SCHEMA LEVEL OBJECTS

A database is not just a bunch of tables, even though that is where most of the work is done. There are stored procedures, user-defined functions, and cursors that the users create. Then there are indexes and other access methods that the user cannot access directly.

This chapter is a very quick overview of some of the schema objects that a user can create. Standard SQL divides the database users into USER and ADMIN roles. These objects require ADMIN privileges to be created, altered, or dropped. Those with USER privileges can invoke them and access the results.

3.1 CREATE SCHEMA Statement

There is a CREATE SCHEMA statement defined in the standards that brings an entire schema into existence all at once. In practice, each product has very different utility programs to allocate physical storage and define a schema. Much of the proprietary syntax is concerned with physical storage allocations.

A schema must have a name and a default character set, usually ASCII or a simple Latin alphabet as defined in the ISO Standards. There is an optional AUTHORIZATION clause that holds a <schema authorization identifier> for access control. After that the schema is a list of schema elements:

```
<schema element> ::=<domain definition> | <table definition> | <view definition>
| <grant statement> | <assertion definition>
| <character set definition>
| <collation definition> | <translation definition>
```

A schema is the skeleton of an SQL database; it defines the structures of the schema objects and the rules under which they operate. The data is the meat on that skeleton.

The only data structure in SQL is the table. Tables can be persistent (base tables), used for working storage (temporary tables), virtual (VIEWs, common table expressions, and derived
Chapter 3  SCHEMA LEVEL OBJECTS

The differences among these types are in implementation, not performance. One advantage of having only one data structure is that the results of all operations are also tables—you never have to convert structures, write special operators, or deal with any irregularity in the language.

The <grant statement> has to do with limiting access by users to only certain schema elements. The <assertion definition> is still not widely implemented yet, but it is like a constraint that applies to the schema as a whole. Finally, the <character set definition>, <collation definition>, and <translation definition> deal with the display of data. We are not really concerned with any of these schema objects; they are usually set in place by the DBA (database administrator) for the users and we mere programmers do not get to change them.

3.1.1 CREATE TABLE and CREATE VIEW Statements

Since tables and views are the basic unit of work in SQL, they have their own chapters.

3.2 CREATE PROCEDURE, CREATE FUNCTION, and CREATE TRIGGER

Procedural construct statements put modules of procedural code written in SQL/PSM or other languages into the database. They can be invoked as needed. These constructs get their own chapters.

3.3 CREATE DOMAIN Statement

The DOMAIN is a schema element in Standard SQL that allows you to declare an in-line macro that will allow you to put a commonly used column definition in one place in the schema. The syntax is:

```
<domain definition> ::= CREATE DOMAIN <domain name> [AS] <data type>
                        [<default clause>]
                        [<domain constraint>]
                        [<collate clause>]

<domain constraint> ::= [<constraint name definition>]
                      <check constraint definition> [<constraint attributes>]

<alter domain statement> ::= ALTER DOMAIN <domain name> <alter domain action>

<alter domain action> ::= <set domain default clause>
```
It is important to note that a DOMAIN has to be defined with a basic data type and not with other DOMAINS. Once declared, a DOMAIN can be used in place of a data type declaration on a column.

The CHECK() clause is where you can put the code for validating data items with check digits, ranges, lists, and other conditions. Here is a skeleton for US State codes:

```sql
CREATE DOMAIN StateCode AS CHAR(2)
DEFAULT '??'
CONSTRAINT valid_state_code
CHECK (VALUE IN ('AL', 'AK', 'AZ', . . .));
```

Since the DOMAIN is in one place, you do not have to worry about getting the correct data everywhere you define a column from this domain. If you did not have a DOMAIN clause, then you have to replicate the CHECK() clause in multiple tables in the database. The ALTER DOMAIN and DROP DOMAIN statements explain themselves.

### 3.4 CREATE SEQUENCE

Sequences are generators that produce a sequence of values each time they are invoked. You call on them like a function and get the next value in the sequence.

In my earlier books, I used the table “Sequence” for a set of integers from 1 to (n). Since it is now a reserved word, I have switched to “Series” in this book. The syntax looks like this:

```sql
CREATE SEQUENCE <seq name> AS <data type>
START WITH <value>
INCREMENT BY <value>
[MAXVALUE <value>]
[MINVALUE <value>]
[NO] CYCLE;
```

To get a value from it, this expression is used wherever it is a legal data type.

```sql
NEXT VALUE FOR <seq name>
```

If a sequence needs to be reset, you use this statement to change the optional clauses or to restart the cycle.

```sql
ALTER SEQUENCE <seq name>
RESTART WITH <value>; -- begin over
To remove the sequence, use the obvious statement:

```
DROP SEQUENCE <seq name>;
```

Even when this feature becomes widely available, it should be avoided. It is a nonrelational extension that behaves like a sequential file or procedural function rather than in a set-oriented manner. You can currently find it in Oracle, DB2, Postgres, and Mimer products.

### 3.5 CREATE ASSERTION

In Standard SQL, the CREATE ASSERTION allows you to apply a constraint on the tables within a schema but not have the constraint attached to any particular table. The syntax is:

```
<assertion definition> ::= 
    CREATE ASSERTION <constraint name> <assertion check> 
    [<constraint attributes>]

<assertion check> ::= 
    CHECK (<search condition>)
```

As you would expect, there is a DROP ASSERTION statement, but no ALTER ASSERTION statement. An assertion can do things that a CHECK() clause attached to a table cannot do, because it is outside of the tables involved. A CHECK() constraint is always TRUE if the table is empty.

For example, it is very hard to make a rule that the total number of employees in the company must be equal to the total number of employees in all the health plan tables.

```
CREATE ASSERTION Total_Health_Coverage 
    CHECK (SELECT COUNT(*) FROM Personnel) = 
    (SELECT COUNT(*) FROM HealthPlan_1) 
    + (SELECT COUNT(*) FROM HealthPlan_2) 
    + (SELECT COUNT(*) FROM HealthPlan_3);
```

Since the CREATE ASSERTION is global to the schema, table check constraint names are also global to the schema and not local to the table where they are declared.

### 3.5.1 Using VIEWs for Schema Level Constraints

Until you can get the CREATE ASSERTION, you have to use procedures and triggers to get the same effects. Consider a schema for a chain of stores that has three tables, thus:

```
CREATE TABLE Stores
    (store_nbr INTEGER NOT NULL PRIMARY KEY,
```
store_name CHAR(35) NOT NULL,
..);

CREATE TABLE Personnel
(emp_id CHAR(9) NOT NULL PRIMARY KEY,
last_name CHAR(15) NOT NULL,
first_name CHAR(15) NOT NULL,
..);

The first two explain themselves. The third table shows the relationship between stores and personnel, namely who is assigned to what job at which store and when this happened. Thus:

CREATE TABLE JobAssignments
(store_nbr INTEGER NOT NULL
    REFERENCES Stores (store_nbr)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
emp_id CHAR(9) NOT NULL PRIMARY KEY
    REFERENCES Personnel(emp_id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
end_date TIMESTAMP,
CHECK (start_date <= end_date),
job_type INTEGER DEFAULT 0 NOT NULL -- unassigned = 0
    CHECK (job_type BETWEEN 0 AND 99),
PRIMARY KEY (store_nbr, emp_id, start_date));

Let's invent some job_type codes, such as 0 = 'unassigned',
1 = 'stockboy', and so on, until we get to 99 = 'Store Manager',
and we have a rule that each store has at most one manager. In
Standard SQL you could write a constraint like this:

CREATE ASSERTION ManagerVerification
CHECK (1 <= ALL (SELECT COUNT(*)
    FROM JobAssignments
    WHERE job_type = 99
    GROUP BY store_nbr));

This is actually a bit subtler than it looks. If you change the
<~ to ~, then the stores must have exactly one manager if it has
any employees at all.

But as we said, most SQL product still do not allow CHECK() constraints that apply to the table as a whole, nor do they support
the scheme level CREATE ASSERTION statement.

So, how to do this? You might use a trigger, which will involve proprietary, procedural code. In spite of the SQL/PSM Standard,
most vendors implement very different trigger models and use
their proprietary 4GL language in the body of the trigger.
We need a set of TRIGGERs that validates the state of the table after each INSERT and UPDATE operation. If we DELETE an employee, this will not create more than one manager per store. The skeleton for these triggers would be something like this.

```
CREATE TRIGGER CheckManagers
AFTER UPDATE ON JobAssignments -- same for INSERT
IF 1 <= ALL (SELECT COUNT(*)
    FROM JobAssignments
    WHERE job_type = 99
    GROUP BY store_nbr)
THEN ROLLBACK;
ELSE COMMIT;
END IF;
```

But being a fanatic, I want a pure SQL solution that is declarative within the limits of most current SQL products.
Let’s create two tables. This first table is a Personnel table for the store managers only and it is keyed on their employee identification numbers. Notice the use of DEFAULT and CHECK() on their job_type to assure that this is really a “managers only” table.

