Noname manuscript No.
(will be inserted by the editor)

Semantic Integrity Support in SQL:1999 and Commercial (Object-)Relational Database Management Systems

Can Türker[1], **Michael Gertz**[2]

[1] Swiss Federal Institute of Technology (ETH) Zurich, Institute of Information Systems, ETH Zentrum, CH-8092 Zurich, Switzerland
e-mail: `tuerker@inf.ethz.ch`

[2] University of California, Davis, Department of Computer Science, One Shields Avenue, Davis, CA 95616, USA
e-mail: `gertz@cs.ucdavis.edu`

The date of receipt and acceptance will be inserted by the editor

Abstract. The correctness of the data managed by database systems is vital to any application that utilizes data for business, research, and decision making purposes. To guard databases against erroneous data not reflecting real-world data or business rules, semantic integrity constraints can be specified during database design. Current commercial database management systems provide various means to implement mechanisms to enforce semantic integrity constraints at database run-time.

In this paper, we give an overview of the semantic integrity support in the most recent SQL-standard SQL:1999, and we show to what extent the different concepts and language constructs proposed in this standard can be found in major commercial (object-)relational database management systems. In addition, we discuss general design guidelines that point out how the semantic integrity features provided by these systems should be utilized in order to implement an effective integrity enforcing subsystem for a database.

Key words: semantic integrity constraints — SQL:1999 — object-relational databases — constraint enforcement

1 Introduction

In today's complex information system infrastructures, the semantic correctness of the data managed by database management systems is vital to the more general aspect of quality of the data and thus their usage by diverse types of applications. Semantic correctness of data basically deals with the issue to what extent the data managed by a database system reflect the real-world or artifacts consumed and operated on by applications. The specification of what data is semantically correct builds one of the most important tasks in the database design process [Mai83,Bis98]. In this process, data correctness requirements are gathered from users, application developers, and business regulations, and are translated into *semantic integrity constraints* (or *integrity constraints* for short). Integrity constraints, formulated in some language used in database design, specify conditions on the database objects which have to be satisfied in order to correctly reflect the real-world and universe of discourse. Integrity constraint specifications are typically translated into constraint enforcing mechanisms provided by the database management system used to implement the database.

Semantic integrity constraints serve several purposes in database design, implementation, and run-time. Besides specifying admissible data, they also often embed portions of the application semantics and thus provide some kind of application specific domain knowledge. Furthermore, at database run-time, integrity constraints can be exploited for query optimization purposes and thus can lead to a better performance for query evaluation.

The importance of the specification and enforcement of semantic integrity constraints in databases was already recognized and addressed in the Seventies by several authors [HM75,Sto75,EC75,Bro78,Cod79]. Thereafter, semantic integrity constraints in databases have been studied from various aspects. The results of the different research efforts are documented in many publications. Several dissertations, e.g., [Bro78,Gre92,Deß93,Ger96,Tür99b] and chapters in database textbooks, e.g., [Ull88,Dat95,EN94,SKS97,SST97] are devoted to semantic integrity issues in databases. There are also textbooks, such as [Mai83,AHV95,dB95], that primarily focus on theoretical foundations of semantic integrity constraints. In the late Eighties and early Nineties there were a few overview papers on the foundations of integrity constraints in relational database systems [BM88,Dat90,GA93]. These presentations were independent of any particular integrity constraint specification language, in particular neither the SQL-standard existing at that time nor SQL-dialects as provided by major database management systems were considered.

This paper surveys the state-of-the-art in the field of semantic integrity constraints in (object-)relational databases. It is the first paper that provides a profound overview and comparison of semantic integrity support in the new SQL-standard [Int99][1], commonly referred to as SQL:1999 (formerly known as SQL3), and in major commercial database management systems. In particular, we compare the semantic integrity features provided by SQL:1999 and the database management systems Oracle, IBM DB2, Informix, Microsoft SQL-Server, Sybase, and Ingres. We do so by comparing simple integrity features such as primary key and foreign key constraints up to complex features, such as database triggers. We discuss the major differences and similarities among these systems with respect to the definition, semantics, administration, and actual enforcement of various classes of integrity constraints at database run-time.

Interestingly, despite numerous formal and general approaches to integrity-centered database design, in particular in the area of active databases, e.g., [CW90, WC96, CCW00], advanced integrity features provided by commercial systems considered in this paper are rarely used. One reason for this, among others to be discussed in this paper, is that many general approaches typically assume systems with ideal integrity features and system behavior. As we will show in this paper, in practice many systems have limitations that require a deviation from standard formal approaches to constraint design and enforcement. We therefore provide some major guidelines and rules tailored towards designing and implementing integrity constraints using diverse types of constraint enforcing mechanisms realized in commercial systems.

This survey paper is structured as follows: In Section 2, we outline the basic database schema and transaction issues necessary for the rest of the paper. In Section 3, we discuss in detail the diverse integrity features proposed in SQL:1999. In Section 4, we then compare how the different components of the standard are realized in major commercial database management systems. The design and implementation of integrity constraints using the features present in these systems is discussed in Section 5. Besides a brief outline of general guidelines on integrity constraint design, such as constraint decomposition and checking for consistency and non-redundancy of constraints, this section includes in particular the discussion of various design aspects specific to the systems discussed in this paper. We conclude the paper in Section 6 with a discussion of what has been done and what still needs to be done with respect to semantic integrity support in database management systems.

2 Preliminaries

2.1 Basic Notions of (Object-)Relational Models

The main construct for representing data in the relational data model is a relation [Cod70]. A relation is associated with a schema and an instance. A relation schema $\mathsf{R}(\mathsf{c}_1{:}\mathsf{d}_1, \ldots, \mathsf{c}_n{:}\mathsf{d}_n)$ specifies the name R of the relation, the name of each attribute c_i, and the domain d_i of each attribute c_i. A domain is typically referred to by a domain name and has a set of associated values. Examples of basic domains (built-in data types) in SQL are **INTEGER**, **SMALLINT**, **DECIMAL**, **VARCHAR**, or **DATE**, to name a few. Each domain includes the special value **NULL**, meaning that the value is not known or not applicable.

An instance of a relation schema, called *relation*, is a set of tuples in which each tuple has the same structure as the relation schema, i.e., the same number of attributes and the values of the attributes are taken from the corresponding domains. A relational database (or *database*, for short) is a collection of relations. A relational *database schema* is a set of relation schemas.

A relation can naturally be viewed as a *table* in which each tuple is a *row*, and all rows have the same number of columns, where the values of a column are taken from the same domain. For a row r in a table R, denoted $\mathsf{r} \in \mathsf{R}$, the value of a column c_i for that row is denoted $\mathsf{r.c}_i$. A table is either a *base table* or a *derived table*. A derived table is a table that is derived from one or more other tables by the evaluation of a query expression. A *view* is a named derived table.

In the following, we will use the terms table/row/columns instead of (the mathematical notions) relation/tuple/attribute, because the former are also used in SQL as well as in commercial database management systems. Actually, tables are *multisets* of rows, i.e., in contrast to relations, tables allow duplicate rows unless duplicates are explicitly excluded by a uniqueness constraint.

SQL:1999 extends the relational model of SQL-92 (also known as SQL2) [MS93, DD97] towards an *object-relational* model by supporting

- type constructors (**ROW**, **REF**, and **ARRAY**),
- structured (user-defined) types and subtyping, and
- typed tables and subtables.

Type constructors are used to build *constructed* data types. Unfortunately, the type constructors above are not completely orthogonal, e.g., arrays are only one-dimensional since array elements may not be of an array type. Other collection types, e.g., **SET**, **MULTISET**, and **LIST** have been postponed to the next SQL-standard.

A *structured* type is a named, user-defined abstract data type that is defined by implicitly applying the row type constructor to built-in data types, constructed types, and other structured types. In fact, a structured type consists of a set of attributes and methods. A structured type can also be defined as a *subtype* of another structured type, and thereby inherits the attributes and methods of its supertype.

A *typed* table is a base table that is defined based on a structured type. The rows of a typed table are instances (objects) of the corresponding structured type. A typed table can also be defined as a *subtable* of another typed table. A prerequisite, however, is that the structured type of the subtable is a direct subtype of the structured type of the supertable. The

[1] Recent reports on the status of the SQL-standardization efforts can be found in [EM99a, EM99b, EM00]. There are also two books [For99, GP99] that illustrate the new concepts of the SQL-standard by many examples. Since these books were published before the final release of the SQL-standard, they also contain some material that has been postponed to the next SQL-standard SQL:200n (commonly referred to as SQL4) or changed slightly with respect to the draft.

subtable relationship ensures that all rows of a subtable are also members of the corresponding supertable(s).

In the following, a *structured* column refers to a column that is based on the row type constructor either explicitly or implicitly via a structured type. A *reference* column denotes a column that relies on the reference type constructor applied to any structured type. A *collection-valued* column refers to a column whose type is a collection type. Further details about the object-relational extensions of SQL:1999 will be explained in the following sections as needed to understand the corresponding semantic integrity features.

2.2 Basic Notions and Classification of Integrity Constraints

Semantic integrity constraints are formulas that specify what database states and state transitions (and sequences) are admissible, i.e., correctly reflect the real-world semantics. Such predicates can either be associated with a database schema or they can be specified as procedural components of the implementation of the database schema using a particular database management system.

A database changes over time as rows are inserted, updated, or deleted from tables. Such data change operations occur within transactions. A *transaction* basically is a sequence of operations that performs a single logical function in a database application and adheres to the transaction principles atomicity, integrity, isolation, and durability [Gra78]. A transaction induces a transition from a database state to another database state. Each single data change operation is either a data definition statement such as create, alter, or drop a table or a data change statement like insert, update, or delete. In SQL database management systems, a transaction is implicitly started with the first operation following the terminating statement of the previous transaction. A transaction is terminated either through a `COMMIT` or a `ROLLBACK` (abort). An abort causes changes performed by the transaction to be undone.

It is part of the responsibility of the recovery scheme of the database management system to correctly manage transaction aborts. Note that there are (at least) two types of transaction aborts. The first are "technical" aborts that are initiated by the database management system, for instance, to solve a deadlock situation by aborting one or more deadlocked transactions. The second —and for this paper more interesting— type of transaction aborts is related to ensuring semantic integrity constraints by aborting transactions that produce an illegal database state or correspond to an illegal database state transition.

A database state is *consistent* if it satisfies all defined integrity constraints. In the context of transactions and abort operations it is important to distinguish two notions of consistency: *transaction-level consistency* and *statement-level consistency*. Transaction-level consistency requires a transaction to lead from a consistent database state to another (not necessarily different) consistent database state, with possibly inconsistent intermediate database states. Statement-level consistency requires that each statement results in a consistent state. Statement-level consistency is also the consistency notion that is used in SQL:1999. A transaction where each statement satisfies statement-level consistency naturally also satisfies transaction-level consistency. Depending on what type of consistency level is supported by the underlying database management system, a rollback of the entire transaction or a rollback of just the violating statement can be performed in case of a constraint violation. The latter type of mechanisms allows more flexibility since violations of constraints caused by a statement can be compensated by appropriate repairing statements as part of the current transaction. More sophisticated consistency levels can be introduced by grouping sequences of statements and using so-called savepoints [GR93].

Since integrity constraints in general are statements about properties of database objects (columns, rows, and tables), it is important to understand the *enforcement granularity* associated with each integrity constraint. The enforcement granularity of an integrity constraint provides important pointers on appropriate enforcement mechanisms for the integrity constraint.

In general, three types of enforcement granularity can be distinguished. The enforcement granularity is based on what and how many rows and tables are referred to in an integrity constraint. For this, assume a database schema with a set $\mathcal{R} = \{\mathsf{R}_1, \mathsf{R}_2, \ldots, \mathsf{R}_n\}$ of tables. Integrity constraints can be associated with one or more tables from $\mathcal{R}$ and can refer to one or more rows from each such table.

- *Row constraints.* Integrity constraints of this type are associated with exactly one table R_i and can be evaluated independently for each row $\mathsf{r} \in \mathsf{R}_i$. Typical examples of such integrity constraints include domain constraints. A row constraint can refer to one or more columns of a row.
- *Table constraints.* Evaluating this type of integrity constraint involves at least two rows of the same table R_i. The most simple examples of this type of integrity constraint are unique/primary key constraints. More complex examples include aggregate functions where the evaluation involves all or a subset of rows from R_i.
- *Inter-table constraints.* Integrity constraints of this type consider rows from at least two tables R_i and R_j. A typical example of such an integrity constraint is a foreign key constraint. Most general integrity constraints are of this type.

Beside this data-oriented view on enforcement granularity, there exists also a state-oriented view, focusing on how many database states are necessary to evaluate an integrity constraint.

- *State constraints.* Integrity constraints of this type can be evaluated in a single database state, typically the state and contents of tables, respectively, determined by the end of a transaction or statement, depending on whether transaction-level or statement-level consistency should be achieved.
- *State transition constraints.* The evaluation of integrity constraints of this type requires two consecutive database states. Depending on the consistency notion, these are the

state immediately before the transaction (statement) and the state directly after the end of the transaction (statement). In respective constraint evaluations, typically old and new column values of rows are considered. State transition constraints are often easy to identify since they often impose restrictions on old and new column values involved in database modifications, e.g., updates. A typical example of such an integrity constraint is *"The salary of an employee must not increase by more than five percent"*.

– *State sequence (or temporal) constraints.* These are integrity constraints that refer to more than two (not necessarily consecutive) database states. An example of such an integrity constraint is "*Once an employee has been fired by a department, he cannot be hired again by the same department*". The specification and enforcement of temporal integrity constraints has been discussed in detail in, e.g., [GL96,CT95] and will not be covered in this paper.

3 Integrity Support in SQL:1999

The most recent SQL standard SQL:1999 [Int99] supports the definition of integrity constraints both declaratively as well as procedural in terms of triggers. In the following sections, we give an overview of both methods.

3.1 Declarative Definition of Integrity Constraints

An integrity constraint can be specified declaratively within a create table or alter table statement. In the following, we first illustrate the SQL:1999 constructs that concern the specification of integrity constraints by means of examples. The syntax (BNF grammar) and semantics of these constructs will be presented later.

Every integrity constraint is associated with a descriptor that consists of the name of the constraint, the initial checking mode, and a flag indicating whether or not the checking of the constraint can be deferred until the end of a transaction:

1. Integrity constraints are named schema objects such as tables or views. Within a database schema, an integrity constraint is uniquely identified by its name. If a name is not specified explicitly, then the system will implicitly provide an implementation-dependent name.
2. The checking mode determines the relative time when the integrity constraint has to be checked within a transaction. Two modes are distinguished: *immediate* and *deferred*. If the mode is immediate, then the integrity constraint is *effectively* checked at the end of each statement that might violate the integrity constraint. Otherwise, the checking is deferred until the end of the transaction.
3. The initial checking mode defines the default mode that is valid at the beginning of each transaction. Note that only *deferrable* constraints can be set to the deferred mode. The checking mode of a *non-deferrable* constraint always is immediate. The modes *initially immediate* and *non-deferrable* are implicit, if no other is explicitly specified. If *initially deferred* is specified, then *non-deferrable* shall not be specified, and thus *deferrable* is implicit. In principle all integrity constraints can be specified as deferrable.

SQL:1999 provides the following language constructs that can be used to declaratively specify integrity constraints:

– **NOT NULL** prevents a column from taking the null value,
– **DEFAULT** sets the default value of a column,
– **UNIQUE** defines that a column or set of columns must have unique values within a table,
– **PRIMARY KEY** determines the primary key of a table,
– **FOREIGN KEY** (**REFERENCES**) specifies a foreign key whose values must match to that of a unique/primary key,
– **CHECK** defines an integrity constraint based on a search condition.
– **DOMAIN** creates a (restricted) column domain, and
– **ASSERTION** defines a table-independent integrity constraint based on a search condition.

In fact, these language constructs have already been defined in SQL-92. SQL:1999 provides a few extensions, as we will see later. The following example illustrates the usage of the language constructs above.

