View in Database

View The dynamic result of one or more relational operations operating on the base relations to produce another relation. A view is a *virtual relation* that does not necessarily exist in the database but can be produced upon request by a particular user, at the time of request.

To the database user, a view appears just like a real table, with a set of named columns and rows of data. However, unlike a base table, a view does not necessarily exist in the database as a stored set of data values. Instead, a view is defined as a query on one or more base tables or views. The DBMS stores the definition of the view in the database. When the DBMS encounters a reference to a view, one approach is to look up this definition and translate the request into an equivalent request against the source tables of the view and then perform the equivalent request.
An alternative approach, called
view materialization, stores the view as a temporary table in the database and maintains the currency of the view as the underlying base tables are updated.

Creating a View (CREATE VIEW)

CREATE VIEW ViewName [(newColumnName [, . . .])]
AS subselect [**WITH** [**CASCADED** | **LOCAL**] **CHECK OPTION**]

A view is defined by specifying an SQL SELECT statement. A name may optionally be assigned to each column in the view. If a list of column names is specified, it must have the same number of items as the number of columns produced by the *subselect*. If the list of column names is omitted, each column in the view takes the name of the corresponding column in the *subselect* statement. The list of column names must be specified if there is any ambiguity in the name for a column. This may occur if the *subselect* includes calculated columns, and the AS subclause has not been used to name such columns, or it produces two columns with identical names as the result of a join.

The *subselect* is known as the **defining query**. If WITH CHECK OPTION is specified, SQL ensures that if a row fails to satisfy the WHERE clause of the defining query of the view, it is not added to the underlying base table of the view

Create a view so that the manager at branch B003 can see the details only for staff who work in his or her branch office.

A horizontal view restricts a user's access to selected rows of one or more tables.

```
CREATE VIEW Manager3Staff
AS SELECT *
    FROM Staff
    WHERE branchNo = 'B003';
```

This creates a view called Manager3Staff with the same column names as the Staff table but containing only those rows where the branch number is B003. (Strictly speaking, the branchNo column is unnecessary and could have been omitted from the definition of the view, as all entries have branchNo = 'B003'.) If we now execute this statement:

```
SELECT * FROM Manager3Staff;
```

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000.00	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000.00	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000.00	B003

Create a view of the staff details at branch B003 that excludes salary information, so that only managers can access the salary details for staff who work at their branch.

A vertical view restricts a user's access to selected columns of one or more tables.

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
    FROM Staff
    WHERE branchNo = 'B003';
```

Note that we could rewrite this statement to use the Manager3Staff view instead of the Staff table, thus:

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
    FROM Manager3Staff;
```

Select * from Staff3

staffNo	fName	lName	position	sex
SG37	Ann	Beech	Assistant	F
SG14	David	Ford	Supervisor	M
SG5	Susan	Brand	Manager	F

Grouped and joined views

Create a view of staff who manage properties for rent, which includes the branch number they work at, their staff number, and the number of properties they manage (see Example 6.27).

```
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, COUNT(*)
    FROM Staff s, PropertyForRent p
    WHERE s.staffNo = p.staffNo
    GROUP BY s.branchNo, s.staffNo;
```

It illustrates the use of a subselect
containing a GROUP BY clause (giving a view called a **grouped view**), and containing multiple tables (giving a view called a **joined view).** One of the most frequent reasons for using views is to simplify multi-table queries. Once a joined view has been defined, we can often use a simple single-table query against the view for queries that would otherwise require a multi-table join. Note that we have to name the columns in the definition of the view because of the use of the unqualified aggregate function COUNT in the subselect.

branchNo	staffNo	cnt
B003	SG14	1
B003	SG37	2
B005	SL41	1
B007	SA9	1

Removing a View (DROP VIEW)

A view is removed from the database with the DROP VIEW statement:

```
DROP VIEW ViewName [RESTRICT | CASCADE]
```

DROP VIEW causes the definition of the view to be deleted from the database. For example, we could remove the Manager3Staff view using the following statement:

```
DROP VIEW Manager3Staff;
```

If CASCADE is specified, DROP VIEW deletes all related dependent objects; in other words, all objects that reference the view. This means that DROP VIEW also deletes any views that are defined on the view being dropped. If RESTRICT is specified and there are any other objects that depend for their existence on the continued existence of the view being dropped, the command is rejected. The default setting is RESTRICT.

