r_0.A \in MX_{P_{r_0}}$

BOUND $\equiv MX_{P_{r_0}} := MAX(\{r.A \mid r \in P_{r_0} - r_0\})$

$S:=S-s_0$; PROCEDURE: MAINT

TEST1 $\equiv s_0.B \in MN_{P_{s_0}}$

BOUND $\equiv MN_{P_{s_0}} := MIN(\{s.B \mid s \in P_{s_0} - s_0\})$

Assertion type: $\forall r \exists s$

$R:=R \cup r_0$; PROCEDURE; CHECK

TEST1 $\equiv (\exists m \in MX_{P_{r_0}})(r_0.A \leq m)$

TEST2 $\equiv (\exists m \in MAX(\{s.B \mid s \in S \wedge P(r_0,s)\}))(r_0.A \leq m)$

BOUND $\equiv MX_{P_{r_0}} := (MX_{P_{r_0}} - \{m \in MX_{P_{r_0}} \mid m < r_0.A\}) \cup \{r_0.A\}$

UPDATE: $S \cup s_0$; PROCEDURE; MAINT

TEST1 $\equiv \neg(\exists m \in MX_{P_{s_0}})(s_0.B \leq m)$

BOUND $\equiv MX_{P_{s_0}} := (MX_{P_{s_0}} - \{m \in MX_{P_{s_0}} \mid m < s_0.B\}) \cup \{s_0.B\}$

UPDATE: $R:=R-r_0$; PROCEDURE; MAINT

TEST1 $\equiv r_0.A \in MX_{P_{r_0}}$

BOUND $\equiv MX_{P_{r_0}} := MAX(\{r.A \mid r \in P_{r_0} - r_0\})$

UPDATE: $S:=S-s_0$; PROCEDURE; CHECK

TEST1 $\equiv \neg(s_0.B \in MX_{P_{s_0}})$

TEST2 $\equiv (\forall m \in MAX(\{r.A \mid r \in R \wedge P(r,s_0)\})$
$\exists n \in MAX(\{s.B \mid s \in P_{s_0} - s_0\}))\ (m \leq n)$

BOUND $\equiv MX_{P_{s_0}} := MAX(\{s.B \mid s \in P_{s_0} - s_0\})$

Assertion type: $\exists r \forall s$

UPDATE: $R:=R \cup r_0$; PROCEDURE; MAINT

TEST1 $\equiv \neg(\exists m \in MN'_{P_{r_0}})(m \leq r_0.A)$

BOUND $\equiv MN'_{P_{r_0}} := (MN'_{P_{r_0}} - \{m \in MN'_{P_{r_0}} \mid r_0.A < M\}) \cup \{r_0.A\}$

UPDATE: $S:=S \cup s_0$; PROCEDURE: CHECK

TEST1 $\equiv (\exists m \in MN_{P_{s_0}})(m \leq s_0.B)$

TEST2 $\equiv (\exists m \in MIN(\{r.A \mid r \in R \wedge \forall s \in S\ P(r,s)\})$
$\forall n \in MN_{P_{s_0}})\ (m \leq n)$

BOUND $\equiv MN_{P_{s_0}} := (MN_{P_{s_0}} - \{m \in MN_{P_{s_0}} \mid s_0.B < m\}) \cup \{s_0.B\}$

UPDATE: $R:=R-r_0$; PROCEDURE; CHECK

TEST1 $\equiv \neg(r_0.A \in MN'_{P_{r_0}})$

TEST2 $\equiv (\exists m \in MIN(\{r.A \mid r \in P_{r_0} - r_0$
$\wedge \forall s \in S(P(r,s) \wedge r.A \leq s.B)\})$
$\forall n \in MIN(\{s.B \mid s \in S \wedge P(r_0,s)\}))\quad (m \leq n)$

BOUND $MN'_{P_{r_0}} := MIN(\{r.A \mid r \in P_{r_0} - r_0$
$\wedge \forall s \in S(P(r,s) \Rightarrow r.A \leq s.B)\})$

UPDATE: $S:=S-s_0$; PROCEDURE; MAINT

TEST1 $\equiv s_0.B \in MN_{P_{s_0}}$

BOUND $\equiv MN_{P_{s_0}} := MIN(\{s.B \mid s \in P_{s_0} - s_0\})$

---

5. Comparison with Previous Work

5.1 Comparing Approaches

Few systematic approaches to the implementation of semantic integrity assertions have been published; two well-known examples are the query modification method of Stonebraker [75] and the heuristic program analysis of Hammer and Sarin [78]. Let us compare our method to each of these two.