Example 1. Suppose in our sample application domain we have to manage information about employees. Employees are uniquely identified by an artificial id. We assume that every employee has a first name and a last name. For illustration purposes, we furthermore assume that the combinations of employees' first and last names have to be unique. Every employee has a supervisor, except the top manager who supervises himself. Furthermore, in our sample company, three different types of jobs are distinguished: 'sales', 'manager', and 'clerk'. The "default job" is 'sales'. The salary of an employee must be greater than two thousand but the bonus has to be less than five thousand. Finally, the base salary plus the bonus salary of every employee must be below twenty-five thousand.

```
CREATE DOMAIN jobs CHAR (7)
  DEFAULT 'sales'
  CHECK(VALUE IN ('sales', 'manager', 'clerk'));

CREATE TABLE emp (
  id          SMALLINT PRIMARY KEY,
  lastname    VARCHAR(20) NOT NULL,
  firstname   VARCHAR(15) NOT NULL,
  job         jobs,
  supervisor  SMALLINT REFERENCES emp(id),
  hiredate    DATE DEFAULT CURRENT_DATE,
  salary      DECIMAL(7,2) NOT NULL
              CHECK(salary > 2000),
  bonus       DECIMAL(7,2) NOT NULL
              CHECK(bonus < 5000),
  UNIQUE(firstname, lastname),
  CHECK(salary + bonus < 25000)
);
```

The following assertion specifies that the annual total salary (including bonus) of all employees must be less than five million:

```
CREATE ASSERTION maxyearsal
  CHECK((SELECT 12*(SUM (salary + bonus))
          FROM emp) < 5000000);
```

Note that integrity constraints can also be named, e.g.:

```
CREATE TABLE emp (
  ...
  CONSTRAINT unique_name
             UNIQUE (firstname, lastname),
  CONSTRAINT check_salary
             CHECK(salary + bonus < 25000)
);
```

Constraint names are an important means for debugging database applications as well as for administering integrity constraints within a database, e.g., to dynamically drop or disable an integrity constraint, or to change the checking mode of an integrity constraint. □

Syntax of Declarative Constraints

As stated before, declarative constraints are associated with a checking mode. The specification of the checking mode has to obey the following grammar:

```
<mode> ::= INITIALLY {IMMEDIATE | DEFERRED}
            [[NOT] DEFERRABLE]
           | [[NOT] DEFERRABLE]
            INITIALLY {IMMEDIATE | DEFERRED}
```

SQL classifies declarative constraints into four categories: column constraints, table constraints, domain constraints, and assertions.

Column constraints are specified within the column definition, immediately after the specification of the column data type. There are four types of column constraints: not null constraints, check constraints, unique / primary key constraints, and referential constraints. Column constraints are specified using the following syntax:

```
<column-cons-def> ::= [CONSTRAINT <name>]
                      <column-cons>
                      [<mode>]

<column-cons> ::= NOT NULL
                | CHECK ( <search-condition> )
                | UNIQUE
                | PRIMARY KEY
                | <ref-spec>

<ref-spec> ::= REFERENCES <table-name>
               ( <column-list> )
               [MATCH {SIMPLE | PARTIAL | FULL}]
               [<ref-trig-action>]

<ref-trig-action> ::= ON UPDATE <ref-action>
                       [ON DELETE <ref-action>]
                     | ON DELETE <ref-action>
                       [ON UPDATE <ref-action>]

<ref-action> ::= NO ACTION
               | RESTRICT
               | CASCADE
               | SET NULL
               | SET DEFAULT
```

The specification of a foreign key (references clause) has to obey the following general rules:

- The parent key constraint which is associated with the foreign key constraint must be non-deferrable. The foreign key constraint itself, however, can be deferrable.
- The referenced table must be a base table. Thus, for instance, references to views are not allowed.
- The number of columns of the foreign key must be equal to the number of columns of the parent key. The names of the corresponding columns of the foreign key and the parent key must not necessarily match but their data types must be comparable. For instance, all numbers, i.e., the corresponding data types are comparable. The same holds for all character strings.
- If the references clause does not contain a column list, implicitly the primary key of the parent table is referenced (if the data types match).

With respect to column constraints, SQL:1999 extends the previous standard SQL-92 in two ways. First, it introduces the *restrict* option for referential constraints. Second, it supports a *scope* clause for reference columns, which can be defined using following syntax:

```
<scope> ::= SCOPE <typed-table-name>
            [REFERENCES ARE [NOT] CHECKED
            [ON DELETE <ref-action>]]
```

The default mode is that references are not checked. In this case, dangling references can occur. Although **REFERENCES ARE NOT CHECKED** is default, it can be explicitly specified. If, however, references shall be checked, the keyword **REFERENCES ARE CHECKED** has to be specified explicitly. If this is done without a specification of a referential action, **ON DELETE NO ACTION** is implicit.

We now turn the focus on table constraints, which are defined independently of a column definition. Three types of table constraints are supported: check constraints, unique / primary key constraints, and foreign key constraints. The syntax is similar to column constraints:

```
<table-cons-def> ::= [CONSTRAINT <name>]
                     <table-cons>
                     [<mode>]
```

<table-cons> **::=** **CHECK (** *<search-condition>* **)**
| **UNIQUE (**{*<column-list>* | **VALUE**}**)**
| **PRIMARY KEY (** *<column-list>* **)**
| **FOREIGN KEY (** *<column-list>* **)** *<ref-spec>*

It is important to note that SQL:1999 uses the notion of a table constraint to refer to an integrity constraint that is defined within a table definition but outside the definition of a column. Thus, a table constraint in SQL:1999 can be a row, table, or inter-table constraint according to our classification in Section 2.

Domain constraints are defined within a domain definition, i.e., outside a table definition. The syntax for a domain definition is as follows:

<domain-def> **::=** **CREATE DOMAIN** *<name>*
[**DEFAULT** *<default-value>*]
[*<domain-cons-def-list>*]

<domain-cons-def> **::=** [**CONSTRAINT** *<name>*]
CHECK (*<search-condition>* **)**
[*<mode>*]

Analogously, assertions are defined independently, i.e., outside a table definition. The definition of an assertion has to obey the following grammar:

<assertion-def> **::=** **CREATE ASSERTION** *<name>*
CHECK (*<search-condition>* **)**
[*<mode>*]

As we can see, a check constraint can be defined as a column constraint, a table constraint, a domain constraint, or as an assertion. In all cases, the following rules must hold for the search condition of a check constraint:

- it shall not contain an aggregate function (i.e., **COUNT**, **SUM**, **AVG**, **MIN**, and **MAX**) that is not contained in a subquery;
- it shall not include a call of a non-deterministic routine or of a routine that possibly modifies SQL-data;
- it shall not contain a possibly non-deterministic subquery or function like **CURRENT_DATE**, **CURRENT_TIME**, **CURRENT_USER**, or **SYSTEM_USER**.

Besides, in case of a column check constraint, the search condition must be a Boolean predicate in which the corresponding column is the only variable. In case of a domain check constraint, the search condition must be a Boolean predicate over the corresponding domain (where the domain variable is denoted **VALUE**).

Semantics of Check Constraints

In terms of SQL, a check constraint **CHECK**(SC) defined on a table T is satisfied if and only if

NOT EXISTS(**SELECT** * **FROM** T **WHERE** **NOT**(SC))

yields true. An assertion check constraint **CHECK**(SC) is satisfied if and only if SC yields true or unknown. Remember that SQL follows a three-valued logic with the values *true*, *unknown*, and *false*. In case of a domain check constraint **CHECK**(SC), assume that the corresponding domain is used as type of the column c of table T. Let SC' be the search condition where the domain variable **VALUE** is substituted by the column c. Then, the domain check constraint **CHECK**(SC) used in table T is satisfied if and only if

NOT EXISTS(**SELECT** * **FROM** T **WHERE** **NOT**(SC'))

yields true.

Semantics of Unique and Primary Key Constraints

A uniqueness constraint **UNIQUE**($u_1, ..., u_n$) holds for a table R in a database if and only if there are no two rows r_1, r_2 in R such that the values of all their uniqueness columns u_i match and are not null. More formally ($\perp$ stands for **NULL**):

$$\forall r_1, r_2 \in R\colon \left(\bigwedge_{i=1}^{n} r_1.u_i \neq \perp \wedge r_2.u_i \neq \perp \right) \Rightarrow \left(\bigvee_{i=1}^{n} r_1.u_i \neq r_2.u_i \right)$$

A primary key constraint is a combination of a uniqueness constraint and one or more not null constraints. A constraint **PRIMARY KEY**($u_1, ..., u_n$) holds for a table R if and only if the following holds:

$$\forall r \in R\colon \left(\bigwedge_{i=1}^{n} r.u_i \neq \perp \right) \wedge \forall r_1, r_2 \in R\colon \left(\bigvee_{i=1}^{n} r_1.u_i \neq r_2.u_i \right)$$

Note that no column shall occur more than once within the same unique / primary key definition. Furthermore, each table can have at most one primary key.

Semantics of Foreign Key Constraints

Referential constraints are an important means to express dependencies among (portions of) rows in tables. One typically talks about *referenced* (or *parent*) and *referencing* (or *child*) tables where a subset f_i, ..., f_k of the columns of the referencing table builds the foreign key and refers to the unique / primary key columns u_j, ..., u_l of the referenced table.

Referential constraints can be defined with respect to different match types and referential actions. Depending on the specified match type, a referential constraint is satisfied if one of the following conditions holds:

- If the match type is *simple*, then for each row r of the referencing table R (child table), either at least one of the values of the referencing columns f_1, ..., f_n is a null value or the value of each referencing column f_i, $1 \leq i \leq n$, is equal to the value of the corresponding referenced column u_i for some row s of the referenced table S. More formally:

$$\forall r \in R\colon \left(\bigwedge_{i=1}^{n} r.f_i \neq \perp \right) \Rightarrow \exists s \in S\colon \left(\bigwedge_{i=1}^{n} r.f_i = s.u_i \right)$$

– If the match type is *partial*, then for each row r of the referencing table R, there must be some row s in the referenced table S such that the value of each referencing column f_i is null or equal to the value of the corresponding referenced column u_i in s. More formally:

$$\forall r \in R: \left(\bigvee_{i=1}^{n} r.f_i \neq \perp \right) \Rightarrow \exists s \in S: \left(\bigwedge_{i=1}^{n} r.f_i = \perp \vee r.f_i = s.u_i \right)$$

– If the match type is *full*, then for each row r of the referencing table R, either the value of each referencing column f_i must be a null value or there must be some row s in the referenced table S such that the value of each referencing column f_i is equal to the value of the corresponding referenced column u_i in row s. More formally:

$$\forall r \in R: \left(\bigvee_{i=1}^{n} r.f_i \neq \perp \right) \Rightarrow \exists s \in S: \left(\bigwedge_{i=1}^{n} r.f_i \neq \perp \wedge r.f_i = s.u_i \right)$$

If no match type is defined, then *simple* match is implicit. Having a closer look at the formulas above, we can observe the following facts:

1. Each match type introduces an additional constraint with respect to the previous one. In other words, full match is stronger than partial match and partial match is stronger than simple match.
2. Full match can also be declaratively specified using simple match with an additional check constraint of the following form:

```
CHECK((f1 IS NULL AND... AND fn IS NULL) OR
      (f1 IS NOT NULL AND ... AND
       fn IS NOT NULL))
```

3. The effect of the different match types is equivalent if the referential constraint is specified on exactly one column and/or all referenced or referencing columns are not nullable.

A violation of a referential constraint can occur in the following cases:

1. insert or update operation on the referencing table, and
2. update or delete operation on the referenced table.

Depending on the type of application scenario, sometimes it is more appropriate to maintain referential integrity by performing compensating operations rather than simply rejecting the operation that would violate the referential integrity. Such compensating operations are known as *referential actions*. SQL:1999 allows to define different referential actions to react on updates and deletions performed on the referenced table:

– **CASCADE**: The update/deletion of a referenced row leads to an update/deletion of all rows that reference this row.
– **SET NULL**: Each column f_i of the foreign key of each row that references the updated/deleted row is set to **NULL**. This rule must not be specified if some columns of the foreign key disallow null values.
– **SET DEFAULT**: The update/deletion of a referenced row requires that the foreign keys of all rows that reference this row are set to their default values.
Note that it might happen that there exists no row in the referenced table whose primary key values correspond to the default foreign key. In this case, the update/deletion of the referenced row, and thus the setting to the default foreign key, is disallowed.
– **RESTRICT**: This rule disallows the update/deletion if the row is referenced by a row in any table.
– **NO ACTION**: This rule is similar to the previous one except a few subtle differences, which are described below. **NO ACTION** is implicit if no rule is specified.

As mentioned before, SQL:1999 requires constraint checks to occur effectively at the end each data change statement. The main difference between **RESTRICT** and **NO ACTION** lies in the time of checking (see also Section 3.3). Referential constraints with the restrict rule are checked before all referential constraints of other types. As stated in [Hor92], the restrict rule may cause a data change statement to be rolled back prematurely since referential actions such as cascade or set null may repair intermediate constraint violations. Referential constraints with the no action rule, on the other hand, are checked at the end after all other referential constraints have been processed. Thus, they will not see an intermediate constraint violation unless it remains a violation until the end. Therefore, the no action rule is sometimes called a *soft* restriction while the restrict rule is called a *hard* restriction.

The following example illustrates the specification of referential actions and depicts a further subtle difference between the restrict and no action rules.

Example 2. The following table stores information about departments. Each department has a unique id, a unique name, and is located in one of the cities Zurich, Davis, or Istanbul. It may also have a manager.

```
CREATE TABLE dept (
  id          SMALLINT PRIMARY KEY,
  name        VARCHAR (20) NOT NULL UNIQUE,
  manager     SMALLINT REFERENCES emp(id)
              ON DELETE SET NULL
              ON UPDATE NO ACTION,
  location    Char(9) DEFAULT'Zurich',
  CHECK (location IN ('Zurich', 'Davis', 'Istanbul'))
);
```

The table definition above states that the manager of a department must be an employee of the company. However, the specified referential integrity constraint allows that a department may have no manager. If this shall be disallowed, an additional not null constraint has to be added to the column manager and the **ON DELETE SET NULL** option must be changed into **ON DELETE NO ACTION** or **ON DELETE RESTRICT**.

Assume now that the referenced table emp (see Example 1) contains three rows with the employee ids: 101, 102, and 103, respectively. Furthermore, let the table dept contain a department whose manager is the employee with the id 102. Then, the statement

```
UPDATE emp SET id = id + 1;
```

will be accepted since after the execution the referencing foreign key of the department row has still a parent in the employee table, even though the identity of the referenced employee has changed.

Assume now that **NO ACTION** in the above table definition is replaced by **RESTRICT**. Then, the update statement above will be rejected since the restrict rule does not allow any updates on referenced columns. □

Constraints on Reference Columns

In addition to referential integrity via foreign keys, SQL:1999 also provides the specification of reference types. In this context, a column of a reference type contains the row identifier of the referenced row. A row identifier is a generated, unique value of an artificial column that is implicitly attached to all tables by the database management system. Such reference columns are usually not visible to the database users. Reference columns should also not be confused with columns on which a foreign key is defined. These are two different types of "referencing" columns.

The value of a reference column can be invalidated by deleting the referenced row. To avoid this, referential actions are supported in a fashion analogous to traditional referential constraints: In case of **NO ACTION** and **RESTRICT**, the deletion of the referenced row is disallowed. If **CASCADE** is specified, the deletion of the referenced row implies the deletion of all rows that reference that row. In case of **SET NULL**, the referencing columns are set to null; in case of **SET DEFAULT** to the specified default value. Since SQL:1999 only supports a null specification in the default clause of a reference column, the latter two options are semantically equivalent.

Example 3. A reference column is defined based on a structured type. A structured type can be specified as follows:

```
CREATE TYPE project_type AS (
  id        SMALLINT,
  name      VARCHAR (20),
  manager   SMALLINT
) NOT FINAL;
```

The keyword **NOT FINAL** states that this structured type may have subtypes. Typed tables then can be defined based on this structured type, as shown in the following example:

```
CREATE TABLE external_projects OF project_type (
  REF IS oid SYSTEM GENERATED
);

CREATE TABLE internal_projects OF project_type (
  REF IS oid SYSTEM GENERATED
);
```

SQL:1999 requires that the *reference generation* clause is specified for every root table. This clause indicates the way how object identifiers are generated for each row (object) of such a typed table.

Consider now the following reference column definition:

```
CREATE TABLE emp (
  ...
  project    REF (project_type)
             SCOPE internal_projects
             REFERENCES ARE CHECKED
             ON DELETE CASCADE
);
```

The column project references an instance of the structured type project_type. As we have seen above, there may exist several tables of this structured type. To restrict the reference column to a particular table, for instance, to the table internal_projects, the scope clause has to be used. Here, we also specify that references are automatically checked whenever a row is deleted from the referenced table. In particular, if the deletion concerns a currently referenced row, all rows that reference this row are deleted, too. □

Moreover, note that a reference column may contain the null value. To prevent this, a not null constraint can be defined for this column in the same way as for normal columns.

Inheritance of Integrity Constraints

SQL:1999 supports subtyping and subtables. In a table hierarchy, every integrity constraint implicitly holds for each corresponding subtable. Often this fact is denoted as inheritance of integrity constraints. Integrity constraints are inherited from a supertable to a subtable in the sense that they must also hold for the subtable. Overriding (replacing) of inherited integrity constraints in subtables is not supported.

As a rule, a table shall contain at most one implicit or explicit primary key. An implicit primary key is one that is inherited from a supertable. A table that has an explicit primary key must not be a subtable.