View Resolution

Having considered how to create and use views, we now look more closely at how a query on a view is handled. To illustrate the process of **view resolution**, consider the following query, which counts the number of properties managed by each member of staff at branch office B003. This query is based on the StaffPropCnt view of Example 7.5:

```
SELECT staffNo, cnt
FROM StaffPropCnt
WHERE branchNo = 'B003'
ORDER BY staffNo;
```

View resolution merges the example query with the defining query of the StaffPropCnt view as follows:

(1) The view column names in the SELECT list are translated into their corresponding column names in the defining query. This gives:

```
SELECT s.staffNo AS staffNo, COUNT(*) AS cnt
```

(2) View names in the FROM clause are replaced with the corresponding FROM lists of the defining query:

```
FROM Staff s, PropertyForRent p
```

(3) The WHERE clause from the user query is combined with the WHERE clause of the defining query using the logical operator AND, thus:

```
WHERE s.staffNo = p.staffNo AND branchNo = 'B003'
```

(4) The GROUP BY and HAVING clauses are copied from the defining query. In this example, we have only a GROUP BY clause:

```
GROUP BY s.branchNo, s.staffNo
```

(5) Finally, the ORDER BY clause is copied from the user query with the view column name translated into the defining query column name:

```
ORDER BY s.staffNo
```

(6) The final merged query becomes:

```
SELECT s.staffNo AS staffNo, COUNT(*) AS cnt
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo AND branchNo = 'B003'
GROUP BY s.branchNo, s.staffNo
ORDER BY s.staffNo;
```

staffNo	cnt
SG14	1
SG37	2

Restrictions on Views

The ISO standard imposes several important restrictions on the creation and use of views, although there is considerable variation among dialects.

- If a column in the view is based on an aggregate function, then the column may appear only in SELECT and ORDER BY clauses of queries that access the view. In particular, such a column may not be used in a WHERE clause and may not be an argument to an aggregate function in any query based on the view. For example, consider the view StaffPropCnt of Example 7.5, which has a column cnt based on the aggregate function COUNT. The following query would fail:

  ```
  SELECT COUNT(cnt)
  FROM StaffPropCnt;
  ```

  because we are using an aggregate function on the column cnt, which is itself based on an aggregate function. Similarly, the following query would also fail:

  ```
  SELECT *
  FROM StaffPropCnt
  WHERE cnt > 2;
  ```

  because we are using the view column, cnt, derived from an aggregate function, on the left-hand side of a WHERE clause.

- A grouped view may never be joined with a base table or a view. For example, the StaffPropCnt view is a grouped view, so any attempt to join this view with another table or view fails.

View Updatability

All updates to a base table are immediately reflected in all views that encompass that base table. Similarly, we may expect that if a view is updated, the base table(s) will reflect that change. However, consider again the view StaffPropCnt of Example 7.5. Consider what would happen if we tried to insert a record that showed that at branch B003, staff member SG5 manages two properties, using the following insert statement:

```
INSERT INTO StaffPropCnt
VALUES ('B003', 'SG5', 2);
```

We have to insert two records into the PropertyForRent table showing which properties staff member SG5 manages. However, we do not know which properties they are; all we know is that this member of staff manages two properties. In other words, we do not know the corresponding primary key values for the PropertyForRent table. If we change the definition of the view and replace the count with the actual property numbers as follows:

```
CREATE VIEW StaffPropList (branchNo, staffNo, propertyNo)
AS SELECT s.branchNo, s.staffNo, p.propertyNo
   FROM Staff s, PropertyForRent p
   WHERE s.staffNo = p.staffNo;
```

and we try to insert the record:

```
INSERT INTO StaffPropList
VALUES ('B003', 'SG5', 'PG19');
```

there is still a problem with this insertion, because we specified in the definition of the PropertyForRent table that all columns except postcode and staffNo were not allowed to have nulls (see Example 7.1). However, as the StaffPropList view excludes all columns from the PropertyForRent table except the property number, we have no way of providing the remaining nonnull columns with values.

