CHAPTER 21

Creating Additional Structural Elements

Abstract

This chapter discusses the creation and use of SQL to manage database elements not covered in previous chapters. Topics include views, temporary tables, common table expressions, and indexes. The discussion of temporary tables covers creating the tables, loading them with data, and disposing of them when they are no longer needed. The chapter not only looks at how to create views, but also includes a discussion of deciding which views are needed.

Keywords

SQL
SQL views
SQL temporary tables
SQL common table expressions
CTEs
SQL CTEs
SQL indexes
CREATE INDEX

Views

As you first read in Chapter 5, views provide a way to give users a specific portion of a larger schema with which they can work. Before you actually can create views, there are two things you should consider: which views you really need and whether the views can be used for updating data.

Deciding Which Views to Create

Views take up very little space in a database, occupying only a few rows in a data dictionary table. That being the case, you can feel free to create views as needed.

A typical database might include the following views:

- One view for every base table that is exactly the same as the base table, but with a different name. Then, you prevent end users from seeing the base tables and do not tell the end users the table names; you give end users access only to the views. This makes it harder for end users to attempt to gain access to the stored tables because they do not know their names. However, as you will see in the next section, it is essential for updating that there be views that do match the base tables.
- One view for each primary key–foreign key relationship over which you join frequently. If the tables are large, the actual syntax of the statement may include structures for avoiding the join operation, but still combining the tables.
- One view for each complex query that you issue frequently.
- Views as needed to restrict user access to specific columns and rows. For example, you might recreate a view for a receptionist that shows employee office numbers and telephone extensions, but leaves off home address, telephone number, and salary.

View Updatability Issues

A database query can apply any operations supported by its DBMS's query language to a view, just as it can to base tables. However, using views for updates is a much more complicated issue. Given that views exist only in main memory, any updates made to a view must be stored in the underlying base tables if the updates are to have any effect on the database.

Not every view is updatable, however. Although the rules for view updatability vary from one DBMS to another, you will find that many DBMSs share the following restrictions:[1]

- A view must be created from one base table or view, or if the view uses joined tables, only one of the underlying base tables can be updated.
- If the source of the view is another view, then the source view must also adhere to the rules for updatability.
- A view must be created from only one query. Two or more queries cannot be assembled into a single view table using operations such as union.
- The view must include the primary key columns of the base table.
- The view must include all columns specified as not null (columns requiring mandatory values).
- The view must not include any groups of data. It must include the original rows of data from the base table, rather than rows based on values common to groups of data.
- The view must not remove duplicate rows.

Creating Views

Creating Views

To create a view whose columns have the same name as the columns in the base tables from which it is derived, you give the view a name and include the SQL query that defines its contents:

```
CREATE VIEW view_name AS
    SELECT …
```

For example, if *Antique Opticals* wanted to create a view that included actions films, the SQL is written

```
CREATE VIEW action_films AS
    SELECT item_numb, title
    FROM item
    WHERE genre = 'action';
```

If you want to rename the columns in the view, you include the view's column names in the CREATE VIEW statement:

```
CREATE VIEW action_films (identifier, name)
    AS
    SELECT item_numb, title
    FROM item
    WHERE genre = 'action';
```

The preceding statement will produce a view with two columns named *identifier* and *name*. Note that if you want to change even one column name, you must include *all* the column names in the parentheses following the view name. The DBMS will match the columns following SELECT with the view column names by their positions in the list.

Views can be created from any SQL query, including those that perform joins, unions, and grouping. For example, to simplify looking at customers and their order totals, *Antique Opticals* might create a view like the following:

```
CREATE VIEW sales_summary AS
    SELECT customer_numb, order.order_numb, order.order_date,
      SUM (selling_price)
    FROM order_line JOIN order
    GROUP BY customer_number, orders.order_date,
      orders.order_numb;
```

The view table will then contain grouped data along with a computed column.