Example 4. Assume the following type definition:

```
CREATE TYPE emp_type AS (
  id          SMALLINT,
  lastname    VARCHAR(20),
  firstname   VARCHAR(15),
  job         jobs,
  supervisor  REF(emp),
  hiredate    DATE DEFAULT CURRENT_DATE,
  salary      DECIMAL(7,2),
  bonus       DECIMAL(7,2)
) NOT FINAL;
```

Based on this structured type, a subtype can be defined using the **UNDER** clause as follows:

```
CREATE TYPE mng_type UNDER emp_type AS (
  projects    SMALLINT ARRAY[10]
) NOT FINAL;
```

The subtype mng_type inherits all properties (attributes and methods) of the type emp_type. Apart from the definition of a default clause, SQL:1999 does not support the specification of integrity constraints within a type definition. Integrity constraints are specified within table, assertion, or domain definitions.

The structured types above can be used for the definition of typed tables, for instance, as follows:

```
CREATE TABLE emp OF emp_type (
  REF IS oid SYSTEM GENERATED,
  lastname    WITH OPTIONS NOT NULL,
  firstname   WITH OPTIONS NOT NULL,
  supervisor  WITH OPTIONS SCOPE emp
              REFERENCES ARE CHECKED
              ON DELETE NO ACTION,
  salary      WITH OPTIONS NOT NULL
              CHECK(salary > 2000),
  bonus       WITH OPTIONS NOT NULL
              CHECK(bonus < 5000),
  CHECK(salary + bonus < 25000)
);
```

In contrast to the reference generation clause, the definition of integrity constraints is optional. Attribute constraints are defined using the *with options* clause; table constraints are specified analogously to usual untyped tables.

A subtable can be defined using the *under* clause as follows:

```
CREATE TABLE manager OF mng_type UNDER emp (
  salary      WITH OPTIONS NOT NULL
              CHECK(salary > 6000)
);
```

The subtable relationship requires that the type of the subtable must always be a direct subtype of the type of its direct supertable, as it is the case in our example above. When defined, it ensures that all rows of the subtable are also members of the corresponding supertable(s). In this way, the integrity constraints of the supertable(s) are also checked for the rows of each subtable. □

3.2 Procedural Definition of Integrity Constraints by Triggers

SQL:1999 also provides the concept of triggers. A *trigger* is a procedure that is automatically invoked by the database management system in response to specified database events. Triggers can be viewed as event-condition-action (ECA) rules that allow users to implement application logic within the database management system. Triggers can be used to monitor modifications of the database (e.g., for auditing purposes), to automatically propagate database modifications, to support alerts, or to enforce integrity constraints. In this paper, we only focus on the latter feature.

A trigger in SQL:1999 has the following components:

- a unique *name* identifies the trigger within the database;
- a triggering *event* which is an **INSERT**, a **DELETE**, or an **UPDATE** on a base table;
- an *activation time* which is **BEFORE** or **AFTER** executing the triggering event;
- a trigger *granularity* which is **FOR EACH ROW** or **FOR EACH STATEMENT**;
- a trigger *condition* which can be any valid SQL-condition involving complex queries;
- a triggered *action* which can be any valid sequence of SQL-procedure statements;
- a trigger *timestamp* stating when the trigger was created.

A trigger is implicitly activated whenever the specified event occurs. Thus, the database can react on changes made by applications or ad-hoc users. Several triggers may refer to the same event. In case several triggers are fired at the same time, the oldest trigger with respect to its creation timestamps is executed first. If two or more triggers have the same timestamp, e.g., when they are defined in the same create schema statement, then their relative order is implementation-defined. Note that in contrast to general ECA-rules [Buc94, PD99] the execution of the condition and action of SQL:1999 triggers can be neither *deferred* nor *detached*.

The grammar of the trigger definition in SQL:1999 is as follows:

```
<trig-def> ::= CREATE TRIGGER <name>
               {BEFORE | AFTER}
               {INSERT | DELETE | UPDATE
                [OF <column-list>]}
               ON <table-name>
               [REFERENCING <old-new-list>]
               [FOR EACH {ROW | STATEMENT}]
               [WHEN ( <search-condition> )]
               <triggered-SQL-statement>

<old-new> ::= {OLD | NEW} [ROW] [AS] <alias>
              | {OLD | NEW} TABLE [AS] <alias>

<trig-SQL-stmt> ::= <SQL-procedure-stmt>
                    | BEGIN ATOMIC
                        <SQL-procedure-stmt-list>
                      END
```

Depending on the specified triggering event, a trigger is referred to as an *insert*, *update*, or *delete* trigger.

A trigger condition and trigger action can be verified and executed, respectively, for each row affected by the triggering statement or once for the whole triggering event (for each statement). In the former case, the trigger is fired for each row affected by the event. In the latter case, for all rows affected by the triggering statement, the trigger is only fired once. If no rows are affected by the triggering statement, then the trigger is not evaluated. A trigger that is specified using **FOR EACH ROW** is called a *row-level* trigger. Otherwise, if **FOR EACH STATEMENT** is implicitly or explicitly specified, it is called a *statement-level* trigger. The default mode is **FOR EACH STATEMENT**.

The activation times **BEFORE** and **AFTER** specify when the trigger should be fired, that is, either before the triggering event is performed or after the triggering event. Accordingly, a trigger is referred to as a *before* or *after* trigger. The action of a before trigger shall contain neither an SQL-data change statement (insert, update, or delete) nor an invocation of a routine that possibly modifies SQL-data. However, it can contain **SET** statements that change the value of **NEW** transition variables referring to rows that have been modified by the triggering event. In contrast to normal data modifications, a **SET** statement that modifies a transition variable is not considered as a data modification event and thus does not cause any trigger to fire.

Trigger conditions and actions can refer to both old and new values of the rows affected by the triggering event. Such rows can be seen individually or jointly, depending on the trigger granularity, as *transition* tables. If either the condition or action in a trigger refers to modified rows, the *referencing* clause allows transition tables to be explicitly named. The referencing clause of an insert trigger shall include neither **OLD ROW** nor **OLD TABLE**. Analogously, the referencing clause of a delete trigger shall include neither **NEW ROW** nor **NEW TABLE**. Moreover, **OLD ROW** as well as **NEW ROW** shall only be specified for row-level triggers. **OLD TABLE** and **NEW TABLE**, however, shall only be specified for after triggers.

Example 5. In many commercial database systems, triggers are used to implement the different referential actions because that are not completely supported by declarative means. A trigger specification for implementing the referential actions described in Example 2 looks as follows:

```
CREATE TRIGGER on_delete_set_null
  AFTER DELETE ON emp
  REFERENCING OLD AS pre
  FOR EACH ROW
  BEGIN ATOMIC
    UPDATE dept
    SET supervisor = NULL
    WHERE supervisor = pre.id
  END;

CREATE TRIGGER on_update_cascade
  AFTER UPDATE OF id ON emp
  REFERENCING OLD AS pre NEW AS post
  FOR EACH ROW
  WHEN(EXISTS(SELECT *
              FROM dept
              WHERE supervisor = pre.id))
  BEGIN ATOMIC
    UPDATE dept
    SET supervisor = post.id
    WHERE supervisor = pre.id
  END;
```

In the last trigger, a compensating action is specified that "resets" the value of the column supervisor for all updated rows of the table emp. □

Up to now, our considerations were restricted to static integrity constraints alone. Clearly, triggers are predestined to implement state transition constraints, that is, integrity constraints that refer to the (intermediate) database states before and after the triggering event. The specification of such integrity constraints obviously requires the usage of transition tables recording information about rows modified by the triggering event.

Example 6. The following trigger ensures that the salary of an employee can never decrease:

```
CREATE TRIGGER salary_never_decrease
  AFTER UPDATE OF salary ON emp
  REFERENCING OLD AS pre NEW AS post
  FOR EACH ROW
  WHEN(pre.salary > post.salary)
  BEGIN ATOMIC
    SIGNAL 1000 ('Salary must not decrease!');
  END;
```

Note that an explicit rollback of the current transaction is not possible. As shown above, instead an exception is thrown. An exception implicitly causes a partial rollback to the beginning of the triggering action. The application may then respond to the exception, for instance, by rolling back the entire transaction. □

Note that adding a trigger to a non-empty table will not cause any firing. Thus, a trigger that is designed to ensure an integrity constraint may not have the intended effect if the corresponding integrity constraint is not satisfied when the trigger is created.

SQL:1999 does not explicitly say anything about inheritance of triggers. However, due to the subtable relationship, one can assume that triggers are inherited from supertables to subtables similar to integrity constraints. For instance, an insertion into a subtable implies an insertion into the corresponding supertable(s), and thereby can fire an insert trigger defined on the supertable(s). For a detailed discussion on trigger inheritance in a more general context, we refer to [BGM00].

It is important to point out that the concept of a trigger is independent of integrity constraints. Triggers should be used for integrity enforcement purposes only if an integrity constraint cannot be expressed appropriately by declarative means. For a more detailed discussion of SQL:1999 triggers in a more general context, we refer to [For99, KMC99]. An overview of the different facets of triggers in general can be found in [CPM96, WC96, ZCF+97, Cha98, Pat99].

3.3 Constraint Checking and Trigger Execution

If a declarative constraint is violated, an exception is thrown and a rollback to the beginning of the statement that caused the violation is performed. At this point, the application then can appropriately respond to the violation, either by requesting a rollback of the whole transaction (transaction-level consistency) or by proceeding with some other action (statement-level consistency).

We have already mentioned that the checking of an integrity constraint can also be deferred to the commit time of the transaction. The *set constraint mode* statement can be used to change the checking mode of deferrable integrity constraints with respect to the current transaction as follows:

SET CONSTRAINTS unique_name, check_salary
DEFERRED;

The full syntax of the set constraint mode statement is as follows:

<*set-mode*> **::=** **SET CONSTRAINTS**
{**ALL** | <*constraint-name-list*>}
{**IMMEDIATE** | **DEFERRED**}

When the keyword **ALL** is used instead of a list of constraint names, the mode of all deferrable integrity constraints is set to deferred. The commit of a transaction implicitly executes the statement **SET CONSTRAINTS ALL IMMEDIATE**. In this way, all deferred integrity constraints are enforced at commit time.[2]

Since SQL:1999 supports both declarative integrity constraints and procedural triggers, the semantics of their possible interaction has to be defined clearly. This includes the definition of the precise timing of constraint checking and trigger execution. As noted in [Mar91, Mar94], an incorrect evaluation of referential constraints alone can lead to anomalous non-deterministic behavior. Deterministic behavior can be achieved by a fixpoint evaluation of integrity constraints that is independent of the order in which rows and integrity constraints are processed [Hor92]. It should be noted that [RH96] proposed an approach to efficiently verify referential integrity in SQL-92 based on access path support.

Figure 1 sketches the processing order of declarative constraints and triggers. For a detailed discussion about the interaction of constraint checking and trigger processing, we refer to [CPM96, Cha98, KMC99].

Before triggers are executed before the changes of the triggering operation are actually performed on the database. Like declarative constraints, the execution of before triggers must be monotonic due to the fixpoint computation of the transition tables. This is also the reason why data change statements are disallowed in before triggers. After triggers are executed after the triggering operation has been completely performed on the database and all declarative integrity constraints (including domain constraints and assertions) have been enforced. As stated in [CPM96, KMC99], these strict restrictions are needed to ensure the deterministic execution of SQL-statements.

[2] When we talk about commit of a transaction here, we actually mean the "request-to-commit" event of that transaction. A transaction is then committed if it satisfies all specified (and enabled) integrity constraints.

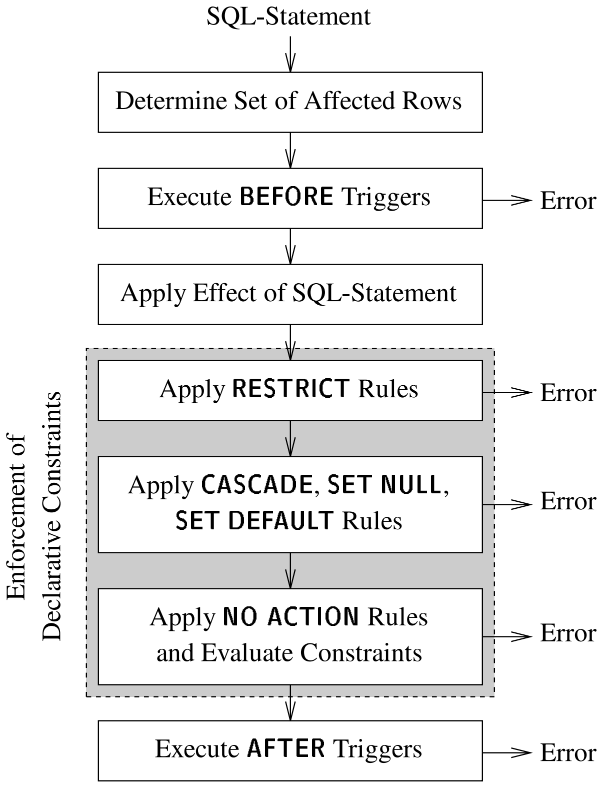

Fig. 1 Trigger Execution and Constraint Checking [CPM96]

An interesting fact is that the order of the execution of before and after triggers, respectively, exclusively depends on the creation timestamps of the triggers. That is, in particular there is no rule stating that row-level triggers have to be executed before statement-level triggers, as it is the case in some commercial systems (see Section 4).

The order of the enforcement of all applicable declarative constraints is as follows: First the referential constraints with the **RESTRICT** semantics are evaluated. If these constraints are satisfied, then the referential constraints with the **CASCADE**, **SET NULL**, or **SET DEFAULT** semantics are enforced. Finally, if all previous integrity constraints are satisfied, not null constraints, unique / primary key constraints, check constraints, domain constraints, assertions, and referential constraints with the **NO ACTION** semantics are evaluated.

As mentioned before, the cascaded effects of referential constraints are evaluated until a fixpoint is reached. Since triggers do not contain the concept of a fixpoint, the constraint model of SQL:1999 cannot be accurately simulated by triggers alone.

4 Comparison of Integrity Support in SQL:1999 and Commercial Database Management Systems

In this section, we compare the semantic integrity features of SQL:1999 [Int99] with those of major commercial (object-) relational database management systems. Our comparison includes the following systems:

- Oracle8i JServer Release 8.1.6 [Ora99],
- IBM DB2 Universal Database Version 7 [IBM00],
- Informix Dynamic Server.2000 Version 9.2 [Inf99],
- Microsoft SQL Server Version 7.0 [Mic99],
- Sybase Adaptive Server Enterprise Edition Version 11.5 [Syb99],
- Ingres II Release 2.0 [Ing99], and
- Sybase Adaptive Server Anywhere Edition Version 6.0.3 [Syb00].

In the following, we will use the abbreviations Oracle, DB2, Informix, MSSQL, Sybase, Ingres, and Sybase^{Any}, respectively, to refer to these systems.

Oracle, DB2, and Informix provide object-relational extensions, whereas MSSQL, Sybase, Ingres, and Sybase^{Any} are pure relational database management systems. Before we compare the means for semantic integrity support, we briefly sketch the object-relational extensions of Oracle, DB2, and Informix.

4.1 Comparison of Object-Relational Extensions

The object-relational models of Oracle, DB2, and Informix differ in various ways. Table 1 sketches the object-relational extensions supported in these systems.

Table 1 Object-Relational Extensions

	SQL:1999	Oracle	DB2	Informix
row type	✓	—	—	✓
reference type	✓	✓	✓	—
collection types	(✓)	(✓)	—	(✓)
structured types	✓	✓	✓	(✓)
type hierarchies	✓	—	✓	✓
typed tables	✓	✓	✓	✓
table hierarchies	✓	—	✓	✓

The row type constructor is only provided by Informix. In contrast, the reference type constructor is supported in Oracle and DB2 but not in Informix. DB2 does not provide any collection type. From the previous section, we know that the array type constructor is the only collection type constructor in SQL:1999. The array type constructor is provided in Oracle, too. Oracle moreover supports the table type constructor, which actually defines a multiset of rows. Unfortunately, these two collection type constructors cannot be nested. That is, the type of an array element as well as of a table component can neither be an array type nor a table type. Informix, on the other hand, supports set, multiset, and list type constructors in an orthogonal fashion.

Structured types are supported by all of these systems. However, we have to point out that Oracle and DB2 (more or less) follow SQL:1999 and support structured types as abstract data types whose instances are objects with object identifiers and methods. Informix, in contrast, provides named row types as structured types. Named row types are not associated with object identifiers and methods. Nevertheless, type hierarchies (subtypes) are supported for named row types, i.e., a named row type can inherit the attributes of another named row type. Subtyping is also supported in DB2 but not in Oracle. In DB2, a structured type can inherit the attributes and methods of another structured type.

In all three systems, typed tables can be defined based on structured types. Table hierarchies (subtables), however, are only supported by DB2 and Informix.

Throughout this paper, we will use the SQL:1999 notions for the concepts presented above, although particularly Oracle uses a different naming. In Oracle structured types are denoted object types, while typed tables are called object tables.

