```
CREATE VIEW MovieProd(movieTitle, prodName) AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```

The view is the same, but its columns are headed by attributes `movieTitle` and `prodName` instead of `title` and `name`.

8.1.4 Exercises for Section 8.1

Exercise 8.1.1: From the following base tables of our running example

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Construct the following views:

a) A view `StudioPres` giving the name, address, and certificate number of all executives who are studio presidents.

b) A view `ExecutiveStar` giving the name, address, gender, birth date, certificate number, and net worth of all individuals who are both executives and stars.

c) A view `RichExec` giving the name, address, certificate number and net worth of all executives with a net worth of at least $5,000,000.

Exercise 8.1.2: Write each of the queries below, using one or more of the views from Exercise 8.1.1 and no base tables.

a) Find the names of those executives who are both studio presidents and worth at least $5,000,000.

b) Find the names of females who are both stars and executives.

! c) Find the names of studio presidents who are also stars and are worth at least $10,000,000.

8.2 Modifying Views

In limited circumstances it is possible to execute an insertion, deletion, or update to a view. At first, this idea makes no sense at all, since the view does not exist the way a base table (stored relation) does. What could it mean, say, to insert a new tuple into a view? Where would the tuple go, and how would the database system remember that it was supposed to be in the view?

For many views, the answer is simply "you can't do that." However, for sufficiently simple views, called *updatable views*, it is possible to translate the

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

Figure 8.2: Trigger to replace an insertion on a view by an insertion on the underlying base table

8.2.4 Exercises for Section 8.2

Exercise 8.2.1: Which of the views of Exercise 8.1.1 are updatable?

Exercise 8.2.2: 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) Write an instead-of trigger to handle an insertion into this view.

b) Write an instead-of trigger to handle an update of the speed.

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

d) Is this view updatable?

Exercise 8.2.3: Suppose we create the view:

```
CREATE VIEW LongDisneyMovies AS
    SELECT title, year, genre FROM Movies
    WHERE studioName = 'Disney' AND length > 120;
```

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

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

c) Is this view updatable?

than a year for a movie, then we would prefer to order the attributes as above; if a year were more likely to be specified, then we would ask for an index on `(year, title)`. □

If we wish to delete the index, we simply use its name in a statement like:

```
DROP INDEX YearIndex;
```

8.3.3 Exercises for Section 8.3

Exercise 8.3.1: For our running movies example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Declare indexes on the following attributes or combination of attributes:

a) `length`.

b) `address` of `Studio`.

c) `year` and `genre`.

8.4 Selection of Indexes

Choosing which indexes to create requires the database designer to analyze a trade-off. In practice, this choice is one of the principal factors that influence whether a database design gives acceptable performance. Two important factors to consider are:

- The existence of an index on an attribute may speed up greatly the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.
- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

8.4.1 A Simple Cost Model

To understand how to choose indexes for a database, we first need to know where the time is spent answering a query. The details of how relations are stored will be taken up when we consider DBMS implementation. But for the moment, let us state that the tuples of a relation are normally distributed

8.4.5 Exercises for Section 8.4

! **Exercise 8.4.1:** In this problem, we consider indexes for the relation

`Ships(name, class, launched)`

from our running battleships exercise. Assume:

i. `name` is the key.

ii. The relation `Ships` is stored over 100 pages.

iii. The relation is clustered on `class` so we expect that only one disk access is needed to find the ships of a given class.

iv. On average, there are 4 ships of a class, and 20 ships launched in any given year.

v. With probability p_1 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE class =` c.

vi. With probability p_2 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE launched =` y.

vii. With probability p_3 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE name =` n.

viii. With probability $1 - p_1 - p_2 - p_3$ the operation on this relation is an insertion of a new tuple into `Ships`.

You can also make the assumptions about accessing indexes and finding empty space for insertions that were made in Example 8.14.

Consider the creation of indexes on `name`, `class`, and `launched`. For each combination of indexes, estimate the average cost of an operation. As a function of p_1, p_2, and p_3, what is the best choice of indexes?

Exercise 8.4.2: Suppose that the relation `StarsIn` discussed in Example 8.14 required 100 pages rather than 10, but all other assumptions of that example continued to hold. Give formulas in terms of p_1 and p_2 to measure the cost of queries Q_1 and Q_2 and insertion I, under the four combinations of index/no index discussed there.

8.5 Materialized Views

A view describes how a new relation can be constructed from base tables by executing a query on those tables. Until now, we have thought of views only as logical descriptions of relations. However, if a view is used frequently enough, it may even be efficient to *materialize* it; that is, to maintain its value at all times. As with maintaining indexes, there is a cost involved in maintaining a materialized view, since we must recompute parts of the materialized view each time one of the underlying base tables changes.

1. Have a list of relations in the FROM clause that is a subset of those in the FROM clause of at least one query of the workload.
2. Have a WHERE clause that is the AND of conditions that each appear in at least one query.
3. Have a list of attributes in the SELECT clause that is sufficient to be used in at least one query.

To evaluate the benefit of a materialized view, let the query optimizer estimate the running times of the queries, both with and without the materialized view. Of course, the optimizer must be designed to take advantage of materialized views; all modern optimizers know how to exploit indexes, but not all can exploit materialized views. Section 8.5.3 was an example of the reasoning that would be necessary for a query optimizer to perform, if it were to take advantage of such views.

There is another issue that comes up when we consider automatic choice of materialized views, but that did not surface for indexes. An index on a relation is generally smaller than the relation itself, and all indexes on one relation take roughly the same amount of space. However, materialized views can vary radically in size, and some — those involving joins — can be very much larger than the relation or relations on which they are built. Thus, we may need to rethink the definition of the "benefit" of a materialized view. For example, we might want to define the benefit to be the improvement in average running time of the query workload divided by the amount of space the view occupies.

8.5.5 Exercises for Section 8.5

! **Exercise 8.5.1:** Suppose the view NewPC of Exercise 8.2.2 were a materialized view. What modifications to the base tables Product and PC would require a modification of the materialized view? How would you implement those modifications incrementally?

Exercise 8.5.2: Complete Example 8.15 by considering updates to either of the base tables.

! **Exercise 8.5.3:** This exercise explores materialized views that are based on aggregation of data. Suppose we build a materialized view on the base tables

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

from our running battleships exercise, as follows:

```
CREATE MATERIALIZED VIEW ShipStats AS
    SELECT country, AVG(displacement), COUNT(*)
    FROM Classes, Ships
    WHERE Classes.class = Ships.class
    GROUP BY country;
```

What modifications to the base tables `Classes` and `Ships` would require a modification of the materialized view? How would you implement those modifications incrementally?

! **Exercise 8.5.4:** In Section 8.5.3 we gave conditions under which a materialized view of simple form could be used in the execution of a query of similar form. For the view of Example 8.15, describe all the queries of that form, for which this view could be used.

8.6 Summary of Chapter 8

- ✦ *Virtual Views*: A virtual view is a definition of how one relation (the view) may be constructed logically from tables stored in the database or other views. Views may be queried as if they were stored relations. The query processor modifies queries about a view so the query is instead about the base tables that are used to define the view.

- ✦ *Updatable Views*: Some virtual views on a single relation are updatable, meaning that we can insert into, delete from, and update the view as if it were a stored table. These operations are translated into equivalent modifications to the base table over which the view is defined.

- ✦ *Instead-Of Triggers*: SQL allows a special type of trigger to apply to a virtual view. When a modification to the view is called for, the instead-of trigger turns the modification into operations on base tables that are specified in the trigger.

- ✦ *Indexes*: While not part of the SQL standard, commercial SQL systems allow the declaration of indexes on attributes; these indexes speed up certain queries or modifications that involve specification of a value, or range of values, for the indexed attribute(s).

- ✦ *Choosing Indexes*: While indexes speed up queries, they slow down database modifications, since the indexes on the modified relation must also be modified. Thus, the choice of indexes is a complex problem, depending on the actual mix of queries and modifications performed on the database.

- ✦ *Automatic Index Selection*: Some DBMS's offer tools that choose indexes for a database automatically. They examine the typical queries and modifications performed on the database and evaluate the cost trade-offs for different indexes that might be created.

- ✦ *Materialized Views*: Instead of treating a view as a query on base tables, we can use the query as a definition of an additional stored relation, whose value is a function of the values of the base tables.