Chapter 8 Views, Indexes

Virtual and Materialized Views

Speeding Accesses to Data

Views

- A *view* is a relation defined in terms of 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

- Declare by:

```
CREATE [MATERIALIZED] VIEW
    <name> AS <query>;
```

- Default is virtual.

Example: View Definition

- CanDrink(drinker, beer) is a view “containing” the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
   SELECT drinker, beer
   FROM Frequents, Sells
   WHERE Frequents.bar = Sells.bar;
```

Example: Accessing a View

- 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 beer FROM CanDrink
WHERE drinker = ’Sally’;
```

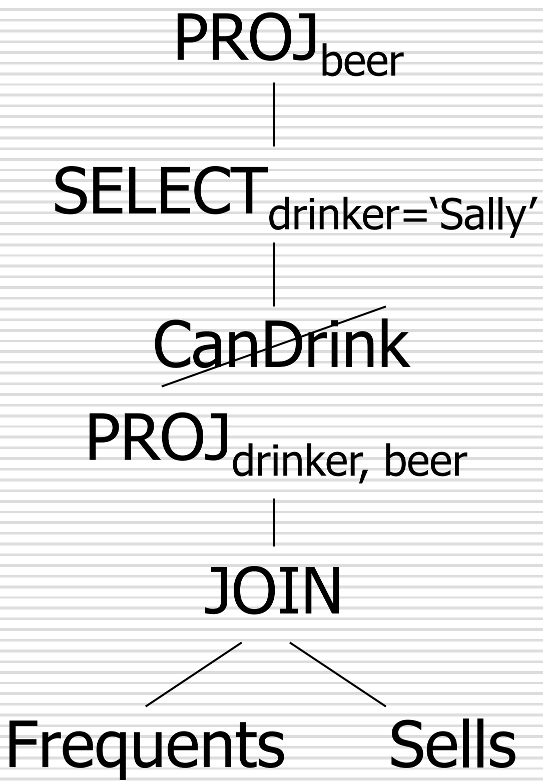
What Happens When a View Is Used?

- **The DBMS starts by interpreting the query as if the view were a base table.**
  - **Typical DBMS turns the query into something like relational algebra.**
- **The queries defining any views used by the query are also replaced by their algebraic equivalents, and "spliced into" the expression tree for the query.**

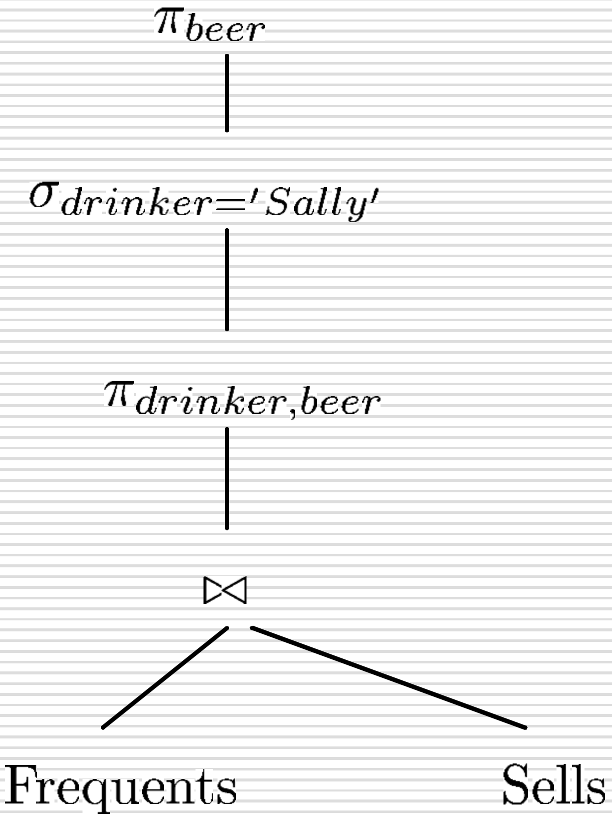
Example: View Expansion

```
SELECT beer
FROM CanDrink
WHERE drinker
= ’Sally’;
```

```
CREATE VIEW
CanDrink AS
SELECT drinker,
beer
FROM Frequents,
Sells
WHERE
Frequents.bar =
Sells.bar;
```



$\text{PROJ}_{\text{beer}}$
$\text{SELECT}_{\text{drinker=‘Sally’}}$
~~CanDrink~~
$\text{PROJ}_{\text{drinker, beer}}$
JOIN
Frequents
Sells



