Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS

Can Türker

Swiss Federal Institute of Technology (ETH) Zurich
Institute of Information Systems, ETH Zentrum
CH–8092 Zurich, Switzerland
tuerker@inf.ethz.ch

1 Introduction

A *database schema* denotes the description of the structure and behavior of a database. Straightforwardly, (database) *schema evolution* refers to changes of the database schema that occur during the lifetime of the corresponding database. It particularly refers to changes of schema elements already stored in the database.

The information about a database schema is stored in the schema catalog. Data stored in these catalogs is referred to as meta-data. In this sense, schema evolution could be seen as a change of the content of the schema catalog.

In an object-relational database model, such as proposed in SQL-99 [Int99], a database schema, among others, consists of the following elements:

- types, tables, and views,
- subtype and subtable relationships,
- constraints and assertions,
- functions, stored procedures, and triggers, and
- roles and privileges.

Thus more precisely, schema evolution can be defined as the creation, modification, and removal of such kinds of schema elements.

Although schema evolution is a well-known and partially well-studied topic, an overview and comparison of schema evolution language constructs provided in the SQL standard as well as in commercial database management systems is still missing. This survey paper intends to fill this gap. First, in Section 2, we give an overview of schema evolution operations supported by the new SQL standard, called SQL-99 [Int99]. Thereafter, in Section 3, we compare major commercial (object-)relational database management systems with respect to the support of these operations and others disregarded in SQL-99. Finally, we conclude the paper with some remarks on open schema evolution issues neglected in SQL-99 as well as in the current implementations of commercial database management systems.

2 Schema Evolution in SQL-99

Before we present the schema evolution language constructs provided in SQL-99 [Int99], we briefly introduce the basic notions and concepts of SQL-99.

H. Balsters, B. de Brock, and S. Conrad (Eds.): FoMLaDO/DEMM 2000, LNCS 2065, pp. 1–32, 2001.

2.1 Basic Schema Elements

The main concept for representing data in SQL-99 is the concept of a table, which is made up by a set of columns and rows. A *table* is associated with a schema and an instance:

- A *schema* of a table specifies the name of the table, the name of each column, and the domains (data types) associated with the columns. A domain is typically referred to by a domain name and has a set of associated values. Examples for basic domains (built-in data types) in SQL-99 are **INTEGER**, **REAL**, **NUMERIC**, **CHAR**, or **DATE**.
- An *instance* of a table schema, called *table*, is a set of rows where each row has the same structure as defined in the table schema, that is, each row the same number of columns and the values of the columns are taken from the corresponding domain.

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.

Besides the standard built-in data types, SQL-99 provides a *row* type constructor, an *array* type constructor and a *reference* type constructor. A row type constructor is used to define a column consisting of a number of *fields*. Any data type can be assigned to a field. The array type constructor is also applicable to any data type, whereas the applicability of the reference type constructor is restricted to user-defined types only.

A *user-defined type* is a *named* data type. SQL-99 distinguishes two kinds of user-defined types: (1) *distinct* types which are copies of predefined data types and (2) *structured* types which define a number of *attributes* and *method* specifications. Every attribute is associated with a data type, which itself can also be a user-defined type.

Structured types can be set into a subtype relationship. A *subtype* implicitly inherits the attributes and method specifications from its *supertype*. Every structured type may have at most one *direct* supertype. That is, SQL-99 does not support *multiple inheritance*.

A table that is created based on a structured type is called a *typed table*. Typed tables can be organized within a table hierarchy. A table can be a subtable of at most one direct supertable. All rows of a subtable are implicitly contained in all supertables of that table. Analogously to (base) tables, views can be typed and organized in view hierarchies.

A table column may rely on a built-in data type, row type, user-defined type, reference type, or collection type. The same holds for an attribute of a structured type.

SQL-99 furthermore supports the following concepts:

- *Domains* are named sets of values that are associated with a default value and a set of domain constraints.
- *Assertions* are named constraints that may relate to the content of individual rows of a table, to the entire content of a table, or to the contents of more than one table.

- *Routines* (*procedures* and *functions*) and *triggers* are named execution units that are used to implement application logic in the database.
- *Roles* and *privileges* are used to implement a security model. A role is a named group of related privileges which can be granted to users or roles.

To sum up, domains, user-defined types, tables, views, assertions, routines (procedures and functions), triggers, roles, and privileges are the basic schema elements in SQL-99. A *database schema* is formed by a set of schema element definitions and it evolves by adding, altering, or removing schema element definitions. It is important to note that some schema evolution operations may also have an effect on the actual database objects, for instance, on the rows of a table. In the following, we will see which language constructs are provided in SQL-99 to evolve a database schema. Before, to give an overview of the available operations, we summarize the main schema evolution operations in Table 1.

Table 1. Main Operations of Schema Evolution in SQL-99

	CREATE	ALTER	DROP
DOMAIN	✓	✓	✓
TYPE	✓	✓	✓
TABLE	✓	✓	✓
VIEW	✓	—	✓
ASSERTION	✓	—	✓
PROCEDURE	✓	✓	✓
FUNCTION	✓	✓	✓
TRIGGER	✓	—	✓
ROLE	✓	—	✓
PRIVILEG	✓	—	✓

2.2 Creating, Altering, and Removing a Domain

The syntax of the definition of a domain is as follows:[1]

CREATE DOMAIN <*domain-name*> [**AS**] <*data-type*>
[<*default-clause*>] [<*domain-constraint-list*>]

<*default-clause*> **::= DEFAULT** <*default-value*>

<*domain-constraint*> **::=** [<*constraint-name*>] <*check-constraint*>
[<*characteristics*>]

[1] In the following grammars, **terminal** and <*non-terminal*> symbols are distinguished using different font types. Optional symbols are enclosed by [] brackets. The symbol | is used to list alternatives.

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

A domain constraint is expressed by a check constraint which restricts the values of the specified data type to the permitted ones. The *default* clause is used to specify a default value for the domain.

The *characteristics* clause specifies the checking mode of a constraint. The checking mode determines the relative time when the constraint has to be checked within a transaction. In the *immediate* mode, the constraint is checked at the end of each database modification (either an insert, update or delete SQL-statement) that might violate the constraint. In the *deferred* mode, the checking is delayed until the end of the transaction.

In addition, the characteristics clause determines the initial checking mode, which must be valid for the constraint at the beginning of every transaction. 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.

The checking mode of a constraint can also be changed during the execution of a transaction using the following command:

```
SET CONSTRAINTS {ALL | <constraint-name-list>}
                {IMMEDIATE | DEFERRED }
```

Example 1. Suppose in our application domain, three different cities are distinguished: 'Munich', 'London', and 'Paris'. The "default city" is 'Munich'. Such a domain can be created as follows:

```
CREATE DOMAIN cities CHAR (6)
  DEFAULT 'Munich'
  CHECK(VALUE IN ('Munich', 'London', 'Paris'));
```

□

The definition of a domain can be changed by setting/removing the default value or by adding/removing a constraint to/from the domain. The syntax of the alter domain statement is as follows:

```
ALTER DOMAIN <domain-name> <alter-domain-action>

<alter-domain-action> ::= SET <default-clause>
                        | DROP DEFAULT
                        | ADD <domain-constraint>
                        | DROP CONSTRAINT <constraint-name>
```

For each column that is based on the domain to be altered by removing the default value, the dropped default value is placed in that column if it does not already contain a default value. Analogously, for each column that is based on the domain to be altered by removing a domain constraint, the dropped domain constraint is attached to the constraint list of that column.

A domain can be dropped from the database schema using the following command:

DROP DOMAIN <*domain-name*> {**RESTRICT** | **CASCADE** }

If **RESTRICT** is specified, then the domain must not be referenced in any of the following: table column, body of an SQL routine, query expression of a view, or search condition of a constraint.

