Pearson New International Edition

Database Systems
The Complete Book

Garcia-Molina Ullman Widom

Second Edition

ALWAYS LEARNING

PEARSON

Pearson New International Edition

Database Systems
The Complete Book
Garcia-Molina Ullman Widom
Second Edition

PEARSON

modification of the view into an equivalent modification on a base table, and the modification can be done to the base table instead. In addition, "instead-of" triggers can be used to turn a view modification into modifications of base tables. In that way, the programmer can force whatever interpretation of a view modification is desired.

2.1 View Removal

An extreme modification of a view is to delete it altogether. This modification may be done whether or not the view is updatable. A typical `DROP` statement is

```
DROP VIEW ParamountMovies;
```

Note that this statement deletes the definition of the view, so we may no longer make queries or issue modification commands involving this view. However dropping the view does not affect any tuples of the underlying relation `Movies`. In contrast,

```
DROP TABLE Movies
```

would not only make the `Movies` table go away. It would also make the view `ParamountMovies` unusable, since a query that used it would indirectly refer to the nonexistent relation `Movies`.

2.2 Updatable Views

SQL provides a formal definition of when modifications to a view are permitted. The SQL rules are complex, but roughly, they permit modifications on views that are defined by selecting (using `SELECT`, not `SELECT DISTINCT`) some attributes from one relation R (which may itself be an updatable view). Two important technical points:

- The `WHERE` clause must not involve R in a subquery.
- The `FROM` clause can only consist of one occurrence of R and no other relation.
- The list in the `SELECT` clause must include enough attributes that for every tuple inserted into the view, we can fill the other attributes out with `NULL` values or the proper default. For example, it is not permitted to project out an attribute that is declared `NOT NULL` and has no default.

An insertion on the view can be applied directly to the underlying relation R. The only nuance is that we need to specify that the attributes in the `SELECT` clause of the view are the only ones for which values are supplied.

Example 5: Suppose we insert into view `ParamountMovies` of Example 1 a tuple like:

```
INSERT INTO ParamountMovies
VALUES('Star Trek', 1979);
```

View `ParamountMovies` meets the SQL updatability conditions, since the view asks only for some components of some tuples of one base table:

```
Movies(title, year, length, genre, studioName, producerC#)
```

The insertion on `ParamountMovies` is executed as if it were the same insertion on `Movies`:

```
INSERT INTO Movies(title, year)
VALUES('Star Trek', 1979);
```

Notice that the attributes `title` and `year` had to be specified in this insertion, since we cannot provide values for other attributes of `Movies`.

The tuple inserted into `Movies` has values `'Star Trek'` for `title`, 1979 for `year`, and `NULL` for the other four attributes. Curiously, the inserted tuple, since it has `NULL` as the value of attribute `studioName`, will not meet the selection condition for the view `ParamountMovies`, and thus, the inserted tuple has no effect on the view. For instance, the query of Example 3 would not retrieve the tuple (`'Star Trek'`, 1979).

To fix this apparent anomaly, we could add `studioName` to the `SELECT` clause of the view, as:

```
CREATE VIEW ParamountMovies AS
   SELECT studioName, title, year
   FROM Movies
   WHERE studioName = 'Paramount';
```

Then, we could insert the *Star-Trek* tuple into the view by:

```
INSERT INTO ParamountMovies
VALUES('Paramount', 'Star Trek', 1979);
```

This insertion has the same effect on `Movies` as:

```
INSERT INTO Movies(studioName, title, year)
VALUES('Paramount', 'Star Trek', 1979);
```

Notice that the resulting tuple, although it has `NULL` in the attributes not mentioned, does yield the appropriate tuple for the view `ParamountMovies`. □

We may also delete from an updatable view. The deletion, like the insertion, is passed through to the underlying relation R. However, to make sure that only tuples that can be seen in the view are deleted, we add (using `AND`) the condition of the `WHERE` clause in the view to the `WHERE` clause of the deletion.

Example 6 : Suppose we wish to delete from the updatable `ParamountMovies` view all movies with "Trek" in their titles. We may issue the deletion statement

```
DELETE FROM ParamountMovies
WHERE title LIKE '%Trek%';
```

This deletion is translated into an equivalent deletion on the `Movies` base table; the only difference is that the condition defining the view `ParamountMovies` is added to the conditions of the `WHERE` clause.

```
DELETE FROM Movies
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

is the resulting delete statement. □

Similarly, an update on an updatable view is passed through to the underlying relation. The view update thus has the effect of updating all tuples of the underlying relation that give rise in the view to updated view tuples.

Example 7 : The view update

```
UPDATE ParamountMovies
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

is equivalent to the base-table update

```
UPDATE Movies
SET year = 1979
WHERE title = 'Star Trek the Movie' AND
    studioName = 'Paramount';
```

□

2.3 Instead-Of Triggers on Views

When a trigger is defined on a view, we can use `INSTEAD OF` in place of `BEFORE` or `AFTER`. If we do so, then when an event awakens the trigger, the action of the trigger is done instead of the event itself. That is, an instead-of trigger intercepts attempts to modify the view and in its place performs whatever action the database designer deems appropriate. The following is a typical example.

