Data Modelling and Databases - Chapter 8 (Book)

- Author: Ruben Schenk
- Date: 05.05.2021
- Contact: ruben.schenk@inf.ethz.ch

8. Views and Indexes

We begin this chapter by introducing virtual views, which are relations that are defined by a query over other relations. Virtual views are not stored in the database, but can be queried as if they existed. The query processor will replace the view by its definition in order to execute the query.

Views can also be materialized, in the sense that they are constructed periodically from the database and stored there. The existence of these materialized views can be speed up the execution of queries.

8.1 Virtual Views

Relations that are defined with a CREATE TABLE statement actually exist in the database. They are *persistent*, in the sense that they can be expected to exist indefinitely and not to change unless they are explicitely told to change by a SQL modification statement.

There is another class of SQL relations, called `(virtual) views`, that do not exist physically. Rather, they are defined by an expression much like a query.

8.1.1 Declaring Views

The simplest form of `view definition` is:

```
CREATE VIEW <view-name> AS <view-definition>;
```

The view definition is an SQL query.

Example: Suppose we want to have a view that is part of the *Movies* relation, specifically, the titles and yyears of the movies made by Paramaount Studios. We can define this view by:

```
/* Code 8.1: Views in SQL. */
CREATE VIEW ParamountMovies AS
    SELECT title, year
    FROM Movies
    WHERE studioName = 'Paramount';
```

8.1.2 Querying Views

A view may be queried exactly as if it were a stored table. We mention its name in a FROM clause and rely on the DBMS to produce the needed tuples by operating on the relations used to define the virtual view.

Example: We may query the view *ParamountMovies* just as if it were a stored table, for instance:

```
/* Code 8.3: Querying virtual views. */
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

finds the movies made by Paramount in 1979.

8.1.3 Renaming Attributes

Sometimes, we might prefer to give a view's attributes names of our own choosing. We may specify the attributes of the view by listing them, surrounded by parentheses, after the name of the view in the CREATE VIEW statement.

Example: For instance, we could write:

```
CREATE VIEW MovieProd(movieTitle, prodName) AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerNum = certNum;
```

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.

For many views, it is simply not possible to do that. However, for sufficiently simple views, called `updatable views`, it is possible to translate modification of the view into an equivalent modification on the base table, and the modification can be done to the base table instead.

8.2.1 View Removal

An extrem 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*.

8.2.2 Updatable View

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. Some 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.

Example: Suppose we insert into view *ParamountMovies* a tuple like:

```
/* Code 8.5: Inserting into views. */
INSERT INTO ParamountMovies
VALUES('Star Trek', 1979);
```

Theinsertion on *ParamountMovies* is executed as if it were the same insertion on *Movies*:

```
/* 8.5: Continuation. */
INSERT INTO Movies(title, year)
VALUES('Star Trek', 1979);
```

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 the `AND`) the condition of the WHERE clause in the view to the WHERE clause of the deletion.

Example: Suppose we wish to delete from the updatable *ParamountMovies* view all movies with "Trek" in their titles:

```
/* Code 8.6: Deletion via updatable views. */
    DELETE FROM ParamountMovies
    WHERE title LIKE '%Trek%';
```

This deletion is translated into an equivalent deletion on the *Movies* base table:

```
/* Code 8.6: Continuation. */
DELETE FROM Movies
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

Similarly, an `update` one an updatable view is passed through to the under lying relation.

8.2.3 Instead-Of Triggers on Views

When a trigger is defined on a view, we canb 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.

Example: Let us consider the following instead-of trigger:

```
/* Code 8.8: Instead-of triggers in SQL. */
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');
```

Much of the trigger is unsurprising. We see the keyword `INSTEAD OF` on line (2), establishing that an attempt to isnert into *ParamountMovies* will never take place. Rather, lines (5) and (6) is the action that replaces the attempted insertion.

8.3 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)* paris, 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 of attribute A.

We shall refer to the attributes of the index as the `index key` when a distinction needs to be made.

8.3.1 Motivation for Indexes

When relations are very large, it becomes expensive to scan all the tuples of a relation to find those tuples that match a given condition.

