Data & Knowledge Engineering 10 (1993) 187–223
North-Holland

DATAK 167

Integrity control in relational database systems – An overview

Paul W.P.J. Grefen and Peter M.G. Apers
University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands

Received 28 January 1992
Accepted 25 September 1992

Abstract

Grefen, P.W.P.J. and P.M.G. Apers, Integrity control in relational database systems – An overview, Data & Knowledge Engineering 10 (1993) 187–223.

This paper gives an overview of research regarding integrity control or integrity constraint handling in relational database management systems. The topic of constraint handling is discussed from two points of view. First, constraint handling is discussed by identifying a number of important research issues, and by treating each issue in detail. Second, a number of projects is described that have resulted in the realization of database management systems supporting integrity constraints; the various projects are compared with respect to a number of system characteristics. Together, both approaches give a broad overview of the state of the art in the field at this moment.

Keywords. Integrity control; relational database; survey.

1. Introduction

The complexity of modern database applications requires powerful facilities for controlling the semantic correctness of the data in the database. The consequences of the absence of such facilities are convincingly illustrated in [8, 69], where some striking anomalies in the records of a health organization are presented. The requirement of semantic data control has led to a number of research projects in the context of relational database systems. The first projects started in the mid-seventies, together with the development of the first relational database management systems, like System R and INGRES. The basic issues have become clear by now, but research continues into new areas, like easy specification of constraints and definition of complex constraint types, verification and optimization of constraint specifications, efficient constraint enforcement algorithms and parallelism in constraint enforcement.

This paper focusses on constraint handling in relational systems, but, where interesting, side trips are made to other types of database systems.

1.1. Structure of this paper

This paper gives a broad overview of research performed in the field of semantic data control or integrity constraint handling in the context of relational database systems. This

Correspondence to: P.W.P.J. Grefen, University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands. Email: grefen@cs.utwente.nl

concept is explained in the section below. Section 2 gives a short formal background on databases, integrity constraints, and transactions. This section also introduces the example database used throughout this paper. The research in the field of integrity constraint handling is discussed from two points of view in Sections 3 and 4 of this paper. In the first place, integrity constraint handling is discussed by identifying a number of important research issues, and treating each issue in detail. In the second place, a number of projects are described that have resulted in the realization of database systems supporting integrity constraints; the various projects are compared with respect to a number of system characteristics. Together, both approaches give a broad overview of the state of the art in the field at this moment.

1.2. The integrity concept

As stated above, in a database management system, the correctness or accuracy of data is of great importance. There are a number of ways in which incorrect data may occur in a database. The following disciplines in database technology try to prevent certain classes of errors [35, 51, 31]:

Security control deals with preventing users from accessing and modifying data in a database in unauthorized ways; the security control subsystem of a DBMS keeps record of the authorization of users to perform certain operations on certain data and checks this authorization upon database access [29, 34].

Concurrency control deals with the prevention of inconsistencies caused by concurrent access of multiple users or applications to a database; the concurrency control subsystem orchestrates the access to the database, in most cases using a locking or timestamping technique [29, 59, 34].

Reliability control deals with the prevention of errors due to the malfunctioning of system hardware or software; the reliability control subsystem uses recovery techniques to reinstall a correct database after system crashes [29, 59], and techniques like replication of data to increase the reliability of the system [16].

Integrity control deals with the prevention of semantic errors made by users due to their carelessness or lack of knowledge; the integrity control subsystem uses integrity rules to verify the database and operations on the database.

This paper is concerned only with the integrity control discipline.

The term *integrity* as used throughout this paper refers to the *correctness* or *validity* of the data in the database, as defined explicitly by means of integrity rules or *integrity constraints*.[1] This implies that neither the full correctness with respect to the part of the real world modeled by the database is guaranteed in general, nor the completeness of the facts stored in the database; a detailed discussion of this topic can be found in [67].

1.3. Integrity control allocation

An important question to be answered is who or what is responsible for keeping a

[1] Note that there is quite some confusion in terminology here: the terms *integrity*, *consistency*, *validity*, *correctness*, etc. may be used differently by different authors.

database consistent with a number of specified integrity constraints (integrity control). In general, the task of integrity control can be allocated in three ways:

Application designer The integrity control task can be left to the application designer or ad hoc user of the database system. As such, no robustness with respect to integrity is offered at all: applications and ad hoc transactions are required to be individually correct with respect to the integrity constraints.

Transaction designer The integrity control task can be the responsibility of the transaction designer. This means that transactions are required to be correct with respect to specified constraints. In this situation, applications can only make use of predefined transactions.

Database Management System The database management system can have the responsibility for integrity control. This means that arbitrary transactions can be executed on the system.

In the first case, the integrity of the database is not guaranteed by a central specification of the database itself, but only by the design of the (numerous and ever changing) applications operating on the database. Consequently, the chance of improper constraint control is high. Further, changes of some constraint definitions require modification of all applications including integrity control with respect to these definitions. This is an undesirable situation, although it may occur frequently in practice.

In the second case, the responsibility for integrity control is part of the transaction design process [38, 92]. Again, changes to constraint definitions require modification of all transactions including integrity control with respect to these definitions. In a situation with many constraints, transactions can get very complex. Further, this approach is hardly feasible in a situation with a high rate of ad hoc transactions.

If the integrity control task is allocated with the database management system, the integrity of the database is effectively ruled by a central set of constraint definitions. Applications and transactions can be fully unaware of integrity control. A disadvantage of this approach may be a reduced flexibility with respect to constraint handling (a different handling of constraint violations per application or transaction type is not possible).

The latter two approaches are described in research, and the choice between them depends on many criteria like the application domain, the actual usage of a database system, performance requirements etc. This paper is mainly concerned with fully automatic integrity control with transparency to all users of the database system, and therefore mainly adopts the DBMS-based approach. Some attention is devoted to the transaction-based approach as well, however.

2. A formal background for integrity constraints

As stated above, the integrity of a database is stated explicitly by means of integrity constraints, i.e. rules that define properties to be satisfied by the database. This section gives a formal background on integrity constraints that serves as a basis for the sequel of this paper. First, some elementary database notions are introduced in short. Then, the concept of integrity constraint is introduced in a database context. Finally, the relation between integrity constraints and transactions is discussed. This section concludes with the introduction of a simple example database that will be used throughout the paper to illustrate concepts and techniques.

2.1. Elementary database notions

The database notions to be used in the formal description of database integrity are relation schemas and states, database schemas and states, and database transitions. These notions are defined below.

Definition 2.1. *A relation schema* $\mathcal{R}$ consists of a relation name and a list of attributes $\langle A_1, \ldots, A_n \rangle$. Each attribute A_i is defined on a domain $\mathrm{dom}(A_i)$. The type of $\mathcal{R}$ is defined as $\mathrm{dom}(\mathcal{R}) = \mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_n)$. A *relation* or *relation instance* R of relation schema $\mathcal{R}$ consists of the relation name of $\mathcal{R}$ and a set of elements in $\mathrm{dom}(\mathcal{R})$.

Definition 2.2. A *database schema* $\mathcal{D}$ is a set of relation schemas $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$. A *database* or *database instance* D of database schema $\mathcal{D}$ is a set of relation instances $\{R_1, \ldots, R_n\}$. The set of all possible database instances of schema $\mathcal{D}$ is called the *database universe* $U_{\mathcal{D}}$, so $U_{\mathcal{D}} = \mathrm{dom}(\mathcal{R}_1) \times \cdots \times \mathrm{dom}(\mathcal{R}_n)$.

Definition 2.3. A *database transition* of database schema $\mathcal{D}$ is an ordered pair of database states $\langle D^{t_1}, D^{t_2} \rangle$ of schema $\mathcal{D}$, with $t_1, t_2 \in \mathbb{N}$ and $t_1 < t_2$. The values t_1 and t_2 are the logical times of the database states.

Usually, a database transition describes two successive states of the database, so $t_2 = t_1 + 1$ in the definition above. This type of transition is called a *single-step transition*. If not stated otherwise, the term transition is used for single-step transitions in this paper.

2.2. Databases and integrity constraints

Below, the concept of *integrity constraint* is defined. In this definition, constraints are divided into *state constraints* that describe properties of database states, and *transition constraints* that describe properties of database transitions. This distinction is necessary to come to a correct definition of integrity constraints; a more detailed classification of constraints is discussed in Section 3.

Definition 2.4. Let $\mathcal{D}$ be a database schema. A *state constraint* I^s is a boolean function that is evaluated over a database state D from the database universe $U_{\mathcal{D}}$ defined on $\mathcal{D}$:

$$I^s : U_{\mathcal{D}} \rightarrow \mathrm{bool}\,.$$

Definition 2.5. A *correct database state* $D \in U_{\mathcal{D}}$ satisfies each element of a set of state constraints $\mathcal{I}^s = \{I_1^s, \ldots, I_m^s\}$ defined on $\mathcal{D}$. The set of correct database states with schema $\mathcal{D}$ and constraint set $\mathcal{I}^s$ is denoted as:

$$U_{\mathcal{D}}^{\mathcal{I}^s} = \left\{ D \in U_{\mathcal{D}} \,\middle|\, \bigwedge_{i=1}^{i=m} I_i^s(D) \right\}.$$

A state constraint describes the static properties of a database, i.e. the properties that a database should satisfy at one given moment.

Definition 2.6. Let $\mathcal{D}$ be a database schema. A *transition constraint* I^t is a boolean function

that is evaluated over a pair of database states or database transition $\langle D_1, D_2 \rangle$ defined on $\mathscr{D}$ [2].

$$I^t: U_{\mathscr{D}} \times U_{\mathscr{D}} \rightarrow \text{bool}.$$

Definition 2.7. A *correct database transition* $\langle D_1, D_2 \rangle$ defined on schema $\mathscr{D}$ satisfies each element of a set of transition constraints $\mathscr{I}^t = \{I_1^t, \ldots, I_n^t\}$ defined on $\mathscr{D}$. The set of correct database transitions with schema $\mathscr{D}$ and constraint set $\mathscr{I}^t$ is denoted as:

$$V_{\mathscr{D}}^{\mathscr{I}^t} = \left\{ \langle D_1, D_2 \rangle \in U_{\mathscr{D}} \times U_{\mathscr{D}} \middle| \bigwedge_{i=1}^{i=n} I_i^t(D_1, D_2) \right\}.$$

A transition constraint describes the correct transitions of a database; as such, it describes dynamic properties of a database. Therefore, transition constraints are also referred to as *dynamic constraints*.

2.3. Transactions and integrity constraints

The integrity of the database in terms of integrity constraints as defined above has its effect on the set of transactions that are allowed to be executed and successfully committed against a database.

The fact that each database state has to satisfy all state constraints means that the execution of a transaction T on a correct state D may never result in a database state that violates any constraint in $\mathscr{I}^s$; so the following should hold:

$$\left(\bigwedge_{i=1}^{i=m} I_i^s(D) \right) \Rightarrow \left(\bigwedge_{i=1}^{i=m} I_i^s(T(D)) \right),$$

or in short notation:

$$\mathscr{I}^s(D) \Rightarrow \mathscr{I}^s(T(D)).$$

The fact that each database transition has to satisfy all transition constraints means that given a correct database state, the execution of a transaction T may never imply a transition that violates any constraint in $\mathscr{I}^t$; so the following should always hold:

$$\left(\bigwedge_{i=1}^{i=m} I_i^s(D) \right) \Rightarrow \left(\bigwedge_{i=1}^{i=n} I_i^t(D, T(D)) \right),$$

or again in short:

$$\mathscr{I}^s(D) \Rightarrow \mathscr{I}^t(D, T(D)).$$

Given these observations, we can define the correctness of a transaction as follows.

Definition 2.8. A transaction T is *correct* with respect to a correct database state D and a set of integrity constraints $\mathscr{I}$ if and only if a committed execution of T on D does not imply a database transition that violates any transition constraint in $\mathscr{I}$, and the post-transaction database state $T(D)$ does not violate any state constraint in $\mathscr{I}$. A transaction that does not comply with this requirement is called *incorrect*.

The concept of *correctness* of a transaction is essentially different from the concept of *safeness* of a transaction as defined below.

[2] We limit ourselves here to single-step transitions; some remarks on general transitions can be found in Section 3.1.