The ISO standard specifies the views that must be updatable in a system that conforms to the standard. The definition given in the ISO standard is that a view is updatable if and only if:

- DISTINCT is not specified; that is, duplicate rows must not be eliminated from the query results.
- Every element in the SELECT list of the defining query is a column name (rather than a constant, expression, or aggregate function) and no column name appears more than once.
- The FROM clause specifies only one table; that is, the view must have a single source table for which the user has the required privileges. If the source table is itself a view, then that view must satisfy these conditions. This, therefore, excludes any views based on a join, union (UNION), intersection (INTERSECT), or difference (EXCEPT).
- The WHERE clause does not include any nested SELECTs that reference the table in the FROM clause.
- There is no GROUP BY or HAVING clause in the defining query.

In addition, every row that is added through the view must not violate the integrity constraints of the base table. For example, if a new row is added through a view, columns that are not included in the view are set to null, but this must not violate a NOT NULL integrity constraint in the base table. The basic concept behind these restrictions is as follows:

Updatable view

For a view to be updatable, the DBMS must be able to trace any row or column back to its row or column in the source table.

WITH CHECK OPTION

Rows exist in a view, because they satisfy the WHERE condition of the defining query. If a row is altered such that it no longer satisfies this condition, then it will

disappear from the view. Similarly, new rows will appear within the view when an insert or update on the view causes them to satisfy the WHERE condition. The rows that enter or leave a view are called **migrating rows**.

Generally, the WITH CHECK OPTION clause of the CREATE VIEW statement prohibits a row from migrating out of the view. The optional qualifiers LOCAL/CASCADED are applicable to view hierarchies, that is, a view that is derived from another view. In this case, if WITH LOCAL CHECK OPTION is specified, then any row insert or update on this view, and on any view directly or indirectly defined on this view, must not cause the row to disappear from the view, unless the row also disappears from the underlying derived view/table. If the WITH CASCADED CHECK OPTION is specified (the default setting), then any row insert or update on this view and on any view directly or indirectly defined on this view must not cause the row to disappear from the view.

This feature is so useful that it can make working with views more attractive than working with the base tables. When an INSERT or UPDATE statement on the view violates the WHERE condition of the defining query, the operation is rejected. This behavior enforces constraints on the database and helps preserve database integrity. The WITH CHECK OPTION can be specified only for an updatable view, as defined in the previous section.

WITH CHECK OPTION

Consider again the view created in Example 7.3:

```
CREATE VIEW Manager3Staff
AS SELECT *
    FROM Staff
    WHERE branchNo = 'B003'
WITH CHECK OPTION;
```

with the virtual table shown in Table 7.3. If we now attempt to update the branch number of one of the rows from B003 to B005, for example:

```
UPDATE Manager3Staff
SET branchNo = 'B005'
WHERE staffNo = 'SG37';
```

then the specification of the WITH CHECK OPTION clause in the definition of the view prevents this from happening, as it would cause the row to migrate from this horizontal view. Similarly, if we attempt to insert the following row through the view:

```
INSERT INTO Manager3Staff
VALUES('SL15', 'Mary', 'Black', 'Assistant', 'F', DATE'1967-06-21', 8000, 'B002');
```

then the specification of WITH CHECK OPTION would prevent the row from being inserted into the underlying Staff table and immediately disappearing from this view (as branch B002 is not part of the view).

What is View Materialization and discuss its advantage?

END