For example, consider the following query:

```
SELECT *
FROM Movies
WHERE studioName = `Disney` AND year = 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.

8.3.2 Declaring Indexes

Suppose we want to have an index on attribute *year* for the relation *Movies*. Then we say:

```
CREATE INDEX YearIndex ON Movies(year);
```

The result will be that an index whose name is *YearIndex* will be created on attribute *year* of the relation *Movies*. Henceforth, SQL queries that specify a year may be executed by the SQL query processor in such a way that only those tuples of *Movies* with the specified year are ever examined.

Example: Since *title* and *year* form a key for *Movies*, we might expect it to be common that values for both these attributes will be specified, or neither will. The following is a typical declaration of an index on these two attributes:

```
/* Code 8.10: Indexes in SQL. */
CREATE INDEX KeyIndex ON Movies(title, year);
```

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

```
DROP INDEX YearIndex;
```

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 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 queries on the respective attribute.
- On the other hand, every index built for one or more attribute makes insertions, deletions, and updates to the respective 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 queries. For the moment, let us state that the tuples of a relation are normally distributed among many pages of a disk. Therefore, to examine even one tuple requires that the whole page be brought into main memory.

8.4.2 Some Useful Indexes

Often, the most useful index we can put on a relation is an `index on its key`. There are two reasons:

1. Queries in which a value for the key is specified are common.
2. Since there is at most one tuple with a given key value, the index returns either nothing or one location for a tuple. Thus, at most one page must be retrieved to get that tuple into main memory.

There are two situations in which an index can be effective, even if it is not on a key:

1. If the attribute is almost a key, that is, relatively few tuples have a given value for that attribute.
2. If the tuples are *clustered* on that attribute. We `cluster` a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible.

8.4.3 Calculating the Best Index to Create

It might seem that the more indexes we create, the more likely it is that an index useful for a given query will be available. However, if modifications are the most frequent action, then we should be very conservative about creating indexes, since each modification on a relation R forces us to change any index on one or more of the modified attributes of R.

Following an example of how to calculate the best index:
Let us consider the relation `StarsIn(movieTitle, movieYear, starName)`. Suppose that there are three database operations that we somestimes perform on this relation:

```
/* Q1 */
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = 2;
/* for some constant s */
```

```
/* Q2 */
SELECT starName
FROM StarsIn
WHERE movieTitle = t AND movieYear = y;
/* for some constants t and y */
/* I */
INSERT INTO StarsIn VALUES(t, y, s);
/* for some constants t, y, and s. */
```

Let us make the following assumptions about the data:

1. `StarsIn` occupies 10 pages, so if we need to examine the entire relation the cost is 10.
2. On the average, a star has appeared in 3 movies and a movie has3 stars.
3. Even with an index, it will take 3 disk accesses to find the 3 tuples for a star or movie.
4. One disk access is needed to read a page of the index. If an index page must be modified, then another disk access is needed to write back the modified page (for example in an insertion).

The following table gives the costs of each of the three operations:

Action	No Index	Star Index	Movie Index	Both Indexes
Q_1	10	4	10	4
Q_2	10	10	4	4
I	2	4	4	6
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

The final row gives the average cost of an action, on the assumption that the fraction of the time we do Q_1 is p_1 and the fraction of the time we do Q_2 is p_2, therefore, the fraction of the time we do I is $1 - p_1 - p_2$. Depending on p_1 and p_2, any of the four choices of index/no index can yield the best average cost for the three actions.

8.4.4 Automatic Selection of Indexes to Create

Tuning a database is aprocess that includes not only index selection, but the choice of many different parameters. There are a number of tools that have been designed to take the responsibility from the database designer and have the system tune itself, or at least advise the designer on good choices.

Here is an outline of how the index-selection portion of tuning advisors work:

1. The first step is to establish the query workload. We may be able to examine a log and find a set of representative queries and database modifications for the database at hand.
2. The designer may be offered the opportunity to specify some constraints, e.g., indexes that must, or must not, be chosen.
3. The tuning advisor generates a set of possible `candidate indexes`, and evaluates each one. Typical queries are given to the query optimizier of the DBMS.
4. The index set resulting in the lowest cost for the given workload is suggested to the designer, or it is automatically created.

8.5 Materialized Views

A view describes how a new relation can be constructed from base tables by executing a query on those tables. If a view is used frequently enoughy, it may even be efficient to `materialize` it, that is, to maintain its value at all times.

In principle, the DBMS needs to recompute a materialized view every time one of its base tables changes in any way. For simple views, it is possible to limit the number of times we need to consider changing the materialized view.

Example: Suppose we frequently want to find the name of the producer of a given movie. We might find it advantageous to materialize a view as follows:

```
/* Code 8.15: Materialized views. */
CREATE MATERIALIZED VIEW MovieProd AS
    SELECT title, year, name
    FROM Movies, MovieExec
    WHERE producerNum = certNum
```

It is important to note that many common types of materialized views do allow the view to be maintained `incrementally`. That is, we never have to reconstruct the whole view from scratch. Rather, insertions, deletions, and updates to a base table can be implemented in a join view by a small number of queries to the base tables followed by modification statements on the materialized view.

8.5.2 Periodic Maintenance of Materialized Views

There is another setting in which we may use materialized views, yet not have to worry about the cost or complexity of maintaining them up-to-date as the underlying base tables change. For example, an analyst may not need the exact current state of a database for its analysis.

What is usually done is to create materialized views, but not to try to keep them up-to-date as the base tables change. Rather, the materialized views are reconstructed periodically (typically each night), when other activity in the database is low. The materialized views are only used by analysts, and their data might be out of date by as much as 24 hours.

8.5.3 Rewriting Queries to Use Materialized Views

A materialized view can be referred to in the FROM clause of a query, just as a virtual view can. However, becuase a materialized view is stored in the database, it is possible to rewrite a query to use a materialized view, even if that view was not mentioned in the query as written.

Suppose we have a mterialized view V defined by a query of the form

```
SELECT L_v
FROM R_v
WHERE C_v
```

where L_V is a list of attributes, R_V is a list of relations, and C_V is a condition. Similarly, suppose we have a query Q of the same form:

```
SELECT L_q
FROM R_q
WHERE C_q
```

Here are the conditions under which we can repalce part of the query Q by the view V.

1. The realtions in list R_V all appear in the list R_Q.
2. The condition C_Q is equivalent C_V AND C for some condition C.
3. If C is needed, then the attributes of relations on list R_V that C mentions are attributes on the list L_V.
4. Attributes on the list L_Q that come from relations on the list R_V are also on the list L_V.

If all these conditions are met, then we can rewrite Q to use V, as follows:

1. Replace the list R_Q by V and the realtions are in list R_Q but not on R_V.
2. Repalce C_Q by C. If C is not needed (i.e., $C_V = C_Q$), then there is no WHERE clause.

8.5.4 Automatic Creation of Materialized Views

The ideas that were discussed in Section 8.4.4 for indexes can apply as well to materialized views. We first need to establish or approximate the query workload. An automated materialized-view-selection advisor needs to generate `candidate views`. The process can be limited if we remember that there is no point in creating a materialized view that does not help for at least one query of our expected workload.

We can limit ourselves to candidate materialized views that:

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.

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.

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.

Instead-Of-Triggers

SQL allows a special type of trigger to apply to a virtual view.

Indexes

While not part of the SQL standard, commercial SQL systems allow the declaration of `indexes` on attributes.

Choosing Indexes

While indexes speed up queries, they slow down database modifications, since the indexes on the modified relations must also be modified.

Automatic Index Selection

Some DBMS's offer tools that choose indexes for a database automatically.

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.

Maintaining Materialized Views

As the base tables change, we must make the corresponding changes to any materialized view whose value is affected by the change.

Rewriting Queries to Use Materialized Views

The condition under which a query can be rewritten to use a materialized view are complex. However, if the query optimizier can perform such rewritings, then an automatic design tool can consider the improvement in performance that results from creating materialized views.