Temporary Tables

A temporary table is a base table that is not stored in the database, but instead exists only while the database session in which it was created is

A temporary table is a base table that is not stored in the database, but instead exists only while the database session in which it was created is active. At first glance, this may sound like a view, but views and temporary tables are somewhat different:

- A view exists only for a single query. Each time you use the name of a view, its table is recreated from existing data.
- A temporary table exists for the entire database session in which it was created.
- A view is automatically populated with the data retrieved by the query that defines it.
- You must add data to a temporary table with SQL INSERT commands.
- Only views that meet the criteria for view updatability can be used for data modifications.
- Because temporary tables are base tables, all of them can be updated. (They can be updated, but keep in mind that any modifications you make won't be retained in the database.)
- Because the contents of a view are generated each time the view's name is used, a view's data are always current.
- The data in a temporary table reflect the state of the database at the time the table was loaded with data. If the data from which the temporary table was loaded are modified after the temporary table has received its data, then the contents of the temporary table may be out of sync with other parts of the database.

If the contents of a temporary table become outdated when source data change, why use a temporary table at all? Wouldn't it be better simply to use a view whose contents are continually regenerated? The answer lies in performance. It takes processing time to create a view table. If you are going to use data only once during a database session, then a view will actually perform better than a temporary table because you don't need to create a structure for it. However, if you are going to be using the data repeatedly during a session, then a temporary table provides better performance because it needs to be created only once. The decision therefore results in a trade-off: Using a view repeatedly takes more time, but provides continuously updated data; using a temporary table repeatedly saves time, but you run the risk that the table's contents may be out of date.

Creating Temporary Tables

Creating a temporary table is very similar to creating a permanent base table. You do, however, need to decide on the *scope* of the table. A temporary table may be *global*, in which case it is accessible to the entire application program that created it.[2] Alternatively, it can be *local*, in which case it is accessible only to the program module in which it was created.

To create a global temporary table, you add the keywords GLOBAL TEMPORARY to the CREATE TABLE statement:

```
CREATE GLOBAL TEMPORARY TABLE
    (remainder of CREATE statement)
```

By the same token, you create a local temporary table with

```
CREATE LOCAL TEMPORARY TABLE
    (remainder of CREATE statement)
```

For example, if *Antique Opticals* was going to use the order summary information repeatedly, it might create the following temporary table instead of using a view:

```
CREATE GLOBAL TEMPORARY TABLE order_summary
    (customer_numb int,
    order_numb int,
    order_date date,
    order_total numeric (6,2),
    PRIMARY KEY (customer_numb, order_numb);
```

Loading Temporary Tables with Data

To place data in a temporary table, you use one or more SQL INSERT statements. For example, to load the order summary table created in the preceding section, you could type

```
INSERT INTO order_summary
    SELECT customer_numb, order.order_numb, order.order_date,
      SUM (selling_price)
    FROM order_line JOIN order
    GROUP BY customer_number, orders.order_date,
      orders.order_numb
```

You can now query and manipulate the *order_summary* table, just as you would a permanent base table.

Disposition of Temporary Table Rows

When you write embedded SQL (SQL statements coded as part of a program written in a high-level language such as C++ or Java), you have control over the amount of work that the DBMS considers to be a unit (a *transaction*). Although we will cover transactions in depth in Chapter 22, at this point you need to know that a transaction can end in one of two ways: It can be *committed* (changes made permanent) or it can be *rolled back* (its changes undone).

By default, the rows in a temporary table are purged whenever a transaction is committed. If you want to use the same temporary tables in multiple transactions, however, you can instruct the DBMS to retain the rows by including ON COMMIT PRESERVE ROWS to the end of the table creation statement:

```
CREATE GLOBAL TEMPORARY TABLE order_summary
    (customer_numb int,
    order_numb int,
    order_date date,
    order_total numeric (6,2),
    PRIMARY KEY (customer_numb, order_numb
    ON COMMIT PRESERVE ROWS);
```

Because a rollback returns the database to the state it was in before the transaction begins, a temporary table will also be restored to its previous state (with or without rows).

Common Table Expressions (CTEs)

A *common table expression* (CTE) is yet another way of extracting a subset of a database for use in another query. CTEs are like views in that they generate virtual tables. However, the definition of a CTE is not stored in the database, and it must be used immediately after it is defined.