Definition 2.9. A transaction T is *safe* with respect to a database schema $\mathcal{D}$ and a set of integrity constraints $\mathcal{I}$ if and only if a committed execution of T on any correct state $D \in U_{\mathcal{D}}$ does not imply a database transition that violates any transition constraint in $\mathcal{I}$, and the post-transaction database state $T(D)$ does not violate any state constraint in $\mathcal{I}$. A transaction that is not safe is called *unsafe*.

Note that the above definition of integrity constraints only considers the database states before and after the execution of transactions. This approach is based on the fact that transactions are considered to be atomic units of operations against a database; intermediate database states during transaction execution have therefore no semantics. Some approaches to integrity control, however, do take the intermediate database states during transaction execution into consideration (see Section 3.2), thereby rejecting the atomicity principle. Other approaches do not use the transaction concept at all, but limit themselves to single operations against the database (sometimes even to single-tuple operations).

2.4. Example database

Below an example beer database is introduced that is used in the examples of this paper. The database schema is shown in *Table 1*; it consists of a relation *beer* describing for each beer the name, brewery, type, and alcohol percentage, a relation *brewery* describing for each brewery the name and location, and a relation *drinker* describing for each drinker his/her favorite beer and the quantities bought and drunk of this beer.

The example constraints are shown in *Table 1* in first order logic notation. The identifiers x and y denote tuple variables; the other identifiers refer to relations and attributes of the example database. Constraint *I*1 is a simple domain constraint on an attribute of relation *beer*; it states that a beer in the database contains at least some alcohol. Constraint *I*2 is a referential integrity constraint from relation *beer* to relation *brewery*, stating that every beer in the database is brewed by an existing brewery. Constraint *I*3 is a unique key constraint, stating that the name of a brewery in relation *brewery* is unique. Tuple constraint *I*4 states that every drinker in the database has bought at least as much beer as he or she drinks. Finally, *I*5 is a dynamic constraint stating that the quantity of beer drunk by a person never decreases. In this constraint, the notation $drinker_{old}$ refers to the pre-transaction state of relation *drinker*.

Table 1
Example database definition

Relation	Attributes
beer	name, brewed_by, type, alcperc
brewery	name, city, country
drinker	name, beer, qty_bought, qty_drunk
Constraint	**Definition**
*I*1	$(\forall x \in beer.alcperc)(x > 0)$
*I*2	$(\forall x \in beer.\ brewed\_by)(\exists y \in brewery.name)(x = y)$
*I*3	$(\forall x, y \in brewery)((x.name = y.name) \Rightarrow (x = y))$
*I*4	$(\forall x \in drinker)(x.qty\_bought \geq x.qty\_drunk)$
*I*5	$(\forall x \in drinker, y \in drinker_{old})$ $((x.name = y.name) \Rightarrow (x.qty\_drunk \geq y.qty\_drunk))$

3. Research topic overview

This section discusses a number of important research topics in the field of integrity constraint handling in relational database systems. These topics can be classified into four categories; these categories and their topics are listed below.

Constraint types and semantics: the various types of integrity constraints discussed in the field of relational database systems; the various approaches to full transaction support in constraint handling, particularly with regard to transaction atomicity.

Constraint specification: the various approaches to the specification of integrity constraints, and the classes of constraint specification languages;

Constraint preprocessing: the verification of newly defined constraints, i.e. checking if new constraints are correct and meaningful with respect to a number of criteria; the translation of constraints from the formalism used for constraint specification to the formalism used for constraint enforcement by the system; the optimization of constraint definitions, i.e. the transformation of constraint definitions to obtain forms that are semantically equivalent, but that can be enforced more efficiently;

Constraint enforcement: the various approaches to the actual evaluation or enforcement of constraints and the execution of violation response actions; the effects of database system distribution and parallel processing capabilities on constraint handling; special system support aiming at improving the efficiency of constraint enforcement, both in terms of software and hardware; the analysis and evaluation of the performance of the constraint enforcement process in transaction execution.

In the sections to follow each of these topics is discussed.

3.1. Constraint types

Various constraint types and their semantics are an important research topic. It is however hard, if not impossible, to design a complete constraint type classification scheme. An overview of important constraint types is given in *Table 2*. Below, some constraint types are discussed in detail.

3.1.1 Domain and nonnull constraints

Domain constraints[3] restrict the values that attributes of tuples can assume. Usually, domain constraints specify a certain range of values for the attribute, like example constraint *I*1. Sometimes, a domain constraint enumerates the possible values for an attribute, like:

$$(\forall x \in beer.type)(x \in \{\text{'}pilsener\text{'}, \text{'}ale\text{'}, \text{'}stout\text{'}\}) .$$

In the relational model, the enumerated values can be stored in a separate, single-attribute relation; the domain constraint has to be implemented then as a referential integrity constraint. In this way, a 'variable enumerated domain constraint' can be defined [9].

Nonnull constraints can be seen as a special case of domain constraints, since they restrict the domain of attributes. Some theoretical work on nonnull constraints can be found in [41].

[3] Some authors (e.g. [37]) use the term *type constraint* instead of domain constraint.

Table 2
Constraint taxonomy

State	Attribute	Domain
		Nonnull
	Tuple	Attribute comparison
	Relation	Uniqueness (key)
		Functional dependency
		Aggregate
		Transitive closure
	Database	Referential integrity
		Interrelation aggregate
Transition	Attribute	Domain
	Tuple	Attribute comparison
	Relation	Aggregate
	Database	Interrelation aggregate

3.1.2 Referential integrity constraints

The concept of referential integrity is a central issue in the relational data model, since it is an important way to specify semantic links between the various relations in a database. Therefore, this type of constraint has received quite some attention, even though it is just another type of constraint from a constraint handling viewpoint.

The original idea of referential integrity is described in [25]. Extensions to this idea are given in [28], leading to more usable semantics for constraints. An important extension is allowing foreign keys to contain null values. The combination of null values and multi-attribute foreign keys can lead to various kinds of problematic situations, however, since it is hard to describe the semantics of referential integrity constructs with foreign key values that are partly null [30].

In [64] the safeness of referential integrity structures is discussed. A structure is considered to be unsafe if it causes certain data manipulation problems as a consequence of the order of enforcement of referential integrity constraints or the order of tuple manipulations in a transaction, or if no correct transition sequence exists that transforms one given consistent database state in another given consistent state.

Referential integrity constraints are even more important if the relational data model is used for the representation of object-oriented structures. This topic is discussed for example in [63].

3.1.3 Recursive constraints

Recursive queries have been the topic of a reasonable amount of research by now, especially transitive closure queries. Integrity constraints with recursive rules have been discussed much less, however. The topic is discussed in [19] in the context of deductive databases with logic programming interfaces. In the context of relational database systems, the topic is described briefly in [46, 47], and mentioned very briefly in [79].

In many applications, the use of constraints with transitive closure constructs is natural; examples are the following:

- in an employee database, one may want to specify that no one can be his/her own chief, either directly or indirectly;

- in a database representing a network (e.g. railroads), one may want to specify that every node in the network is reachable from every other node.

3.1.4 Transition constraints

In most cases, transition or dynamic constraints described in literature are 'single-step' transition constraints, i.e. constraints that are evaluated on a pair of pre-transaction and post-transaction states of a database. Constraint *I5* from the example database illustrates this kind of constraint:

$$(\forall x \in drinker,\ y \in drinker_{old})((x.name = y.name) \Rightarrow$$
$$(x.qty\_drunk \geq y.qty\_drunk)) .$$

Mostly, these constraints are tuple constraints, i.e. they relate old and new attribute values in a single tuple. As illustrated by the above example, the notion of a key (attribute *name* in this case) is important for the specification of these constraints.

An example of a transition constraint on the relation level is the constraint shown below, stating that the number of breweries in the database cannot decrease:

$$COUNT(brewery) \geq COUNT(brewery_{old}) .$$

3.1.5 Temporal constraints

It is possible to describe dynamic constraints that are not limited to single-step transitions. These *temporal constraints* can be specified using the temporal qualifiers *always* and *sometime* and their bounded versions *always . . . until* and *sometime . . . before* [33], or other forms of temporal logic [23].

Two remarks can be made with respect to general temporal constraints. In the first place, one can question the relevance to practice of this type of constraints. In the second place, practical solutions for the enforcement of this type of constraints are either restricted or very inefficient (if possible at all) [33]. A restricted approach is discussed in [23], where a method is described for the enforcement of temporal constraints taking only past database states into consideration. This method makes use of redundant information to be able to enforce temporal constraints without storing the entire history of a database.

3.1.6 Fuzzy constraints

Fuzzy relational database systems are systems that can deal with fuzzy or incomplete data. In these systems, integrity constraints involve fuzzy constructs as well [73]. Examples of fuzzy constraints in an employee database are (taken from [73]):

- Many managers have a high income (fuzzy domain constraint).
- Employees having similar jobs and experience must have almost equal salary (fuzzy data dependency).

The specification of fuzzy constraints requires a specification language including fuzzy operators.

3.2. Transaction support

This section discusses the way constraint handling approaches take transactions into consideration. A number of issues are of interest here. The first issue concerns the enforcement granularity, with which we mean the complexity of update operations against the database the integrity constraint enforcement takes into consideration. The next issue concerns transaction atomicity under constraint enforcement; this is considered important to obtain full transaction semantics. Finally, the concept of integrity checkpoints is described; this concept allows the specification of points in a transaction where the integrity constraints

should be enforced, thereby avoiding the complete rollback of complex transactions. These issues are discussed below in detail.

Note that the concept of nested or multi-level transactions (see e.g. [5]) can make the semantics of constraints more complicated. A proposal for constraints in a multi-level transaction environment is given in [75, 54].

3.2.1 Enforcement granularity

In principle, three levels of constraint enforcement granularity can be distinguished:

Tuple update If the enforcement granularity is tuple update, constraints are enforced after every single tuple update operation. In some approaches this is even limited to insertion and deletions of single tuples [6, 80].

Set update If the enforcement granularity is set update, constraints are enforced after every update operation that may involve the update of multiple tuples.

Transaction If the enforcement granularity is transaction, constraints are enforced at the end of a transaction, possibly consisting of multiple set updates.

3.2.2 Transaction atomicity

Conceptually, the execution of a transaction T should always satisfy the atomicity property; this means that the effect of any execution of T on the initial database state D must be such that either the effects of T are completed fully, or D remains unchanged. So, if T consists of the actions $a_1; \ldots, a_n$, the following must hold:

$$(T(D) = D^{a_n}\downarrow) \vee (T(D) = D)\,,$$

where $D^{a_n}\downarrow$ denotes the database state after the execution of a_n minus the temporary relations created by T. The enforcement of constraints as a consequence of updates performed by a transaction T should not violate this property. To be able to support transaction atomicity, the enforcement granularity as defined above must clearly be at the transaction level. Further, the constraint violation response action must be such that transaction atomicity is not violated. An example of an approach that does not satisfy this requirement is the *query modification technique* [81] (see Section 3.7.2).

In the context of more complex transaction models (e.g. nested transactions [54, 75]), the concept of transaction atomicity may have to be redefined.

3.2.3 Integrity checkpoints

In the case of complex transactions consisting of multiple operations, the abort of the complete transaction as a consequence of a constraint violation may be costly, either in terms of the rollback operation or in terms of the work already performed. Therefore, some approaches to constraint enforcement use the concept of *integrity checkpoints*, which are points in a transaction where all constraints are enforced. When a constraint violation occurs after an integrity checkpoint, the transaction can be undone back to this checkpoint, and does not have to be undone completely. User-defined integrity checkpoints are suggested in the System R approach to constraint handling [91]. It should be noted, however, that a partial transaction rollback does not fit nicely in the concept of transaction atomicity. As described in Section 2.3, the principle of transaction atomicity implies that only the pre-transaction and post-transaction database states are taken into account in determining the correctness of a transaction.

In the SABRE project, integrity checkpoints are used to find points in a transaction where

certain constraints can be enforced, such that they do not have to be postponed until commit time [79, 91]. In this approach, integrity checkpoints are automatically established by the system through an analysis of the transaction and the constraints.