Let c be a column of a table t that is based on a domain d. If **CASCADE** is specified, then removing d results in the following modifications of c:

- The domain d is substituted by a copy of its data type.
- The default clause of d is included in c, if c does not contain an own default clause.
- The constraints of d are added to the table t.

2.3 Creating, Altering, and Removing a User-Defined Type

The main corpus of the syntax of the definition of a user-defined type is as follows:

CREATE TYPE <*type-name*> [**UNDER** <*type-name*>]
[**AS** {<*predefined-type*> | <*attribute-def-list*>}]
[{**INSTANTIABLE** | **NOT INSTANTIABLE** }]
{**FINAL** | **NOT FINAL** }
[<*ref-type-spec*>] [<*method-spec-list*>]

<*attribute-def*> ::= <*attribute-name*> <*data-type*>
[<*ref-scope-check*>] [<*default-clause*>]

<*ref-scope-check*> ::= **REFERENCES ARE** [**NOT**] **CHECKED**
ON DELETED <*ref-action*>

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

<*ref-type-spec*> ::= **REF USING** <*predefined-type*>
| **REF FROM** (<*attribute-name-list*>)
| **REF IS SYSTEM GENERATED**

An attribute is a component of a structured type. A reference attribute is based on the reference type. The *reference scope check* clause shall only be specified for such reference attributes. Using this clause, one can specify whether and how to react on the deletion of a referenced instance. The *reference type specification* defines the way how the reference is created.

Every user-defined type is *instantiable* by default, that is, an instance of a user-defined type can be created unless it is explictly disallowed by specifying the keyword **NOT INSTANTIABLE**.

The *under* clause is used to create a subtype of another structured type. In this way, type hierarchies can be built. The under clause shall not be used for distinct types since it is obviously not reasonable. Let type2 be a subtype of type1, then type2 inherits all attributes of type1. Here, type2 shall not contain any attribute that has the same name as an inherited one. That is, attribute redefinition is not allowed.

The *final* clause indicates whether or not the structured type can be used as a supertype. Surprisingly, the keyword **NOT FINAL** must always be specified within the definition of a structured type. If the under clause is specified, the reference type specification is prohibited. For each attribute of a structured type *observer* and *mutator* methods are generated. These methods are used to access and modify the value of an attribute.

In case of the definition of a distinct type, the keyword **FINAL** must always be specified, while neither the under clause nor the reference type specification are allowed.[2]

Example 2. The following statement defines a distinct type:

```
CREATE TYPE swiss_francs AS DECIMAL (12,2) FINAL;
```

A structured type is defined as follows:

```
CREATE TYPE address AS (
  street      VARCHAR(35),
  number      DECIMAL(4),
  zip         DECIMAL(5),
  city        VARCHAR(25),
  country     VARCHAR(30)
) NOT FINAL;
```

The types defined above can now also be used within the definition of another structured type:

```
CREATE TYPE employee AS (
  id          SMALLINT,
  name        ROW(first VARCHAR(15), last VARCHAR(20)),
  address     address,
  supervisor  REF(employee) REFERENCES ARE CHECKED
                            ON DELETE SET NULL,
  hiredate    DATE,
  salary      swiss_francs
) NOT FINAL;
```

In this case references are checked automatically whenever an instance of the referenced type is deleted. If the deletion concerns an actually referenced instance, then the attribute supervisor of the referencing instance is set to **NULL**.

We now define a subtype of the structured type above:

[2] Since the keywords **NOT FINAL** and **FINAL** must always be used without any options, it is not understandable why they have been introduced.

```
CREATE TYPE manager UNDER employee AS (
  bonus       swiss_francs
) NOT FINAL;
```

Managers are thus modeled as special employees having an additional bonus salary. □

An existing structured type can also be changed by adding new attributes or method specifications and by removing existing attributes or method specifications. The main corpus of the syntax of the alter type statement looks as follows:

```
ALTER TYPE <type-name> <alter-type-action>

<alter-type-action> ::= ADD ATTRIBUTE <attribute-def>
                      | DROP ATTRIBUTE <attribute-name> RESTRICT
                      | ADD <method-spec>
                      | DROP <routine> RESTRICT

<routine> ::= {PROCEDURE | FUNCTION } <routine-name>
```

The attribute or routine to be dropped shall not be contained in any of the following: body of an SQL routine, query expression of a view, search condition of a constraint or assertion, or trigger action.

A user-defined type is dropped using the following command:

```
DROP TYPE <type-name> {RESTRICT | CASCADE }
```

If **RESTRICT** is specified, then the user-defined type to be dropped, among others, shall not be referenced in any of the following: another user-defined type, expression of a view, search condition of a constraint or assertion, or trigger action.

2.4 Creating, Altering, and Removing a Table

As already mentioned, there are two types of tables: (1) usual tables as known from the previous SQL standard and (2) typed tables which are based on a structured type.

The main corpus of the syntax of a table definition is follows:

```
CREATE TABLE <table-name>
                {( <table-element-list>)
                 | OF <type-name> [UNDER <table-name>]
                   [( <table-element-list>) ]}

<table-element> ::= <column-def>
                  | <table-constraint-def>
                  | REF IS <column-name> <ref-generation>
                  | <column-name> WITH OPTIONS <option-list>
```

```
<column-def> ::= <column-name> <type-or-domain-name>
                 [<ref-scope-check>] [<default-clause>]
                 [<column-constraint-def-list>]

<column-constraint-def> ::= [CONSTRAINT <constraint-name>]
                            <column-constraint> [<characteristics>]

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

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

<table-constraint-def> ::= [CONSTRAINT <constraint-name>]
                           <table-constraint> [<characteristics>]

<table-constraint> ::= UNIQUE ( VALUE )
                     | UNIQUE ( <column-name-list> )
                     | PRIMARY KEY ( <column-name-list> )
                     | CHECK ( <search-condition> )
                     | FOREIGN KEY ( <column-name-list> ) <ref-spec>

<ref-generation> ::= SYSTEM GENERATED
                   | USER GENERATED
                   | DERIVED

<option-list> ::= [<scope-clause>] [<default-clause>]
                  [<column-constraint-def-list>

<scope-clause> ::= SCOPE <table-name>
```

A usual table is defined by specifying a column list, whereas a typed table is defined using the *of* clause with the name of a structured type. In the latter case, the attributes of the structured type determines the schema of the table. The column options are used to define default values and constraints for a typed table.

Using the *under* clause, table hierarchies can be built by setting typed tables into a subtable relationship. The typed table specified in the under clause refers to the direct supertable of the created typed table. Every typed table may have at most one direct supertable. Besides, a subtable must not have an explicit primary key.

Let table1 be a table of type type1 and table2 a table of type type2. If table1 occurs in the under clause of the definition of table2, then type2 must be a direct subtype of type1.

Example 3. The following statement defines a typed table based on the structured type introduced in Example 2:

```
CREATE TABLE employees OF employee;
```

A subtable of this table is defined using the under clause, for instance, as follows:

```
CREATE TABLE managers OF manager UNDER employee;
```

Note this table has the same schema as the following usual table:

```
CREATE TABLE managers (
  id          SMALLINT,
  name        ROW(first VARCHAR(15), last VARCHAR(20)),
  address     address,
  supervisor  REF(employee) REFERENCES ARE CHECKED
                            ON DELETE SET NULL,
  hiredate    DATE,
  salary      swiss_francs,
  bonus       swiss_francs
);
```

A main difference between these two kinds of managers tables is that the rows of the typed table can be referenced in the sense of object-orientation. That is, there may be a reference column referring to an instance of that typed table. In this case, the value of the reference column is a row (or object) identifier associated with a row of the typed table. In contrast, the only way to reference a row in a usual table is to use the foreign key concept. Here, the value of the (referencing) foreign key must match the value of a (referenced) unique/primary key of that table. □

The definition of a table can be changed using the alter table statement, which has the following syntax:

```
ALTER TABLE <table-name> <alter-table-action>

<alter-table-action> ::= ADD [COLUMN] <column-def>
                       | ALTER [COLUMN] <column-name> <col-action>
                       | DROP <column-name> {RESTRICT | CASCADE}
                       | ADD <table-constraint-def>
                       | DROP <constraint-name> {RESTRICT | CASCADE}


<col-action> ::= SET <default-clause>
               | DROP DEFAULT
               | ADD <scope-clause>
               | DROP SCOPE {RESTRICT | CASCADE}
```

The alter table statement can only be applied to base tables. A typed table, however, cannot be altered. Usual base table can be altered by adding and

removing columns and constraints. Besides, an existing column of such a table can be altered by setting/removing the default value. Furthermore, the scope of a reference column can be added or removed. A scope can only be added if the reference column does not already have one.