```
CREATE TABLE Job_99_Assignments
(store_nbr INTEGER NOT NULL PRIMARY KEY
    REFERENCES Stores (store_nbr)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
emp_id CHAR(9) NOT NULL
    REFERENCES Personnel (emp_id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
end_date TIMESTAMP,
CHECK (start_date <= end_date),
job_type INTEGER DEFAULT 99 NOT NULL
    CHECK (job_type = 99));
```

This second table is a Personnel table for employees who are not store manager and it is also keyed on employee identification numbers. Notice the use of DEFAULT for a starting position of unassigned and CHECK() on their job_type to assure that this is really a No managers allowed table.

```
CREATE TABLE Job_not99_Assignments
(store_nbr INTEGER NOT NULL
    REFERENCES Stores (store_nbr)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
emp_id CHAR(9) NOT NULL PRIMARY KEY
    REFERENCES Personnel (emp_id)
```
ON UPDATE CASCADE
ON DELETE CASCADE,
start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
end_date TIMESTAMP,
CHECK (start_date <= end_date),
job_type INTEGER DEFAULT 0 NOT NULL
   CHECK (job_type BETWEEN 0 AND 98) -- no 99 code
);

From these two tables, build this UNION-ed view of all the job assignments in the entire company and show that to users.

CREATE VIEW JobAssignments (store_nbr, emp_id, start_date,
   end_date, job_type)
AS
(SELECT store_nbr, emp_id, start_date, end_date, job_type
   FROM Job_not99_Assignments
UNION ALL
SELECT store_nbr, emp_id, start_date, end_date, job_type
   FROM Job_99_Assignments)

The key and job_type constraints in each table working together will guarantee at most one manager per store. The next step is to add INSTEAD OF triggers to the VIEW or write stored procedures, so that the users can insert, update, and delete from it easily. A simple stored procedure, without error handling or input validation, would be:

CREATE PROCEDURE InsertJobAssignments
   (IN store_nbr INTEGER, IN new_emp_id CHAR(9), IN new_start_date DATE, IN new_end_date DATE, IN new_job_type INTEGER)
LANGUAGE SQL
IF new_job_type <> 99
THEN INSERT INTO Job_not99_Assignments
   VALUES (store_nbr, new_emp_id, new_start_date,
   new_end_date, new_job_type);
ELSE INSERT INTO Job_99_Assignments
   VALUES (store_nbr, new_emp_id, new_start_date,
   new_end_date, new_job_type);
END IF;

Likewise, a procedure to terminate an employee:

CREATE PROCEDURE FireEmployee (IN new_emp_id CHAR(9))
LANGUAGE SQL
IF new_job_type <> 99
THEN DELETE FROM Job_not99_Assignments
   WHERE emp_id = new_emp_id;
ELSE DELETE FROM Job_99_Assignments
   WHERE emp_id = new_emp_id;
END IF;
If a developer attempts to change the Job_Assignments VIEW directly with an INSERT, UPDATE, or DELETE, they will get an error message telling them that the VIEW is not updatable because it contains a UNION operation. That is a good thing in one way because we can force them to use only the stored procedures.

Again, this is an exercise in programming a solution within certain limits. The TRIGGER is probably going give better performance than the VIEW.

### 3.5.2 Using PRIMARY KEYs and ASSERTIONs for Constraints

Let's do another version of the “stores and personnel” problem given in a previous section.

```sql
CREATE TABLE JobAssignments
(emp_id CHAR(9) NOT NULL PRIMARY KEY -- nobody is in two Stores
REFERENCES Personnel (emp_id)
ON UPDATE CASCADE
ON DELETE CASCADE,
store_nbr INTEGER NOT NULL
REFERENCES Stores (store_nbr)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

The key on the SSN will assure that nobody is at two stores and that a store can have many employees assigned to it. Ideally, you would want a constraint to check that each employee does have a branch assignment.

The first attempt is usually something like this:

```sql
CREATE ASSERTION Nobody_Unassigned
CHECK (NOT EXISTS
(SELECT *
FROM Personnel AS P
LEFT OUTER JOIN
JobAssignments AS J
ON P.emp_id = J.emp_id
WHERE J.emp_id IS NULL
AND P.emp_id
IN (SELECT emp_id FROM JobAssignments
UNION
SELECT emp_id FROM Personnel)));
```

However, that is overkill and does not prevent an employee from being at more than one store. There are probably indexes on the SSN values in both Personnel and JobAssignments, so getting a COUNT() function should be cheap. This assertion will also work.
CREATE ASSERTION Everyone_assigned_one_store
CHECK ((SELECT COUNT(emp_id) FROM JobAssignments)
    = (SELECT COUNT(emp_id) FROM Personnel));

This is a surprise to people at first because they expect to see a JOIN to do the one-to-one mapping between personnel and job assignments. But the PK-FK requirement provides that for you. Any unassigned employee will make the Personnel table bigger than the JobAssignments table, and an employee in JobAssignments must have a match in personnel. The good optimizers extract things like that as predicates and use them, which is why we want Declarative Referential Integrity (DRI) instead of triggers and application side logic.