3.3. Constraint specification

The specification of integrity constraints is an important research issue, because the specification formalism can determine both the functionality of the constraint handling mechanism and the userfriendliness in database design. Four approaches to constraint specification can be distinguished:

Language-oriented In the language-oriented approach, constraints are specified in some kind of data definition language. This can either be an extension to the general data definition language of the system or a special-purpose constraint definition language.

Form-oriented In the form-oriented approach, constraints are specified by filling in conditions in forms generated by the system. Mostly, these forms are a representation of existing relations, like in the QBE system [97, 98] (see also Section 4.2).

Toolbox-oriented The toolbox-oriented approach allows the construction of constraint specification with the use of toolboxes providing building blocks and construction primitives for constraints. An example of this approach can be found in the SuperBase system [86].

Graphics-oriented In the graphical approach, constraints are specified by arranging graphical symbols representing constraint constructs in a graphical workspace. An example in the context of entity-relationship schema definition is the graphical schema editor of the SUPER database visual environment [3].

The form-oriented, toolbox-oriented, and graphics-oriented approaches usually concentrate on userfriendliness, whereas the language-oriented approach tends to put more stress on expressive power. Therefore, the language-based approach can be considered the most general. Since user interfaces are not a main topic in this paper, the sequel of this section will be devoted to the language-based approach.

In its most complete form from an operational point of view, the specification of an integrity constraint I is a triplet $[t, c, a]$ with the following elements:

Triggers The *trigger set* t specifies the update types that may violate I;

Condition The *constraint condition* c specifies the boolean predicate that has to be satisfied by either a database state or a database transition;

Action The *violation response action* specifies the action to be executed upon a constraint violation.

In other approaches, like [17, 18], this triplet is denoted in the fully equivalent notation of a production rule:

WHEN t
IF c
THEN a

Note that the specification of triggers is of an operational nature; in general, the triggers can be derived from the constraint rule [17, 18, 46, 47].

Below, the specification of these elements in language-oriented approaches is discussed.

The most important part of a constraint is the condition. Therefore, attention is first paid to a number of condition specification language classes. Next, the specification of triggers and violation response actions is discussed. The section is concluded with a few words on the procedural approach to the specification of constraints; this approach uses an integrated imperative specification of triggers, condition, and violation response action.

3.3.1 Relational algebra based approach

In the relational algebra approach, constraints are expressed in (some extension of) the relational algebra. A common way to express a constraint is to specify a relational expression that calculates the tuples in the database that do not satisfy the constraint. In this way, example constraints $I1$ and $I2$ are expressed as follows:

$I1$: $\sigma_{\neg alcperc>0} beer$

$I2$: $beer - \pi_{name,brewed\_by,type,alcperc}(beer \bowtie_{brewed\_by=name} brewery)$

Some approaches add special-purpose constructs to the relational algebra to be able to specify constraints (see for instance the approach in the PRISMA project in Section 4.6).

3.3.2 Relational calculus based approach

In the relational calculus approach, two variants can be used: the tuple and domain calculus [90]. In tuple relational calculus example constraints $I1$ and $I2$ are expressed by describing the violating tuples as follows:

$I1$: $\{t \mid beer(t) \wedge \neg(t[3] > 0)\}$

$I2$: $\{t \mid (\forall u)(beer(t) \wedge (\neg brewery(u) \vee t[2] \neq u[1]))\}$

In domain relational calculus we have for the same constraints:

$I1$: $\{tuvw \mid beer(tuvw) \wedge \neg(w > 0)\}$

$I2$: $\{tuvw \mid (\forall x, y, z)(beer(tuvw) \wedge (\neg brewery(xyz) \vee u \neq x))\}$

Instead of 'pure' relational calculus, also query languages based on relational calculus can be used. The SQL language is frequently used for constraint specification, for instance in [2, 17]. Using the notation of [17], constraints are expressed by an SQL predicate denoting the tuples that violate the constraints. The example constraints are specified in this notation as follows:

$I1$: *alcperc* < 0

$I2$: *brewed_by NOT IN (SELECT name FROM brewery)*

3.3.3 Logic programming approach

In a logic programming approach, constraints are specified in a Prolog-like syntax. Constraints can be specified by logic clauses that define the correct tuples. The example constraints then can look like:

$I1$: $beer(\_, \_, \_, Z) :- Z > 0$.

$I2$: $beer(\_, Y, \_, \_) :- brewery(Y, \_, \_)$.

In [19] a predicate *incorrectdb* is introduced that states if a database state is incorrect. Using this approach, the example constraints look as follows:

$I1$: $incorrectdb :- beer(\_, \_, \_, Z),\ Z < 0$.

$I2$: $incorrectdb :- beer(\_, Y, \_, \_),\ not(brewery(Y, \_, \_))$.

The logic approach is especially used in the deductive database field. In [69] logic is used for constraints and meta-constraints.

3.3.4 Special-purpose languages

Several proposals have been made for special-purpose languages or language constructs to facilitate the specification of constraints. Some examples are discussed below.

Constraint equations

In [66] the use of *constraint equations* is discussed. Constraint equations are declarative expressions of constraints that require consistency between several relations.

The following example of a constraint equation specifies that the projects of a manager are to be the same as the set of projects his employees work on:

$$manager.project == manager.employee.project$$

Constructs for common constraints

Several approaches to constraint specification use simple special-purpose language constructs for commonly used constraints and provide a general mechanism for other constraints. This approach is used for example in the SABRINA system [40, 91]. Referential integrity constraint *I2* can be specified in this system as follows:

ASSERT REFERENTIAL DEPENDENCY
beer.brewed_by FROM brewery.name

3.3.5 Constraint triggers

Many approaches to constraint specification allow or require the specification of *triggers* with the constraint rule, indicating the types of updates that may violate the constraint. Triggers are used for several purposes:

System simplification Explicit specification of triggers avoids part of automatic constraint analysis by the system. This simplifies the constraint handling system, but puts the burden on the shoulders of the database designer.

Enforcement efficiency Allowing the database designer to specify application characteristics in the form of triggers may enhance the efficiency of constraint enforcement. The database designer may know for example that certain update types will never violate certain constraints in practice.

Enhanced functionality In some cases the possibility to specify triggers can enhance the functionality of the constraint handling subsystem. This is illustrated by the following example on relation *drinker*. If an application requires that only tuples with attribute *qty_drunk* equal to 0 may be deleted, the following constraint may be specified:

on delete drinker: qty_drunk = 0

The use of triggers provides a procedural but simple mechanism for the specification of this type of constraints. If the values of deleted tuples can be addressed explicitly, this constraint can be specified more declaratively.[4]

[4] Using the notation for differential sets as explained in Section 3.6.1, the example constraint can be specified in a logic formalism as follows:

$$(\forall x \in drinker^{-}.\ qty\_drunk)(x = 0)\ .$$

3.3.6 Violation response actions

The default violation response action in most approaches to constraint handling is transaction abort. In some cases, the application may require other actions, such as cascading updates [29]. We take the referential integrity constraint as an example here; one case of this constraint can be specified as:

$$I2 = [t, r, a]$$

$$t = \{insert(beer)\}$$
$$r = brewed\_by\ NOT\ IN\ (SELECT\ name\ FROM\ brewery)$$
$$a = INSERT\ INTO\ brewery\ VALUES\ (brewed\_by,null,null)$$

Note that the specification of a violation response action has imperative semantics. Therefore, a purely declarative view on constraints cannot be used when actions are specified. This complicates the semantics of constraint enforcement.

A different approach to specifying constraint violation actions is the specification of integrity dependencies [61]. These dependencies denote which variables in constraints can be modified upon a violation detection (dependent variables) and which cannot (independent variables). Take constraint *I4* of the example database as an illustration of this technique to describe violation response actions. The definition of this constraint is the following:

$$(\forall x \in drinker)(x.qty\_bought \geq x.qty\_drunk)$$

If *qty_bought* would be declared the dependent variable and *qty_drunk* the independent variable, a violation of this constraint would be repaired by increasing the value of *qty_bought* to that of *qty_drunk*.

3.3.7 Procedural approach

The procedural approach to the specification of integrity constraints uses programming language concepts to build 'constraint enforcement procedures'. This kind of constraint specification is of an imperative nature, whereas the approaches to the specification of rules as discussed before are of a declarative nature. An example is the way in which constraints are specified in the SYBASE system [64, 88]. A partial specification of example integrity constraint *I2* is given in *Table 3*. The shown trigger procedure specifies the effect of the deletion of *brewery* tuples. It will be clear that this approach to constraint specification is less friendly and more error-prone than the declarative approach.

Table 3
Example constraint in SYBASE

```
create trigger delete_brewery on brewery for delete as
begin
  declare @del_beer int
  select @del_beer = count(*)
  from deleted, beer
  where deleted.name = beer.brewed_by
  if @del_beer > 0
  begin
    raiserror 1 "deletion of brewery failed"
    rollback transaction
  end
end
```

3.4. Constraint verification

When new integrity constraints are defined on a database, it has to be checked if the constraints themselves are valid syntactically and semantically. This process is called *constraint verification* here. The verification of a new constraint includes the following issues (see also [89, 10]):

- Checking the syntactic correctness or *well-formedness* of the constraint with respect to the syntactic rules for specifying constraints.
- Checking the semantic correctness of the individual constraint. Relevant aspects are for example: do all used relations and attributes exist, are comparisons made only between compatible operands[5]?
- Checking if the constraint is not violated by the database state at the time the constraint is defined. If the constraint does not hold at definition time, it is likely that no update on the database will succeed any more. Clearly, this check can only be performed for state constraints.
- Checking if the constraint is not implied by already existing constraints. If so, the new constraint is superfluous. Although a redundant set of constraints is semantically correct, it is undesirable from viewpoints of efficiency and constraint maintenance.
- Checking if the constraint is not contradicted by already existing constraints, i.e. checking the consistency of the constraint set. If the set of constraints is inconsistent, no valid database states or transitions can exist if the constraint is accepted. Note that for state constraints, checking the constraint successfully against the current database state is sufficient to prove that there is no contradiction.
- Checking if the violation response action does not trigger an infinite process of compensating actions due to the interaction with other constraints (see e.g. [18]).

The relevance of the first three issues can be seen as generally accepted. These checks should be supported by every real world database system supporting constraints and pose no fundamental problems. The last three issues are mainly discussed in the research area. Some fundamental problems have to be solved here.

3.4.1 Consistency of a set of constraints

The issue of consistency of a set of constraints has received some attention in literature.

Practical work on constraints and meta-constraints defining the validity of a set of constraints is described in [69]. A method to detect a contradiction between constraints is described in a logic programming context. The approach is rather informal and based on a single application.

A formal treatment based on logic of constraint set consistency is given in [11]. A constraint set is consistent when it is satisfiable. This work distinguishes between *finitely satisfiable* and *infinitely satisfiable*; finitely satisfiable implies the existence of a finite model for the set of constraints, whereas infinitely satisfiable implies the existence of infinite models only. Constraint sets are considered acceptable only when they are finitely satisfiable, because databases can have finite contents only. The following set of constraints defined on a personnel database is an example of a consistent but not finitely satisfiable set (taken from [11]):

- everybody works for somebody;
- nobody works for himself;
- if x works for y and y works for z, then x works for z.

[5] This issue is related to the concept of *underlying domain* as introduced in the ALPHA language [24]. This concept allows the database designer to specify that certain attributes are not compatible, even though they have the same basic domain.

3.4.2 Finiteness of compensating actions

The issue of infinite compensating actions is addressed in [17, 18]. The problem of guaranteeing termination of the constraint enforcement process in case of compensating violation response actions is described to be 'almost certainly undecidable in the general case'. Therefore, the infinite action detection algorithms perform a worst case analysis of the constraint violation actions using a *triggering graph*, and issue a warning in case of potential cyclic behavior. The actual decision is further left to the discretion of the database designer.

3.5. Constraint translation

The requirements on languages for constraint specification by a database designer and constraint enforcement by the database system are generally different. Specification languages focus on ease of use and declarative semantics, whereas enforcement languages require efficiency and imperative semantics. Therefore, a translation between the two types of languages may be necessary. The approach as described in [43, 46] is an example. Here, constraints are specified in first-order logic and enforced in an extension to the relational algebra.

Further, some systems allow users to use special language constructs to define commonly used constraints in an easy way. These language constructs have to be translated to the general constraint specification formalism (or directly to the enforcement formalism).