Comparing our method to [Stonebraker 75], we see three main differences: the types of assertions studied, the role of aggregates in assertopms and the cost of consistency testing.

Our class of assertions is a subset of those studied by Stonebraker [75]. We studied only two-variable assertions with certain forms (two-free assertions). Stonebraker studied assertions with any number of variables and with any logical structure. He divided assertions into categories based on the number of variables in the assertion and on the role in the assertion of the relation being updated. Our class of two-free assertions is not directly comparable to his categories, in that two-free assertions include two-variable assertions from each of these categories. We note that the categorizatcn of assertions by Hammer and McLeod [75] is similar to that of [Stonebraker 75], and the above comments apply to their categorization as well.

The impact of aggregates on consistency tests is markedly different in each method. In query modification, assertions involving aggregates are among the most difficult to test, because testing the assertion requires calculating the aggregate. In contrast, our method maintains stored aggregates; often, it is even cost beneficial to transform assertions without aggregates into assertions involving aggregates.

The cost of testing consistency also differs from method to method. In query modification, the modified update often requires significantly more work for evaluation than does the original update. For example, for multivariate assertions with more than one tuple ranging over the relating being updated, the number of clauses to be evaluated is exponential in the number of variables ranging over the relation being updated. This usually leads to a high cost of evaluating the assertion. Even when the number of clauses is small, the modified update may access many tuples of relations referenced in the assertion.

Example 9 - Update Modification

Assertion: same as Ex.1a.
Update: SELLS:=SELLS∪(432,'toy','whistle',30,.75).
Modified Update: Insert (432,'toy','whistle',30,.75) into SELLS where
∀buys∈BUYS(buys.ITEM='whistle')⇒buys.COST≤.75).

All tuples in BUYS must be checked for ITEM≠'whistle' or COST≤.75 before the insertion is done. If BUYS is inverted on ITEM, then |BUYS[ITEM='whistle']| tuples must accessed, compared to only one tuple in our method. □

Section 5.2 discusses cost comparisons more fully.

Hammer and Sarin [Sarin 77; Hammer and Sarin 78] discuss faster methods of evaluating assertions by using knowledge about the update transaction and the assertion to identify specific conditions which may cause a semantic integrity violation. Testing these conditions is often less costly than evaluating the complete assertion on the current database state. This method depends on an analysis of the particular assertion and update transaction. And, the analytic technique is essentially mechanical theorem proving, which is typically slow. By contrast, our algorithms apply to all simple updates and a given class of assertions. And, they do not require any prior analysis of the actual update transaction; all of the analysis is done *a priori* and can be summarized in a table. At run-time, the analysis is no more complex than a table look-up to obtain the appropriate procedures.

5.2 Cost Estimates

It is difficult to quantify the cost of different integrity enforcement methods, yet this task is essential for precisely comparing them. As with any cost model, it is difficult to capture all the factors which affect the final cost and to assign relative costs to each factor. Integrity enforcement costs cannot be accurately determined independent of an actual machine and database because they depend on such factors as the structure of the assertion to be verified, the type and frequency of updates, the storage structure of the database, and even on the actual values in the database and in the update.

Although we cannot define a general and mathematically precise cost model, we can focus on several of the major factors affecting cost. The role of each of these factors in different verification methods helps to determine the condition under which each method works best. The cost of our method is chiefly dependent on:

1. the type of assertion (∀∀, ∀∃, etc.)
2. the ratio of deletions to insertions
3. the probability that the bound of an equivalent set will not change with each update
4. the average size of equivalence sets, and
5. the cost of evaluating P.