If **RESTRICT** is specified for the *drop column* clause, then the column to be dropped shall not be contained in any of the following: body of an SQL routine, query expression of a view, search condition or triggered action of a trigger, or search condition of a table constraint.

A primary key can only be added to a table that has no supertable.

If **RESTRICT** is specified for the *drop constraint* clause, then the following must hold: neither a table constraint nor a view shall be dependent on the table constraint to be dropped and its name shall not be contained in the body of any SQL routine body.

A table is dropped from the database using the following command:

DROP TABLE <*table-name*> {**RESTRICT** | **CASCADE** }

Removing a table means that the table schema as well as the table instance are removed together with the corresponding privileges.

If **RESTRICT** is specified, then the table to be dropped shall not have any subtable, and moreover it shall not be referenced in any of the following: body of an SQL routine, scope of the declared type of an SQL routine parameter, query expression of a view, search condition or triggered action of a trigger, search condition of a check constraint of another table, search condition of an assertion, or a referential constraint of another referenced table. If **CASCADE** is specified, such dependent schema elements are dropped implicitly.

2.5 Creating and Removing a View

SQL-99 supports two types of views: (1) usual views and (2) typed views that are based on a structured type.

The main corpus of the syntax of the view definition is as follows:

CREATE VIEW <*table-name*>
{**(** <*column-name-list*>**)**
| **OF** <*type-name*> [**UNDER** <*table-name*>]
[**(** <*column-option-list*>**)**]}
AS <*query-expression*>
[**WITH CHECK OPTION**]

<*column-option*> **::=** <*column-name*> **WITH OPTIONS** <*scope-clause*>

A usual view is defined by a column list, whereas a typed view is specified using the *of* clause which determines the schema of the view. The column option list is used to specify the scope of reference columns.

The *under* clause is used to create a subview of another typed view. In this way, view hierarchies can be built. The typed view specified in the under clause

refers to the direct superview of the created typed view. Every typed view may have at most one direct supertable.

Let view1 be a view of type type1 and view2 be a view of type type2. If view1 occurs in the under clause of the definition of view2, then type2 must be a direct subtype of type1.

The *check* option ensures that all data modification statements performed on the view will be validated against the query expression of that view.

Example 4. Assuming there is a structured type employee and a typed table employees, the following statement defines a typed view:

```
CREATE VIEW cheap_employees OF employee AS (
  SELECT * FROM employees WHERE salary < 5000
);
```

□

A view definition cannot be altered, but it can be dropped using the following statement:

DROP VIEW <*table-name*> {**RESTRICT** | **CASCADE** }

If **RESTRICT** is specified, then the view to be dropped shall neither have any subviews nor it shall be referenced in any of the following: body of an SQL routine, scope of the declared type an SQL routine parameter, query expression of another view, search condition or triggered action of a trigger, search condition of a check constraint of another table, search condition of an assertion, or a referential constraint of another referenced table. If **CASCADE** is specified, such dependent schema elements are dropped implicitly.

2.6 Creating and Removing an Assertion

An assertion is created using the following statement:

CREATE ASSERTION <*assertion-name*> **CHECK (** <*search-condition*>**)**
[<*characteristics*>]

In constrast to the search condition of a column constraint or a table constraint, the search condition of an assertion may also refer to more than one row of one or more tables, that is, table-level and database-level check constraints can be defined within an assertion.

An existing assertion is dropped from the database using the following statement:

DROP ASSERTION <*assertion-name*>

2.7 Creating, Altering, and Removing a Routine

A routine in SQL-99 refers to a procedure or function. The main corpus of the syntax of a procedure and function definition is as follows:

CREATE PROCEDURE <*routine-name*> **(** <*parameter-list*>**)**
<*routine-characteristics*> <*routine-body*>

CREATE FUNCTION <*routine-name*> **(** <*parameter-list*>**)** <*returns-clause*>
<*routine-characteristics*> <*routine-body*>

Loosely spoken, a function is a procedure with an additional *return* clause. A routine can be specified with different characteristics. For instance, a routine can be either an SQL or an external routine, it can be deterministic or non-deterministic, and it can be a routine that only reads or modifies SQL data. The routine body consists of an SQL procedure statement.

A routine can also be altered and dropped, respectively, using the following commands:

ALTER <*routine*> <*alter-routine-characteristics*> **RESTRICT**

DROP <*routine-name*> {**RESTRICT** | **CASCADE** }

If **RESTRICT** is specified, then the routine to be dropped shall not be referenced in any of the following: body of an SQL routine, query expression of a view, search condition of a check constraint or assertion, or triggered action of a trigger. If **CASCADE** is specified, such dependent schema elements are dropped implicitly.

2.8 Creating and Removing a Trigger

The syntax of a trigger definition is as follows:

CREATE TRIGGER <*trigger-name*>
{**BEFORE** | **AFTER** }
{**INSERT** | **DELETE** | **UPDATE** [**OF** <*column-name-list*>]}
ON <*table-name*> [**REFERENCING** <*old-or-new-values-list*>]
[**FOR EACH** {**ROW** | **STATEMENT** }]
[**WHEN** **(** <*search-condition*>**)**]
<*SQL-procedure-stat*> | **BEGIN ATOMIC** <*SQL-procedure-stat-list*> **END**

<*old-or-new-values*> **::=** {**OLD** | **NEW** } [**ROW**] [**AS**] <*correlation-name*>
| {**OLD** | **NEW** } **TABLE** [**AS**] <*table-alias*>

A trigger is implicitly activated when the specified event occurs. 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. Valid triggering events are the execution of insert, update, or delete statements. 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). Trigger conditions and actions can refer to both old and new values of the rows affected by the triggering event.

An existing trigger can be dropped using the following command:

DROP TRIGGER <*trigger-name*>

2.9 Creating and Removing Roles

The create role statement has the following syntax:

CREATE ROLE <*role-name*> [**WITH ADMIN OPTION** <*grantor*>]

After the creation of a role, no privileges are associated with that role. These have to be added using the grant statement, as described in the following. The *admin* option is used to give the grantee the right to grant the role to others, to revoke it from other users or roles, and to drop or alter the granted role.

An existing role is dropped using the following statement:

DROP ROLE <*role-name*>

2.10 Granting and Revoking Privileges

Privileges are granted to a user or role using the grant statement, which has the following syntax:

GRANT {**ALL PRIVILEGES** | <*privileges-or-role-name-list*>}
TO <*grantee-list*>
[**WITH HIERARCHY OPTION**] [**WITH GRANT OPTION**]
[**WITH ADMIN OPTION**] [**GRANTED BY** <*grantor*>]

The *hierarchy* option can only be applied to privileges on typed tables or typed views. It specifies that the granted privileg is also valid for all subtables (subviews) of a typed table (typed view). The *grant* option is used to specify that the granted privileg is also grantable, that is, the user is allowed to give others the privileg to access and use the named object. In general, the hierarchy and grant options shall only be specified when privileges are granted while the admin option shall only be specified when roles are granted.

A granted privileg or role can be revoked from a user or role using the revoke command. The syntax of the revoke command is as follows:

REVOKE [{**GRANT** | **HIERARCHY** | **ADMIN** } **OPTION FOR**]
{**ALL PRIVILEGES** | <*privileges-or-role-name-list*>}
FROM <*grantee-list*> [**GRANTED BY** <*grantor*>]
{**RESTRICT** | **CASCADE** }

Analogously to the grant statement, the hierarchy and grant option shall only be specified when privileges are revoked, while the admin option can only be specified when roles are revoked.