3.6. Constraint optimization

The execution cost of constraint enforcement is one of the major problems in the field of constraint handling. Therefore, constraint optimization is of great importance. The topic has many similarities with query optimization (see e.g. [16]).

Various types of optimization can be applied to integrity constraints:

- The amount of data to be checked in constraint enforcement can be reduced by evaluating the constraints only on the relevant parts of relations.
- The constraint rules can be manipulated algebraically to obtain equivalent rules that can be evaluated cheaper.
- In a distributed system with fragmented relations, fragmentation knowledge can be used to reduce the number of fragments used in a constraint.
- The constraint rules can be transformed using semantic knowledge about the application being modelled by the database and about the database itself.

These topics are discussed below in more detail.

3.6.1 Data reduction

A reduction of the amount of data to be checked in constraint enforcement can be obtained by inspecting only those parts of relations that have been changed in a relevant way. This is usually accomplished by the use of *differential sets* [79, 40, 43]. In this approach, each relation consists of three sets:

Base set The base set contains the tuples that have not been changed by the current transaction. For a relation R, this set can be denoted as R^0.

New set The new set contains tuples that have been inserted and the new values of tuples that have been modified by the current transaction. The new set is usually denoted as R^+.

Old set The old set contains tuples that have been deleted and the old values of tuples that have been modified by the current transaction. The old set is usually denoted as R^-.

Using these sets, constraint rules can be reformulated. Take as an example the domain constraint rule r_1 below that can be reformulated into r_2:

r_1: $(\forall x \in beer.alcperc)(x > 0)$
r_2: $(\forall x \in beer^+.alcperc)(x > 0)$.

Differential sets can be easily implemented using indexes or markings on the tuples in a relation to mark the sets. The principle of data reduction is also used in [17, 18].

3.6.2 Syntactical rule manipulation

Various approaches to the manipulation of integrity constraint rules are described in literature. In [68] the simplification of integrity constraints in first-order logic is described. The basic principle of the described method relies on the instantiation of formulas obtained by substituting for some of their variables the (or some of the) constants occurring in the inserted (or deleted, or updated) tuple. The technique is only applicable to transactions updating multiple tuples, if the order of the operations in a transaction is immaterial. In [58] the simplification of constraints in the context of general transactions is described. The simplification is based on the prefix of the constraint (quantifiers). In [43, 47] the rewriting is discussed of constraints on fragmented relations specified in a first-order logic formalism. The rewriting consists of pushing quantifiers through set unions and is comparable with pushing operations through unions in relational algebra as described in [16].

3.6.3 Usage of fragmentation knowledge

Similar to the use of fragmentation knowledge in query optimization [16], fragmentation knowledge can be used for the optimization of constraints in systems with fragmented relations [43, 47]. An important example is a referential integrity constraint between two horizontally fragmented relations. If the referencing relation is fragmented on the referencing attributes (foreign key) and the referenced relation is fragmented on the referenced attributes (key) using the same fragmentation algorithm, the enforcement of the referential integrity constraint can be reduced from a pairwise checking of all combinations of fragments of the two relations to a pairwise checking of the compatible fragments only.

3.6.4 Semantical rule manipulation

Constraint rules can be manipulated by applying semantic knowledge about application and/or database. A knowledge-based approach to constraint optimization and enforcement is described in [71]. In this approach, a constraint manager keeps a knowledge base with semantic knowledge about application and database. The knowledge is extracted from the database schema definition, from user-defined constraints, and from the results of continuous monitoring of the database state. The knowledge is used to optimize and enforce constraints. The ideas are, however, a long way from practical applicability.

3.7. Constraint enforcement

This section first discusses the constraint enforcement strategy: prevention or detection of constraint violations. Next, attention is paid to the handling of the various kinds of system responses to constraint violations.

3.7.1 Violation prevention versus violation detection

Essentially, there are two strategies to enforce integrity constraints [79, 40]:

Violation prevention is used, if integrity constraints are enforced before the updates of a

transaction are actually applied to the database. In case of a violation, the transaction is aborted, and there is no need to undo any changes to the database.

Violation detection is used, if integrity constraints are enforced after the updates of a transaction have actually been applied to the database. In case of a violation, the transaction is aborted, and the changes performed by the transaction to the database have to be undone.

From a complexity point of view, the violation detection technique is preferable to the prevention technique, since updates can be applied to the database, the constraints can be evaluated against the database, and standard recovery techniques can be used to undo updates in case of a violation. For this reason, the detection technique is used in most of the earlier systems, like DB2 [91]. From a performance point of view, the prevention method is generally preferable, since it eliminates the need to undo changes to a database, which is costly in a disk-based environment. In a main-memory database system, however, this does not hold. Further, in a main-memory system like PRISMA/DB [95, 1], it is hard to distinguish between the database storage and the transaction workspace, since both are managed by the same processes in the distributed memory of the system. In such a situation, it is questionable if the distinction between prevention and detection has to be made (see below).

3.7.2 Violation prevention techniques

Constraint violation prevention can be carried out with a number of rather different techniques. These techniques are described in detail below.

Transaction analysis

One approach to constraint violation detection is to analyze update transactions statically, i.e. without taking the database state upon which they will be executed into consideration. This process is called verifying the safeness or consistency of the transaction. As stated in Section 2, a transaction T is *safe* or *consistent* with respect to a set of integrity constraints $\mathcal{I}$ and a database schema $\mathcal{D}$, if it cannot violate the integrity of any database state conforming to the schema:

$$safe(T) \Leftrightarrow (\forall D \in \mathcal{D})(\mathcal{I}(D) \Rightarrow \mathcal{I}(D, T(D))) .$$

If the transaction is found to be consistent, it can be executed against the database without any further constraint enforcement. If the transaction is found to be inconsistent, it is simply rejected. In [38], the consistency of transactions is analyzed with Hoare's axiomatic approach to program correctness. In [92] a knowledge-based approach is described. A positive aspect of the technique is that it can be applied at transaction definition time and does not imply any constraint enforcement overhead at transaction execution time. An evident shortcoming of the technique is, however, that it cannot be used for arbitrary transactions and constraints, since it does not take the dynamic properties of the database into account.

Query modification

In the *query modification* approach to constraint enforcement, updates to be performed on the database are modified such that they cannot violate the integrity of the database [81]. After modification, the updates can be executed without any checks. Updates are modified by extending the update qualification predicate with a predicate derived from a constraint definition. Take as an example relation *beer* and domain constraint $I1$ as described in Section 2.4. The following update statement:

UPDATE beer
SET alcperc = alcperc − 1
WHERE brewed_by = "Geineken"

is modified to the following statement:

UPDATE beer
SET alcperc = alcperc − 1
WHERE brewed_by = "Geineken" AND alcperc > 1

The query modification approach has two major drawbacks. In the first place, the approach is not applicable to all constraint types [77]. In the second place, it does not comply with the transaction atomicity principle since it merely 'filters out' tuples violating a constraint, but does change the other tuples [46, 47]. If the *beer* relation contains the tuples shown in *Table 4*, the update shown above changes two tuples of brewery *Geineken*, but leaves one tuple as it is, and is thus not executed atomically as defined originally.

Transaction workspace

Using the transaction workspace approach to constraint violation prevention, the updates to be performed to the database are carried out in a temporary workspace of the transaction. Next, the constraints are evaluated on this workspace and, if necessary, the database. If a constraint violation is detected, the updates in the workspace are simply not propagated to the database. If possible, the transaction workspace is kept in main memory to avoid unnecessary disk accesses. The transaction workspace method is used in the SABRE/SABRINA project [79, 40]; details can be found in Section 4.4.

3.7.3 Violation response actions

When an integrity constraint violation occurs, the system has to execute a *violation response action*. In general, three classes of response actions can be distinguished:

Error message The most simple (but least satisfactory) approach to handling a constraint violation is issuing an error message to the user or application causing the violation. Note that this approach fully relies on the user to get the database back into a consistent state. Constraints that trigger this type of violation response action are called *soft assertions* in [35].

Transaction abort A commonly used approach to handling a violation is triggering a transaction abort. In case of constraint enforcement through prevention, the updates performed by the transaction causing the violation are simply not propagated to the database. In case of enforcement through detection, the updates of the transaction have to be undone, possibly by using the recovery mechanism of the system.

Table 4
Example beer relation

Name	Brewed_by	Type	Alcperc
Pilsner Gold	Geineken	Pilsner	5.0
Super Bock	Geineken	Bock	7.0
Ultra Light	Geineken	Light	0.5
Extra Stout	Huinness	Stout	6.0

Compensating updates The most complex but also most flexible approach to handling a constraint violation is allowing the database designer to specify compensating updates as violation response actions. A theoretical treatment of compensating updates can be found in [14].

It is also possible, of course, to use a combination of the three approaches described above, to give the database designer maximum flexibility.

3.8. Distribution and parallelism

Constraint handling should be an integral part of distributed database systems too. Distribution of the database over various sites or nodes has two effects.

In the first place, relations are generally fragmented and allocated on various sites or nodes [16]. This brings two new problems to constraint handling. Constraints specified in terms of relations have to be translated to constraints specified in terms of relation fragments; this problem is discussed in [43, 47]. Also, it has to be decided where the constraint enforcement processes have to take place; a discussion of this issue can be found in [78].

In the second place, distributed systems often offer parallel processing capabilities. These capabilities can be used to support parallel constraint enforcement, thereby reducing the response times of constraint enforcement; this is very important, since the processing costs of constraint enforcement are generally considered one of the main problems in constraint handling. It may be clear that parallel processing is more appropriate in distributed database systems in a multi-processor environment, than in a wide-area network environment.

3.8.1 Relation fragmentation

The translation of constraints from the relation to the fragment level can be performed in the same way as regular queries (see e.g. [16]). The translation consists of the following steps [43, 47]:

Canonical fragment form The constraints are first translated to the canonical fragment form. This form is obtained by replacing all references to global relations by the reconstruction of these relations from their fragments.

Distributed fragment form As a second step, the canonical fragment form is distributed, to avoid the reconstruction of global relations and to improve the possibilities for parallel enforcement.

Optimized fragment form The distributed fragment form can then be optimized in a number of ways to obtain constraints that can be enforced more efficiently. An optimization that is closely related to relation fragmentation is the omission of parts of constraints that can never be violated due to fragmentation constraints.

In general, this translation of constraints can be performed at constraint definition time. A retranslation may be necessary, however, when the fragmentation and/or allocation of fragments used in the constraints changes.

3.8.2 Parallel enforcement

In principle, the same parallel processing possibilities can be used for constraint enforcement as for regular query execution. The following parallelism taxonomy can be used [95]:

Inter-transaction parallelism In a multi-user database system, several transactions can be executed in parallel. As a consequence of this, the constraint enforcement process of these transactions can be executed in parallel, as far as this does not violate the serializability of concurrently executing transactions.

Inter-constraint parallelism Within one transaction, the enforcement of several constraints can be executed in parallel. This process is closely related to the scheduling of the execution of various actions within one transaction [48].

Intra-constraint parallelism Within the enforcement of one constraint, we have the same possibilities for parallelism as within the execution of one single query, by executing the various relational operators in parallel. Intra-constraint parallelism can be subdivided further in:

Task spreading Task spreading parallelism is obtained by executing the same operator on the various fragments of a relation or intermediate result in parallel.

Pipelining Pipelining parallelism is obtained by executing the various stages of relational operators within the execution of a constraint in parallel.

3.9. Special system support

One of the main problems of constraint enforcement is the large cost associated with it. Therefore, techniques have been developed to improve the efficiency of constraint enforcement. In general, three approaches can be distinguished. First, the database system can keep special (redundant or summary) data that facilitate the evaluation of constraints. Secondly, the system can support special-purpose low level algorithms for constraint enforcement. Thirdly, the system can be equipped with special-purpose hardware to speed up constraint enforcement. The first two approaches can be used in general database systems. The last approach is usually restricted to high-performance database machines. The various approaches are discussed below in detail.

3.9.1 Redundant data

One way to reduce the cost for integrity constraint enforcement is to have the database system keep redundant data that can be used in the evaluation of integrity constraints. A form of redundant data that is used very frequently in database systems is an index. This type of low level data will not be considered here.

In [6] the use of redundant data is described for the enforcement of certain classes of integrity constraints called *two-free* assertions. The redundant data required for constraint enforcement are aggregates (minima and maxima of domains). The correctness of the algorithms is demonstrated using Hoare's program logic. The method is only applicable to single-tuple inserts and deletes.

In the SABRE system [77, 79] redundant aggregate values are maintained by the system to reduce the costs of enforcing constraints with aggregates.

3.9.2 Low level algorithms

In addition to a general constraint enforcement mechanism that can handle all kinds of constraints, a system may be equipped with special-purpose low level algorithms that can enforce heavily used constraint types more efficiently than the general mechanism. Constraint types that may be suited for this approach are the structural constraints of the

relational model: domain, referential integrity, and key uniqueness constraints (see example constraints *I1–I3*).

In the PRISMA project this approach is taken [44]. The parallel data management layer of the PRISMA database system is equipped with a special-purpose interface for the specification of certain kinds of constraints that can be enforced more or less autonomously by this layer as part of the two-phase commit protocol.

3.9.3 Hardware support

A way to improve the performance of a database machine is the use of special-purpose hardware. Examples are database filters that perform a first phase of filtering on disk I/O and join processors that can perform join operations very efficiently. These special-purpose hardware devices may also be used to speed up integrity constraint enforcement.

In [70] the use of filter chips for constraint enforcement is discussed. These filter chips are designed to perform selection, projection, and join operations to improve the performance of general database operations.

3.10. Constraint enforcement performance

There has not been much attention in literature for the performance analysis and evaluation of integrity constraint enforcement in relational database systems. With performance analysis, the quantitative analysis of models of constraint enforcement algorithms is meant. With performance evaluation, the evaluation of constraint enforcement algorithms through measurements on actual systems is meant.

3.10.1 Performance analysis

In [4] a quantitative analysis of three constraint enforcement strategies is made:

- transaction compile time enforcement, meaning that the execution of a transaction is not started before all constraints are evaluated to hold; thus, the correctness of the transaction is established before it is actually executed;
- transaction run time enforcement, meaning that the constraints are enforced during transaction execution while the actual updates have not yet been carried out; note that this is the same as violation prevention as described in [40];
- post transaction execution enforcement, in which the constraints are evaluated after all updates have been performed against the database; this is the same as violation detection as described in [40].

The costs of the three methods are measured only in I/O costs. A number of further simplifications are made that limit the practical usability of the proposed model. According to the paper, run time constraint enforcement is superior to any of the other methods examined for realistic database operations (i.e. with most transactions committing). In some situations, compile time enforcement yields a better performance.

In the context of the SABRE project (see Section 4.4) some attention has been devoted to the performance modeling of constraint enforcement [77, 78]. The cost of constraint enforcement is modeled as the number of I/O operations necessary to read and write data from/to disk. Both domain and referential integrity constraints are discussed. The method for enforcement of domain constraints is compared quantitatively with the query modification method [81]; the analysis shows a general superiority of the SABRE approach.[6]

[6] This observation is, however, mainly caused by the fact that the analysis is based on the assumption that tuples violating the constraint have to be presented to the user. As a consequence of this choice, the query modification approach requires the execution of an additional retrieval operation, whereas the SABRE approach has the violating tuples readily available.

3.10.2 Performance evaluation

The actual performance evaluation of existing constraint enforcement systems has hardly got any attention in literature. The only known work in this field is that performed in the context of the SABRE and PRISMA projects.

In [77] the enforcement of domain and referential integrity constraints is evaluated. For these experiments, the SABRE system runs on a MULTICS system and manages a small toy database. As may be expected, the experiments show that the enforcement of referential integrity constraints is more costly than the enforcement of domain constraints. In [79] similar experiments are described, but now with SABRE running on a SM90 machine and managing an adapted Wisconsin database [7]; the relation sizes are still fairly small (3000 tuples).

A performance evaluation of constraint enforcement in a parallel main-memory environment has been performed in the context of the PRISMA project [49]. In the evaluation, the DBMS runs on an 8-node shared-nothing POOMA multiprocessor. The experiments contain enforcement of domain and referential integrity constraints on medium-size databases (10 000 and 50 000 tuples). The evaluation focuses on the effects on integrity control of parallel constraint enforcement and main-memory data storage.

4. Research system overview

This section provides an overview of a number of research projects in which database systems were or are developed that support integrity constraints. The following systems are discussed: System R and DB2, QBE and OBE, INGRES, SABRE/SABRINA, POSTGRES, PRISMA, Starburst, and a few small scale projects. An introduction to a number of these projects can be found in [91]. It should be clear that the systems discussed in this section are not the only relational database systems that support integrity constraints. Some other systems are commercially available that have complete integrity constraint handling, like SYBASE [87]; there is, however, hardly any scientific literature available discussing the details of these systems.

The larger projects are all described in three parts. The first part provides a short introduction to the project. The second part discusses the specification and preprocessing of integrity constraints; preprocessing includes verification, compilation, simplification, and optimization. The third part describes the constraint enforcement algorithms. The smaller projects are described shorter. The last part of this section presents a number of comparisons between the systems described; in one comparison, a few other commercial systems are included as well.

Note that the description of the systems is mainly based on the original design of the systems. Some systems, like INGRES, have been extended later with mechanisms not discussed in this paper.

4.1. System R and DB2

System R was developed by the IBM San Jose Research Laboratory between 1975 and 1979 [2, 22]. The system is one of the first complete prototypes of a relational database management system. In this project the SQL language (originally called SEQUEL) has been designed [21, 34]. The techniques developed in the system R project have been incorporated into the commercial DB2 (and SQL/DS) system.

A functional description of a complete integrity subsystem for System R is given in [35]. The functionality described in this paper is rather complete; the actual implementation in

system R and DB2 is, however, rather restricted (only uniqueness constraints on keys are supported in System R [91], recent versions of DB2 also include other constraints types like referential integrity [32]).

4.1.1 Constraint specification

The definition of an integrity constraint is established by an assertion, expressed in SQL predicates, on a relation or a view. Any constraint that may be expressed by predicates in the WHERE clause of an SQL query may be defined. Transition constraints are supported by allowing the explicit specification of the OLD and NEW states of data. Further, key uniqueness constraints can be defined.

Constraints can be declared to be IMMEDIATE, with enforcement immediately after an update operation, or DELAYED, with enforcement at the end of a transaction. Newly defined constraints are validated against the current database state and stored in the data dictionary of the system.

4.1.2 Constraint enforcement

Updates performed by transactions are made on a copy of the base relations, and constraints are evaluated on the updated copy. In case of a violation, the pre-transaction state of the database is restored by application of the recovery mechanism of the system. Transactions may specify integrity points, however, invoking the enforcement of the constraints during transaction execution. This allows a partial transaction restart from the last integrity point in case of a constraint violation.

4.2. QBE and OBE

The Query-by-Example (QBE) relational product was developed by the IBM Thomas J. Watson Research Center in the mid-seventies [97]. The emphasis in this project was on an easy to use form-oriented database language. QBE has been adapted to an integrated office automation system, called OBE [99].

QBE has an integrated support for integrity constraints [98]; the emphasis is on constraints specification, not on constraint enforcement.

4.2.1 Constraint specification

In QBE integrity constraints are specified in the same form-driven way as relations are defined or queries are specified [97, 98, 90]. In forms representing database relation skeletons the user can specify the constraints in an easy way, using constants, variables, keywords, and simple constructs. The formalism supports a full range of constraint types, among which domain constraints, uniqueness (key) constraints, functional dependencies, inter-attribute comparisons, and transition constraints [97]. Further, the user can use triggers to define upon which events a constraint should be enforced.

Table 5 shows a simplified example of constraints defined on a relation EMPLOYEE; the

Table 5
Example QBE constraint definitions

Employee	Name	Salary	Manager	Department
(1) KEY	*			
(2) CONSTR(I, U)		<10 000		Sales
(3) CONSTR(I, U)	Tom		¬Tom	
(4) CONSTR(D)	COUNT Tom > 2			Dep
(5) CONSTR(I, U)	Tom		COUNT Ed = 1	

constraints are shown in the form specified by the user. The constraint definitions have the following meaning.

(1) The attribute NAME is the key of relation EMPLOYEE.
(2) The salaries in the Sales department must be less than 10 000. Both Sales and 10 000 are constants in this constraint. The constraint has to be enforced upon insert or update operations on the relation.
(3) An employee cannot be his own manager. The underlined word indicates a variable; the name is arbitrary.
(4) Every department must have at least 3 employees. The constraint has to be enforced upon delete operations on the relation.
(5) Each employee has exactly one manager (this is an example of a functional dependency constraint).

4.2.2 Constraint enforcement

The QBE system does not support the explicit notion of transactions. Update commands can, however, be grouped (a user can submit an entire screenful of commands to the system). Integrity constraints are enforced at the end of the execution of the entire group, thereby allowing the database to go through incorrect states during the execution of a group of commands.

4.3. INGRES

The INGRES project was conducted at the University of California at Berkeley from 1973 to 1980. It is one of the first major efforts to build a relational database management system. A complete description of the development of INGRES can be found in [82]. From 1981, INGRES was marketed as a commercial product by Relational Technology Inc. The original data definition and manipulation language of INGRES is called QUEL, a language based on tuple relational calculus [34].

Integrity constraint support has been one of the research topics of the project. This has led to the development of the well-known method of *query modification*, described in [81].

4.3.1 Constraint specification

In INGRES, the definition of an integrity constraint is expressed as a QUEL qualification. Although algorithms for processing all types of constraints that may be expressed by predicates are given in [81], the original implementation was restricted to single-variable constraints without aggregates (domain and transition constraints).[7] According to [77], constraints on deleted tuples (for instance in referential integrity constraints) and constraints that require duplicate elimination cannot be specified.

When an integrity constraint is defined on a relation, it is verified against the current state of that relation. Accepted constraints are stored in the data dictionary of the system, which is organized as a metabase containing among others the relation INTEGRITY.

4.3.2 Constraint enforcement

Integrity constraints are enforced by means of query modification. This means that the qualification of an update operation is modified with the qualification representing the constraints defined on the relation being updated (a simple conjunction of the qualifiers is used). As such, query modification performs constraint enforcement by prevention of constraint violation, not by detection.

[7] Recent commercial versions of INGRES include more general constraint handling mechanisms, see e.g. [64].

As mentioned before, query modification merely places a filter on update operations to remove incorrect values, it does not comply with transaction semantics (the atomicity of transactions is violated).

4.4. SABRE and SABRINA

SABRE and SABRINA[8] are relational database management systems developed in the SABRE project at INRIA in cooperation with MASI laboratory in France [39]. The project started in the early eighties. Most of the solutions implemented in the prototype have been incorporated into a commercial product marketed since 1986.

One of the main research topics of the project is a complete support of semantic data integrity [77, 79]. Further topics that are relevant here are the support of efficient transaction management and the possibility to experiment with parallelism.

The constraint enforcement algorithms of SABRE are extended in [78] to distributed database systems. An introduction to the basic ideas can also be found in [40].

4.4.1 Constraint specification

In SABRE and SABRINA, an integrity constraint is specified as a tuple relational calculus assertion or by a special construct corresponding to predefined types of integrity constraints (like referential integrity). A constraint specification in relational calculus is labeled with the update type upon which the constraint must be enforced. New and old states of relations can be specified explicitly to enable the specification of transition constraints.

The constraint specifications are translated by a constraint compiler into a set of assertions, each referring to a base relation for a particular type of update. Some classes of assertions can be further simplified to improve the efficiency of constraint enforcement [79]. The assertions are evaluated against the current database state; if no violation occurs, they are stored in the data dictionary of the system.

4.4.2 Constraint enforcement

During transaction execution, tuples to be updated are isolated in two temporary relations called differential relations. One contains tuples to be inserted, the other contains tuples to be deleted. The integrity constraints are evaluated against these differential relations. If all constraints are satisfied, the updates in the differential sets are applied to the actual database; if a violation occurs, the updates are rejected and the user is informed. As such, the system uses a constraint violation prevention method, rather than a detection method.

Theoretically, integrity constraints must be enforced at transaction end. The system can detect, however, that some constraints can be enforced at the end of a single update operation. Using this technique, it is possible to determine integrity enforcement points within a transaction, enabling partial validation of the updates of a transaction. The current implementation of the system enforces all constraints at transaction end.

Constraints are divided into three categories according to their enforcement cost and complexity:

(1) individual assertions (single-relation constraints such as domain and uniqueness constraints);
(2) multi-relation assertions (like referential integrity constraints);
(3) assertions with aggregate functions.

For each category, an efficient enforcement algorithm is defined. For aggregate constraints,

[8] The earlier papers from the SABRE project use SABRE as the name for the DBMS, whereas later papers use SABRINA.

redundant information is maintained in the database to avoid the repetitive calculation of the aggregate values.

A quantitative analysis of the constraint enforcement algorithms for domain and referential integrity constraints can be found in [77].

4.5. POSTGRES

POSTGRES is an extended relational database management system designed at the University of California at Berkeley during the second half of the eighties [83, 85]. The system is the successor to the INGRES system, adding many new features like support for complex objects, extendibility for user-defined data types and facilities for active databases in the form of a rule system. The system is designed as close to the standard relational model as possible. POSTGRES uses the POSTQUEL language for data definition and manipulation [74]; this language is based on QUEL [55].

POSTGRES does not have special-purpose support for integrity control, but its rule system can be used for integrity constraint enforcement [84]. The first version of the rule system has been implemented [84, 85]; the description below is based on this version. Because of functional shortcomings and fundamental complexity of the first version of the rule system, it will be replaced by a second version in due course [85].

4.5.1 Constraint specification

POSTGRES supports the query language POSTQUEL [74], which is based on its predecessor QUEL [55]. Any POSTQUEL command can be tagged with special modifiers which change the commands to *rules*. These rules can be used for integrity control. Two modifiers are relevant in the context of integrity control. A POSTQUEL command tagged with the *always* modifier logically appears to run forever. Update commands used in this way can maintain the integrity of the database. *Table 6* shows how an example domain constraint *I*1 can be specified using this construct. A POSTQUEL command tagged with the *refuse* modifier can never run. This construct can be used to specify update operations that would invalidate the integrity of the database. Constraint *I*2 in *Table 6* is a partial specification of a referential integrity constraint using the *refuse* modifier.

To avoid ambiguity in multiple rule semantics and to allow the specification of exception situations, POSTGRES allows the specification of a priority level per rule.

The second version of the rule system will be based on a more conventional production rule syntax for rule specification.

4.5.2 Constraint enforcement

In POSTGRES, integrity constraint enforcement consists of the execution of rules that specify the constraints. The execution has two important characteristics that can be changed by a rule optimizer to obtain optimal performance.

In the first place, the time of triggering of rules tagged with the *always* modifier can be chosen. The first option is *early evaluation* or *forward chaining*, meaning that a rule is executed when an update changes part of the database such that the execution of the rule is

Table 6
Example constraints in POSTGRES

I1	delete always beer where beer.alcperc(= 0
I2	refuse delete brewery where brewery.name = beer.brewed_by

needed to obtain a correct state again. The second option is *late evaluation* or *backward chaining*, meaning that a rule is executed when a database action requires access to part of the database of which the integrity was violated by a prior update.

In the second place, the granularity of locking for rules can be chosen.[9] The options are relation level and tuple level locking. Clearly, relation level locking is preferable in the case when few rules are defined on a large quantity of data, whereas tuple level locking is preferable in the situation with many rules concerning small quantities of data. The locking strategy can be enhanced to allow dynamic escalation of tuple level locks to relation level locks.

In the case of rules with small scope, rule execution is based on markers per tuple that trigger actions; in the case of rules with large scope, a query rewrite scheme is used comparable to query modification [81].

POSTGRES builds a graph representing rule invocations to avoid cyclical triggering of multiple rules. If this graph is not tree-shaped, the system generates an error message.

4.6. *PRISMA*

In the PRISMA project a parallel, main-memory, relational database system has been developed [95, 1], designed to run on a shared-nothing, multiprocessor hardware architecture.[10] The second operational prototype of the PRISMA database management system (PRISMA/DB1) includes a complete constraint handling subsystem based on *transaction modification* [46, 47]. This subsystem uses parallelism for the efficient enforcement of constraints on fragmented relations. Transaction modification is used to support a broad class of constraint types [47]; the current implementation of PRISMA/DB1 only supports domain, uniqueness, and referential integrity constraints, however.

4.6.1 *Constraint specification*

In the transaction modification approach, constraints are specified in a first-order logic formalism (the current implementation uses a simple ad hoc formalism). Constraint definition is performed with full fragmentation and location transparency [16]. Newly defined constraints are handled by a DBMS component called the *constraint compiler*.

The constraint compiler has three tasks in preparing constraints for transaction modification. First, it optimizes the constraints in a number of ways [43, 47]. Secondly, it removes fragmentation transparency. Thirdly, it translates constraints into an extension of the relational algebra [43, 47], called XRA [45]; the translated constraints are directly executable by the relational execution layer of the system. The example constraints are shown in XRA format in *Table 7*.[11]

Table 7
Example constraints in XRA format

I2	alarm (sel (not alcperc > 0), beer)
I2	alarm (diff (uniq (proj (brewed_by. beer)), proj (name, brewery)))

[9] The locking strategy for rules in POSTGRES is rather complex, since the system supports no less than 13 kinds of locks [84].

[10] The PRISMA project should not be confused with the PRISM project [76] or the PRIMA project [54, 75]. Both projects also pay attention to integrity constraint handling: PRISM deals with a knowledge based approach to constraint management, and PRIMA deals with constraint handling in a complex object database.

[11] The *alarm* operator is specially included in XRA to allow the specification of constraints; the operator triggers a transaction abort if its argument is non-empty.

The constraint compiler can also decide to have some constraint types enforced autonomously by the relational execution layer of the system to reduce the overhead in constraint enforcement through transaction modification [44].

4.6.2 Constraint enforcement

In PRISMA/DB, constraints are always enforced at transaction commit time, thereby ensuring full transaction semantics. During transaction execution, the transaction management layer of the system analyses the transaction to record which relation fragments are updated by the transaction. When the transaction management layer reaches the commit point of a transaction, it retrieves the constraints of updated fragments from the data dictionary of the system, and performs some filtering on these. Next, the actual transaction modification takes place by concatenating the constraint specifications to the transaction. The modified transaction is executed in the same way as an unmodified transaction, using the full possibilities for parallel execution [95, 48, 96]. In case of a constraint violation, the modified transaction is either aborted using the normal abort protocol, or the modified transaction performs compensating updates to reach a correct database state.

Autonomous enforcement of constraints by the relational execution layer of the system takes place without any intervention from the transaction management layer. In case of a constraint violation, the transaction is aborted using the normal abort protocol.

4.7. Starburst

Starburst is a prototype relational database management system under development at the IBM Almaden Research Center [50, 62]. One of the goals of the Starburst project is to build an extensible system that can support non-traditional applications and can serve as a testbed for innovations and improvements in database technology. Constraint handling in Starburst relies on a general-purpose production rules facility that is developed as an extension to Starburst [93, 62]. This facility allows the definition of database operations that are executed automatically whenever certain events occur or certain conditions are met. The production rules mechanism in Starburst can also be used for the maintenance of derived data [20] and the construction of knowledge base systems.

4.7.1 Constraint specification

In Starburst, constraints are specified in a set-oriented database rule language based on SQL [93]. A rule in Starburst is identified by its name and the relation on which the rule is defined. A rule further consists of four parts. The *rule transition predicate* specifies the triggering operations of the rule in terms of insert, delete, and update (on specified columns). The rule is triggered if the specified operations occurred on the specified relation in the net effect of a database transition. *The rule condition* is an arbitrary SQL predicate over the database. If it evaluates to true, the action list of the rule is executed. The *rule action list* consists of arbitrary Starburst database operations, including data manipulation operations, data definition commands and rollback requests. Finally, the *precedence part* of a rule lists the rules of which the execution has to precede, respectively follow, the execution of the rule at hand, and is used to define rule priorities. The condition and action list of a rule may refer to both the current state of the database and transition tables that contain the net changes to the database that have occurred during the database transition.

In [17, 18] a facility is discussed to derive these production rules from constraint definitions stated in an SQL-based language. This facility generates the transition predicate, analyses sets of rules for potential cyclical triggering behaviour, and performs some optimizations on the rule conditions.

4.7.2 Constraint enforcement

When a transaction (a sequence of SQL operations) is executed on Starburst, this may cause a change in the database state. This state change creates the first transition that triggers a set of rules. Taking specified precedences between rules into consideration, one rule from this set is chosen. If the condition of the chosen rule evaluates to false, another rule is chosen from the set. If the condition evaluates to true, the actions of the rule are executed. The database state change composed of the original transaction and the execution of the actions of the rule constitutes the next relevant transition for rule evaluation. A given rule is triggered at a certain point if its transition predicate holds with respect to the (composite) transition since the point at which its condition was most recently evaluated; if the rule's condition has not been evaluated before, the transition is taken from the start of the transaction. If a rollback action is encountered during rule execution, the system undoes the transaction, including the effects of previously executed rule actions, and rule processing terminates. Otherwise, rule processing terminates when the set of triggered rules is empty or when no triggered rule has a condition that evaluates to true; in this case the entire transaction is committed.

Rule condition evaluation and rule action execution is handled by the Starburst query processor, thereby ensuring serializability of transactions including rule actions. The rule system itself must enforce concurrency control for transactions that affect the set of rules in the system, i.e. for transactions that include any form of rule definition. Also, the rule system must provide means to roll back the effects of transactions that affect the set of rules in the system, to guarantee atomicity of this kind of transactions in the case of transaction abort.

4.8. Other systems

4.8.1 System by small

At the University of London a Constraint Enforcement System has been developed for a database system based on the Binary Relational Model [80]. In this approach, constraints are specified in a subset of first order predicate calculus. Each constraint definition is translated into a single query specified in a logic programming formalism. The translation takes place in a few simple rewriting steps. Upon updates against the database, the updated tuples are matched against the set of constraints. Matched constraints are evaluated against the database using a resolution based theorem prover. If a constraint is violated, the update operation is aborted, and the violating tuples are presented to the user. The presented algorithms are suited for single tuple insert and delete operations only; handling of multi-tuple update operations requires modification of the algorithms. General constraints are supported by the system, with the exception of transition constraints and aggregate constraints. A small scale implementation of the system has been completed.

4.8.2 AIM

AIM is an extension to the INGRES system to provide a nontrivial semantic control mechanism [26]. The system has been developed at the University of Dortmund, Germany.[12] Following the AIM approach, integrity constraints are partitioned into two categories: constraints that are enforced upon every relevant update of the database, and constraints that are checked periodically (batch constraints). A constraint is specified using a QUEL-

[12] This AIM system has no relation to the IBM project with the same name, in which a DBMS for extended NF2 relations was designed [27].

based formalism. Constraint enforcement takes place by submitting a query with as qualification the negation of the QUEL predicate stated in the constraint. For most constraint types, AIM implements a detection and rollback strategy for constraint enforcement. For some simple types of constraints, the system can use a prevention strategy, making use of temporary relations. Further, the system supports transactions, by enforcing constraints at transaction end (except for the constraints enforced by prevention).

4.9. System comparison

Below, the systems described above are compared on a number of characteristics with respect to constraint handling: supported constraint types, constraint preprocessing characteristics, and constraint enforcement characteristics.

4.9.1 Supported constraint types

An overview of the integrity constraint types supported by implemented systems is given in *Table 8*.[13] The table lists most systems discussed in this section plus a few commercial systems.

Note that POSTGRES and Starburst do not have integrated support for specific constraint types, but provide a general production rule-based extension to handle a large spectrum of constraint types (see Sections 4.7 and 4.5 for details). Similarly, SYBASE has specific support for some constraint types, but provides a trigger mechanism for more general types.

From this table it may be clear that some systems actually support only a very limited set of constraint types, although some of them are based on a complete-conceptual design and have been under development for a considerable period (like DB2 and INGRES).

4.9.2 Constraint preprocessing characteristics

In *Table 9* the following characteristics of constraint preprocessing in existing systems are listed:

Constraint verification This column in the table lists if the system supports constraint verification as discussed in Section 3.4. Note that syntactic and simple semantic checks (attribute types) on individual constraints are considered standard here, and are not listed in the table.