4.2 Comparison of Declarative Integrity Constraints

Table 2 summarizes the support of declarative integrity constraints in the various systems. The symbol '✓' states that the language construct is provided in the corresponding SQL dialect; '(✓)' marks the cases in which the concept is supported but with a syntax different from SQL:1999. For instance, all reference systems support the foreign key clause but only with implicit simple match semantics, i.e., without providing the keyword `MATCH SIMPLE`.

The interesting issues of the comparison will be discussed in more detail in the following.

Nullability and Default Values

Specification of not null constraints. In all reference systems, a not null constraint is defined either by using the keyword `NOT NULL` directly in the column specification or in a check clause of the form `CHECK`(column `IS NOT NULL`).

Default nullability. Whereas in nearly all of these systems columns are nullable by default unless null values are explicitly disallowed, in Sybase columns are implicitly defined as non-nullable. Null values have to be explicitly allowed using the keyword `NULL` in the column specification.

Restrictions on the order of the specification of declarative constraints and default values. In Oracle and Sybase, the specification of the default value must precede the specification of other integrity constraints. In Informix, the specification order is: default value, not null constraint, and then other integrity constraints in arbitrary order. DB2, MSSQL, Ingres, and Sybase^{Any} have no restrictions on the order.

Nullability and object-relational type extensions. Not null constraints can be specified for columns of constructed or structured types in the same way as usual columns (of standard built-in data types). In Oracle and DB2, not null constraints can also be specified individually for any component

Table 2 Comparison of Declarative Integrity Support

			SQL:1999	Oracle	DB2	Informix	MSSQL	Sybase	Ingres	Sybase^{Any}
NOT NULL			✓	✓	✓	✓	✓	✓	✓	✓
DEFAULT			✓	✓	✓	✓	✓	✓	✓	✓
UNIQUE			✓	✓	✓	✓	✓	✓	✓	✓
PRIMARY KEY			✓	✓	✓	✓	✓	✓	✓	✓
FOREIGN KEY	MATCH	SIMPLE	✓	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)
		PARTIAL	✓	—	—	—	—	—	—	—
		FULL	✓	—	—	—	—	—	—	—
	ON DELETE	NO ACTION	✓	(✓)	✓	(✓)	(✓)	(✓)	(✓)	—
		RESTRICT	✓	—	✓	—	—	—	—	✓
		CASCADE	✓	✓	✓	✓	—	—	—	✓
		SET NULL	✓	✓	✓	—	—	—	—	✓
		SET DEFAULT	✓	—	—	—	—	—	—	✓
	ON UPDATE	NO ACTION	✓	(✓)	✓	(✓)	(✓)	(✓)	(✓)	—
		RESTRICT	✓	—	✓	—	—	—	—	✓
		CASCADE	✓	—	—	—	—	—	—	✓
		SET NULL	✓	—	—	—	—	—	—	✓
		SET DEFAULT	✓	—	—	—	—	—	—	✓
CHECK	column-level		✓	✓	✓	✓	✓	✓	✓	✓
	row-level		✓	✓	✓	✓	✓	✓	✓	✓
	table-level		✓	—	—	—	—	—	—	—
	database-level		✓	—	—	—	—	—	—	—
DOMAIN			✓	—	—	—	—	—	—	✓
ASSERTION			✓	—	—	—	—	—	—	—

of a structured column. The following check clause shows the definition of a not null constraint for the component city of the structured column address (whose values are supposed to be instances of a structured type consisting of several components like city, street, and zip):

```
CHECK(address.city IS NOT NULL)
```

The following example illustrates a problem that has to be considered when dealing with structured types and not null constraints.

Example 7. Assume the following type and table definitions (given in SQL:1999 syntax):

```
CREATE TYPE name_type AS (
  first VARCHAR (20), last VARCHAR (20)
);
CREATE TABLE emp (
  ...
  name   name_type NOT NULL
);
```

The following insertion is rejected due to a violation of the not null constraint above:

```
INSERT INTO emp(name)
VALUES (null);
```

However, the following insertion is accepted:

```
INSERT INTO emp(name)
VALUES (name_type(null, null));
```

Obviously, these two values have different semantics. In order to disallow the latter one, too, not null constraints have to be defined on each component of the column name, for instance, as follows:

```
CREATE TABLE emp (
  ...
  name   name_type NOT NULL,
  CHECK((name.first IS NOT NULL) AND
        (name.last IS NOT NULL))
);
```

Such check constraints are possible in Oracle and DB2 but not in Informix. □

In Table 3, we have summarized the support of not null constraints for the type extensions in the object-relational database systems Oracle, DB2, and Informix.

Table 3 Not Null Constraints and Object-Relational Type Extensions

NOT NULL on	SQL:1999	Oracle	DB2	Informix
row type	✓	*	*	✓
reference type	✓	✓	✓	*
collection type	✓	—	*	✓
structured type	✓	✓	✓	(✓)

The '(✓)' for Informix means that not null constraints are only supported at the structure level for columns of a named row type (structured type). In other words, not null constraints cannot be defined on components of such a column. A '—' states that not null constraints are not supported for columns of corresponding type; a '$*$' says that the corresponding type extension is generally not supported by the system.

Example 8. The following table definition is supported in Informix:

```
CREATE TABLE Person (
  name    ROW(first VARCHAR(15) NOT NULL,
              last VARCHAR(20) NOT NULL)
              NOT NULL,
  phones  LIST(VARCHAR(20) NOT NULL)
              NOT NULL,
  hobbies SET(VARCHAR(25) NOT NULL)
              NOT NULL
);
```

Not null constraints are accepted for row-valued columns of unnamed row types as well as for single components of such columns. In case of collections, all elements must always be not null anyway, i.e., not null is mandatory for collection elements. However, an additional not null constraint can be defined for the whole collection. □

Unique / Primary Key Constraints

Semantics of unique / primary key constraints. In DB2, Informix, MSSQL, Sybase, Ingres, and Sybase^{Any}, the semantics of a uniqueness constraint is defined as follows:

$$\mathsf{UNIQUE}(u_1, ..., u_n) \text{ holds on table R } \Leftrightarrow \forall r_1, r_2 \in R: \left(\bigvee_{i=1}^{n} r_1.u_i \neq r_2.u_i \right)$$

That is, no two rows shall have the same value combination for the uniqueness columns u_1, ..., u_n. Oracle defines the semantics of a uniqueness constraint less restrictive:

$$\mathsf{UNIQUE}(u_1, ..., u_n) \text{ holds on table R } \Leftrightarrow \forall r_1, r_2 \in R: \left(\bigvee_{i=1}^{n} r_1.u_i \neq \perp \vee r_2.u_i \neq \perp \right) \Rightarrow \left(\bigvee_{i=1}^{n} r_1.u_i \neq r_2.u_i \right)$$

Compared to the other systems, Oracle additionally allows that there may exist two rows whose uniqueness columns completely consist of null values. Also note the difference between these two semantics and the definition in SQL:1999.

Nullability of unique / primary key columns. DB2, MSSQL, Ingres, and Sybase^{Any} do not allow unique / primary key constraints on nullable columns.

In Oracle, Informix, and Sybase, the definition of a primary key implicitly defines not null constraints on the corresponding uniqueness columns. This strategy is also proposed in SQL:1999.

Integrity violation message in case of simultaneous violation of multiple uniqueness constraints. Oracle, MSSQL, and Sybase print the name of the uniqueness constraint which has the oldest timestamp (with respect to its recent enabling time). DB2, Informix, Ingres, and Sybase^{Any} only state that there is an integrity violation due to a uniqueness constraint.

Multiple uniqueness constraints on same sets of columns. In Oracle, Informix, Sybase, and Sybase^{Any}, the definition of a uniqueness constraint is rejected if there already exists a uniqueness constraint that is defined on the same order of columns. DB2 and Ingres are even more restricted. Here, as in SQL:1999, the creation of a uniqueness constraint is rejected if there already exists a uniqueness constraint that is defined on the same set of columns. MSSQL has no restrictions concerning the definition of uniqueness constraints on common columns.

Unique / Primary key constraints and object-relational type extensions. Table 4 summarizes the applicability of unique / primary key constraints on columns of row, reference, collection, and structured types in the object-relational database systems Oracle, DB2, and Informix. Oracle allows the definition of unique / primary key constraints on components of a structured column but not on the whole structure (column). Note that the components have to be based on standard built-in data types. DB2 supports the definition of unique / primary key constraints on reference columns.

Table 4 Unique/Primary Key Constraints and Object-Relational Type Extensions

UNIQUE on	SQL:1999	Oracle	DB2	Informix
row type	✓	$*$	$*$	—
reference type	✓	—	✓	$*$
collection type	—	—	$*$	—
structured type	✓	(✓)	—	—

Foreign Key Constraints

Referring back to Table 2, all reference systems implement the simple match rule for referential constraints. MSSQL, Sybase, and Ingres purely rely on the no action rules for deletions from and updates to the referenced table. The '(✓)' marked fields in Table 2 state that the corresponding systems implement this semantics but do not support the keywords **ON DELETE NO ACTION** and **ON UPDATE NO ACTION**. Sybase^{Any}, in contrast, supports a broad range of referential actions. Only the no action keyword is not supported. We have to mention here that Sybase^{Any}'s restrict rule actually corresponds to the no action rule in SQL:1999. The delete cascade rule is furthermore supported by Oracle, DB2, and Informix; the delete set null rule by Oracle and DB2; and the restrict rule by DB2.

If a table in DB2 contains more than one foreign key, then all foreign keys must have the same delete rule as the first foreign key defined in this table. Surprisingly, in case the first foreign key is based on the **ON DELETE SET NULL** rule, DB2 disallows the definition of multiple foreign keys. Actually, these restrictions were introduced to avoid constraint evaluation anomalies as described in [Hor92].

Example 9. DB2 disallows, for instance, the following constraint definitions:

```
CREATE TABLE dept (
  dept_id   SMALLINT NOT NULL PRIMARY KEY
);

CREATE TABLE proj (
  proj_id   SMALLINT NOT NULL PRIMARY KEY,
  dept_id   SMALLINT NOT NULL
            REFERENCES dept(dept_id)
            ON DELETE CASCADE
);

CREATE TABLE emp (
  emp_id    SMALLINT NOT NULL PRIMARY KEY,
  proj_id   SMALLINT NOT NULL,
  dept_id   SMALLINT NOT NULL
            REFERENCES dept(dept_id)
            ON DELETE SET NULL,
  FOREIGN KEY (proj_id, dept_id)
            REFERENCES proj(proj_id, dept_id)
            ON DELETE CASCADE
);
```

These definitions (if accepted) could lead to race conditions. Assume that there is row in the table dept that is referenced by rows in the tables emp and proj. The deletion of the referenced row may have non-deterministic effects depending on which of the two **ON DELETE CASCADE** rules is triggered first. If the referencing rows are first deleted from the table proj, then the corresponding foreign key columns of the referencing rows in the table emp are set to null. In the other case, the referencing rows are also deleted from the table emp.

Sybase^{Any}, however, accepts the definitions above. In our experiments Sybase^{Any} always applied the second strategy described above, i.e., the referencing rows were also deleted from the table emp. □

Constraints on Reference Columns

Reference columns can only be specified in Oracle and DB2. The scope of such columns is restricted using the scope clause either within or outside the column specification. Oracle supports both forms of the scope constraint definition:

```
column-based: ref_column REF ref_type
                         SCOPE IS typed_table
table-based:  SCOPE FOR ref_column IS typed_table
```

DB2 only supports a column-based scope clause as follows:

```
column-based: ref_column REF(ref_type)
                         SCOPE typed_table
```

It is interesting to note that DB2 does not avoid the problem of dangling references. An object may be deleted although it is referenced by other objects. Oracle, in contrast, avoids dangling references by setting the values of the corresponding reference columns to null. Unfortunately, this is even done in case a not null constraint is defined for one of the corresponding reference columns. Another possibility to avoid dangling references in Oracle is to explicitly use referential constraints for reference columns. For instance, a referential constraint with the **ON DELETE CASCADE** option is defined for a reference column as follows:

```
column-based: ref_column REF ref_type
                         REFERENCES typed_table
                         ON DELETE CASCADE
table-based:  ref_column REFERENCES typed_table
                         ON DELETE CASCADE
```

Oracle moreover allows to explicitly specify the **ON DELETE SET NULL** rule for reference columns. This rule is the default setting. If a referential constraint is added to an existing reference column that is scoped, then the referenced table must be the same as the scope table of the reference column. The system adds a scope constraint when a referential constraint is added to an existing unscoped reference column. If the referential constraint is dropped later, the reference column will remain scoped to the referenced table.

Check Constraints

All reference systems support check constraints at the row level. That is, the Boolean expression of the check clause must not contain subqueries, aggregate functions, host variables, stored procedure calls, pseudo-columns (like **ROWID** or **ROWNUM**), columns of different tables, and certain function calls (like **USER** or **CURRENT_DATE**).

Considering object-relational extensions, it is important to know how far columns of constructed / structured types can be used within the Boolean expression of a check constraint. In particular, the following questions are of interest:

1. Can we restrict components of row-valued and structured columns?
2. Can we formulate predicates including path expressions?
3. Can we define predicates on elements of a collection-valued column?

Table 5 summarizes the answers to these questions. Here, a '(✓)' states that a check constraint can only define a not null constraint on a column of the corresponding type. In these cases, none of the questions above can be answered with yes.

The only check constraint that is directly applicable to a row-valued or structured column is a not null constraint. Informix supports this either by specifying the corresponding keyword directly or using a check constraint. However, Informix does not support check constraints on components

Table 5 Check Constraints and Object-Relational Type Extensions

CHECK on	SQL:1999	Oracle	DB2	Informix
row type	✓	*	*	(✓)
reference type	✓	✓	(✓)	*
collection type	✓	(✓)	*	✓
structured type	✓	✓	(✓)	(✓)

of row-valued or structured columns. For instance, one cannot state that the component city of a structured type address_type must begin with a 'D' or 'Z'. That is, path expressions are not allowed with a check constraint. On the other hand, Informix allows the definition of check constraints on collection-valued columns as well as on collection elements.

Example 10. The table definition of Example 8 is now extended by check constraints:

```
CREATE TABLE person (
  name     ROW(first VARCHAR(15) NOT NULL,
               last VARCHAR(20) NOT NULL),
  phones   LIST(VARCHAR(20) NOT NULL),
  hobbies  SET(VARCHAR(25) NOT NULL),
  CHECK(name IS NOT NULL),
  CHECK(phones IS NOT NULL),
  CHECK(hobbies IS NOT NULL),
  CHECK(CARDINALITY(phones) < 4),
  CHECK('Databases' IN hobbies OR
        'Integrity' IN hobbies)
);
```

The first three check constraints illustrate the definition of not null constraints for both row-valued and collection-valued columns, respectively. The fourth check constraint ensures that every person has at most three phone numbers, while the fifth check constraint guarantees that at least one of the terms "Databases" or "Integrity" has to occur in the set-valued column hobbies. □

DB2 does support check constraints neither on components of a structured column nor on components of a referenced object. The former, however, is possible in Oracle.

Example 11. Assume the following table definition in Oracle (where address_type and dept_type are supposed to be previously defined structured types):

```
CREATE TABLE emp (
  name     VARCHAR(35),
  address  address_type,
  dept     REF dept_type SCOPE IS department,
  CHECK(address IS NOT NULL),
  CHECK(dept IS NOT NULL),
  CHECK(address.street IS NOT NULL),
  CHECK(address.city LIKE 'D%' OR
        address.city LIKE 'Z%'),
  CHECK(DEREF(dept) IS NOT NULL)
);
```

The first two check constraints illustrate the definition of not null constraints on structured and reference columns, respectively. As shown by the third and fourth check constraint, Oracle allows to formulate conditions on the components of structured columns. In case of reference columns, however, access to components of a referenced object via dereferencing and attribute selection is not supported. It is only possible to define that the value of a referenced object shall be not null. This is exactly specified by the fifth check constraint. In summary, Oracle allows path expressions in a check constraint if they do not include a dereferencing. □

In the tests we conducted, all reference systems accept inconsistent and redundant constraint specifications. Even check constraints of the form

```
CHECK(salary = 2000 AND salary <> 2000)
```

are accepted by all reference systems. Mostly, only syntax checking is performed. At constraint definition time, Oracle, DB2, Informix, MSSQL, and Ingres check whether or not the column names specified in a check constraint are valid with respect to the table on which the integrity constraint is defined. Sybase and Sybase^{Any} do not detect this error until the next insert, update, or delete operation. In summary, it is up to the designer to specify consistent and non-redundant integrity constraints. We will discuss this issue in Section 5.

Domains and Assertions

The concept of a domain is only supported by Sybase^{Any}. Here, Sybase^{Any} follows the given syntax and semantics of SQL:1999.

The concept of an assertion is not supported by any of the reference systems!