Example 5. The following statement creates a role reademp. This role is associated with the privileg to read the data of all kinds of employees:

```
CREATE ROLE reademp;
GRANT SELECT ON employee TO reademp WITH HIERARCHY OPTION;
```

The hierarchy option ensures that all users associated with the role reademp are also allowed to read the data of any special employee, for instance, the salary of a manager.

It is also possible to revoke only the hierarchy option from the role reademp. This is achieved by executing the following statement:

```
REVOKE HIERARCHY OPTION FOR SELECT ON employee FROM reademp;
```

Using the statement above without the hierarchy option revokes the privileg to select *any* employee. □

3 Comparison of Schema Evolution Constructs in SQL-99 and Commercial DBMS

In this section, we compare the schema evolution language constructs of SQL-99 [Int99] with that of the commercially available (object-)relational database management systems Oracle8i Server (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 (Version 11.5) [Syb99], and Ingres II (Release 2.0) [Ing99]. In the following, we will use the abbreviations Oracle, DB2, Informix, MSSQL, Sybase, and Ingres, respectively, to refer to these systems. In addition, we will use the term *reference systems* to refer to all of these systems as a whole. It should be pointed out that in fact only Oracle, DB2, and Informix could be denoted as object-relational database management systems. The other three systems are pure relational database management systems.

3.1 Domains and Assertions

Neither the concept of a domain nor the concept of an assertion is supported by any reference system.

However, there are a few rudimentary approaches in that directions. Ingres, for instance, provides the concept of an integrity rule which actually corresponds to a row-level assertion. Internally, these integrity rules are stored with a generated integer number, which is used to identify an integrity rule within a table definition. This number is needed, for instance, to drop an integrity rule. An integrity rule is created and dropped, respectively, as follows:

```
CREATE INTEGRITY ON <table-name> IS <search-condition>
DROP INTEGRITY ON <table-name> {ALL | <integer-list>}
```

The creation of an integrity rule fails if the table contains a row that does not satisfy the search condition. In the Ingres manuals, there is a hint to define check constraints within a create table or alter table statement instead of specifying integrity rules that anyway are not conform to the standard.

MSSQL and Sybase provide language constructs for specifying named default values and rules. A named default value is created and dropped, respectively, as follows:

CREATE DEFAULT <*default-name*> **AS** <*constant-expression*>
DROP DEFAULT <*default-name*>

A named default value can then be bound to a table column or a distinct type using the predefined stored procedure **SP_BINDEFAULT** with the following parameters:

SP_BINDEFAULT <*default-name*>, '<*column-or-type-name*>'

Before a named default value can be dropped, it must be unbound from all dependent schema elements using the predefined stored procedure **SP_UNBINDEFAULT**.

In the context of MSSQL and Sybase, a rule defines a domain of acceptable values for a particular table column or distinct type. A rule is created and dropped, respectively, as follows:

CREATE RULE <*rule-name*> **AS** <*search-condition*>
DROP RULE <*rule-name*>

A rule must be unbound using the predefined stored procedure **SP_UNBINDRULE** before it can be dropped.

Similarly to a named default value, a rule is bound to a table column or distinct type using the predefined stored procedure **SP_BINDRULE** with the following parameters:

SP_BINDRULE <*rule-name*>, '<*column-or-type-name*>'

When a rule is bound to a table column or distinct type, it specifies the acceptable values that can be inserted into that column. A rule, however, does not apply to data that already exists in the database at the time the rule is created. It also does not override a column definition. That is, a nullable column can take the null value even though **NULL** is not included in the text of the rule. If both a default and a rule are defined, the default value must fall in the domain defined by the rule. A default value that conflicts with a rule will never be inserted. An error message will be generated each time such a conflicting default value is tried to be inserted. Since a rule performs some of the same functions as a check constraint, the latter, standard way of restricting the values in a column is recommended.

3.2 User-Defined Types

As already mentioned, SQL-99 supports two kinds of user-defined types: distinct type and structured types. Since certain user-defined types have been provided by some of the reference systems prior to the introduction of SQL99, the notions and language constructs used in these systems differ in some cases.

Table 2 gives an overview of the support of the various named and unnamed type constructors in the reference systems. Interestingly, the unnamed array type is supported in none of the reference systems, although it is proposed in SQL-99. On the other hand, the unnamed collection types set, multiset, and list are only provided in Informix. These types were originally included in the preliminary drafts of SQL-99, but they were now postponed to the next version of the standard, which is currently referred to as SQL4.

Table 2. Comparison of User-Defined Types

TYPE		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
Named	DISTINCT	✓	—	✓	✓	✓	✓	—
	OBJECT (Structured)	✓	✓	✓	—	—	—	—
	ROW	—	—	—	✓	—	—	—
	VARRAY	—	✓	—	—	—	—	—
	TABLE	—	✓	—	—	—	—	—
Unnamed	ROW	✓	—	—	✓	—	—	—
	SET	—	—	—	✓	—	—	—
	MULTISET	—	—	—	✓	—	—	—
	LIST	—	—	—	✓	—	—	—
	ARRAY	✓	—	—	—	—	—	—
	REF	✓	✓	✓	—	—	—	—
HIERARCHY (UNDER)		✓	—	✓	✓	—	—	—

In DB2 and Informix, the creation of a distinct type is performed using the following command:

CREATE DISTINCT TYPE <*type-name*> **AS** <*source-type-name*>

In DB2, the statement above must end with an additional keyword **WITH COMPARISONS** unless the source data type is **BLOB**, **CLOB**, **LONG VARCHAR**, **LONG VARGRAPHIC**, or **DATALINK**. This option ensures that the instances of the same distinct type can be compared. Both, DB2 and Informix automatically generate two functions to cast in both directions from the distinct type to its source type and vice versa.

DB2's syntax of the definition of a structured type is more or less the same as proposed in SQL-99. Nevertheless, it is worth to mention that DB2 requires

the specification of an awkward, non-optional keyword **MODE DB2SQL**. On the other hand, the specification of the keyword **NOT FINAL** is optional.

DB2 also provides a means to alter the definition of a structured type. Attributes and methods can be added (dropped) to (from) a structured type. The syntax of the alter type statement is as follows:

ALTER TYPE <*type-name*>
{**ADD ATTRIBUTE** <*attribute-def*>
| **DROP ATTRIBUTE** <*attribute-name*> [**RESTRICT**]
| **ADD** <*method-spec*>
| **DROP METHOD** <*method-name*> [**RESTRICT**]}

The *restrict* option ensures that no attribute or method can be dropped if the structured type they belong to is referenced in any other schema element.

Distinct and structured types are dropped in DB2 using the following commands:

DROP DISTINCT TYPE <*type-name*>
DROP TYPE <*type-name*>

A user-defined type is not dropped if there is any schema element that depends on this type. An error occurs if the user-defined type to be dropped has a subtype or is used within the definition of a column, typed table, typed view, or another structured type.

Informix supports the concept of a structured type under the notion of a *named row type*. The syntax of the definition of a named row type is as follows:

CREATE ROW TYPE <*type-name*> **(** <*attribute-def-list*>**)**
[**UNDER** <*type-name*>]

A named row type can be used to create a typed table or typed view. It can also be assigned to a column of a table or to an attribute of another named row type. The concept of subtyping is supported analogously to SQL-99. That is, attributes and methods are inherited from the supertypes to the subtypes and the redefinition of inherited attributes and methods is not allowed. In a type hierarchy, a named row type cannot be substituted for its supertype or its subtype.

An attribute of a named row type can be defined as non-nullable. Other kinds of constraints, however, cannot be applied to a named row type directly. They have to be defined within the create table or alter table statement.