To get started, let's look at a very simple example. The general format of a simple CTE is

```
WITH CTE_name (columns) AS
      (SELECT_statement_defining_table)
CTE_query
```

For example, a CTE and its query to view all of the rare book store's customers could be written

```
WITH customer_names (first, last) AS
      (SELECT first_name, last_name
      FROM customer)
```

```
        FROM customer)
SELECT *
FROM customer_names;
```

The result is a listing of the first and last names of the customers. This type of structure for a simple query really doesn't buy you much, except that the CTE isn't stored in the database like a view and doesn't require INSERT statements to populate it like a temporary table. However, the major use of CTEs is for *recursive queries*, queries that query themselves. (That may sound a bit circular and it is, intentionally.) The typical application of a recursive query using a CTE is to process hierarchical data, data arranged in a tree structure. It will allow a single query to access every element in the tree or to access subtrees that begin somewhere other than the top of the tree.

As an example, let's create a table that handles the descendants of a single person (in this case, John). As you can see in Figure 21.1, each node in the tree has at most one parent and any number of children. The numbers in the illustration represent the ID of each person.

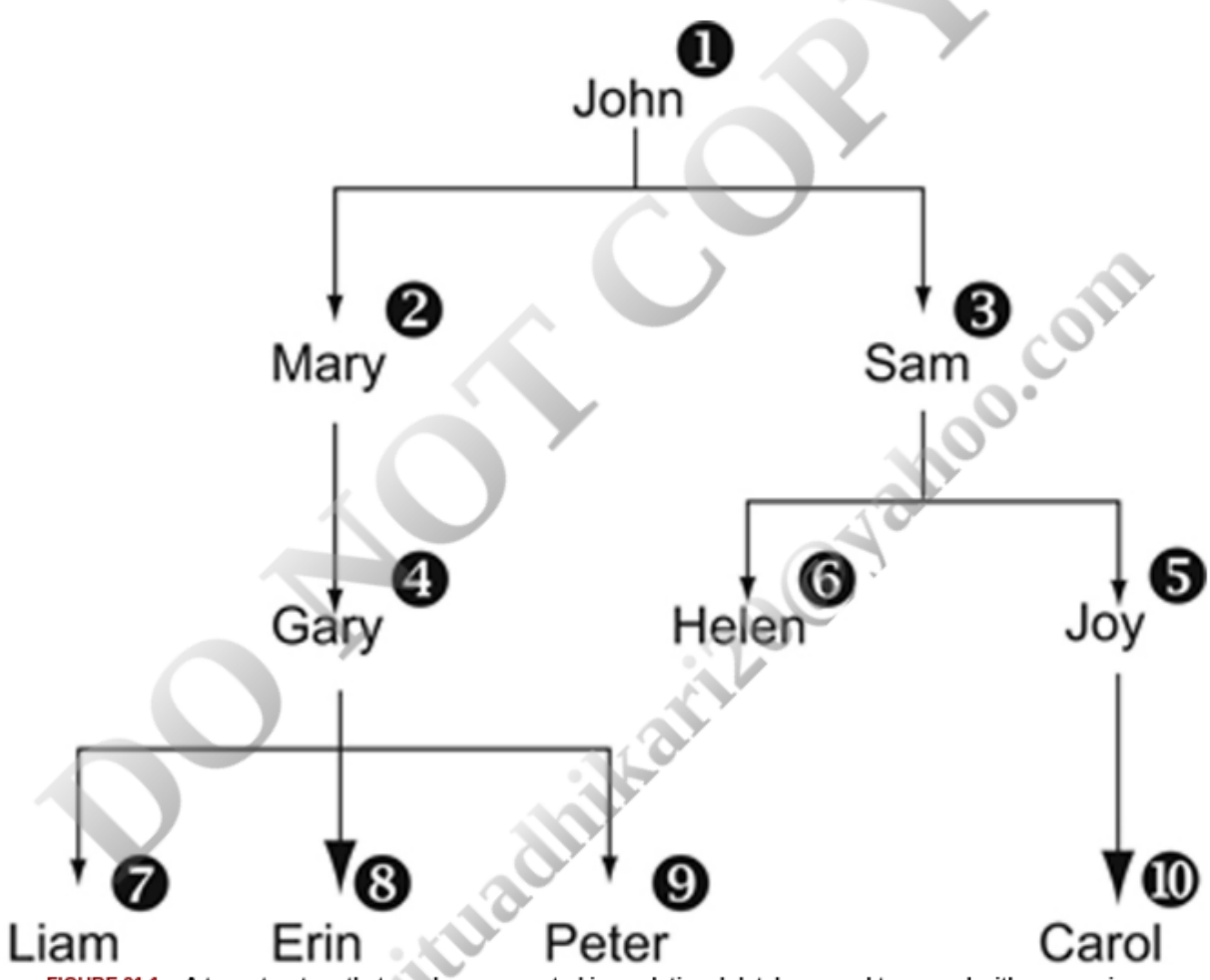

FIGURE 21.1 **A tree structure that can be represented in a relational database, and traversed with a recursive query.**