Table 8
Supported constraint types in implemented systems

	Domain constr.	Unique (key)	Referent. integrity	Funct. depend.	Aggre-gate	Trans-ition	General constr.
DB2	–	yes	–	–	–	–	–
INGRES	yes	yes	–	–	–	–	–
ORACLE	yes	yes	–	–	–	–	–
SABRINA	yes	yes	yes	yes	yes	yes	–
POSTGRES	–	–	–	–	–	–	rules
PRISMA	yes	yes	yes	–	–	–	–
Starburst	–	–	–	–	–	–	rules
SYBASE	yes	yes	yes	–	–	–	triggers
UNIFY	yes	yes	yes	–	–	–	–
Small	yes	yes	yes	yes	–	–	–
AIM	yes	yes	yes	yes	yes	yes	–

[13] This table is partly based on information in [91] and may therefore not reflect the most recent status of certain systems. Some papers by other authors (e.g. [63]) even seem to contradict this information.

Table 9
Constraint preprocessing characteristics in implemented systems

System	Verification	Translation		Optimization
		From	To	
DB2	curr. state	–	–	–
INGRES	curr. state	–	–	–
SABRINA	curr. state	special DDL	rel. calc.	yes
POSTGRES	–	–	–	yes
PRISMA	–	special DDL	rel. algebra	yes
Starburst	–	–	–	–
Small	–	pred. calculus	logic. DML	–
AIM	–	–	–	–

Constraint translation The two columns in the table under translation list if the system supports translation from a specification language to a different internal (enforcement) language, and if so, which language classes are used (see Section 3.3).

Constraint optimization The optimization column lists if the system supports constraint optimization techniques.

4.9.3 Constraint enforcement characteristics

In *Table 10* a number of characteristics of the constraint enforcement algorithms of existing database systems are listed:

Enforcement strategy This column lists the constraint enforcement strategy employed by the system: constraint violation prevention before the database is actually updated, or violation detection after the database has been updated; further details are discussed in Section 3.7.

Transaction atomicity This column specifies whether a system supports full transaction atomicity; see Section 3.2 for a more complete treatment of this issue.

Integrity checkpoints Some systems use the concept of *integrity checkpoints* to establish the correctness of the database at a certain point during transaction execution to enable a partial transaction rollback (see Section 3.2).

Table 10
Constraint enforcement characteristics of implemented systems

System	Enforcement strategy	Transaction atomicity	Integrity checkpoints	Parallel enforcement
DB2	detection	optional	yes	–
INGRES	prevention	–	–	–
SABRINA	prevention	yes	–	*see text*
POSTGRES	prev./detect.	no	no	no
PRISMA	*see text*	yes	–	yes
Starburst	detection	yes	no	–
Small	detection	–	–	–
AIM	detection	optional	–	–

Parallel enforcement To decrease constraint enforcement execution times and thereby transaction response times, parallelism may be used in constraint enforcement; further details can be found in Section 3.8.

Two entries in the table require some comment. First, in the PRISMA system it is hard to distinguish between enforcement through violation prevention of detection because of the data storage organization of this system; this topic was discussed in Section 3.7. Secondly, parallelism is a research issue in the SABRINA project [91], but the literature does not state if parallelism is actually used in constraint enforcement.

5. Conclusions

A number of general remarks can be made with respect to the current state of the research into integrity control in relational databases. A number of important remarks – at least from the authors point of view – are listed below.

In general, there is still a large gap between the theoretic approach to constraint handling and the actual implementation in complete relational database management systems. Most implementations are incomplete, in the sense that they accept only limited constraint types, do not support full transaction semantics etc.

The importance of transactions as atomic units of work against a database has not been generally accepted in the research on integrity constraints. Part of the research does not take transactions into consideration at all, part does take transactions into account, but does not comply with the atomicity principle.

The integrity control mechanism in a database management system can be specially constructed for this task, or can rely on some general mechanism. If a specially constructed mechanism exists, it can be specially designed for a limited set of constraint types, or it can be designed to handle arbitrary constraint types. So, the integrity control mechanism in a DBMS can have one of three levels of generality (see *Fig. 1*): integrity control can be performed by a mechanism designed to handle a limited set of constraint types, the task can be performed by a mechanism designed to handle all constraint types, or it can be performed by a general purpose rule or trigger subsystem that can be used for other tasks too.

An important criterium for the classification of integrity control mechanisms is the fact whether the mechanism employs a violation prevention or detection technique. In a main-memory environment where updates take place in the same memory space where the database is actually stored, it is hardly relevant, however, to distinguish between violation prevention using the transaction workspace model and violation detection.

Only little attention has been devoted to the performance of integrity control subsystems,

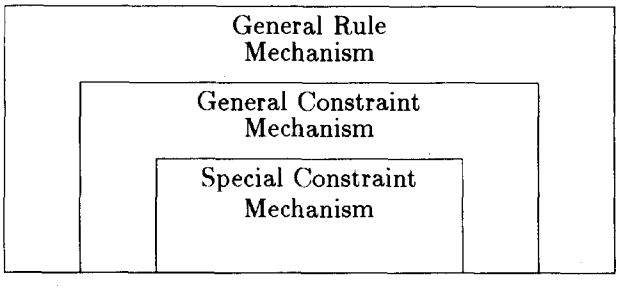


Fig. 1. Integrity control mechanisms.

both in terms of analysis and evaluation of systems. This observation is not in accordance with the general opinion that performance is one of the key problems in real-world integrity control. In contrast to the research in the field of query processing, there has been very little attention for parallelism in constraint enforcement until now. Typical research on integrity control mechanisms hardly takes parallelism into consideration, and typical research on parallel database systems does not pay much attention to integrity control. Only in the PRISMA project parallelism is taken as a primary research topic in the context of integrity constraint handling.

Acknowledgements

We are grateful to Jennifer Widom from IBM Almaden Research Center for her comments on an early version of this work, published as technical report at the University of Twente. The referees of the DKE journal are thanked for their comments on the first submitted version of this paper.

References

[1] P.M.G. Apers, C.A. v.d. Berg, J. Flokstra, P.W.P.J. Grefen, M.L. Kersten and A.N. Wilschut, PRISMA/DB: A parallel, main-memory relational DBMS, *IEEE Trans. Knowledge Data Engrg.* 4 (6) (1992).

[2] M.M. Astrahan et al., System R: A relational approach to database management, *ACM Trans. Database Syst.* 1 (2) (1976).

[3] A. Auddino, E. Amiel and B. Bhargava, Experiences with SUPER, a database visual environment; *Proc. 2nd Internat. Conf. on Database and Expert Systems Applications*, Berlin, Germany (1991).

[4] D. Badal and G. Popek, Cost and Performance Analysis of Semantic Integrity Validation Methods, *Proc. 1979 ACM SIGMOD Internat. Conf. on the Management of Data*, Boston, USA (1979).

[5] C. Beeri, H.J. Schek and G. Weikum, Multilevel transaction management, Theoretical art or practical need?, *Proc. 1988 Internat. Conf. on Extending Database Technology*, Venice, Italy (1988).

[6] P. Bernstein, B. Blaustein and E. Clarke, Fast maintenance of semantic integrity assertions using redundant aggregate data, *Proc. 6th Internat. Conf. on Very Large Data Bases*, Montreal, Canada (1980).

[7] D. Bitton, D.J. DeWitt and C. Turbyfill, Benchmarking database systems: A systematic approach; *Proc. 9th Internat. Conf. on Very Large Data Bases*, Florence, Italy (1983).

[8] BMJ Editorial, Some hospital statistics, *British Med. J.* 1 (1965).

[9] E.O. de Brock, *De Grondslagen van Semantische Databases* (Academic Service, 1989) (in Dutch).

[10] M.L. Brodie, Specification and verification of data base semantic integrity, Ph.D. Thesis, Dept. of Computer Science, University of Toronto, Toronto, Canada, 1978.

[11] F. Bry and R. Manthey, Checking consistency of database constraints: A logical basis, *Proc. 12th Internat. Conf. on Very Large Data Bases*, Kyoto, Japan (1986).

[12] F. Bry, H. Decker and R. Manthey, A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases, *Proc. Internat. Conf. on Extending Database Technology*, Venice, Italy (1988).

[13] O.P. Buneman and E.K. Clemons, Efficiently monitoring relational databases, *ACM Trans. Database Syst.* (3) (1979).

[14] R.A. de By, The integration of specification aspects in database design, Ph.D. Thesis, University of Twente, 1991.

[15] M.A. Casanova, L. Tucherman and A.L. Furtado, Enforcing inclusion dependencies and referential integrity, *Proc. 14th Internat. Conf. on Very Large Data Bases*, Los Angeles, USA (1988).

[16] S. Ceri and G. Pelagatti, *Distributed Databases, Principles and Systems* (McGraw-Hill, Englewood Cliffs, NJ, 1984).

[17] S. Ceri and J. Widom, Deriving production rules for constraint maintenance, IBM Research Report RJ7348, IBM Almaden Research Center, San Jose, USA, 1990.

[18] S. Ceri and J. Widom, Deriving production rules for constraint maintenance, *Proc. 16th Internat. Conf. on Very Large Data Bases*, Brisbane, Australia (1990).

[19] S. Ceri, G. Gottlob and L. Tanca, *Logic Programming and Databases* (Springer-Verlag, Berlin, 1990).

[20] S. Ceri and J. Widom, Deriving production rules for incremental view maintenance, *Proc. 17th Internat. Conf. on Very Large Data Bases*, Barcelona, Spain (1991).

[21] D.D. Chamberlin et al., SEQUEL 2: A unified approach to data definition, manipulation, and control, *IBM J. Res. Develop.* 20 (6) (1976).

[22] D.D. Chamberlin, A.M. Gilbert and R.A. Yost, A history of System R and SQL/Data System, *Proc. 7th Internat. Conf. on Very Large Data Bases*, Cannes, France (1981).

[23] J. Chomicki, History-less checking of dynamic integrity constraints, *Proc. 8th IEEE Internat. Conf. on Data Engineering*, Phoenix, USA (1992).

[24] E.F. Codd, A data base sublanguage founded on relational calculus, *Proc. 1971 ACM SIGFIDET Workshop* (1971).

[25] E.F. Codd, Extending the database relational model to capture more meaning, *ACM Trans. Database Syst.* (4) (1979).

[26] A.B. Cremers and G. Domann, AIM – An Integrity Monitor for the database system INGRES, *Proc. 9th Internat. Conf. on Very Large Data Bases*, Florence, Italy (1983).

[27] P. Dadam et al., A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies, *Proc. 1986 ACM SIGMOD Internat. Conf. on the Management of Data*, Washington D.C., USA (1986).

[28] C.J. Date, Referential integrity, *Proc. 7th Internat. Conf. on Very Large Data Bases*, Cannes, France (1981).

[29] C.J. Date, *An Introduction to Database Systems, Volume II* (Addison-Wesley, Reading, MA, 1983).

[30] C.J. Date, Referential integrity and foreign keys: Further considerations, in *Relational Database Writings 1985–1989* (Addison-Wesley, Reading, MA, 1990).

[31] C.J. Date, A Contribution to the Study of Database Integrity, in *Relational Database Writings 1985–1989* (Addison-Wesley, Reading, MA, 1990).

[32] IBM DATABASE 2 Referential integrity usage guide, IBM Corporation, 1989.

[33] H.D. Ehrich, U.W. Lipeck and M. Gogolla, Specification, semantics and enforcement of dynamic database constraints, *Proc. 1984 ACM SIGMOD Internat. Conf. on the Management of Data*, Singapore (1984).

[34] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems* (Benjamin/Cummings, Menlo Park, LA, 1989).

[35] K.P. Eswaran and D.D. Chamberlin, Functional specifications of a subsystem for data base integrity, *Proc. 1st Internat. Conf. on Very Large Data Bases*, Framingham, USA (1975).

[36] K.P. Eswaran, Aspects of a trigger subsystem in an integrated database system, *Proc. 2nd Internat. Conf. on Software Engineering* (1976).

[37] H. Gallaire, Impacts of logic on data bases, *Proc. 7th Internat. Conf. on Very Large Data Bases*, Cannes, France (1981).