Inheritance of Integrity Constraints

Subtyping and subtables are supported by DB2 and Informix. Following the inheritance rules of SQL:1999, integrity constraints are inherited from supertables to subtables in both systems. While DB2 ensures that each table has at most one primary key, this is not the case in Informix.

Example 12. The following type and table definitions are allowed in Informix:

```
CREATE ROW TYPE atype (
  x INTEGER
);

CREATE TABLE atable OF TYPE atype (
  PRIMARY KEY(x)
);

CREATE ROW TYPE btype (
  y INTEGER
) UNDER atype;
```

```
CREATE TABLE btable OF TYPE btype (
  PRIMARY KEY(y)
) UNDER btable;
```

The first statement defines a type that is used in the second statement to create a typed table. The third statement defines a subtype of the type atype. This subtype is then used to create a subtable of the table atable.

Since the integrity constraints are inherited from the supertables to the subtables, a subtable may implicitly have multiple primary keys. This, however, contradicts the general SQL:1999 requirement that a table must have at most one implicit or explicit primary key. In fact, Informix interprets this requirement as at most *one explicit* primary key per table. □

DB2 is even more restricted than SQL:1999. It does not allow the definition of explicit unique / primary key constraints for subtables.

Order of Constraint Checking

The order of the constraint checking can be crucial for optimizing the overall performance of an application system. We were interested in finding out the checking order among unique / primary key constraints (U), referential constraints with no action semantics (F), and check constraints (C). This information, however, is not available in the manuals of the systems.

To find out the actual order of constraint checking in the reference systems, we first created a table containing an integrity constraint of each type above and then formulated a series of update queries that violate at least two of these three integrity constraints. The results of our tests are summarized in Table 6, where '✓' marks the violated integrity constraints and the corresponding fields contain the abbreviation of the integrity constraint that the corresponding systems return as cause for the integrity violation. Consider the first row stating that an update query violates a uniqueness and a foreign key at the same time. In this case, all reference systems return an integrity violation message stating that the integrity violation is caused by a uniqueness constraint. Thus, we can assume that uniqueness constraints are checked before foreign key constraints.

Table 6 Order of Constraint Checking

U	F	C	**Oracle**	**DB2**	**Informix**	**MSSQL**	**Sybase**	**Ingres**	**Sybase^{Any}**
✓	✓	—	U	U	U	U	U	U	U
✓	—	✓	C	U	C	U	C	U	C
—	✓	✓	C	F	C	C	C	C	C
✓	✓	✓	C	U	C	U	C	U	C

According to the results depicted in Table 6, the following constraint checking orders are (assumed to be) applied in the reference systems: C, U, F (Oracle, Informix, Sybase, and Sybase^{Any}); U, C, F (MSSQL and Ingres); U, F, C (DB2). We have to mention that this is a purely experimental result, and depending on the strategy a query optimizer uses in a particular setting, the order of the checking may be different. Given that often the detailed realization of query processing and optimization is not known in current systems, it will be difficult to come up with analytically correct ordering scheme for the different systems.

It should be mentioned that the integrity constraints above are checked after processing all referential constraints with the restrict, cascade, set null, and set default rules. Remember that only a subset (if any) of these referential actions are actually supported by the reference systems.

4.3 Comparison of Trigger Features

In 1987, Sybase was the first commercial database management system that provided the concept of a trigger. Currently, triggers are available in all reference systems. However, since the concept of a trigger was not standardized until the recent SQL:1999 standard, the implementations in the various systems differ with respect to syntax (and semantics) and capabilities. The grammar of trigger definitions in the various systems are so different that no trigger specification that is formulated in one system can be used without any modifications in another system. Particularly, the action parts differ in the various approaches because each system has its own "trigger programming language". Figure 7 compares the capabilities of the supported triggers mechanisms.

Our comparison is based on the following properties:

- *Number of events per trigger.* Potential events that can lead to the firing of a trigger are insert, update, and delete operations. In SQL:1999, DB2, Informix, and MSSQL, a trigger is based on exactly one event. Oracle, Sybase, and Ingres, in contrast, allow the definition of a trigger on multiple events on the same table.
  Informix does not allow the definition of a delete trigger on a table that contains a foreign key with the `ON DELETE CASCADE` option.
- *Number of triggers per events.* Ingres allows the definition of at most one trigger for each event. Informix applies the same rule for insert and delete events. In case of update events, however, multiple triggers are possible if the column lists of the update events are mutually exclusive. The other systems do not have any restrictions with respect to the number of triggers per events.
- *Trigger granularity.* The granularity of a trigger can be `FOR EACH ROW` (R) or `FOR EACH STATEMENT` (S). MSSQL and Sybase only provide statement-level triggers, whereas the other systems support both kinds of triggers. `FOR EACH STATEMENT` is default in Oracle, Informix, and Sybase^{Any}, whereas `FOR EACH ROW` is default in Ingres. In DB2 the granularity has to be specified explicitly.

Table 7 Comparison of Trigger Features

	SQL:1999	Oracle	DB2	Informix	MSSQL	Sybase	Ingres	Sybase^{Any}
#events per trigger	1	n	1	1	1	n	n	1
#triggers per event	n	n	n	1/n	n	1	n	n
trigger granularity	R/S	R/S	R/S	R/S	S	S	R/S	R/S
transition variables	R/T	R	R/T	R	T	T	R	R/T
activation time	B/A	B/A	B/A	B/A	A	A	A	B/A
activation condition	✓	✓	✓	✓	✓	✓	✓	✓

– *Transition variables.* Within a trigger, one might refer to transition row variables (R) and/or to transition tables (T). In SQL:1999, DB2, and Sybase^{Any}, `NEW` and `OLD` are used to refer to the new and old transitions row variables, while `NEW TABLE` and `OLD TABLE` are used to refer to the new and old transition tables. MSSQL and Sybase only provide transition tables, since they only support statement-level triggers. The new and old transition tables are named `INSERTED` and `DELETED`, respectively. In Oracle, Informix, and Ingres, on the other hand, only transition row variables are supported.
– *Trigger activation time.* A trigger is fired before (B) or after (A) the specified event is handled. MSSQL, Sybase, and Ingres only provide after triggers, whereas the other systems support both kinds of triggers. However, DB2 and Sybase^{Any} do not allow to specify `NEW TABLE` and `OLD TABLE` as well as `FOR EACH STATEMENT` for before triggers.
– *Trigger activation condition.* The activation of a trigger can be associated with a condition. All reference systems support such conditional triggers, although they differ in the way they specify the Boolean expression.
  In SQL:1999, DB2, Informix, MSSQL and Sybase the activation condition can be any Boolean SQL-expression, i.e., it may even contain subqueries. Oracle and Ingres, on the other hand, only allow row-level Boolean SQL-expressions that do not contain subqueries.
– *Trigger action.* In SQL:1999 and DB2, before triggers may not contain any SQL-data change statements. The action part of an after trigger, in contrast, can be any SQL-statement block including SQL-data change statements and procedure calls as well.
  Oracle, Informix, and Sybase^{Any}, however, allow SQL-data change statements within before triggers. Informix has the restriction that the trigger shall not modify the table on which it was fired. This restriction is also applied to after triggers in Informix and Oracle as well.
  In Oracle, a trigger action is specified by a PL/SQL (Programming Language extension of Oracle's SQL) statement block. In Informix and Sybase^{Any}, the action part consists of a list of SQL-data change statements and procedure/function calls.
  In Ingres, a trigger action consists of a single procedure call.
– *Execution order of triggers with respect to declarative integrity constraints.* The execution models of Oracle, DB2, and Sybase^{Any} are analogous to that of SQL:1999 (see Subsection 3.3). The execution order is: executing before triggers, checking of declarative integrity constraints, executing after triggers.
  Informix first executes all fired triggers and then checks the corresponding declarative integrity constraints. Using `SET CONSTRAINTS ALL DEFERRED`, Informix effectively defers the checking of all integrity constraints before it executes the triggering statement. After the trigger action is completed, it initiates the checking of the integrity constraints that were deferred by effectively executing `SET CONSTRAINTS ALL IMMEDIATE`. Thereby, Informix allows to write triggers that resolve any constraint violation that the triggering event would create, e.g., if there is no match for a newly inserted or updated foreign key, then such a key is automatically inserted.
  Since MSSQL, Sybase, and Ingres support only after triggers, their execution model is quite simple. They check all declarative integrity constraints before executing a trigger.
– *Conflict resolution.* A conflict occurs when more than one trigger is fired by an event. SQL:1999, DB2, MSSQL, and Sybase use a creation timestamp-based approach to solve such conflicts. The trigger with the oldest creation timestamp is executed first. Informix resolves conflicts based on column ordering. This is possible since Informix allows multiple update triggers only if they rely on different columns. In Oracle, Ingres, and Sybase^{Any}, row-level triggers are executed before statement-level triggers. The execution order among row-level triggers and statement-level triggers, respectively, is different in these systems. Oracle first executes the trigger with the newest creation timestamp, Ingres first executes the trigger with the oldest creation timestamp, and Sybase^{Any} requires a user-defined execution order (which must be explicitly specified using the *order* clause in the trigger definition).
– *Trigger inheritance.* DB2 and Informix support subtyping and subtables. In Informix, however, triggers are not inherited from supertable to subtables. That is, a trigger defined on an event that occurs on a supertable will not be implicitly triggered by the corresponding event on a subtable. DB2, in contrast, supports trigger inheritance. For instance, an insertion into a subtable fires not only the insert triggers defined on that subtable but also the insert triggers defined on the corresponding supertables. The execution order is determined by the trigger creation timestamp. Note that an inherited trigger is not redefined

but overloaded in the sense that several triggers may rely on the same event.

Oracle and Informix moreover support the enabling and disabling of triggers.

4.4 Database Schema Evolution and Integrity Constraints

A database schema usually evolves during its lifetime. New integrity constraints are added to a table, existing ones are removed from a table, and the checking of integrity constraints is either enabled or disabled. These issues can often occur at the same time. For instance, an integrity constraint is added to a table in the disabled mode or a disabled integrity constraint is enabled but without verifying it against the current content of the corresponding table. In Table 8, we give an overview of the support of such features in SQL:1999 and commercial database management systems. For an overview of general schema evolution features in SQL:1999 and commercial database management systems, we refer to [Tür01].

According to Table 8, Oracle supports a quite broad range of schema evolution features that are related to integrity constraints. One can add, drop, enable, and disable integrity constraints. An interesting fact is that SQL:1999 does not provide any construct for enabling and disabling integrity constraints. The gray shaded fields emphasize the default settings. The '(✓)' marked fields state that the concept is supported implicitly. For instance, the add clause and the drop clause are provided by all reference systems in its standard form. However, extensions like **ENABLE**, **DISABLE**, or **CASCADE** are not supported explicitly by the '(✓)' marked systems. It is interesting to note that only SQL:1999 and Oracle provide language constructs for setting the initial constraint checking and constraint deferrability modes. In the following, we will discuss the different features in more detail.

Adding integrity constraints. An integrity constraint can be added to a table using the alter table statement as follows:

```
ALTER TABLE tab ADD constraint-spec;
```

By default, a newly added integrity constraint is enabled and validated. *Enabled* means that future changes of the content of the table will be verified against this integrity constraint (unless it has been disabled until then). Future modifications that violate the integrity constraint will be rejected. *Validated* means that the table content is verified against the integrity constraint when the latter is added to the table. If the table contains rows that do not satisfy the integrity constraint, the latter cannot be added to the table.

All reference systems support these two modes. Moreover, Oracle and Informix allow to add an enabled integrity constraint even in case there is a row in the table that does not satisfy the integrity constraint. However, in this case an exception clause has to be specified as follows:

```
Oracle:   ALTER TABLE tab ADD constraint-spec
                          EXCEPTIONS INTO exc;
Informix: ALTER TABLE tab ADD constraint-spec
                          FILTERING;
```

Both systems also allow to add a disabled integrity constraint to a table. By default, such an integrity constraint is not validated when it is added to the table. A disabled integrity constraint is added as follows:

```
Oracle:   ALTER TABLE tab ADD constraint-spec
                          DISABLE;
Informix: ALTER TABLE tab ADD constraint-spec
                          DISABLED;
```

Oracle and MSSQL allow to add an enabled integrity constraint that is not validated when it is added to the table. In this case, some rows in the table may violate the integrity constraint. However, future modifications of the table will be verified against the newly added integrity constraint. Such an integrity constraint is defined as follows:

```
Oracle:   ALTER TABLE tab ADD constraint-spec
                          NOVALIDATE;
MSSQL:    ALTER TABLE tab WITH NOCHECK
                          ADD constraint-spec;
```

Dropping integrity constraints. An integrity constraint can be dropped from a table using the alter table statement as follows:

```
ALTER TABLE tab DROP constraint-name;
```

In some systems, a cascade mode can be attached to the drop clause. Two cascade modes are usually distinguished. The **RESTRICT** mode disallows the dropping of the integrity constraint if there is another integrity constraint that depends on the integrity constraint to be dropped. For instance, a foreign key constraint depends on a unique / primary key constraint. In this case, the unique / primary key constraint cannot be dropped unless all depending foreign key constraints are dropped before or the cascade mode is set to **CASCADE**. If **CASCADE** is specified, an integrity constraint is dropped together with all its depending integrity constraints.

The specification of one of these modes is mandatory in Ingres, whereas it is optional in Oracle. The other systems do not support the specification of these modes. The default mode is **RESTRICT** in Oracle, Informix, MSSQL, and Sybase and **CASCADE** in DB2.

Enabling integrity constraints. A deactivated integrity constraint can be enabled in the validate mode as follows:[3]

```
Oracle:   ALTER TABLE tab ENABLE constraint;
Informix: SET CONSTRAINTS cons-name ENABLED;
```

The enabling of an integrity constraint can also be performed in the exception/filtering mode. All rows that violate the enabled integrity constraint are removed from the table into an exception/violations table.

The enabling of a disabled integrity constraint in the novalidate mode is specified as follows:

[3] Here constraint stands for **CONSTRAINT** constraint-name, **PRIMARY KEY**, or **UNIQUE** (column-list).

Table 8 Comparison of Integrity Constraint Evolution Features

ALTER TABLE			SQL:1999	Oracle	DB2	Informix	MSSQL	Sybase	Ingres	Sybase^{Any}
ADD	ENABLE	VALIDATE	(✓)	✓	(✓)	✓	✓	(✓)	(✓)	(✓)
		VALIDATE + EXCEPTION	—	✓	(×)	✓	—	—	—	—
		NOVALIDATE	—	✓	(×)	—	✓	—	—	—
	DISABLE		—	✓	(×)	✓	—	—	—	—
DROP	RESTRICT		✓	✓	—	(✓)	(✓)	(✓)	✓	(✓)
	CASCADE		✓	✓	(✓)	—	—	—	✓	—
ENABLE	VALIDATE		—	✓	(×)	✓	—	—	—	—
	VALIDATE + EXCEPTION		—	✓	(×)	✓	—	—	—	—
	NOVALIDATE		—	✓	(×)	—	(✓)	—	—	—
DISABLE	RESTRICT		—	✓	—	(✓)	(✓)	—	—	—
	CASCADE		—	✓	(×)	—	—	—	—	—
SET	INITIALLY	IMMEDIATE	✓	✓	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)
		DEFERRED	✓	✓	—	—	—	—	—	—
	DEFERRABLE		✓	✓	(✓)	(✓)	—	—	—	—
	NOT DEFERRABLE		✓	✓	—	—	(✓)	(✓)	(✓)	(✓)

```
Oracle:   ALTER TABLE tab ENABLE NOVALIDATE
                                 constraint;
MSSQL:    ALTER TABLE tab CHECK constraint;
```

In Oracle, the novalidate option is ignored for unique/primary key constraints. Such integrity constraints are validated even if the novalidate mode is specified. Unfortunately, the status of such an integrity constraint is then noted in the data dictionary as invalidated (although it was validated).

Disabling integrity constraints. An enabled integrity constraint can be disabled using either:

```
Oracle:   ALTER TABLE tab DISABLE constraint;
Informix: SET CONSTRAINTS cons-name DISABLED;
MSSQL:    ALTER TABLE tab NOCHECK constraint;
```

Cascading disabling of integrity constraints is only supported by Oracle. For this, the keyword `CASCADE` has to be attached to the disable clause.

Also, DB2 provides means to switch off the checking of integrity constraints. We have marked this in Table 8 with a special symbol (×) because DB2's approach differs from the other approaches with respect to the language constructs for enabling and disabling integrity constraints and the fact that it allows only very restricted access to tables with disabled integrity constraints as well. In fact, the content of a table in the check pending state can neither be queried using a select statement nor manipulated via an insert, update, or delete statement; but DB2 provides the warehouse features `LOAD INSERT` and `LOAD REPLACE` to append or replace the content of a table in the check pending state. However, one cannot simply disable integrity constraints and then work with the corresponding tables as usual.

Using the statement

```
SET INTEGRITY FOR tab OFF;
```

the integrity checking for table `tab` and all dependent tables (that reference to table `tab` via a foreign key) are switched off. Now, for instance, new integrity constraints can be added using the alter table statement, or the load utilities can be invoked. Later, integrity checking can be switched on again for table `tab` either in the validated mode using

```
SET INTEGRITY FOR tab
IMMEDIATE CHECKED;
```

or in the invalidated mode using

```
SET INTEGRITY FOR tab
ALL IMMEDIATE UNCHECKED;
```

In the latter mode, the table may contain some rows that do not satisfy all corresponding integrity constraints (including the newly added ones). If the former mode is used in such a situation, the table remains in the check pending state unless the following extension is used to automatically remove the constraint violating rows into an exception table:

```
SET INTEGRITY FOR tab
IMMEDIATE CHECKED
FOR EXCEPTION IN tab USE exctab;
```

The dependent tables that were automatically switched off now have to be switched on manually, i.e., the user / application has to list the names of the corresponding tables instead of the single table name as done above.

Finally, we have to mention that DB2 moreover supports incremental checking of the appended content. To do so, the keyword INCREMENTAL has to be specified in front of the exception clause. For more details on DB2-specific features, we refer the reader to [Cha98].

Changing the checking time of integrity constraints. In Oracle and Informix, the checking time of an integrity constraint can be changed, e.g., as follows:

```
SET CONSTRAINTS constraint DEFERRED;
```

Note that in Oracle the deferrability status of an integrity constraint cannot be altered. If it should be changed, the integrity constraint must first be dropped and then re-created with the desired status.

4.5 Information about Integrity Constraints

Integrity constraints are schema objects. Information about the stored integrity constraints can be found in the data dictionary. In SQL:1999, the following data dictionary views are related to integrity constraints:

- assertions: Contains metadata about an assertion, e.g., its name and checking mode.
- check_constraints: Provides metadata about check constraints, e.g., the names and check clauses.
- constraint_column_usage: Refers to columns that are involved in referential constraints, uniqueness constraints, check constraints, or assertions.
- constraint_table_usage: Identifies tables that are used by referential constraints, uniqueness constraints, check constraints, or assertions.
- domain_constraints: Provides metadata about all domain constraints, e.g., the names of the associated domains.
- key_column_usage: Identifies columns that are involved in keys.
- referential_constraints: Maintains metadata about referential constraints, e.g., the names of the corresponding key constraints, the match options, and the update/delete rules.
- table_constraints: Maintains metadata about table constraints, e.g., the corresponding constraint types and table names.
- triggers: Provides metadata about triggers, e.g., the triggering events and action conditions.
- trigger_column_usage: Refers to columns that are used by triggers.
- trigger_table_usage: Identifies the tables that are used by triggers.

It interesting to point out that only MSSQL actually implements the information schema as proposed in SQL:1999. The implementations of the other systems differ in the view names and column names as well as in the content of the data dictionary views.

Figure 9 summarizes the names of the data dictionary views provided by the reference systems (except Sybase^{Any}). Sybase^{Any} provides information about integrity constraints and triggers via the tables syscolumn and systrigger, respectively. In Oracle, the prefix xxx has to be substituted by one of the abbreviations dba, user, and all. By using the prefix dba, one gets access to all metadata that is available for the database administrator (of course, the current user must have the corresponding dba access rights), the prefix user is used to query metadata about the database objects owned by the current user, and the prefix all is used to retrieve metadata about all database objects that are accessible to the current user.

Interestingly, Oracle and DB2 require that the names of database objects are written in upper letters when used in queries against data dictionary views. Informix and Ingres require lower case letters. Sybase clearly distinguishes both cases (e.g., two distinct tables with the names Emp and emp, respectively, are allowed). MSSQL and Sybase^{Any} make no distinction between upper and lower case letters; it can handle both, i.e., the names Emp and emp would here refer to the same table.

5 Design Rules for Integrity Constraints

Given the support for the specification and enforcement of semantic integrity constraints in commercial database management systems, the question now is how to utilize respective concepts and mechanisms effectively in database design and implementation. Existing work on integrity constraint design and enforcement typically deals with models of ideal database management systems that provide all integrity features in terms of, e.g., execution model, trigger granularity or transition variables. In practice, however, the limitations of certain integrity features in current systems can have a strong impact on an integrity design and implementation methods.

In this section, we provide some general guidelines and rules for utilizing the integrity maintaining mechanisms discussed in the previous sections. In particular, we discuss how system-specific features for constraint maintenance fit into these guidelines and design issues. However, we do not give a comprehensive overview of the role of integrity constraints in database design in general, but rather indicate aspects specific to implementing typical integrity constraints using the declarative and procedural mechanisms provided by the systems. This section shall be understood as an overview of integrity design issues that a developer of an effective integrity subsystem should have in mind in addition to the knowledge about the semantic integrity features supported by the various systems.

In Section 5.1, we outline some basic issues regarding the preparation of integrity constraints obtained from the require-

Table 9 Overview of Integrity Constraint Related Data Dictionary Views

metadata about	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
table constraints	xxx_constraints	syscat.tabconst	sysconstraints	table_constraints	sysconstraints	iiconstraints
check constraints	xxx_constraints	syscat.checks	syschecks	check_constraints	sysconstraints	iiconstraints
referential constraints	xxx_constraints	syscat.references	sysreferences	referential_constraints	sysreferences	iiref_constraints
key columns	xxx_cons_columns	syscat.keycoluse	syscoldepend	key_columns_usage	syskeys	iiindexcolumns
triggers	xxx_triggers	syscat.triggers	systriggers	triggers	sysprocedures	iirules

ments analysis for implementing respective enforcing mechanisms. In Section 5.2, we detail design issues regarding the translation of integrity constraints into declarative and procedural constraint enforcing mechanisms provided by the systems discussed in this paper.

5.1 Constraint Preparation

Before translating a set of integrity constraints into constraint enforcing mechanisms, two tasks need to be applied: (1) *constraint decomposition*, and (2) *checking constraints for consistency and non-redundancy*. These tasks are essential not only to ease the translation process but also to realize meaningful and efficient constraint enforcement.

Constraint Decomposition. Integrity constraints are usually gathered during the database design phase. While some types of integrity constraints can already be expressed using the concepts provided by the data model used for the design, other integrity constraints are simply formulated in natural language statements or logic-based expressions. Such statements and expressions often tend to be fairly complex since designers often combine several integrity aspects about related objects (rows). Although this is a good design strategy and has been addressed by introducing the concept of an ASSERTION in SQL:1999, none of the commercial database management systems supports this feature. Thus, complex integrity constraints need to be appropriately decomposed and "distributed" over one or more constraint enforcing mechanisms such as table constraints and/or triggers.

More precisely, given an integrity constraint C, constraint decomposition is the process of deriving constraints C_1, ..., C_k from C such that each C_i is either a state (state transition) row, table or inter-table constraint. That is, each C_i is exactly of one of the classes of integrity constraints introduced in Section 2.2. Here we do not consider the three classes of temporal constraints since these are often already formulated as appropriate state transition constraints. We show the basic idea of constraint decomposition by means of an example.

Example 13. Assume a foreign key constraint stating that every employee must work in a department. Furthermore, let C1 be an integrity constraint stating *"For all employees, the salary must be between two thousand and five thousand, and the total of all salaries in a department must be less than the department's budget"*. This integrity constraint can be formulated as follows:

$$\forall \mathsf{e} \in \mathsf{emp}\colon \mathsf{e.salary} > 2000 \land \mathsf{e.salary} < 5000 \land$$
$$\forall \mathsf{d} \in \mathsf{dept}\colon \mathsf{sum}\{\mathsf{e.salary} \mid \mathsf{e} \in \mathsf{emp} \land \mathsf{e.dept} = \mathsf{d.id}\} < \mathsf{d.budget}$$

It is easy to see that this constraint can be decomposed into two independent constraints $\mathsf{C1}_1$ (first line) and $\mathsf{C1}_2$ (second line). $\mathsf{C1}_1$ is a row constraint and $\mathsf{C1}_2$ is an inter-table constraint. Note that the similar substatement *"The total of all salaries in a department must be less than four hundred thousand"* is a table constraint and can be evaluated on rows from the table emp only using a grouping by department id.

Both $\mathsf{C1}_1$ and $\mathsf{C1}_2$ are state constraints, that is, they can be evaluated in a single database state. The integrity constraint *"The salary of a manager can increase by at most five percent"* is a state transition constraint, restricting possible updates on employees' salary. This integrity constraint can be formulated using the prefixes old.R and new.R to refer to the extensions of table R in a transaction's pre- and post-state, respectively:

$$\forall \mathsf{e} \in \mathsf{new.emp}, \mathsf{e}' \in \mathsf{old.emp}\colon \mathsf{e.id} = \mathsf{e}'.\mathsf{id} \land \mathsf{e}'.\mathsf{job} = \text{'manager'} \Rightarrow \mathsf{e.salary} <= \mathsf{e}'.\mathsf{salary} * 1.05 \quad \square$$

Constraint decomposition is hard to do just on the natural language level, breaking a statement into smaller, independent substatements. This is why we advocate a logic-based formalism. Such a formalism not only provides us with a concise notation for integrity constraints but also provides a more precise framework to check constraints for consistency and non-redundancy.

Interestingly, the basic concepts and techniques for constraint decomposition and simplification have already been introduced in the early Eighties [BBC80,BB82,Nic82]. To the best of our knowledge, however, there exists no complete framework for constraint decomposition in the object-relational model (or the object-oriented model). A complicating issues in this model definitely is the association of constraints with the "right" subtables and subcomponents in case subtables and structured types, respectively, are supported by the system. The development of such an algorithmic framework and its appropriate integration into commercial database design tools would greatly enhance integrity-centered design of object-relational databases.

5.1.1 Consistency and Redundancy of Constraints. Naturally, a set of (decomposed) integrity constraints has to satisfy two important criteria. First, it has to be *consistent*. A set $\mathcal{C}$ of integrity constraints is said to be *consistent* if there exists at least one (non-empty) database state such that each integrity constraint $C_i \in \mathcal{C}$ is satisfied. Second, it has to be *non-redundant*. That is, no integrity constraint in $\mathcal{C}$ should be logically implied by other integrity constraints in $\mathcal{C}$. Amazingly, inconsistencies can easily be introduced in constraint specifications.

Example 14. The following specification of a simple primary / foreign key combination is accepted by all reference systems we considered.

```
CREATE TABLE dept (
  ...
  id   SMALLINT PRIMARY KEY CHECK(id > 100)
);

CREATE TABLE emp (
  ...
  dept SMALLINT NOT NULL CHECK(dept < 100),
  FOREIGN KEY (dept) REFERENCES dept(id)
);
```

Clearly, due to the row constraints, the domains of the foreign key and primary key are disjoint. Thus, there cannot be any match between the values of these keys. Since there is also a not null constraint defined on the foreign key, this set of integrity constraints is inconsistent. □

Obviously, neglecting the verification of constraint consistency (sometimes also called *satisfiability*) can have a severe impact on the resulting implementation of constraint enforcing mechanisms in the database. For certain types of integrity constraints, however, verification methods have been proposed in the literature.

Example 15. Assume the following set of row constraints associated with the table emp:

$$\forall e \in \mathsf{emp}: \mathsf{e.salary} \geq 5000$$

$$\forall e \in \mathsf{emp}: \mathsf{e.bonus} \leq 2000$$

$$\forall e \in \mathsf{emp}: \mathsf{e.salary} \leq \mathsf{e.bonus} + 2500$$

One can easily see that this set of integrity constraints is inconsistent. For these types of check constraints, the verification problem can also be solved algorithmically. For instance, [RH80, GSW96a] provide efficient graph-based satisfiability algorithms for integrity constraints that are based on predicates of the forms $(x\,\theta\,c)$ and $(x\,\theta\,y + c)$, where x and y are numeric variables (column names), c is a numeric constant, and θ is a comparison operator from $\{<, \leq, =, \geq, >\}$. □

Similar algorithms have also been introduced for verifying the consistency of check constraints consisting of conjunctive predicates based on character string variables [Tür99a].

While the above example specifications of inconsistent sets of integrity constraints result in the fact that there exists no database state satisfying all integrity constraints, the problem of redundant integrity constraints has less severe consequences but nevertheless can result in a database performance penalty as well as in additional expenditure for schema maintenance.

Example 16. One of the following integrity constraints is redundant, thus does not need to be checked by the system at database run-time:

$$\forall e \in \mathsf{emp}: \mathsf{e.salary} \leq 5000$$

$$\forall e \in \mathsf{emp}: \mathsf{e.bonus} \leq 2000$$

$$\forall e \in \mathsf{emp}: \mathsf{e.salary} + \mathsf{e.bonus} \leq 8000$$

Here, the third integrity constraint is implied by the conjunction of the first two ones. Redundancy is not so obvious in the following case:

$$\forall e \in \mathsf{emp}: \mathsf{e.salary} < 5000$$

$$\forall d \in \mathsf{dept}: \mathsf{count}\{\mathsf{e.id} \mid e \in \mathsf{emp} \wedge \mathsf{e.dept} = \mathsf{d.id}\} \leq 10$$

$$\forall d \in \mathsf{dept}: \mathsf{sum}\{\mathsf{e.salary} \mid e \in \mathsf{emp} \wedge \mathsf{e.dept} = \mathsf{d.id}\} \leq 100000$$

Based on the first two integrity constraints, the third integrity constraint can never be violated and thus needs not be verified at database run-time. □

Interestingly, advanced consistency and redundancy checking procedures are provided, for instance, in [BCL86, Ull89, SKN89, Sri93, GSW96a, GSW96b]. For an overview of consistency checking algorithms for integrity constraints we refer to [Tür99b]. Since the knowledge about the consistency and redundancy problems and of their solutions is highly important for a good design of a database, a database designer has to be aware of these problems and possibilities. The proposed algorithms could easily be brought together to solve the constraint consistency and redundancy problems in the presence of one of the basic types of integrity constraints used in many application domains, namely check constraints with conjunctive predicates of a special form, unique / primary and foreign key constraints, and not null constraints. From a research point of view, however, still much work is needed on consistency and redundancy issues, in particular with respect to the types of constraints supported in today's commercial database systems:

- Since different types of integrity constraints are not independent of each other, combined consistency and redundancy verification algorithms have to be developed. For instance, as shown in [Tür99b], efficient solutions are possible for combinations of special aggregate constraints [RSSS98] and basic linear arithmetic constraints.
- Since database tables are used with multiset semantics in the SQL-standard as well as in SQL-dialects of commercial database management systems [GdB94], previous results have to be considered under multiset semantics.
- The problem of checking consistency and redundancy becomes even more prominent when inheritance of integrity constraints comes into play. In this context, it is hard to manually check the consistency of integrity constraints since the database designer has to be aware of all implicit integrity constraints that a table inherits from its supertables. The connection between integrity constraints and object-objected database concepts such as subtyping and

subclasses is discussed in [FM93, BdBZ93, Tür99a]. The latter work particularly deals with the inheritance of integrity constraints and shows that there are integrity constraints that cannot be inherited to subclasses.
– A more precise framework on what actually a meaningful database comprises is necessary. For example, if we change the comparison operators in Example 14 to include equality, i.e., if we use `CHECK (id >= 100)` and `CHECK (dept <= 100)`, database states exist where all integrity constraints are satisfied. Such states, however, would only contain at most one department id for employees. Thus, models are necessary that make quantitative statements about possible admissible database states.

5.2 Declarative versus Procedural Specifications

In this section, we discuss several design issues and rules for implementing constraint enforcing mechanisms in form of declarative and procedural (trigger) mechanisms provided by the systems discussed in this paper. For this, we assume a set of decomposed integrity constraints that is consistent and non-redundant.

General Issues

In general, enforcing integrity constraints by declarative constraints (and even triggers) often is less costly than enforcing the equivalent rules by issuing SQL-statements in an application. In a networked client-server environment, this effect can be severe since SQL-statements as well as query results need to be transmitted over the network. Another advantage of declarative constraints is that their semantics is more clear. Thus, the database management system is able to optimize internal operations that enforce an integrity constraint.

For best performance and to preserve logical data independence, it is advisable to define and enable integrity constraints in the database system and develop applications to rely on them, rather than on verifying SQL-statements in the applications. However, in certain scenarios, enforcing an integrity constraint by an application might provide a faster feedback to the user. For instance, in case of an insert operation that requires twenty values before it is issued to the system. In this case, the application may notify the user immediately after entering a value that violates an integrity constraint.

In the sequel, we focus on implementing constraint enforcing mechanisms for the following six types of integrity constraints: state and state transition forms of row, table, and inter-table constraints. Although all types of constraints can be implemented by only using triggers, because of performance reasons it is advisable to use declarative mechanisms when possible. Table 10 gives an overview of what mechanisms should be used for the different types of integrity constraints (with respect to the integrity enforcement capabilities of current commercial database systems). Note that integrity constraints that contain aggregate functions are listed explicitly because they play an important role in many applications.