Relational databases are notoriously bad at handling this type of hierarchically structured data. The typical way to handle it is to create a relation, something like this:

```
genealogy (person_id, parent_id, person_name)
```

Each row in the table represents one node in the tree. For this example, the table is populated with the 10 rows in Figure 21.2. John, the node at the top of the tree, has no parent ID. The *parent_ID* column in the other rows is filled with the person ID of the node above it in the tree. (The order of the rows in the table is irrelevant.)

person_id	parent_id	person_name
1		John
2	1	Mary
3	1	Sam
4	2	Gary
5	3	Joy
6	3	Helen
7	4	Liam
8	4	Erin
9	4	Peter
10	5	Carol

FIGURE 21.2 **Sample data for use with a recursive query.**

We can access every node in the tree by simply accessing every row in the table. However, what can we do if we want to process just the people who are Sam's descendants? There is no easy way with a typical SELECT to do that. However, a CTE used recursively will identify just the rows we want.

The syntax of a recursive query is similar to the simple CTE query, with the addition of the keyword RECURSIVE following WITH. For our particular example, the query will be written:

```
WITH RECURSIVE show_tree AS
      (SELECT
      FROM genealogy
      WHERE person_name = 'Sam'
      UNION ALL
      SELECT g.*
      FROM genealogy AS g, show_tree AS st
      WHERE g.parent_id = st.person_id)
SELECT *
FROM show_tree
ORDER BY person_name;
```

The result is

person_id	parent_id	person_name
10	5	Carol
6	3	Helen
5	3	Joy
3	1	Sam

The query that defines the CTE called *show_tree* has two parts. The first is a simple SELECT that retrieves Sam's row and places it in the result table as well as in an intermediate table that represents the current state of *show_tree*. The second SELECT (below UNION ALL) is the recursive part. It will use the intermediate table in place of *show_tree* each time it executes and add the result of each iteration to the result table. The recursive portion will execute repeatedly until it returns no rows.

Here's how the recursion will work in our example:

1. Join the intermediate result table to *genealogy*. Because the intermediate result table contains just Sam's row, the join will match Helen and Joy.
2. Remove Sam from the intermediate table and insert Helen and Joy.
3. Append Helen and Joy to the result table.
4. Join the intermediate table to *genealogy*. The only match from the join will be Carol. (Helen has no children, and Joy has only one.)
5. Remove Helen and Joy from the intermediate table and insert Carol.
6. Append Carol to the result table.
7. Join the intermediate table to *genealogy*. The result will be no rows, and the recursion stops.

CTEs cannot be reused; the declaration of the CTE isn't saved. Therefore, they don't buy you much for most queries. However, they are enormously useful if you are working with tree-structured data. CTEs and recursion can also be helpful when working with bill of materials data.

Creating Indexes

As you read in Chapter 7, an index is a data structure that provides a fast access path to rows in a table based on the value in one or more columns (the index key). Because an index stores key values in order, the DBMS can use a fast search technique to find the values, rather than being forced to search each row in an unordered table sequentially.

You create indexes with the CREATE INDEX statement:

```
CREATE INDEX index_name
ON table_name (index_key_columns)
```

For example, to create an index on the *title* column in *Antique Opticals item* table, you could use

```
CREATE INDEX item_title_index
ON item (title);
```

By default, the index will allow duplicate entries and keeps the entries in ascending order (alphabetical, numeric, or chronological, whichever is appropriate). To require unique indexes, add the keyword UNIQUE after CREATE:

```
CREATE UNIQUE INDEX item_title_index
ON item (title);
```

To sort in descending order, insert DESC after the column whose sort order you want to change. For example, *Antique Opticals* might want to create an index on the order date in the *order* relation in descending order so that the most recent orders are first: