Lecture 5

Database Objects

(part 2)

OUTLINES

OUTLINES

Views

- A *view* is a relation defined by a query over stored tables (called *base tables*) and other views.
- Two kinds:
  1. *Virtual* = not stored in the database; just a query for constructing the relation.
  2. *Materialized* = actually constructed and stored.

Declaring Views

- Declared by:

  CREATE VIEW <view-name>

  AS <SQL query>;

- Notice: Relations(Base relations),

  Tables(Base tables),

  Views

  Their differences

Example : Suppose we want to have a view that is a part of the Movie(title, year , length , incolor, studioName, producerC) Relation, specifically, the titles and years of the movies made by Paramount Studios.

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

Example: Our goal is a relation MovieProd with movie titles and the names of their producers.

```
CREATE VIEW MovieProd AS
      SELECT title, name
      FROM Movies, MovieExec
      WHERE producerC = cert;
```

The query defining the view involves two relations

Querying Views

- Query a view as if it were a base table.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- Example query:

```
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

Querying Views: an example

ParamontMovie(movieTitle, movieYear)

Renaming Attributes

VIEW definitions

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

QUERY on VIEW

```
SELECT title
FROM ParamountMovies WHERE year = 1979;
```

Has the same effect as(equivalent query) on base table

```
SELECT title
FROM Movie
WHERE studioName = 'Paramount' AND year = 1979;
```

Querying Views: an example


```
SELECT DISTINCT starName
FROM ParamountMovies, StarsIn
WHERE title = movieTitle  AND year = movieYear;
```

Queries involving both views and base tables

Equivalent query on table:

Interpreting the use of a virtual view as a subquery

```
SELECT DISTINCT starName
FROM  (SELECT  title, year
          FROM Movie
          WHERE studioName = 'Paramount'
          ) pm, StarsIn
WHERE pm.title = movieTitle  AND pm.year = movieYear;
```

OUTLINES

Modifying Views

- In limited circumstances it is possible to execute an **Insertion**, **Deletion**, or **Update** to a view

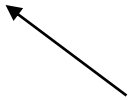
- Modifying Data Through a View
  - ***updatable views***, translate the modification of the view into an equivalent modification on a base table: and the modification can be done to the base table instead

Modifying Views

- View removal

DROP VIEW ParamountMovies;

DROP TABLE Movies;



Deletion the base table and make all the views on it unusable

updatable views

- permit modifications on views that
  - Defined by selecting (using **SELECT,** not **SELECT DISTINCT)** some attributes from one relation *R*
  - 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
  - (optional) The list of **SELECT** must includes enough attributes that for every tuple inserted into the view, we can fill the other attributes out with NULL values or the proper default
- Any modifications are translated into an equivalent modifications on a base table.

Example: Insertion

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

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

What happen? **Can not be seen in View!**

title	year	length	inColor	studioName	producerC#
Star Trek	1979	0	NULL	NULL	NULL

Example: Intersion

To fix this problem[includes enough attributes]

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

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

title	year	length	inColor	studioName	producerC#
Star Trek	1979	0	NULL	Paramount	NULL

Example: Deletion

Delete from the updatable ParamountMovie view all movies with "Trek" in their titles.

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

This deletion is translated into an equivalent deletion on the Movie base table: the only differences 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' ;
```

Example: Update

An update on an updatable view is passed through to the underlying relation.

```
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' ;
```

Question

- Why some views are not updatable?

Triggers on Views

- Generally, it is impossible to modify a virtual view, because it doesn't exist.
- But an INSTEAD OF trigger lets us intercept view modifications in a way that makes sense.

Example

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

The view is updatable, but it has the unexpected flaw that when we insert a tuple into ParamountMovies.

A better solution is to create an instead-of trigger on this view.

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

OUTLINES

OUTLINES

Indexes in SQL

Selection of Indexes

Materialized Views

Indexes

- *Index* = data structure used to speed access to tuples of a relation, given values of one or more attributes.
- Query (SELECT) more efficient
- Could be a hash table, but in a DBMS it is always a balanced search tree with giant nodes (a full disk page) called a *B-tree*.

Motivation

- 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.

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

Motivation

- Indexes may also be useful in queries that involve a join,
- How do indexes on title of Movies and cert of MovieExec work?

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

Declaring & Deleting Indexes

- Declaring
  - No standard!
  - Typical syntax:

```
CREATE INDEX YearIndex ON Movies(year);
CREATE INDEX KeyIndex ON Movies(title, year);
```

- Deleting

```
DROP INDEX YearIndex;
DROP INDEX KeyIndex;
```

Using Indexes

- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.
- Example:
  - 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
  - Decrease in the time needed to answer the query

Using Indexes

- Multiattribute index
  - The key is the concatenation of the attributes in some order
  - We can even use this index to find all the tuples with a given value in the first of the attributes
  - The choice of the order in which the attributes are listed

```
CREATE INDEX KeyIndex ON
Movies(title, year);
```

OUTLINES

Selection of Indexes

- Choosing which indexes to create is a **trade-off**
  - Index on an attribute may speed up greatly the execution of query
    - in which a value, or range of values, is specified for that attribute,
    - and may speed up joins involving that attribute as well
  - Index makes insertions, deletions and updates to that relation more complex and time-consuming.
    - modification(update, delete, insert) is about twice as expensive as selection
    - each modification of R forces to change any index on that modified attribute(s) of R.

A Simple Cost Model

- Time spent answering a query
  - To examine even one tuple requires that the whole page be brought into main memory
  - It costs little more time to examine all the tuples on a page than to examine only one
  - We assume every page we need must be retrieved from the disk

Some Useful Indexes

- The most useful index we can put on a relation is an index on its key
  - Queries in which a value for the key is specified are common. Thus, an index on the key will get used frequently
  - 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

Some Useful Indexes

- Example

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

- With the right indexes, the whole query might be done with as few as two page reads
- How to do this?

Some Useful Indexes

- When the index is not on a key
  - It may or may not be able to improve the time

- Two situations in which an index can be effective, even if it is not on a key
  - If the attribute is almost a key; that is, relatively few tuples have a given value for that attribute
  - 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

Some Useful Indexes

- When the index is not on a key
  - It may or may not be able to improve the time
- Example

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

```
CREATE INDEX TitleIndex ON
Movies(title);
```

What happens compared with the index on the key?

Some Useful Indexes

- When the index is not on a key
  - It may or may not be able to improve the time
- Example

SELECT * FROM Movies WHERE year = 1990;

– The only index we have on Movies

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

- The tuples are not clustered by year
  - Say stored alphabetically by title
- The tuples are clustered on year

Calculating the Best Indexes

- One must be very careful how one estimates the relative frequency of modifications and queries
  - Modification, twice as expensive as query
  - The most common query and modification forms

Calculating the Best Indexes

- Depend on which queries and modifications are most likely to be performed on the Database.
- Example StarsIn(movieTitle, movieYear, starName)

*Q*1：SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = *s*;

*Q*2: SELECT starName
FROM StarsIn
WHERE movieTitle = *t* AND movieYear = *y*;

*I*1： INSERT INTO StarsIn VALUES (*t*, *y*, *s*);

Basic assumptions:

StarsIn is stored in 10 disk blocks (pages)

On the average, a star has appeared in 3 movies and a movie has 3 stars;

Example

- Costs associated with the three actions, as a function of which indexes are selected

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
	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

the fraction of $Q1$ is p1;

the fraction of $Q2$ is p2,

the fraction of I is 1-p1-p2.

- p1=p2=0.1, (2+8p1+8p2) is the smallest
- p1=p2=0.4, (6-2p1-2p2) is the smallest
- p1=0.5, p2=0.1, (4+6p2) is the smallest
- P1=0.1, p2=0.5, (4+6p1) is the smallest

Automatic Selection of Indexes to Create

- "Tuning" a database is a process
  - That includes not only index selection,
  - But the choice of many different parameters
- How the index-selection portion of tuning advisors work
  - To establish the query workload
  - The designer may be offered the opportunity to specify some constraints, e.g., indexes that must, or must not, be chosen
  - The tuning advisor generates a set of possible candidate indexes, and evaluates each one
  - The index set resulting in the lowest cost is created

Automatic Selection of Indexes to Create

- A “greedy” approach to choosing indexes
  - Initially, with no indexes selected, evaluate the benefit of each of the candidate indexes. If at least one provides positive benefit, then choose that index
  - Then, reevaluate the benefit of each of the remaining candidate indexes, assuming that the previously selected index is also available. Again, choose the index that provides the greatest benefit
  - In general, repeat the evaluation of candidate indexes under the assumption that all previously selected indexes are available. Pick the index with maximum benefit, until no more positive benefits can be obtained

OUTLINES

Materialized Views

- If a view is used frequently enough, it may even be efficient to ***materialize*** it. (More Efficient!!)
  - To maintain its values at all times
  - Cost: recompute the *Materialized View* each time one of the underlying base tables changes.
  - All the changes to the Materialized View are *incremental*

```
CREATE MATERIALIZED VIEW MovieProd AS
    SELECT title,year,name
    FROM Movies, MovieExec
    WHERE producerC = cert;
```

Materialized Views

All the changes to the Materialized View are *incremental.*

It is more efficiently than a re-execution of the query.

Scene 1) Insert a new movie into Movies(title, year, producerC):

INSERT INTO Movies VALUES(“Kill Bill”, 2003, 23456);

Then we only need to look up cert=23456 in MovieExec (for cert is the key). So

SELECT name FROM MovieExec WHERE cert = 23456;

As this query return name = “Quentin Tarantino”, then the DBMS can insert the proper tuple into MovieProd by

INSERT INTO MovieProd VALUES(‘Kill Bill’, 2003, ‘Quentin Tarantino’);

Materialized Views

Scene 2) Delete a movie from Movies (title, year, producerC):

DELETE FROM Movies WHERE title = 'Dumb & Dumber' and year = 1994;

The DBMS will:

DELETE FROM MovieProd WHERE title = 'Dumb & Dumber' and year = 1994;

Scene 3) Update a movie in Movies (title, year, producerC):

UPDATE Movies SET title = 'Dumb & Dumber 2'
WHERE title='Dumb & Dumber' AND year = 1994;

The DBMS will:

UPDATE Moviesprod SET title = 'Dumb & Dumber 2'
WHERE title='Dumb & Dumber' AND year = 1994;

Scene 4) Insert a tuple into MovieExec (cert, name)

INSERT INTO MovieExec VALUES(34567, 'Max Bialystock');

The DBMS will:

INSERT INTO MovieProd
SELECT title, year,"Max Bialystock" FROM Movies WHERE producerC=34567;

Materialized Views

Scene 5) Delete a tuple from MovieExec

```
DELETE FROM MovieExec WHERE cert = 45678;
```

The DBMS will:

```
DELETE FROM MovieProd
  WHERE (title, year) IN
    (SELECT title, year  FROM Movies WHERE procuderC = 45678);
```

Scene 6) Update to MovieExec involving cert

```
UPDATE MovieExec SET cert =12345 where cert=34567
```

The DBMS will:

```
DELETE FROM MovieProd
    WHERE (title, year) IN (
        SELECT title, year FROM Movies, MovieExec
        WHERE cert = 34567 AND cert = producerC);
  INSERT INTO MovieProd
    SELECT title, year, name
    FROM Movies, MovieExec
    WHERE cert = 12345 AND cert = producerC;
```

```
SELECT name
FROM MovieExec
WHERE cert = 45678
```

Can we delete/Update all movies from MovieProd that have the producer name?

Periodic Maintenance

- Problem: each time a base table changes, the materialized view may change.
  - Cannot afford to recompute the view with each change.
- Solution: Periodic reconstruction of the materialized view, which is otherwise “out of date.” (typically each night)

Example: A Data Warehouse

- Wal-Mart stores every sale at every store in a database.
- Overnight, the sales for the day are used to update a *data warehouse* = materialized views of the sales.
- The warehouse is used by analysts to predict trends and move goods to where they are selling best.

Rewriting Queries to Use Materialized Views

- A materialized view *V* defined by

SELECT L_V
FROM R_V
WHERE C_V

- Suppose we have a query Q

SELECT L_Q
FROM R_Q
WHERE C_Q

Rewriting Queries to Use Materialized Views

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

1. The relations in list R_V all appear in the list R_Q.
2. The condition C_Q is equivalent to C_V **AND** C for some condition C. As a special case, C_Q could be equivalent to C_V, in which case the "**AND** C" is unnecessary.
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.

Rewriting Queries to Use Materialized Views

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

a) Replace the list R_Q by V and the relations that are on list R_Q but not on R_V.

b) Replace C_Q by C. If C is not needed (i.e., $C_V = C_Q$), then there is no **WHERE** clause.

Rewriting Queries to Use Materialized Views

- Example:

```
SELECT starName
FROM StarsIn, Movies, MovieExec
WHERE movieTitle= title AND movieYear = year AND producer = cert
        AND  name= ‘Max Bialystock’
```

- Rewrite

```
CREATE MATERIALIZED VIEW MovieProd AS
    SELECT title, name, year
        FROM Movies, MovieExec
        WHERE producer = cert
SELECT starName
    FROM StarsIn, MovieProd
    WHERE movieTitle= title AND movieYear = year AND  name= ‘Max Bialystock’
```

Movies, MovieExec

AND producer = cert

Automatic Creation of Materialized Views

- The ideas for indexes can apply as well to materialized views
  - To establish or approximate the query workload
  - An automated materialized-view-selection advisor needs to generate candidate views
  - Far more difficult than generating candidate indexes
  - All modern optimizers know how to exploit indexes, but not all can exploit materialized views