You will need to have a stored procedure that inserts into both tables as a single transaction. The updates and deletes will cascade and clean up the job assignments.

Let’s change the specs a bit and allow employees to work at more than one store. If we want to have an employee in multiple Stores, we could change the keys on JobAssignments, thus.

CREATE TABLE JobAssignments
(emp_id CHAR(9) NOT NULL
  REFERENCES Personnel (emp_id)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
store_nbr INTEGER NOT NULL
  REFERENCES Stores (store_nbr)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
PRIMARY KEY (emp_id, store_nbr));

Then use a COUNT(DISTINCT ..) in the assertion:

CREATE ASSERTION Everyone_assigned_at_least_once
CHECK ((SELECT COUNT(DISTINCT emp_id) FROM JobAssignments)
    = (SELECT COUNT(emp_id) FROM Personnel));

You must be aware that the uniqueness constraints and assertions work together; a change in one or both of them can also change this rule.

3.6 Character Set Related Constructs

There are several schema level constructs for handling characters. You can create a named set of characters for various languages or special purposes, define one or more collation sequences for them, and translate one set into another.
Today, the Unicode Standards and vendor features are commonly used. Most of the characters actually used have Unicode names and collations defined already. For example, SQL text is written in Latin-1, as defined by ISO 8859-1. This is the set used for HTML, consisting of 191 characters from the Latin alphabet. This the most commonly used character set in the Americas, Western Europe, Oceania, Africa, and for standard romanizations of East-Asian languages.

Since 1991, the Unicode Consortium has been working with ISO and IEC to develop the Unicode Standard and ISO/IEC 10646: the Universal Character Set (UCS) in tandem. Unicode and ISO/IEC 10646 currently assign about 100,000 characters to a code space consisting of over a million code points, and they define several standard encodings that are capable of representing every available code point. The standard encodings of Unicode and the UCS use sequences of one to four 8-bit code values (UTF-8), sequences of one or two 16-bit code values (UTF-16), or one 32-bit code value (UTF-32 or UCS-4). There is also an older encoding that uses one 16-bit code value (UCS-2), capable of representing one-seventeenth of the available code points. Of these encoding forms, only UTF-8’s byte sequences are in a fixed order; the others are subject to platform-dependent byte ordering issues that may be addressed via special codes or indicated via out-of-band means.

### 3.6.1 CREATE CHARACTER SET

You will not find this syntax in many SQLs. The vendors will default to a system level character set based on the local language settings.

```sql
<character set definition> ::= CREATE CHARACTER SET <character set name> [AS] <character set source> [<collate clause>]
<character set source> ::= GET <character set specification>
<collate clause> ::= NO PAD | PAD SPACE
```

The `<collate clause>` usually is defaulted also, but you can use named collations.

### 3.6.2 CREATE COLLATION

```sql
<collation definition> ::= CREATE COLLATION <collation name> FOR <character set specification> FROM <existing collation name> [<pad characteristic>]
<pad characteristic> ::= NO PAD | PAD SPACE
```
The \texttt{pad characteristic} option has to do with how strings will be compared to each other. If the collation for the comparison has the NO PAD characteristic and the shorter value is equal to some prefix of the longer value, then the shorter value is considered less than the longer value. If the collation for the comparison has the PAD SPACE characteristic, for the purposes of the comparison, the shorter value is effectively extended to the length of the longer by concatenation of \texttt{space}s on the right. SQL normally pads a shorter string with spaces on the end and then matches them, letter for letter, position by position.

### 3.6.3 CREATE TRANSLATION

This statement defines how one character set can be mapped into another character set. The important part is that it gives this mapping a name.

\begin{verbatim}
<transliteration definition> ::= 
CREATE TRANSLATION <transliteration name>
   FOR <source character set specification>
   TO <target character set specification>
   FROM <transliteration source>

<source character set specification> ::= 
<character set specification>

<translation source> ::= 
<existing transliteration name> | <transliteration routine>

<existing transliteration name> ::= <transliteration name>

<transliteration routine> ::= <specific routine designator>
\end{verbatim}

Notice that I can use a simple mapping, which will behave much like a bunch of nested REPLACE() function calls, or use a routine that can do some computations. The reason that having a name for these translations is that I can use them in the TRANSLATE() function instead of that bunch of nested REPLACE() function calls. The syntax is simple:

\begin{verbatim}
TRANSLATE (<character value expression> USING 
   <transliteration name>)
\end{verbatim}

DB2 and other implementations generalize TRANSLATE() to allow for target and replacement strings, so that you can do a lot of edit work in a single expression. We will get to that when we get to string functions.