With the caveat that our cost equations are only rough estimates, we proceed to characterize the impact of the above factors. We use the resulting formulas only to help compare relative costs and do not try to derive absolute costs from them.

Cost Constants:

Q_S = cost of evaluating P for an updated tuple r_0, i.e., of finding $\{s \in S | P(r_0,s)\}$

Q_R = cost of evaluating P for an updated tuple s_0, i.e., of finding $\{r \in R | P(r,s_0)\}$

M_R = average size of P_{r_0}

M_S = average size of P_{s_0}

c = cost of comparing ($\leq$) two values
d = cost of one database access
P_R = probability that an update changes an R bound
P_S = probability that an update changes an S bound.

Using the algorithms in Section 4.2 and Fig.4, we can derive cost formulas. We describe the derivation for insertions and deletions of R tuples for ∀∀-assertions, and list the formulas for other types of assertions and updates later. We assume that Lemma 1 or 2 holds, so that we have only bound value for each equivalence set (this usually seems to hold in practice since the underlying domain is the set of integers, reals, etc.) Thus, each MIN and MAX set contains only one element. So, accessing a bound costs d and testing a tuple against a bound costs c. Also, assume that P produces one equivalence set per relation (this is only relevant for ∀∃-assertions.)

The algorithm for inserting r_0 for a ∀∀-assertion is CHECK(A,R:=R r_0TEST1 ,TEST2,BOUND). Filling in the appropriate tests and bound assignments we get:

1. if $\exists m \in MX = MAX(\{r.A \mid r \in P_{r_0}\}) r_0.A \leq m$ then $R := R \cup r_0$;
2. else if $\forall m \in MIN(\{s.B \mid s \in S \wedge P(r_0,s)\}) r_0.A \leq m$
3. then begin $MX := (MX - \{m \in MX \mid m\ r_0.A\}) \cup \{r_0.A\}$;
4. $R := R \cup r_0$;
end

Evaluating Line 1 involves accessing the stored bound and comparing it to r_0.A, for a cost of d+c.

Line 2 is only executed if the bound must be changed ($r_0.A > MAX(\{r.a \mid r \in P_{r_0}\})$), so it only adds to the total cost with probability p_R. Evaluating Line 2 involves evaluating P to find the correct S equivalence set, accessing the stored bound, and comparing it with r_0.A, Thus, line 2 costs $p_R(Q_S+d+c)$.

The rest of the algorithm adds no significant cost. If the test succeeds and Line 3 is executed, we simply store the new bound value. We have already accessed the proper equivalence set, and changing the bound involves no new computation. Line 4 is just the cost of inserting the tuple.

Query modification for insertion to R for a ∀∀-assertion requires evaluating P, accessing each S tuple which satisfies P, and comparing it to the inserted tuple. In our system, the S tuples satisfying P would constitute an equivalence set, so we can denote the number of these S tuples as M_S. Therefore, the cost formula is Q_S+M_S*d+c.

Cost formulas for other types of assertions are: 1) Insert to relation with universally quantified tuple variable, d+c+p(Q+d+c), 2) Insert to relation with existentially quantified tuple variable, d+c, 3) Delete from relation with univerally quantified tuple variable, d+c+p(Mdc), and 4) Delete from relation with existentially quantified tuple variable, d+c+p(Q+Mdc+c). By Fig. 3, the test for deletion for ∀∀-assertions is trivial, so the cost of query modification in this case is 0.

Note that for ∀∀-assertions, insertion and deletion of S tuples are exactly analogous to those of R tuples.

In performing a comparison of the two methods, we make the following simplifying assumptions: 1) each operation (insert to R, insert to S, delete from R, delete from S) occurs with the same frequency; 2) the average equivalence set size is the same for R as for S ($M_R=M_S$; we will use $M=M_R=M_S$);3) the query processor is equally efficient given and R tuple or an S tuple ($Q_R=Q_S$; we will use Q to mean either Q_R or Q_S); 4) c is a unit cost, and evaluating P costs more than a database access, which in turn is more than c (c<d<Q); and 5) the probability that the bound of an equivalence set will change with a given update is 1/M. Assumption 1 allows us to simply add together the costs for each operation. We drop the c in each formula by Assumption 4 and make the simplifications in Assumption 2, 3, and 5.