Besides these two kinds of user-defined types, Informix also supports the unnamed *row* type as well as the collection types *set*, *multiset*, and *list*. Complex data types are created by combining these type constructors in any order.

Distinct types and named row types are dropped in Informix using the following commands:

DROP TYPE <*type-name*> **RESTRICT**
DROP ROW TYPE <*type-name*> **RESTRICT**

Since the keyword **RESTRICT** is mandatory, a user-defined type cannot be dropped if the database contains any schema element that depends on this type.

Oracle distinguishes three kinds of user-defined types: *object types*, *varying array types*, and *table types*, which are defined according to the following syntax:

CREATE TYPE <*type-name*> **AS OBJECT (<***attr-method-spec***>)**
CREATE TYPE <*type-name*> **AS VARRAY OF** <*data-type*>
CREATE TYPE <*type-name*> **AS TABLE OF** <*data-type*>

Oracle does not support the concept of subtyping. The notion of an object type corresponds to the notion of a structured type in SQL-99. Distinct types are not supported. Instead, two named collection types are provided. A varying array type defines an ordered multiset of elements, each of which has the same data type. The data type of the elements have to be one of the following: built-in data type, reference type, or object type. The cardinality of the multiset must be explicitly specified. A table type in fact defines an unordered multiset of elements, each of which has the same data type. The elements can be either instances of an object type or values of a built-in type. The cardinality of this multiset is not restricted. A collection type, however, cannot contain any other collection type. That is, a varying array type, for instance, cannot contain any elements that are varying arrays or tables. In this way, nesting of tables is restricted to one level.

Nevertheless, Oracle allows to alter a type definition, either by recompiling a type definition or by replacing the object type. The syntax of the alter type statement looks as follows:

ALTER TYPE <*type-name*> **AS**
{**COMPILE** | **REPLACE AS OBJECT (<***attr-method-spec***>)** }

A user-defined type is dropped in Oracle using the following command:

DROP TYPE <*type-name*> [**FORCE**]

This statement removes a user-defined type if there is no schema element in the database that relies on this type. If there is such a dependent schema element, the *force* option can be used to drop the type and to mark all columns that use this type as *unused*.

The concept of a distinct type is supported in MSSQL and Sybase, too. In these systems, a distinct type is created and dropped, respectively, by executing the following predefined stored procedures:

SP_ADDTYPE <*type-name*>, '<*predefined-type*>'
SP_DROPTYPE <*type-name*>

A distinct type cannot be dropped if it is referenced in any other schema element.

Table 3 summarizes the various schema evolution operations related to user-defined types.

Table 3. Comparison of Type Constructs

			SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
Distinct Type	CREATE		✓	—	✓	✓	(✓)	(✓)	—
	ALTER		—	—	—	—	—	—	—
	DROP		—	—	✓	—	(✓)	(✓)	—
		RESTRICT	✓	—	—	✓	—	—	—
		CASCADE	✓	—	—	—	—	—	—
Structured Type (Object/Row)	CREATE		✓	✓	✓	✓	—	—	—
		UNDER	✓	—	✓	✓	—	—	—
	ALTER		✓	(✓)	✓	—	—	—	—
	DROP		—	✓	✓	—	—	—	—
		RESTRICT	✓	—	—	✓	—	—	—
		CASCADE	✓	(✓)	—	—	—	—	—

3.3 Tables

In all reference systems, the relational part of a table definition basically follows the proposal of SQL-99. In the following, we therefore focus more on the object-relational extensions of a table definition.

While typed tables are supported in Oracle, DB2, and Informix, table hierarchies (subtables) are only provided in DB2 and Informix. The main corpus of the definition of typed tables in all three systems more or less follows the definition of SQL-99. Here, the systems mainly differ in the naming of the same concepts. DB2 uses the terms of SQL-99 (one should better say that SQL-99 uses the terms of DB2), Informix also uses the term typed table but the notion of a named row type instead of the term structured type, and Oracle calls these concepts object tables and object types.

In all three systems, user-defined types can be used as data type of a column. As we could see in the previous subsection, various type constructors are provided in the different systems to create complex data types.

Tables can be altered in all reference systems. However, the provided alter table constructs differ in several ways. Table 4 gives an overview of the different constructs.

As we can see there, all reference systems provide means to add new columns and constraints to a table. Sybase, however, has the restriction that the newly added column must be nullable. In all reference systems, it is also possible to drop an existing constraint from a table. A column, however, can only be dropped in Oracle, Informix, MSSQL, and Ingres.

Some of the reference systems distinguishes between different *drop* options. Ingres is the only reference system that follows the proposal of SQL-99 with respect to the removing of a column or constraint. The specification of the drop option is mandatory and it can be decided between the options **RESTRICT** and

Table 4. Comparison of **ALTER TABLE** Constructs

ALTER TABLE <*table-name*>		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
ADD COLUMN		✓	✓	✓	✓	✓	✓	✓
ALTER COLUMN	SET <*data-type*>	—	(✓)	(✓)	(✓)	(✓)	—	—
	SET DEFAULT	✓	(✓)	—	(✓)	—	(✓)	—
	DROP DEFAULT	✓	—	—	—	—	—	—
	CONSTRAINT	—	(✓)	—	(✓)	(✓)	—	—
	ADD SCOPE	✓	—	✓	—	—	—	—
	DROP SCOPE	✓	—	—	—	—	—	—
DROP COLUMN		—	✓	—	✓	✓	—	—
	RESTRICT	✓	—	—	—	—	—	✓
	CASCADE	✓	—	—	—	—	—	✓
ADD CONSTRAINT		✓	✓	✓	✓	✓	✓	✓
DROP CONSTRAINT		—	✓	✓	✓	✓	✓	—
	RESTRICT	✓	—	—	—	—	—	✓
	CASCADE	✓	✓	—	—	—	—	✓
ADD TYPE		—	—	—	✓	—	—	—

CASCADE. The former requires that the drop column or drop constraint statement is rejected if there is a schema element that depends on the schema element to be dropped. The latter implicitly drops the dependent schema elements. Oracle supports the cascade option in combination with the drop constraint construct. If **CASCADE** is not explicitly specified, the default mode implements the restrict semantics. The other reference systems do not support any drop options. The default mode of the drop constraint statement is **RESTRICT** in MSSQL and Sybase, while it is **CASCADE** in DB2 and Informix. A drop column construct (without a drop option) is also provided in Oracle, Informix, and MSSQL. The default mode is **RESTRICT** in Oracle and MSSQL, while it is **CASCADE** in Informix.

All reference systems except Ingres provide an alter column construct. In Table 4 we used the symbol '(✓)' to mark the constructs that do not follow the syntax of the SQL-99 proposal. The syntax and semantics of the alter column constructs provided in the different systems differ in several ways:

Oracle: **ALTER TABLE** <*table-name*> **MODIFY** <*column-name*>
[<*data-type*>] [<*default-clause*>] [**NOT NULL**]

DB2: **ALTER TABLE** <*table-name*> **ALTER COLUMN** <*column-name*>
{**SET DATA TYPE** <*data-type*>
| **ADD SCOPE** <*typed-table-name*>}

Informix: **ALTER TABLE** <*table-name*> **MODIFY** <*column-name*>
<*data-type*> [<*default-clause*>] [<*constraint*>]

MSSQL: **ALTER TABLE** <*table-name*> **ALTER COLUMN** <*column-name*> <*data-type*> [**NOT NULL** | **NULL**]

Sybase: **ALTER TABLE** <*table-name*> **REPLACE** <*column-name*> <*default-clause*>

That is, Oracle and Informix provides a construct to alter the data type, default value, and constraints of a column. In fact, Oracle only allows the specification of a not null constraint; other constraints have to be added or dropped using the add constraint and drop constraint statements, as discussed previously. In MSSQL, a column can be altered by changing its data type and defining a not null constraint for this column. DB2 provides a construct that can be used either to change the data type of a column or to add a scope to a reference column. Sybase can replace the default value of a column.