Table 10 Declarative versus Procedural Mechanisms

Constraint Type	Declarative	Trigger
row		
not null	✓	—
check	✓	—
table		
check	—	✓
aggregate	—	✓
primary key	✓	—
unique	✓	—
foreign key	✓	(✓)
inter-table		
check	—	✓
aggregate	—	✓
foreign key	✓	(✓)
dependencies	—	✓
transition		
row, table, inter-table	—	(✓)

– *Not null.* In all reference systems, a not null constraint can be specified declaratively, either directly using the keyword `NOT NULL` or a check clause.
– *Check (at row or column level).* The check clause can be used to specify Boolean predicates on the rows of a table, e.g., simple column-constant or column-column comparisons.
– *Check (at table or inter-table level).* These types of integrity constraints include all constraints that cannot be specified for a table using the above declarative mechanisms. Typical examples are column value comparisons of pairs of rows from the same table. For example, the integrity constraints "*A manager's salary must be greater than a clerk's salary*", or "*An employee must work in the same department as his supervisor*". Again, for the same reason as aggregate constraints, such check constraints must be formulated using triggers.
– *Aggregate (table and inter-table).* Constraints that include the aggregate functions `MIN`, `MAX`, `SUM`, `COUNT`, or `AVG` cannot be specified in a declarative fashion. This includes even very simple integrity constraints such as "*Every department must have at least two employees*". Although the SQL-standard supports complex check conditions that can be associated with a table, none of the reference systems supports declarative specifications for this type of integrity constraint. They have to be specified using triggers.
– *Primary key.* All reference systems provide declarative language constructs to specify a primary key for a table.
– *Unique.* This type of integrity constraint is supported by all reference systems through the unique clause. Note that some reference systems, e.g., Oracle, set up special index structures for both primary keys and unique columns to

favor efficient constraint checking and foreign key related joins and accesses.

- *Foreign key.* In all reference systems, a foreign key can be specified either using the references clause, referring to a single column unique / primary key, or the foreign key clause, referring to a composite (multi-column) unique / primary key. Recall from Section 3 that with foreign key constraints compensating actions can be associated that are executed in case the constraint is violated. Since some database management systems do not provide the various update and delete rules, however, these have to be implemented by triggers.
- *Dependencies.* Such integrity constraints include all types of dependencies among rows from different tables that cannot be expressed through foreign key constraints. Typical constraints of this category are inclusion dependencies where a column value of a row must occur as a column value of another row such that neither of the columns is a primary key. For example, the integrity constraint "*The location of a project must be the same as the location of the department managing the project*" specifies an inclusion dependency.
- *State transition constraints.* These types of integrity constraints must always be implemented using triggers since in all systems transition variables are only supported in triggers.

The above descriptions already give some quite useful starting points on what type of constraint enforcing mechanism (declarative or procedural) is applicable to what type of integrity constraint. Note that in order to classify an integrity constraint into one of the above categories, it is important to have it decomposed.

Deriving Triggers from Constraint Specifications

Despite the fact that all reference systems considered in this paper use a different syntax for trigger specifications, implementing triggers based on constraint specifications follows some simple common rules [CW90, CCW00]. Basically, for each integrity constraint, first critical operations have to be determined, and then the appropriate trigger type has to be selected that verifies the integrity constraints in case of such an operation. As we will discuss in this section, the latter aspect is highly system-dependent.

Critical Operations. Given an integrity constraint C, the first task is to determine *critical operations*, that is, database modifications that can lead to a violation of C. Such operations are insert, update, and delete statements on tables referenced in C. These operations eventually determine triggering events on base tables as part of a trigger specification. If one misses a critical operation, the integrity constraint can be violated without the system reacting on the violation. Analogously, if an operation has been identified that actually can never violate the integrity constraint, implementing a respective trigger would be meaningless and only leads to a decrease in the system performance.

Typically, an integrity constraint is formulated in a way such that all qualified rows from a (combination of) table(s) have to satisfy a condition. That is, an integrity constraint has the form

$$\forall \mathsf{r}_1 \in \mathsf{R}_1, \ldots, \mathsf{r}_n \in \mathsf{R}_n \colon \langle \mathsf{cond} \rangle$$

where $\langle \mathsf{cond} \rangle$ is a logical expression over columns from the tables $\mathsf{R}_1, \ldots, \mathsf{R}_n$ using the usual logical connectives $\wedge$, $\vee$, $\neg$, and comparison operators to formulate predicates. $\langle \mathsf{cond} \rangle$ can include further quantifiers, typically used to specify existentially quantified variables. The qualifiers used for tuple variable indicate operations that can violate the integrity constraint.

The identification of critical operations from constraint specifications has already been discussed in detail in several works, e.g., [Nic82, CW90]. Here we only summarize the basic results:

1. for tables R_i that are covered by universally quantified variables, insert operations are critical,
2. for tables R_i that are covered by existentially quantified variables, delete operations are critical, and
3. In both cases update operations on columns used in comparisons are critical.

Note that critical update operations can be refined further based on the type of comparison columns are involved in. Such refined updates prove to be useful for specifying triggering conditions. The same rule can basically be applied to state transition constraints, where quantified variables range over transition tables. Critical operations are then based on the underlying base tables.

In case of aggregate constraints, all operations that might change the element set of the underlying aggregate function potentially are critical operations. However, considering the semantics (monotonicity) of aggregate functions, the set of critical operations can be refined. Monotonic aggregate functions, for instance, have the nice property that depending on the comparison operations involved into the constraint predicate either insertions into or deletions from the element set of the aggregate function are not critical. Here, we do not discuss the various rules that hold for the different combinations of aggregate functions and comparison operators. For a detailed discussion of the relationship among aggregate functions, monotonicity properties, and integrity constraints, we refer to [Tür99b].

Identifying critical operations becomes even more complicated in the object-oriented model. Specific data model considerations must be included in respective design frameworks as proposed, for instance, in [BCP96] for active database systems.

Choosing a Trigger Type. Determining a critical set of operations for a set of integrity constraints eventually results in a set of $\langle \mathsf{table}, \mathsf{operation} \rangle$-pairs that specify critical operations on tables. These pairs already determine the triggering events. The next task is to determine for each such pair the right trigger type, that is, trigger granularity, activation time, and triggered condition. For each aspect, we start with some general guidelines and then address some more specific, system-dependent issues:

- State constraints that include aggregate functions always require at least one statement-level trigger, defined on the table and rows over which the aggregation is performed. Conceptually, a row-level trigger can be used to incrementally check the constraint before/after a row has been modified. Besides the fact that such an approach is much less efficient, most systems moreover do not allow the access to the table being modified by the triggering operation in a row-level trigger. For example, in an after row-level trigger in Oracle, the table on which the triggering event occurred cannot be accessed without resulting in a "mutating table" error.
- Because the reference systems support different types of transition variables, triggers that monitor and verify critical operations derived from state transition constraints have to be of row-level type in Oracle, Informix, and Ingres, and of statement-level type in MSSQL and Sybase (both only provide transition tables). In DB2, either type of trigger can be used to verify state transition constraints.
- In all reference systems, only row-level triggers allow a when clause verifying properties of rows to be checked. This holds in particular for rows involved in the verification of state transition constraints. For example, using transition row variables, in Oracle the values of modified rows can be checked in the when clause.

In general, all types of state constraints shown in Table 10 can be verified in a statement-level trigger. For performance reasons, however, it seems reasonable that row-level triggers are preferable. A row-level trigger often allows an immediate check of a single modified row. A statement-level trigger, in contrast, first has to accumulate all rows modified by a statement before these rows can be checked using either a transition table (if provided) or the whole table. Choosing between a row-level or statement-level trigger thus depends on how many rows actually have been modified and need to be checked for constraint violations and in particular if transition tables are supported. If transition tables are not supported but transition variables for rows, a row-level trigger is always preferable. MSSQL and Sybase only provide statement-level triggers in which transition tables can be used to check modified rows for constraint violations. Thus, aforementioned immediate checks of modified rows cannot be done in these systems. Oracle, in contrast, only supports row transition variables for row-level triggers. If a statement-level trigger is used in Oracle, there is no way to access only the modified rows and to tailor the checking of constraints (in particular state transition constraints) in such a trigger. It should also be mentioned that if the system provides both types of transition variables, then a row-level trigger can be simulated by a statement-level trigger. This, of course, is only possible if the table on which the trigger is defined has a primary key, otherwise modified tuples cannot always be distinguished.

Before versus After Trigger. Associating the "right" activation time with a trigger is highly system-dependent. In almost all cases, after triggers are sufficient. Before triggers associated with a table are typically evaluated before declarative constraints are evaluated for the table. Note that there are several subtle differences among the implementation of the SQL:1999 activation time feature in current systems. For example, in DB2 and Sybase^{Any} **FOR EACH STATEMENT** is not supported for before triggers and rows cannot be modified in before triggers, whereas in Oracle modifications of affected rows are only allowed in before row-level triggers. A major problem in deciding among the two different activation times for verifying constraints using triggers is the lack of information about how current systems actually realize the different activation times, compared to what has been proposed in the literature on active database systems. Furthermore, the execution model of triggers has to be seen in context of the respective execution model for checking declarative constraints. It thus seems desirable to include this aspect in a flexible evaluation framework, together with the aforementioned trigger granularity aspects, and to provide this framework in system specific integrity-centered database design tools. Such a framework capable to consider the variety of existing systems with their specific features and execution models, however, is very likely to be non-trivial.

Triggered Condition. The final step in implementing constraint checking triggers is to specify the verifying condition in the trigger. This is a crucial design task since one cannot use the original constraint but has to use its negation. That is, one has to specify a condition that checks whether there exists a row (or a combination of rows) for which the integrity constraint is violated. If such a row (or combination) exists, the triggering action specifies how to react upon the constraint violation, which typically is a rollback of the transaction. Note that in some commercial database systems (e.g., DB2), the trigger condition specified in the when clause can contain such a verifying condition. In other systems (e.g., Oracle), the condition must be evaluated in the trigger body.

Furthermore, we would like to point out one important aspect on grouping verifying conditions into single triggers. Different integrity constraints, of course, can have the same critical operations. Thus, it seems reasonable to specify respective verifying conditions in one trigger associated with a ⟨table, operation⟩-pair, assuming they can be enforced using the same trigger granularity. For Sybase and Informix, this is the only option since only one trigger per event is allowed (see also Section 4). Having all verifying conditions in one trigger also is beneficial regarding the performance of the system. However, practice often shows that integrity constraints are modified, added, or become obsolete. If verifying conditions for a ⟨table, operation⟩-pair have been specified in one trigger, the whole trigger needs to be modified. Also, in case an integrity constraint has to be (temporarily) disabled, a trigger can only be disabled as a whole. In certain situations, it is desirable to disable integrity constraints of a table temporarily for performance reasons, e.g. when loading large amounts of data into the table using a loader, when performing batch operations that make massive changes to the table, or when exporting or importing a table.

It is important to note that after a batch load or replace triggers are not fired since there is no semantics for whether the data was inserted, updated, or deleted, and in case of a batch replace the old rows (of the table prior to the replacement) are lost. This is another strong reason for why to prefer declarative constraints to procedural constraints implemented in triggers when possible. Otherwise, integrity constraints implemented in triggers must be checked explicitly by every application that performs a batch load or replace.

Reactive Constraint Enforcement

The default response of a database system in case of a constraint violation is to rollback the transaction containing the violating operation. That is, all database modifications up to the transaction pre-state are undone. For some application scenarios, such a behavior can be too rigorous in that long duration transactions (in terms of database modifications) are undone just because of a "simple" constraint violation. That is why all systems provide features to react on constraint violations with actions different from a rollback. We call the implementation of respective mechanisms *reactive constraint enforcement*. We already presented such mechanisms in Sections 3 and 4 where referential actions associated with foreign keys have been introduced. For constraint enforcing triggers, reactive constraint enforcement is possible as well since trigger actions support database modifications. In the following, we will give a short overview of realizing reactive constraint enforcement using triggers in commercial systems.

Implementing reactive mechanisms using triggers essentially occurs through specifying database modifications, i.e., insert, update, and delete statements in a trigger body. The database modifications then can be executed in case the integrity constraint (or portion thereof) verified by the trigger is violated. There are many crucial design issues with respect to implementing such repairing database modifications. First, the modifications, if executed, build a part of the current transaction that contains the operation(s) violating the integrity constraint verified by the trigger. That is, supposedly repairing modifications can lead to new constraint violations, depending on how the modifications are determined at trigger execution time. This can lead to several types of interactions among constraint enforcing (and violation repairing) triggers, mainly regarding the confluence and termination of possible trigger executions.

Confluence deals with the issue of ordering among triggers. That is, if a set of triggers is fully prioritized (based on, e.g., trigger creation time), then trigger processing is fully deterministic and confluence among the triggers can be guaranteed. Confluence is also guaranteed if no two triggers will be activated at the same time. This can be achieved, of course, by implementing all constraint verifications and repairing modifications in one trigger for a $\langle$table, operation$\rangle$-pair. *Termination* is much more difficult to deal with since it is necessary to analyze whether or not a set of triggers may activate each other indefinitely. Regarding the aspects of trigger confluence and termination, a substantial amount of work has been published (see, e.g., [WC96, CFPT94] for an overview or [AWH92, BCW94, CW90, FP97, ZH90] for more detailed discussions).

Another important issue for implementing reactive triggers is the specification of the "right" repairing database modifications. A standard approach typically adopted for applications is to undo a violating database modifications.

Example 17. Assume the integrity constraint "*The salary of a manager must be greater than the salary of a clerk*".

$$\forall e \in \mathsf{emp}, e' \in \mathsf{emp}: \mathsf{e.job} = \text{'manager'} \wedge \mathsf{e'.job} = \text{'clerk'} \Rightarrow \mathsf{e.salary} > \mathsf{e'.salary}$$

Obviously, the pair $\langle$emp, update(salary)$\rangle$ is a critical operation. Based on this pair, the following trigger (in SQL:1999 syntax) implements a reactive constraint enforcement mechanism in that a violating update on an employee's salary is undone:

```
CREATE TRIGGER salary_clerk_manager
  AFTER UPDATE OF salary ON emp
  REFERENCING OLD AS pre NEW AS post
  FOR EACH ROW
  WHEN((post.job = 'manager' AND
         EXISTS(SELECT *
                FROM emp
                WHERE job = 'clerk' AND
                      salary >= post.salary))
        OR
        (post.job = 'clerk' AND
         EXISTS(SELECT *
                FROM emp
                WHERE job = 'manager' AND
                      salary < post.salary)))
  BEGIN ATOMIC
    UPDATE emp
    SET salary = pre.salary
    WHERE id = post.id
  END;
```

The two conditions in the when clause build the violating condition for the integrity constraint. The update clause in the trigger's body undoes the update of the row that violates the integrity constraint. Note that for the other critical operations similar triggers have to defined. □

A more detailed discussion about reactive constraint enforcement mechanisms can be found, for instance, in [CFPT94, FP94, Ger96].

6 Conclusions

In this paper, we have provided a detailed comparison of semantic integrity features in the SQL:1999 standard and the major commercial (object-)relational database management systems Oracle, IBM DB2, Informix, Microsoft SQL-Server, Sybase, and Ingres. In addition, we have provided general guidelines for the design and implementation of integrity constraints using the mechanisms and functionality provided by these systems.

A main observation is that current commercial database management systems basically support the entry level of the previous standard SQL-92 [MS93,DD97]. With regard to the specification of declarative constraints, these systems support the specification of default values, not null constraints, primary and foreign keys, and check constraints at the column and row level. Some systems support the specification of referential actions as simple type of reactive constraints enforcement as well. The biggest discrepancy between the current SQL-standard and the reference systems considered in this paper is on the specification and usage of database triggers. The reason for this is pretty simple: triggers have been provided by most systems prior to the introduction of SQL:1999. That is why for almost all these systems not only the syntax for trigger specifications is quite different from the standard, but also the functionality of the triggers, e.g., in terms of transition tables and trigger execution orders. Also none of the reference systems supports the specification of check constraints on table or database level; assertions are in general not available and are unlikely to be offered in the near future.

We conclude that although sufficient means for implementing constraint enforcing mechanisms —either declaratively and/or in form of database triggers— are provided by all reference systems, practice shows that the role and importance of integrity constraints is often underestimated so that (non-trivial) integrity constraints are often not considered in database design. Obviously, there is a strong need in today's application scenarios to ensure the semantic correctness of the data managed by database systems. However, database designers seem to be quite reluctant to make use of respective means to maintain and enforce semantic integrity constraints. So what are the reasons for this and how can this gap be closed?