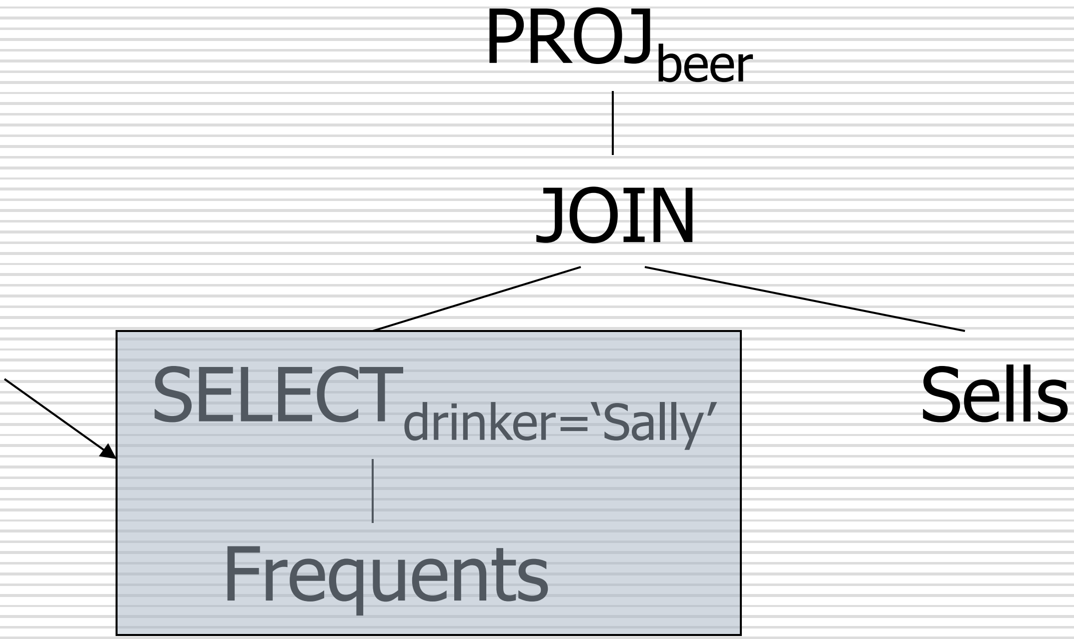
DMBS Optimization

- **It is interesting to observe that the typical DBMS will then "optimize" the query by transforming the algebraic expression to one that can be executed faster.**
- **Key optimizations:**

1. **Push selections down the tree.**
2. **Eliminate unnecessary projections.**

Example: Optimization

Notice how most tuples are eliminated from Frequents before the expensive join.


$\text{PROJ}_{\text{beer}}$
JOIN
$\text{SELECT}_{\text{drinker='Sally'}}$
Frequents
Sells

Modifying Views

- View Removal

  ```
  Drop view canDrink;
  ```

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Triggers on Views

- Generally, it is impossible to modify a virtual view, because it doesn’t exist.
- But an INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.
- Example: View Synergy has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer.

Example: The View

Pick one copy of each attribute

```
CREATE VIEW Synergy AS
SELECT Likes.drinker, Likes.beer, Sells.bar
  FROM Likes, Sells, Frequents
  WHERE Likes.drinker =
  Frequents.drinker
      AND Likes.beer = Sells.beer
      AND Sells.bar = Frequents.bar;
```

Natural join of Likes, Sells, and Frequents

Interpreting a View Insertion

- We cannot insert into Synergy --- it is a virtual view.
- But we can use an INSTEAD OF trigger to turn a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents.
  - Sells.price will have to be NULL.

The Trigger

```
CREATE TRIGGER ViewTrig
    INSTEAD OF INSERT ON Synergy
    REFERENCING NEW ROW AS n
    FOR EACH ROW
    BEGIN
        INSERT INTO LIKES VALUES(n.drinker, n.beer);
        INSERT INTO SELLS(bar, beer) VALUES(n.bar, n.beer);
        INSERT INTO FREQUENTS VALUES(n.drinker, n.bar);
    END;
```

Materialized Views

- 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.”

Example

```
CREATE MATERIALIZED VIEW CanDrink AS
     SELECT drinker, beer
     FROM Frequents, Sells
     WHERE Frequents.bar = Sells.bar;
```

Indexes

- *Index* = data structure used to speed access to tuples of a relation, given values of one or more attributes.
- 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*.

Declaring Indexes

- No standard!
- Typical syntax:

```
CREATE INDEX BeerInd ON
   Beers(manf);
CREATE INDEX SellInd ON Sells(bar,
   beer);
```

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: use BeerInd and SellInd to find the prices of beers manufactured by Pete’s and sold by Joe. (next slide)

Using Indexes --- (cont.)

```
SELECT price FROM Beers, Sells
WHERE manf = ’Pete’’s’ AND
    Beers.name = Sells.beer AND
    bar = ’Joe’’s Bar’;
```

1. Use BeerInd to get all the beers made by Pete’s.
2. Then use SellInd to get prices of those beers, with bar = ’Joe”s Bar’

Database Tuning

- A major problem in making a database run fast is deciding which indexes to create.
- Pro: An index speeds up queries that can use it.
- Con: An index slows down all modifications on its relation because the index must be modified too.

Example: Tuning

- Suppose the only things we did with our beers database was:
  1. Insert new facts into a relation (10%).
  2. Find the price of a given beer at a given bar (90%).
- Then SellInd on Sells(bar, beer) would be wonderful, but BeerInd on Beers(manf) would be harmful.

Tuning Advisors

- A major research thrust.
  - Because hand tuning is so hard.
- An advisor gets a *query load*, e.g.:
  1. Choose random queries from the history of queries run on the database, or
  2. Designer provides a sample workload.

Tuning Advisors --- (2)

- The advisor generates candidate indexes and evaluates each on the workload.
  - Feed each sample query to the query optimizer, which assumes only this one index is available.
  - Measure the improvement/degradation in the average running time of the queries.

Some useful suggestions

- □ Index on its key.
- □ Index on the following two cases:

1. If the attribute is almost a key
2. If the tuples are clustered on that attribute.

➡ To decrease the cost of accessing data

Some useful suggestions (cont.)

- If we are doing mostly insertion,very few queries, then we do not want an index
- If we are doing a lot of queries, and the number of queries specifying movies and starts are roughly equally frequent, then both index are desired.

Summary of chapter 8

- Views (virtual and materialized)
- Updatable views
- Indexes (creation, use)