Informix allows to convert a usual table into a typed one. This modification is performed using the *add type* clause. The added type must be compatible with the impicit type of the usual table, that is, they must have exactly the same attributes with respect to their names and data types.

Although all reference systems provide a drop table statement, they implement this statement with different options and semantics (for an overview see Table 5).

Table 5. Comparison of **DROP TABLE** Constructs

		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
DROP TABLE		—	✓	✓	✓	✓	✓	✓
	RESTRICT	✓	—	—	✓	—	—	—
	CASCADE	✓	✓	(✓)	✓	—	—	—

DB2 applies the *cascade or invalidate* semantics, meaning that the content of the table is removed together with all dependent indexes, constraints, and privileges, while dependent views, procedures, functions, and triggers are only invalidated. If a table contains a subtable, it cannot be dropped before all its subtables are dropped. DB2 provides the *hierarchy* option to drop all tables of a table hierarchy. Note that the functionality of the option is included in the *cascade* option of SQL-99.

In MSSQL and Sybase, a drop table statement removes the table definition together with all data, indexes, triggers, constraints, and privileges for that table. Any view, stored procedures, default, or rule that references the dropped table must be dropped explicitly.

Oracle applies the restricts semantics to drop a table. Nevertheless, it supports the specification of the cascade option to drop a table together with all

dependent constraints. As in DB2, dependent views, procedure, functions, and triggers are not dropped. They are only invalidated and can later be used if the table is re-created.

Informix supports the restrict semantics as well as the cascade semantics. If neither **RESTRICT** nor **CASCADE** is specified, the drop table statement is executed with the cascade semantics.

Ingres drops a table implicitly with the cascade semantics, although it does not support the explicit specification of this option.

3.4 Views

The three object-relational systems Oracle, DB2, and Informix support both usual (untyped) views as well as typed views. However, subviews (view hierarchies) are only provided in DB2. Table 6 gives an overview of various schema evolution operations defined on the concept of a view.

Table 6. Comparison of **VIEW** Constructs

		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
CREATE	VIEW ...	✓	✓	✓	✓	✓	✓	✓
	VIEW ... OF ...	✓	✓	✓	✓	—	—	—
	VIEW ... OF ... UNDER	✓	—	✓	—	—	—	—
ALTER VIEW ...		—	✓	✓	—	✓	—	—
DROP	VIEW ...	—	✓	✓	✓	✓	✓	✓
	VIEW ... RESTRICT	✓	—	—	✓	—	—	—
	VIEW ... CASCADE	✓	—	(✓)	✓	—	—	—

Let us now first consider the main corpus of the syntax of the view definition in Oracle:

CREATE VIEW <*table-name*>
[**(** <*column-name-list*>**)** | **OF** <*type-name*>]
AS <*query-expression*>
[**WITH** {**CHECK OPTION** | **READ ONLY** }]

A typed view is defined using the *of* clause, as in SQL-99. Oracle provides the *read-only* option which ensures that no insert, update, or delete can be performed through the view on the underlying base table(s). The well-known *check* option validates inserts, updates, and deletes against the query expression of the view and rejects invalid changes through the view.

The view definition in Informix is very similar to the previous one, except that the read only option is not supported and the keyword **OF TYPE** must be used instead of the keyword **OF** to define a typed view:

CREATE VIEW <*table-name*>
[**(** <*column-name-list*>**)** | **OF TYPE** <*type-name*>]
AS <*query-expression*> [**WITH CHECK OPTION**]

Compared to the previous view definitions, DB2 provides a more advance one, which allows to define view hierarchies based on typed views. For the definition of a subview, DB2' create view statement, however, requires the ackward, non-optional keywords **MODE DB2SQL** and **INHERIT SELECT PRIVILEGES**. The main part of the create view statement looks as follows:

CREATE VIEW <*table-name*>
[**(** <*column-name-list*>**)** | **OF** <*type-name*>
[**MODE DB2SQL**
UNDER <*table-name*>
INHERIT SELECT PRIVILEGES]]
AS <*query-expression*> [**WITH CHECK OPTION**]

Oracle, DB2, and MSSQL provide an alter view statement. However, since the alter view statement is not standardized yet, the different implementations provide different functionality under the same label. In Oracle, the alter view statement only recompiles a view:[3]

ALTER VIEW <*table-name*> **COMPILE**

In DB2, the alter view statement modifies an existing view by altering a reference column to add a scope. The syntax of this statement is as follows:

ALTER VIEW <*table-name*> **ALTER** [**COLUMN**] <*column-name*>
ADD SCOPE <*typed-table-name*>

Finally, in MSSQL, the alter view statement replaces a previously created view without affecting dependent stored procedures or triggers and without changing privileges. The syntax of this variant is as follows:

ALTER VIEW <*table-name*> [**(** <*column-name-list*>**)**]
AS <*query-expression*> [**WITH CHECK OPTION**]

Concerning the drop view statement, all six reference systems more or less closely follows the proposal of SQL-99. In Oracle, DB2, MSSQL, and Sybase, the execution of a drop view statement invalidates all views that are based on the view to be dropped. DB2 additionally provides the *hierarchy* option, which is similar to the *cascade* option in SQL-99. The hierarchy option is used to implicitly drop all views of a view hierarchy. The syntax of the corresponding statement is as follows:

DROP VIEW HIERARCHY <*table-name*>

Here, table name refers to the name of a root view.

[3] A view can be replaced in Oracle using the keyword **CREATE OR REPLACE VIEW** in create view command.

In Informix, a view can be dropped following either the restrict or the cascade semantics. **RESTRICT** ensures that the drop view operation fails if any existing view is defined on the view to be dropped. **CASCADE** guarantees that all such dependent view are implicitly dropped, too. If none of these keywords is explicitly specified, the drop operation is executed with the cascade semantics. Ingres also applies this strategy, but without providing any options. The remaining reference systems apply the restrict semantics.

3.5 Procedures, Functions, and Triggers

All reference systems support the creation and deletion of routines and triggers. However, since the programming languages to define these routines and triggers differ in several ways, we omit a comparison of the various programming styles. Instead, we address some other interesting issues.

As the alter view statement, the alter routine and alter trigger statements are not standardized yet. Nevertheless, they are included in some of the reference systems. For instance, the alter procedure statement is used in Oracle to recompile a (stand-alone) procedure:

Oracle: **ALTER** {**PROCEDURE** | **FUNCTION** | **TRIGGER**} <*routine-name*> **COMPILE**

MSSQL allows to alter an existing procedure without changing privileges and without affecting any dependent stored procedures or triggers. Analogously, MSSQL provides an alter trigger statement that replaces the definition of an existing trigger.

Oracle, Informix, and MSSQL even support the enabling and disabling of triggers:

Oracle: **ALTER TRIGGER** <*trigger-name*> {**ENABLE** | **DISABLE**}
ALTER TABLE <*table-name*> {**ENABLE** | **DISABLE**} **ALL TRIGGERS**

Informix: **SET TRIGGERS** <*trigger-name-list*> {**ENABLED** | **DISABLED**}

MSSQL: **ALTER TABLE** <*table-name*> {**ENABLE** | **DISABLE**} **TRIGGER** {**ALL** | <*trigger-name-list*>}

In Oracle, DB2, MSSQL, and Sybase, the execution of a drop routine statement invalidates all schema elements that are based on the routine to be dropped. In Informix and Ingres, a procedure is dropped implicitly with the cascade semantics.

3.6 Roles and Privileges

Roles and privileges are supported by all six reference systems in a very similar way as proposed in SQL-99. Table 7 gives an overview of the corresponding language constructs.