[38] G. Gardarin and M. Melkanoff, Proving consistency of database transactions, *Proc. 5th Internat. Conf. on Very Large Data Bases*, Rio de Janeiro, Brazil (1979).

[39] G. Gardarin et al., Design of a multiprocessor relational database system; *IFIP 9th World Computer Congress*, Paris, France (1983).

[40] G. Gardarin and P. Valduriez, *Relational Databases and Knowledge Bases* (Addison-Wesley, Reading, MA, 1989).

[41] B.S. Goldstein, Constraints on null values in relational databases, *Proc. 7th Internat. Conf. on Very Large Data Bases*, Cannes, France (1981).

[42] J. Gray, The transaction concept: virtues and limitations, *Proc. 7th Internat. Conf. on Very Large Data Bases*, Cannes, France (1981).

[43] P.W.P.J. Grefen and P.M.G. Apers, Parallel handling of integrity constraints on fragmented relations, *Proc. Internat. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland (1990).

[44] P.W.P.J. Grefen, J. Flokstra and P.M.G. Apers, Parallel handling of integrity constraints, *Proc. PRISMA Workshop on Parallel Database Systems*, Noordwijk, The Netherlands (1990).

[45] P.W.P.J. Grefen, A.N. Wilschut and J. Flokstra, PRISMA/DB 1.0 user manual. Memorandum INF91-06, University of Twente, The Netherlands, 1991.

[46] P.W.P.J. Grefen and P.M.G. Apers, Integrity constraint enforcement through transaction modification, *Proc. 2nd Internat. Conf. on Database and Expert Systems Applications*, Berlin, Germany (1991).

[47] P.W.P.J. Grefen, Integrity control in parallel database systems, Ph.D. Thesis, University of Twente, 1992.

[48] P.W.P.J. Grefen, Dynamic action scheduling in a parallel database system, *Proc. 4th Internat. Conf. on Parallel Architectures and Languages Europe*, Paris, France (1992).

[49] P.W.P.J. Grefen, J. Flokstra and P.M.G. Apers, Performance evaluation of constraint enforcement in a parallel main-memory database system, *Proc. 3rd Internat. Conf. on Database and Expert System Applications*, Valencia, Spain (1992).

[50] L. Haas et al., Starburst midflight: As the dust clears, *IEEE Trans. Knowledge Data Engrg.* (1) (1990).

[51] M.M. Hammer and D.J. McLeod, Semantic integrity on a relational data base system, *Proc. 1st Internat. Conf. on Very Large Data Bases*, Framingham, USA (1975).

[52] M.M. Hammer and D.J. McLeod, A framework for database semantic integrity, *Proc. 2nd Internat. Conf. on Software Engineering*, San Francisco, USA (1976).

[53] M.M. Hammer and S.K. Sarin, Efficient

monitoring of database assertions, *Proc. 1978 ACM SIGMOD Internat. Conf. on the Management of Data*, Dallas, USA (1978).

[54] T. Härder, K. Meyer-Wegener, B. Mitschang and A. Sikeler, PRIMA – A DBMS prototype supporting engineering applications, *Proc. 13th Internat. Conf. on Very Large Data Bases*, Brighton, UK (1987).

[55] G. Held et al., INGRES: A relational data base system, *Proc. 1975 Nat. Computer Conf.*, Anaheim, USA (1975).

[56] L.J. Henschen, W.W. McCune and S.A. Naqvi, Compiling constraint-checking programs from first-order formulas, *Advances in Database Theory*, Vol. 2 (Plenum Press, New York, 1984).

[57] Y.C. Hong and Y.W. Stanley, Associative hardware and software techniques for integrity control, *ACM Trans. Database Syst.* 6 (1981).

[58] A. Hsu and T. Imielinsky, Integrity checking for multiple updates, *Proc. 1985 ACM SIGMOD Internat. Conf. on the Management of Data*, Austin, USA (1985).

[59] H.F. Korth and A. Silberschatz, *Database System Concepts* (McGraw-Hill, New York, 1986).

[60] A.M. Kotz, K.R. Dittrich and J.A. Mülle, Supporting semantic rules by a generalized event/trigger mechanism, *Proc. 1988 Internat. Conf. on Extending Database Technology*, Venice, Italy (1988).

[61] G.M.E. Lafue, Semantic integrity dependencies and delayed integrity checking, *Proc. 8th Internat. Conf. on Very Large Data Bases*, Mexico City, Mexico (1982).

[62] G.M. Lohman, B. Lindsay, H. Pirahesh and K.B. Schiefer, Extensions to Starburst: objects, types, functions, and rules, *Comm. ACM* 34 (10) (1991).

[63] V.M. Markowitz, Referential integrity revisited: An object-oriented perspective, *Proc. 16th Internat. Conf. on Very Large Data Bases*, Brisbane, Australia (1990).

[64] V.M. Markowitz, Safe referential integrity structures in relational databases, *Proc. 17th Internat. Conf. on Very Large Data Bases*, Barcelona, Spain (1991).

[65] V.M. Markowitz, Problems underlying the use of referential integrity mechanisms in relational database management systems, *Proc. 7th Internat. Conf. on Data Egnineering*, Japan (1991).

[66] M. Morgenstern, Constraint equations: Declarative expression of constraints with automatic enforcement, *Proc. 10th Internat. Conf. on Very Large Data Bases*, Singapore (1984).

[67] A. Motro, Integrity = Validity + Completeness, *ACM Trans. Database Syst.* 14 (4) (1989).

[68] J.M. Nicolas, Logic for improving integrity checking in relational data bases, *Acta Inform.* 18 (1982).

[69] H. Noble and T. Abbod, Meta-rules and semantic integrity constraints in databases, *Proc. 5th British Nat. Conf. on Databases*, Canterbury, UK (1986).

[70] M. Penaloza and E. Ozkarahan, Integrating integrity constraints with database filters implemented in hardware, *Proc. 6th Internat. Workshop on Database Machines*, Deauville, France (1989).

[71] X. Qian and G. Wiederhold, Knowledge-based integrity constraint validation, *Proc. 12th Internat. Conf. on Very Large Data Bases*, Kyoto, Japan (1986).

[72] X. Qian and D.R. Smith, Integrity constraint reformulation for efficient validation, *Proc. 13th Internat. Conf. on Very Large Data Bases*, Brighton, UK (1987).

[73] K.V.S.V.N. Raju and A.K. Majumdar, Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems, *ACM Trans. Database Syst.* 13 (2) (1988).

[74] L. Rowe and M. Stonebraker, The POSTGRES data model, *Proc. 13th Internat. Conf. on Very Large Data Bases*, Brighton, UK (1987).

[75] H. Schöning, Preserving consistency in nested transactions, Department of Computer Science; University of Kaiserslautern, Germany, 1989.

[76] A. Shepherd and L. Kerschberg, PRISM: A knowledge based system for semantic integrity specification and enforcement in database systems, *Proc. 1884 ACM SIGMOD Internat. Conf. on the Management of Data*, Boston, USA (1984).

[77] E. Simon and P. Valduriez, Design and implementation of an extendible integrity subsystem, *Proc. 1984 ACM SIGMOD Internat. Conf. on the Management of Data*, Boston, USA (1984).

[78] E. Simon and P. Valduriez, Integrity control in distributed database systems, MCC Technical Report Number DB-103-85, MCC, Austin, USA, 1985.

[79] E. Simon and P. Valduriez, Design and analysis of a relational integrity subsystem, MCC Technical Report Number DB-015-87, MCC, Austin, USA, 1987.

[80] C. Small, An implementation of a constraint enforcement system, *Proc. 5th British Nat. Conf. on Databases*, Canterbury, UK (1986).

[81] M. Stonebraker, Implementation of integrity constraints and views by query modification, *Proc. 1975 ACM SIGMOD Internat. Conf. on the Management of Data*, San Jose, USA (1975).

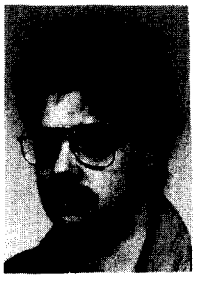
[82] M. Stonebraker, ed., *The INGRES Papers* (Addison-Wesley, Reading, MA, 1986).

[83] M. Stonebraker and L. Rowe, The design of POSTGRES, *Proc. 1986 ACM SIGMOD Internat. Conf. on the Management of Data*, Washington DC, USA (1986).

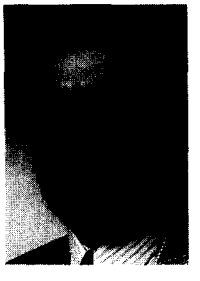
[84] M. Stonebraker, E.N. Hanson and S. Potamianos, The POSTGRES rule manager, *IEEE Trans. Software Engrg.* 14 (7) (1988).

[85] M. Stonebraker, L.A. Rowe and M. Hirohama,

The implementation of POSTGRES, *IEEE Trans. Knowledge Data Engrg.* 2 (1) (1990).
[86] SuperBase user and reference manual, Precision Software Limited, Irving, USA, 1990.
[87] SYBASE SQL server, Sybase Inc., Emeryville, USA, 1989.
[88] Transact-SQL user's guide, Sybase Inc., Emeryville, USA, 1989.
[89] D.C. Tsichritzis and F.H. Lochovsky, *Data Models* (Prentice-Hall, Englewood Cliffs, NJ, 1982).
[90] J.D. Ullman, Principles of database systems, second edition (Computer Science Press, Rockville, MD, 1982).
[91] P. Valduriez and G. Gardarin, *Analysis and Comparison of Relational Database Systems* (Addison-Wesley, Reading, MA, 1989).
[92] X.Y. Wang, N.J. Fiddian and W.A. Gray, The development of a knowledge-based database transaction design assistant, *Proc. 2nd Internat. Conf. on Database and Expert Systems Applications*, Berlin, Germany (1991).
[93] J. Widom, R.J. Cochrane and B.G. Lindsay, Implementing set-oriented production rules as an extension to Starburst, *Proc. 17th Internat. Conf. on Very Large Data Bases*, Barcelona, Spain (1991).
[94] G.A. Wilson, A conceptual model for semantic integrity checking, *Proc. 6th Internat. Conf. on Very Large Data Bases*, Montreal, Canada (1980).
[95] A.N. Wilschut, P.W.P.J. Grefen, P.M.G. Apers and M.L. Kersten, Implementing PRISMA/DB in an OOPL, *Proc. 6th Internat. Workshop on Database Machines*, Deauville, France (1989).
[96] A.N. Wilschut and P.M.G. Apers, Dataflow query execution in a parallel main-memory environment, *Proc. 1st Internat. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, USA (1991).
[97] M.M. Zloof, Query-By-Example: The invocation and definition of tables and forms, *Proc. 1st Internat. Conf. on Very Large Data Bases*, Framingham, USA (1975).
[98] M.M. Zloof, Security and integrity within the Query-by-Example database management language, IBM Research Report RC6982, IBM Thomas J. Watson Research Center, Yorktown Heights, USA, 1978.
[99] M.M. Zloof, Office-By-Example: A business language that unifies data and word processing and electronic mail, *IBM Systems J.* 21 (3) (1982).



Paul W.P.J. Grefen is an assistant professor in the Information Systems Group at the University of Twente in the Netherlands. He received his Ph.D. from the University of Twente on the topic of integrity control in parallel database systems. As a full-time researcher he has worked on the PRISMA research project from 1987 to 1992. This project has resulted in a working prototype of a parallel main-memory database machine with full database functionality, including parallel integrity control. His current research interests include integrity control, active database systems, and groupware systems.



Peter M.G. Apers is a professor at the University of Twente in the Netherlands. He received his Ph.D. from the Vrije University in Amsterdam. His Ph.D. thesis was on distributed query processing and data allocation. Before and after working on his thesis he was a visiting researcher at the University of California in Santa Cruz and Stanford University. He was the European Program chairman of the Very Large Data Bases Conference in 1989 held in Amsterdam. Currently he heads a group working on object-oriented data models, complex object databases, optimization of logical query languages, parallelism in database management systems, and database machines. He serves on the editorial board of *Data & Knowledge Engineering* and *Distributed and Parallel Database Systems*. He is a member of ACM and IEEE (CS).