Why Some Views Are Not Updatable

Consider the view `MovieProd` of Example 2, which relates movie titles and producers' names. This view is not updatable according to the SQL definition, because there are two relations in the `FROM` clause: `Movies` and `MovieExec`. Suppose we tried to insert a tuple like

```
('Greatest Show on Earth', 'Cecil B. DeMille')
```

We would have to insert tuples into both `Movies` and `MovieExec`. We could use the default value for attributes like `length` or `address`, but what could be done for the two equated attributes `producerC#` and `cert#` that both represent the unknown certificate number of DeMille? We could use `NULL` for both of these. However, when joining relations with `NULL`'s, SQL does not recognize two `NULL` values as equal. Thus, `'Greatest Show on Earth'` would not be connected with `'Cecil B. DeMille'` in the `MovieProd` view, and our insertion would not have been done correctly.

Example 8: Let us recall the definition of the view of all movies owned by Paramount:

```
CREATE VIEW ParamountMovies AS
    SELECT title, year
    FROM Movies
    WHERE studioName = 'Paramount';
```

from Example 1. As we discussed in Example 5, this view is updatable, but it has the unexpected flaw that when you insert a tuple into `ParamountMovies`, the system cannot deduce that the `studioName` attribute is surely Paramount, so `studioName` is `NULL` in the inserted `Movies` tuple.

A better result can be obtained if we create an instead-of trigger on this view, as shown in Fig. 2. Much of the trigger is unsurprising. We see the keyword `INSTEAD OF` on line (2), establishing that an attempt to insert into `ParamountMovies` will never take place.

Rather, lines (5) and (6) is the action that replaces the attempted insertion. There is an insertion into `Movies`, and it specifies the three attributes that we know about. Attributes `title` and `year` come from the tuple we tried to insert into the view; we refer to these values by the tuple variable `NewRow` that was declared in line (3) to represent the tuple we are trying to insert. The value of attribute `studioName` is the constant `'Paramount'`. This value is not part of the inserted view tuple. Rather, we assume it is the correct studio for the inserted movie, because the insertion came through the view `ParamountMovies`. □

```
CREATE TRIGGER ParamountInsert
INSTEAD OF INSERT ON ParamountMovies
REFERENCING NEW ROW AS NewRow
FOR EACH ROW
INSERT INTO Movies(title, year, studioName)
VALUES(NewRow.title, NewRow.year, 'Paramount');
```

Figure 2: Trigger to replace an insertion on a view by an insertion on the underlying base table

2.4 Exercises for Section 2

Exercise 2.1: Which of the views of Exercise 1.1 are updatable?

Exercise 2.2: Suppose we create the view:

```
CREATE VIEW DisneyComedies AS
    SELECT title, year, length FROM Movies
    WHERE studioName = 'Disney' AND genre = 'comedy';
```

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the length for a movie (given by title and year) in this view.

Exercise 2.3: Using the base tables

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
```

suppose we create the view:

```
CREATE VIEW NewPC AS
SELECT maker, model, speed, ram, hd, price
FROM Product, PC
WHERE Product.model = PC.model AND type = 'pc';
```

Notice that we have made a check for consistency: that the model number not only appears in the `PC` relation, but the `type` attribute of `Product` indicates that the product is a PC.

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the price.

d) Write an instead-of trigger to handle a deletion of a specified tuple from this view.

3 Indexes in SQL

An *index* on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A. We could think of the index as a binary search tree of (key, value) pairs, in which a key a (one of the values that attribute A may have) is associated with a "value" that is the set of locations of the tuples that have a in the component for attribute A. Such an index may help with queries in which the attribute A is compared with a constant, for instance $A = 3$, or even $A \leq 3$. Note that the key for the index can be any attribute or set of attributes, and need not be the key for the relation on which the index is built. We shall refer to the attributes of the index as the *index key* when a distinction needs to be made.

The technology of implementing indexes on large relations is of central importance in the implementation of DBMS's. The most important data structure used by a typical DBMS is the "B-tree," which is a generalization of a balanced binary tree. We shall take up B-trees when we talk about DBMS implementation, but for the moment, thinking of indexes as binary search trees will suffice.

3.1 Motivation for Indexes

When relations are very large, it becomes expensive to scan all the tuples of a relation to find those (perhaps very few) tuples that match a given condition. For example, consider the first query we examined:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

There might be 10,000 `Movies` tuples, of which only 200 were made in 1990.

The naive way to implement this query is to get all 10,000 tuples and test the condition of the `WHERE` clause on each. It would be much more efficient if we had some way of getting only the 200 tuples from the year 1990 and testing each of them to see if the studio was Disney. It would be even more efficient if we could obtain directly only the 10 or so tuples that satisfied both the conditions of the `WHERE` clause — that the studio is Disney and the year is 1990; see the discussion of "multiattribute indexes," in Section 3.2.

Indexes may also be useful in queries that involve a join. The following example illustrates the point.

Example 9: Recall the query

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```