As depicted there, the concept of a role is provided in all reference systems except DB2. Concerning the creation of a role, Oracle, Informix, and Ingres

Table 7. Comparison of **ROLE**, **GRANT**, and **REVOKE** Constructs

			SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
CREATE ROLE			✓	✓	—	✓	(✓)	(✓)	✓
DROP ROLE			✓	✓	—	✓	(✓)	(✓)	✓
ALTER ROLE			✓	✓	—	✓	(✓)	(✓)	✓
GRANT			✓	✓	✓	✓	✓	✓	✓
	WITH GRANT OPTION		✓	✓	✓	✓	✓	✓	✓
REVOKE			—	✓	✓	✓	✓	✓	—
		RESTRICT	✓	—	—	✓	—	—	✓
		CASCADE	✓	✓	—	✓	—	—	✓
	GRANT OPTION FOR		—	—	✓	—	✓	✓	—
		RESTRICT	✓	—	—	—	—	—	✓
		CASCADE	✓	—	—	—	✓	✓	✓

closely follows the SQL-99 proposal. MSSQL and Sybase, in contrast, implement the concept of a role by providing predefined stored procedures. In MSSQL, a role is created by executing the stored procedure **SP_ADDAPPROLE** with the following parameters:

SP_ADDAPPROLE <*role-name*>, <*password*>

After creation a role is inactive by default. It can be activated by executing the stored procedure **SP_SETAPPROLE** with the same parameters as above. A role is dropped by executing the stored procedure **SP_DROPAPPROLE** with the name of the role:

SP_DROPAPPROLE <*role-name*>

In Sybase, a role is granted and revoked, respectively, by executing the stored procedure **SP_ROLE** as follows:

SP_ROLE {'**GRANT**' | '**REVOKE**'}, <*predefined-role*> <*user-name*>

Sybase supports three predefined roles:

1. **SA_ROLE** (system administrator),
2. **SSO_ROLE** (system security officer), and
3. **OPER_ROLE** (operator).

A role is switched on or off, respectively, using **SET ROLE** {**ON** | **OFF**}.

With respect to the grant statement, all reference systems follow the SQL-99 proposal. Even the *grant* option is provided by all systems. In case of the revoke statement, however, there are some minor differences in the various implementations.

In Sybase, the revoke statement is implemented with the cascade semantics, that is, the removal of a privilege implies the removal of all dependent privileges. The cascade semantics is also default in Informix. Oracle, Informix, and Ingres provide the keyword **CASCADE** to explicitly specify this semantics. The restricted semantics prevents from revoking a privilege if there is a dependent privilege.

Applying the revoke statement with the *grant option* revokes the right to grant the granted privilege to others. If additionally the *cascade* option is used, the transitively granted privileges are revoked, too. This option is supported in MSSQL, Sybase, and Ingres.

3.7 Constraints

Although constraints are part of a table definition, we discuss their evolution separately and in more detail due to their importance. As mentioned before, new constraints can be added to a table and existing ones removed from a table. In addition, and in contrast to SQL-99, the checking of constraints can be enabled and disabled in some of the reference systems. These issues can even be combined. For instance, a constraint can be added to a table in the disabled mode or a disabled constraint can be enabled but without verifying it against the current content of the corresponding table. Table 8 gives an overview of the support of constraint evolution constructs in SQL-99 and in the reference systems.

Table 8. Comparison of Constraint Evolution Constructs

			SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
ADD	ENABLE	VALIDATE	(✓)	✓	(✓)	✓	✓	(✓)	(✓)
		VALIDATE + EXCEPTION	—	✓	—	✓	—	—	—
		NOVALIDATE	—	✓	—	—	✓	—	—
	DISABLE		—	✓	—	✓	—	—	—
DROP			—	✓	✓	✓	✓	✓	—
	RESTRICT		✓	—	—	—	—	—	✓
	CASCADE		✓	✓	—	—	—	—	✓
ENABLE	VALIDATE		—	✓	—	✓	—	—	—
	VALIDATE + EXCEPTION		—	✓	—	✓	—	—	—
	NOVALIDATE		—	✓	—	—	✓	—	—
DISABLE			—	✓	—	✓	✓	—	—
	RESTRICT		—	—	—	—	—	—	—
	CASCADE		—	✓	—	—	—	—	—

According to Table 8, Oracle supports the full range of schema evolution constructs that are related to constraints. Another interesting fact is that SQL-99 does not provide any means to enable and disable constraints. The gray shaded fields highlight the default settings. The '(✓)' marked fields state that the concept is supported implicitly. For instance, the *add* and *drop* clauses 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. In the following we will discuss the different constructs in more detail.

When a constraint is added to a table using the statement

ALTER TABLE <*table-name*> **ADD** <*table-constraint*>

the newly added constraint is enabled and validated by default. *Enabled* means that future modifications of the content of the table will be verified against this constraint (unless it disabled in the meanwhile). *Validated* means that the content of table is verified against the constraint when the latter is added to the table.

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

Oracle: **ALTER TABLE** <*table-name*> **ADD** <*table-constraint*> **EXCEPTIONS INTO** <*table-name*>

Informix: **ALTER TABLE** <*table-name*> **ADD** <*table-constraint*> **FILTERING**

The rows that do not satisfy the newly added constraint are removed from the table to an exception/diagnostic table, which can be named explicitly in Oracle. Both systems also allow to add a disabled constraint to a table. By default, such a constraint is not validated when it is added to the table. A disabled constraint is specified as follows:

Oracle: **ALTER TABLE** <*table-name*> **ADD** <*table-constraint*> **DISABLE**

Informix: **ALTER TABLE** <*table-name*> **ADD** <*table-constraint*> **DISABLED**

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

Oracle: **ALTER TABLE** <*table-name*> **ADD** <*table-constraint*> **NOVALIDATE**

MSSQL: **ALTER TABLE** <*table-name*> **WITH NOCHECK ADD** <*table-constraint*>

In all reference systems, a constraint can be dropped from a table as follows:

ALTER TABLE <*table-name*> **DROP** <*constraint-name*>

Some of the reference systems support the specification of *drop* option, which is either **RESTRICT** or **CASCADE**. **RESTRICT** disallows the removal of the constraint if there is another constraint that depends on the constraint to be dropped. If **CASCADE** is specified, the constraint is dropped together with all its depending 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 it is **CASCADE** in DB2.

A disabled constraint can be enabled in the validate mode as follows:[4]

Oracle: **ALTER TABLE** <*table-name*> **ENABLE** <*constraint*>

Informix: **SET CONSTRAINTS** <*constraint-name*> **ENABLED**

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

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

Oracle: **ALTER TABLE** <*table-name*> **ENABLE NOVALIDATE** <*constraint*>

MSSQL: **ALTER TABLE** <*table-name*> **CHECK** <*constraint*>

An enabled constraint can be disabled as follows:

Oracle: **ALTER TABLE** <*table-name*> **DISABLE** <*constraint*>

Informix: **SET CONSTRAINTS** <*constraint-name*> **DISABLED**

MSSQL: **ALTER TABLE** <*table-name*> **NOCHECK** <*constraint*>

The statements are executed in all three systems with the restrict semantics. Cascaded disabling of constraints is only supported by Oracle. For that, the keyword **CASCADE** has to be attached to the *disable* clause.

3.8 Renaming Schema Elements

In the following, we present a useful schema evolution operation that is already implemented in some of the reference systems, although it is not included in SQL-99.

The renaming of a table is supported by Oracle, DB2, Informix, MSSQL, and Sybase. In Oracle, DB2, and Informix, the syntax of the rename statement is as follows:

RENAME TABLE <*old-table-name*> **TO** <*new-table-name*>

[4] Here, <*constraint*> stands for one of the following: **CONSTRAINT** <*constraint-name*>, **PRIMARY KEY**, or **UNIQUE** (<*column-list*>).

Oracle additionally provides the following alternative way to change the name of a table:

ALTER TABLE <*old-table-name*> **RENAME TO** <*new-table-name*>

Informix even allows to rename a particular column of a table applying the rename command with the following syntax:

RENAME COLUMN <*table-name*>**.** <*old-col-name*> **TO** <*new-col-name*>

MSSQL and Sybase provide a predefined stored procedure which is executed with the following parameters to rename a schema element:

SP_RENAME <*old-name*>, <*new-name*>

This procedure is applicable to names that refer to tables, columns, defaults, constraints, rules, triggers, views, and distinct types.