Query modification: 2(q+Md)

Our method: $2(d+1/m(Q+d)+d+(1/M)Md$
$= 2(Q/M+(3+1/M)d)$

Comparing these formulas, we see that our method is more efficient (given all the assumptions), when the average equivalence set size is 3 or more. Depending on the cost of evaluating P relative to the cost of a database access, our method may also be efficient for M=2.

Under other assumptions, our method may be even more efficient. For example, if Assumption 1 were changed to model a situation where insertions outnumber deletions, our method would compare even more favorably to query modification. On insertions a bound change only requires comparison with the previous bound; it is not necessary to access all the tuples in the equivalence set. Deletions that cause bound changes are more costly, in that each tuple in the equivalence set must be accessed; but even this process only involves accessing the same number of tuples that query modification accessses on each insertion (i.e., the set of tuples which satisfy P). Furthermore, in most cases (p_R<1) bound maintenance will not have to be done for each deletion.

Although the above formulas yield no absolute cost values, they help identify the conditions under which our method is most useful. Whenever stored aggregates of values in an equivalence set can be used frequently to avoid accessing tuples individually, the efficiency of checking integrity offsets the cost of maintaining bounds.

6. Conclusion

The approach to semantic integrity we have described consists of designing a class of assertions that can be efficiently enforced using suitable tactics, and then fully analyzing the compile-time and run-time enforcement algorithms. In this paper, we worked through the analysis for two-free assertions using redundant aggregate data as a tactic. We have carried out this analysis on other classes of assertions with equal success. We believe this approach offers the best hope of developing a semantic integrity subsystem for a database system with acceptable performance.

Acknowledgements

We gratefully acknowledge Marco Casanova, Nathan Goodman, John Smith and Umeshwar Dayal, whose careful reviews of early drafts led to many terminological and organizational improvements. We also wish to thank the referees for their helpful comments and suggestions.

References

[Berstein and Blaustein 80]
Bernstein, P. A.; and Blaustein, B. T., "Efficient Maintenance of Semantic Integrity Assertions Containing Two Tuple Variables", Technical Report Aiken Computation Laboratory, Harvard University, to appear.

[Clarke 79]
Clarke, E. M., "Programming Language Constructs for Which It Is Impossible to Obtain Good Hoare Axiom Systems", JACM 26, 1 (Jan. 1979),129-147.

[Codd 72]
Codd, E. F.,"Relational Completeness of Data Base Sublanguages", in *Data Base Systems*, Courant Computer Science Symposia Seris, Vol.6, Prentice-Hall, Englewood Cliffs, NJ 1972, pp.65-90.

[Dijkstra 76]
Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Hammer and McLeod 75]
Hammer, M. M.; and McLeod, D. J.,"Semantic Integrity in a Relational Data Base System", in *Proceedings First International Conference on Very Large Data Bases*, 1975, pp.25-47.

[Hammer and Sarin 78]
Hammer, M. M. and Sarin, S.,"Efficient Monitoring of Database Assertions", *Proceedings of 1978 SIGMOD Conference on Management of Data*, ACM, NY , 1978.

[Hoare 69]
Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", in *CACM*, Vol.12, No.10, October, 1969.

[Hoare and Wirth 73]
Hoare, C. A. R., and Wirth, N., "An Axiomatic Definition of the Programming Language PASCAL", in *Acta Informatica*, Vol.2, 1973, pp.335-355.

[Sarin 77]
Sarin, S. K., "Automatic Synthesis of Efficient Procedures for Database Integrity Checking", Master's Thesis, Massachusetts Institute of Technology, Sept., 1977.

[Stonebraker 75]
Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification", *Proceedings 1975 ACM-SIGMOD Conference*,pp.65-78.