First, design tools tailored to the design and implementation of integrity constraints are necessary. Current database design tools available for the systems considered in this paper basically only support the specification of declarative constraints and triggers. It is astonishing that there is not more functionality such as consistency and redundancy checks for declarative constraints (recall the simple inconsistent specification of a check constraint in Section 5). Efficient algorithms for these tasks are known and can easily be integrated into these tools. It should be noted that checking certain properties of integrity constraint specifications occurs at database design time and not at database run-time. The additional time spent on integrity constraint design will eventually pay off in better data quality (as well as in better performance in case redundant checking of integrity constraints can be avoided).

With respect to the specification of constraint enforcing triggers, useful functionality can easily be added to design tools as well. For example, if triggers contain compensating database modifications, the design tool can easily derive dependencies among the triggers, thus pointing out potential cycles to the designer. A variety of tools that offer such functionality have been realized at Universities and can easily be integrated into commercial design tools. Regarding the support for designing and implementing constraint enforcing mechanisms, the ball is in the industries court. But without respective requests from customers, this can take some time. Thus, more importantly, customers have to be made aware of the functionality already present in existing systems, and customers have to push this technology into design tools.

Finally, we would like to comment on the biggest concern database designers have regarding the utilization of constraint enforcing mechanisms, namely *performance*. For most databases, constraint verifications are performed in a *batch-oriented* manner, that is, queries that verify integrity constraints are executed at times when there is a low usage of the system. Constraint violating rows are recorded and manually "cleaned" by users periodically. We agree that a sophisticated and complex integrity subsystem can lead to a decrease in performance, but this is only true for systems where high performance (e.g., based on the number of concurrent transactions supported) is required anyway. To the best of our knowledge, there has been no solid scientific research and general framework on the possible performance penalty caused by constraint enforcing mechanisms. There are only a few studies, such as [GFA92,HS95], which focus on performance and optimization of integrity checking in rather specific environments. We thus conclude that such a study is urgent and needs to be conducted in the very near future in a more representative setting based on commercial database systems, as an important argument to utilize constraint enforcing mechanisms and thus to eventually improve the correctness and usability of business critical data.

Acknowledgments: We are grateful to Kerstin Schwarz, Heiko Schuldt, and the anonymous reviewers for their valuable remarks and suggestions on a previous version of this paper.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behaviour of Database Production Rules: Termination, Confluence, and Observable Determisnism. In M. Stonebraker, editor, *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data, San Diego, CA*, ACM SIGMOD Record, Vol. 21, No. 2, pages 59–58, ACM Press, June 1992.

[BB82] P. A. Bernstein and B. Blaustein. Fast Methods for Testing Quantified Relational Calculus Assertions. In M. Schkolnick, editor, *Proc. of the 1982 ACM SIGMOD*

Int. Conf. on Management of Data, Orlando, Florida, pages 39–50. ACM Press, New York, June 1982.

[BBC80] P. A. Bernstein, B. Blaustein, and E. Clarke. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. In A. Taylor, editor, *Proc. of the 6th Int. Conf. on Very Large Data Bases, VLDB'80, Montreal, Quebec, Canada, October 1–3, 1980*, pages 126–136. IEEE Computer Society Press, Los Altos, CA, 1980.

[BCL86] J. A. Blakeley, N. Coburn, and P.-A. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. In W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *Proc. of the 12th Int. Conf. on Very Large Data Bases, VLDB'86, Kyoto, Japan, August 25–28, 1986*, pages 457–466. Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[BCP96] E. Baralis, S. Ceri, and S. Paraboschi. Modularization Techniques for Active Rules Design. *ACM Transactions on Database Systems*, 21(1):1–29, March 1996.

[BCW94] E. Baralis, S. Ceri, and J. Widom. Better Termination Analysis for Active Databases. In N. W. Paton and M. H. Williams, editors, *Rules in Database Systems, Proc. of the 1st Int. Workshop, RIDS'93, Edinburgh, Scotland, August 1993*, Workshops in Computing, pages 163–179. Springer-Verlag, London, 1994.

[BdBZ93] H. Balsters, R. A. de By, and R. Zicari. Typed Sets as a Basis for Object-Oriented Database Schemas. In O. Nierstrasz, editor, *ECOOP'93 — Object-Oriented Programming, Proc. of the 7th European Conf., Kaiserslautern, Germany, July 1993*, Lecture Notes in Computer Science 707, pages 161–184. Springer-Verlag, Berlin, 1993.

[BGM00] E. Bertino, G. Guerrini, and I. Merlo. Trigger Inheritance and Overriding in an Active Object Database System. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):588–608, 2000.

[Bis98] J. Biskup. Achievements of Relational Database Schema Design Theory Revisited. In B. Thalheim and L. Libkin, editors, *Semantics in Databases*, Lecture Notes in Computer Science 1358, pages 29–54. Springer-Verlag, Berlin, 1998.

[BM88] E. Bertino and D. Musto. Correctness of Semantic Integrity Checking in Database Management Systems. *Acta Informatica*, 26:25–57, 1988.

[Bro78] M. L. Brodie. *Specification and Verification of Data Base Semantic Integrity*. Dissertation, Technical Report CSRG-91, University of Toronto, April 1978.

[Buc94] A. P. Buchmann. Active Object Systems. In A. Dogac, M. T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, Nato ASI Series, pages 201–224. Springer-Verlag, Berlin, 1994.

[CCW00] S. Ceri, R. Cochrane, and J. Widom. Practical Applications of Triggers and Constraints: Success and Lingering Issues. In A. El Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB'2000, Proc. of the 26th Int. Conf. on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, pages 254–262. Morgan Kaufmann Publishers, Palo Alto, CA, 2000.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic Generation of Production Rules for Integrity Maintenance. *ACM Transactions on Database Systems*, 19(3):367–422, September 1994.

[Cha98] D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod79] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[CPM96] P. Cochrane, H. Pirahesh, and N. M. Mattos. Integrating Triggers and Declarative Constraints in SQL Database Sytems. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proc. of the 22nd Int. Conf. on Very Large Data Bases, VLDB'96, Bombay, India, September 3–6, 1996*, pages 567–578. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[CT95] J. Chomicki and D. Toman. Implementing Temporal Integrity Constraints Using an Active DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):566–582, August 1995.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Management. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *Proc. of the 16th Int. Conf. on Very Large Data Bases, VLDB'90, Brisbane, Australia, August 13–16, 1990*, pages 566–577. Morgan Kaufmann Publishers, Palo Alto, CA, 1990.

[Dat90] C. J. Date. A Contribution to the Study of Database Integrity. In C. J. Date, editor, *Relational Database Writings 1985–1989*, chapter 7, pages 185–215, Addison-Wesley, Reading, MA, 1990.

[Dat95] C. J. Date. *An Introduction to Database Systems*, Vol. 1. Addison-Wesley, Reading, MA, 6 edition, 1995.

[dB95] B. de Brock. *Foundations of Semantic Databases*. Prentice Hall, New York, 1995.

[DD97] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, Reading, MA, 4 edition, 1997.

[Deß93] S. Deßloch. *Semantic Integrity in Advanced Database Management Systems*. PhD thesis, Universität Kaiserslautern, Fachbereich Informatik, September 1993.

[EC75] K. P. Eswaran and D. D. Chamberlin. Functional Specifications of a Subsystem for Data Base Integrity. In D. S. Kerr, editor, *Proc. of the 1st Int. Conf. on Very Large Data Bases, VLDB'75, September 24–24, 1975, Framingham, MA, USA*, pages 48–68. Morgan Kaufmann Publishers, Los Altos, CA, 1975.

[EM99a] A. Eisenberg and J. Melton. SQL:1999, formerly known as SQL3. *ACM SIGMOD Record*, 28(1):131–138, March 1999.

[EM99b] A. Eisenberg and J. Melton. SQLJ — Part 1: SQL Routines using the Java^{TM} Programming Language. *ACM SIGMOD Record*, 28(4):58–63, December 1999.

[EM00] A. Eisenberg and J. Melton. SQL Standardization: The Next Steps. *ACM SIGMOD Record*, 29(1):63–67, March 2000.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, CA, 2 edition, 1994.

[FM93] A. Formica and M. Missikoff. Integrity Constraints Representation in Object-Oriented Databases. In T. W. Finin, C. K. Nicholas, and Y. Yesha, editors, *Information and Knowledge Management — Expanding the Definition of "Database", Selected Papers of the 1st Conf.*

CIKM'92, Baltimore, Maryland, USA, November, 1992, Lecture Notes in Computer Science 752, pages 89–85. Springer-Verlag, Berlin, 1993.

[For99] P. Fortier. *SQL3 — Implementing the SQL Foundation Standard*. McGraw-Hill, New York, NJ, 1999.

[FP94] P. Fraternali and S. Paraboschi. A Review of Compensating Techniques for Integrity Maintenance. In N. W. Paton and M. H. Williams, editors, *Rules in Database Systems, Proc. of the 1st Int. Workshop, RIDS'93, Edinburgh, Scotland, August 1993*, Workshops in Computing, pages 333–346. Springer-Verlag, London, 1994.

[FP97] P. Fraternali and S. Paraboschi. Ordering and Selecting Production Rules for Constraint Maintenance: Complexity and Heuristic Solution. *IEEE Transactions on Knowledge and Data Engineering*, 9(1), January 1997.

[GA93] P. W. P. J. Grefen and P. M. G. Apers. Integrity Control in Relational Database Systems — An Overview. *Data & Knowledge Engineering*, 10:187–223, 1993.

[GdB94] P. W. P. J. Grefen and R. A. de By. A Multi-Set Extended Relational Algebra: A Formal Approach to a Practical Issue. In *Proc. of the 10th IEEE Int. Conf. on Data Engineering, ICDE'94, Houston, Texas, USA, 14–18 February 1994*, pages 80–89. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[Ger96] M. Gertz. *Diagnosis and Repair of Constraint Violations in Database Systems*, Dissertationen zu Datenbanken und Informationssystemen, Vol. 19. infix-Verlag, Sankt Augustin, 1996.

[GFA92] P. W. P. J. Grefen, J. Flokstra, and P. M. G. Apers. Performance Evaluation of Integrity Control in a Parallel Main-Memory Database System. In A. M. Tjoa and I. Ramos, editors, *Database and Expert System Applications, Proc. of the 3rd Int. Conf., DEXA'92, Valencia, Spain*, pages 96–101. Springer-Verlag, Wien, 1992.

[GL96] M. Gertz and U. W. Lipeck. Deriving Optimized Integrity Monitoring Triggers from Dynamic Integrity Constraints. *Data & Knowledge Engineering*, 20(2):163–193, 1996.

[GP99] P. Gulutzan and T. Pelzer. *SQL-99 Complete, Really*. R&D Books, Lawrence, Kansas, 1999.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Gra78] J. Gray. Notes on Data Base Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems, An Advanced Course*, Lecture Notes in Computer Science 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[Gre92] P. W. P. J. Grefen. *Integrity Control in Parallel Database Systems*. PhD thesis, University of Twente, The Netherlands, 1992.

[GSW96a] S. Guo, W. Sun, and M. A. Weiss. On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):604–616, August 1996.

[GSW96b] S. Guo, W. Sun, and M. A. Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM Transactions on Database Systems*, 21(2):270–293, June 1996.

[HM75] M. M. Hammer and D. J. McLeod. Semantic Integrity in a Relational Data Base System. In D. S. Kerr, editor, *Proc. of the 1st Int. Conf. on Very Large Data Bases, VLDB'75, September 24–24, 1975, Framingham, MA, USA*, pages 25–47. Morgan Kaufmann Publishers, Los Altos, CA, September 1975.

[Hor92] B. M. Horowitz. A Run-time Execution Model for Referential Integrity Maintenance. In F. Golshani, editor, *Proc. of the 8th IEEE Int. Conf. on Data Engineering, ICDE'92, Tempe, Arizona, USA, February 2–3, 1992*, pages 548–556. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[HS95] U. Herzog and R. Schaarschmidt. Parallel Execution of Integrity Constraint Checks. In N. Pissinou, A. Silberschatz, E. K. Park, and K. Makki, editors, *Proc. of the 4th Conf. on Information and Knowledge Management (CIKM'95), Baltimore, Maryland, USA*, pages 218–225, ACM Press, November 1995.

[IBM00] IBM Corporation. *IBM DB2 Universal Database: SQL Reference, Version 7*, 2000.

[Inf99] Informix Corporation, Menlo Park, CA. *Informix Guide to SQL: Syntax, Informix Dynamic Server.2000, Version 9.2*, December 1999.

[Ing99] Ingres Corporation. *Ingres Database Administrator's Guide, Version II*, 1999.

[Int99] International Organization for Standardization (ISO) & American National Standards Institute (ANSI), ANSI/ ISO/IEC 9075-2:99. *ISO International Standard: Database Language SQL - Part 2: Foundation (SQL/Foundation)*, September 1999.

[KMC99] K. Kulkarni, N. Mattos, and R. Cochrane. Active Database Features in SQL3. In N. Paton, editor, *Active Rules in Database Systems*, chapter 10, pages 198–218, Springer-Verlag, New York, 1999.

[Mai83] D. Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, MD, 1983.

[Mar91] V. M. Markowitz. Safe Referential Integrity Structures in Relational Databases. In G. M. Lohmann, A. Sernadas, and R. Camps, editors, *Proc. of the 17th Int. Conf. on Very Large Data Bases, VLDB'91, Barcelona, Spain, September 3-6, 1991*, pages 123–132. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Mar94] V. M. Markowitz. Safe Referential Integrity and Null Constraint Structures in Relational Databases. *Information Systems*, 19(4):359–378, June 1994.

[Mic99] Microsoft Corporation. *Microsoft SQL Server, Version 7.0*, 1999.

[MS93] J. Melton and A. R. Simon. *Understanding the New SQL — A Complete Guide*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Nic82] J.-M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[Ora99] Oracle Corporation. *Oracle8i SQL Reference, Release 8.1.6*, December 1999.

[Pat99] N. W. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag, New York, 1999.

[PD99] N. W. Paton and O. Díaz. Introduction. In N. Paton, editor, *Active Rules in Database Systems*, chapter 1, pages 3–27, Springer-Verlag, New York, 1999.

[RH80] D. J. Rosenkrantz and H. B. Hunt III. Processing Conjunctive Predicates and Queries. In A. Taylor, editor, *Proc. of the 6th Int. Conf. on Very Large Data Bases, VLDB'80, Montreal, Quebec, Canada, October*

1–3, 1980, pages 64–72. IEEE Computer Society Press, Los Altos, CA, 1980.

[RH96] J. Reinert and T. Härder. Access Path Support for Referential Integrity in SQL2. *The VLDB Journal*, 5(3):196–214, July 1996.

[RSSS98] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of Aggregation Constraints. *Theoretical Computer Science*, 193(1–2):149–179, February 1998.

[SKN89] X. Sun, N. N. Kamel, and L. M. Ni. Processing Implications on Queries. *IEEE Transactions on Software Engineering*, 15(10):1168–1175, October 1989.

[SKS97] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, NJ, 3 edition, 1997.

[Sri93] D. Srivastava. Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):315–343, 1993.

[SST97] G. Saake, I. Schmitt, and C. Türker. *Object Databases — Concepts, Languages, Architectures*. International Thomson Publishing, Bonn, 1997. (In German).

[Sto75] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In W. F. King, editor, *Proc. of the 1975 ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA*, pages 65–78, ACM Press, 1975.

[Syb99] Sybase Inc. *Transact-SQL User Guide, Version 11.0*, 1999.

[Syb00] Sybase Inc. *Sybase Adaptive Server Anywhere Reference, Version 6.0.3*, 2000.

[Tür99a] C. Türker. Integrity Constraints and Specialization in Object Databases. In A. P. Buchmann, editor, *BTW'99, Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Freiburg, März 1999* , Informatik aktuell, pages 369–378. Springer-Verlag, Berlin, 1999. (In German).

[Tür99b] C. Türker. *Semantic Integrity Constraints in Federated Database Schemata*, Dissertationen zu Datenbanken und Informationssystemen, Vol. 63. infix-Verlag, Sankt Augustin, 1999.

[Tür01] C. Türker. Schema Evolution in SQL-99 and Commercial (Object-) Relational DBMS. In H. Balsters, B. De Brock, and S. Conrad, editors, *Database Schema Evolution and Meta-Modeling, 9th Int. Workshop on Foundations of Models and Languages for Data and Objects, Dagstuhl Castle, Germany, September 18–22, 2000, Post-Proceedings*, Lecture Notes in Computer Science 2065. Springer-Verlag, Berlin, 2001. *To appear*.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems*. Computer Science Press, Rockville, MD, 1988.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, Rockville, MD, 1989.

[WC96] J. Widom and S. Ceri, editors. *Active Database Systems — Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmaniam, and R. Zicari, editors. *Advanced Database Systems*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[ZH90] Y. Zhou and M. Hsu. A Theory for Rule Triggering Systems. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology — EDBT'90, Proc. of the 2rd Int. Conf. on Extending Database Technology*, Lecture Notes in Computer Science 416, pages 407–421. Springer-Verlag, Berlin, 1990.