Note that renaming a schema element may also effect dependent schema elements. Oracle, for instance, automatically transfers the new name of a table to all dependent constraints, indexes, and privileges, while it invalidates the dependent views, functions, procedures, and triggers. DB2 applies a more strict strategy. The renaming of a table is disallowed if the table contains a check or referential constraint or there is a dependent view, trigger, function, procedure, or another table with a dependent constraint or reference column. If there is no such a dependency, the renaming is performed by updating the schema catalog and transferring the new name to all dependent indexes and privileges. In Informix, in contrast, the renaming is completely transparent, that is, the new name is transferred to the schema catalog as well as to all dependent schema elements.

4 Some Final Remarks

In this paper, we presented and compared the way schema evolution is supported in SQL-99 and in commercially leading (object-)relational database management systems. We will close this paper with a few remarks on some open issues and schema evolution operations that are available neither in SQL-99 nor in one of the reference systems.

An important open issue concerns the consistency of a schema after performing a schema evolution operation. This issue includes the question whether or not a schema definition as a whole is syntactically and semantically (logically) correct. Considering the current implementations of commercial database management systems, we can state that all systems perform syntactic checking. They, for instance, check whether a foreign key definition is correct in the sense that the names and the data types of the referencing and the referenced columns match. However, none of the systems perform (advanced) semantic checking. Suppose there is a table on which the check constraint **CHECK** (y > 0) is defined. Unfortunately, all reference systems accept an alter table statement that adds a new

(obviously contradicting) constraint of the form **CHECK** (y < 0). In fact, the reference systems do not provide any support for detecting inconsistent specifications implied by check constraints. Interestingly, efficient consistency checking procedures for important and often used kinds of constraints are provided, for instance, in [Ull89,SKN89,GSW96a,GSW96b]. Since the knowledge about the consistency problem and its solutions is highly important for a good design and correct evolution of a database, database designers and administrators have be aware of this problem. In object-relational database systems, the problem of inconsistent constraints becomes even more prominent because constraints are implicitly defined for all subtables of a table. In other words, there are constraints that are valid for a table on which they originally were not defined. So it becomes much harder to design, implement, and maintain a semantically correct database (schema).

Now turn the focus on schema evolution in object-oriented databases. Considering the research in this field, for instance, [BKKK87,Ngu89,TK90,SZ90, Bra93] [ABDS94,ST94,RR95,FMZ+95,Bel96,PÖ97], some nice schema evolution operations could be exploited for object-relational databases. For instance, a prominent schema evolution operation in an object-oriented database is the restructuring of a type or table hierarchy. Existing types or tables can be linked via a subtype or subtable, respectively. Such schema evolution operations are not supported by any current object-relational system. One could think about including statements of the forms

SET <*subtype-name*> **UNDER** <*supertype-name*>
SET <*subtable-name*> **UNDER** <*supertable-name*>

or

ALTER TYPE <*subtype-name*> **ADD UNDER** <*supertype-name*>
ALTER TABLE <*subtype-name*> **ADD UNDER** <*supertype-name*>

into the standard as well as commercial systems. Such statements would help to easily set existing types (tables) into a subtype (subtable) relationship. The inverse statements to drop a subtype or subtable relationship could looks as follows:

ALTER TYPE <*subtype-name*> **DROP UNDER** <*supertype-name*>
ALTER TABLE <*subtable-name*> **DROP UNDER** <*supertable-name*>

One might also think about altering a subtype or subtable relationship by redirecting the link to another subtype or subtable, respectively. Another useful schema evolution construct could be the removal of a subtable from a table hierarchy without removing all its subtables. Instead these subtables could be directly linked to the supertable of the dropped table. An inline transformation [SCR01] can be used to flatten a column that is based on a structured type. Such a structured column is substituted by a set of columns which originally were the fields of that structured column.

The list of potentially useful schema evolution operations could be supplemented easily. Therefore, we close this paper by expressing our hope that the

next versions of the SQL standard and in particular of the commercial database systems will provide some more advanced schema evolution language constructs, which are hopefully embedded in a more clear and rigorously developed object-relational model.

Acknowledgments. Thanks to Kerstin Schwarz for useful remarks on a preliminary version of this paper.

References

[ABDS94] E. Amiel, M.-J. Bellosta, E. Dujardin, and E. Simon. Supporting Exceptions to Behavioral Schema Consistency to Ease Schema Evolution in OODBMS. In J. B. Bocca, Matthias Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Data Bases, VLDB'94, Santiago, Chile, September 12–15, 1994*, pages 108–119. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[Bel96] Z. Bellahsene. A View Mechanism for Schema Evolution in Object-Oriented DBMS. In R. Morrison and J. B. Keane, editors, *Advances in Databases: 14th British National Conf. on Databases, BNCOD 14, Edinburgh, UK, July 1996*, Lecture Notes in Computer Science, Vol. 1094, pages 18–34. Springer-Verlag, Berlin, 1996.

[BKKK87] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In U. Dayal and I. Traiger, editors, *Proc. of the 1987 ACM SIGMOD Int. Conf. on Management of Data, San Franscisco, CA*, ACM SIGMOD Record, Vol. 16, No. 3, pages 311–322, ACM Press, 1987.

[Bra93] S. E. Bratsberg. *Evolution and Integration of Classes in Object-Oriented Databases*. Dissertation, The Norwegian Institute of Technology, University of Trondheim, June 1993.

[FMZ+95] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O_2 Object Database System. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proc. of the 21st Int. Conf. on Very Large Data Bases, VLDB'95, Zürich, Switzerland, September 11–15, 1995*, pages 170–182. Morgan Kaufmann Publishers, San Francisco, CA, 1995.

[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.

[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.

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

[Ngu89] G. T. Nguyen. Schema Evolution in Object-Oriented Database Systems. *Data & Knowledge Engineering*, 4(1):43–67, July 1989.

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

[PÖ97] R. J. Peters and M. T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transactions on Database Systems*, 22(1):75–114, March 1997.

[RR95] Y.-G. Ra and E. A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. In P. S. Yu and A. L. P. Chen, editors, *Proc. of the 11th IEEE Int. Conf. on Data Engineering, ICDE'95*, pages 165–172. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[SCR01] H. Su, K. T. Claypool, and E. A. Rundensteiner. Extending the Object Query Language for Transparent Metadata Access. In H. Balsters, B. De Brock, and S. Conrad, editors, *Database Schema Evolution and Meta-Modeling: 9th International Workshop on Foundations of Models and Languages for Data and Objects (FOMLADO/DEMM 2000)*, Lecture Notes in Computer Science Vol. 2065, pages 181–200, Springer-Verlag, 2001.

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

[ST94] M. H. Scholl and M. Tresch. Evolution towards, in, and beyond Object Databases. In K. von Luck and H. Marburger, editors, *Management and Processing of Complex Data Structures, Proc. of the 3rd Workshop on Information Systems and Artificial Intelligence, Hamburg, Germany, February/March 1994*, Lecture Notes in Computer Science, Vol. 777, pages 64–82. Springer-Verlag, Berlin, 1994.

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

[SZ90] A. Skarra and S. B. Zdonik. Type Evolution in an Object-Oriented Database. In A. F. Cárdenas and D. McLeod, editors, *Research Foundations in Object-Oriented and Semantic Database Systems*, chapter 6, pages 137–155, Series in Data and Knowledge Base Systems, Prentice Hall, Englewood Cliffs, NJ, 1990.

[TK90] L. Tan and T. Katayama. Meta Operations for Type Management in Object-Oriented Databases: A Lazy Mechanism for Schema Evolution. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases, Proc. of the 1st Int. Conf., DOOD'89, Kyoto, Japan, December, 1989*, pages 241–258. North-Holland, Amsterdam, 1990.

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