# 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';
```

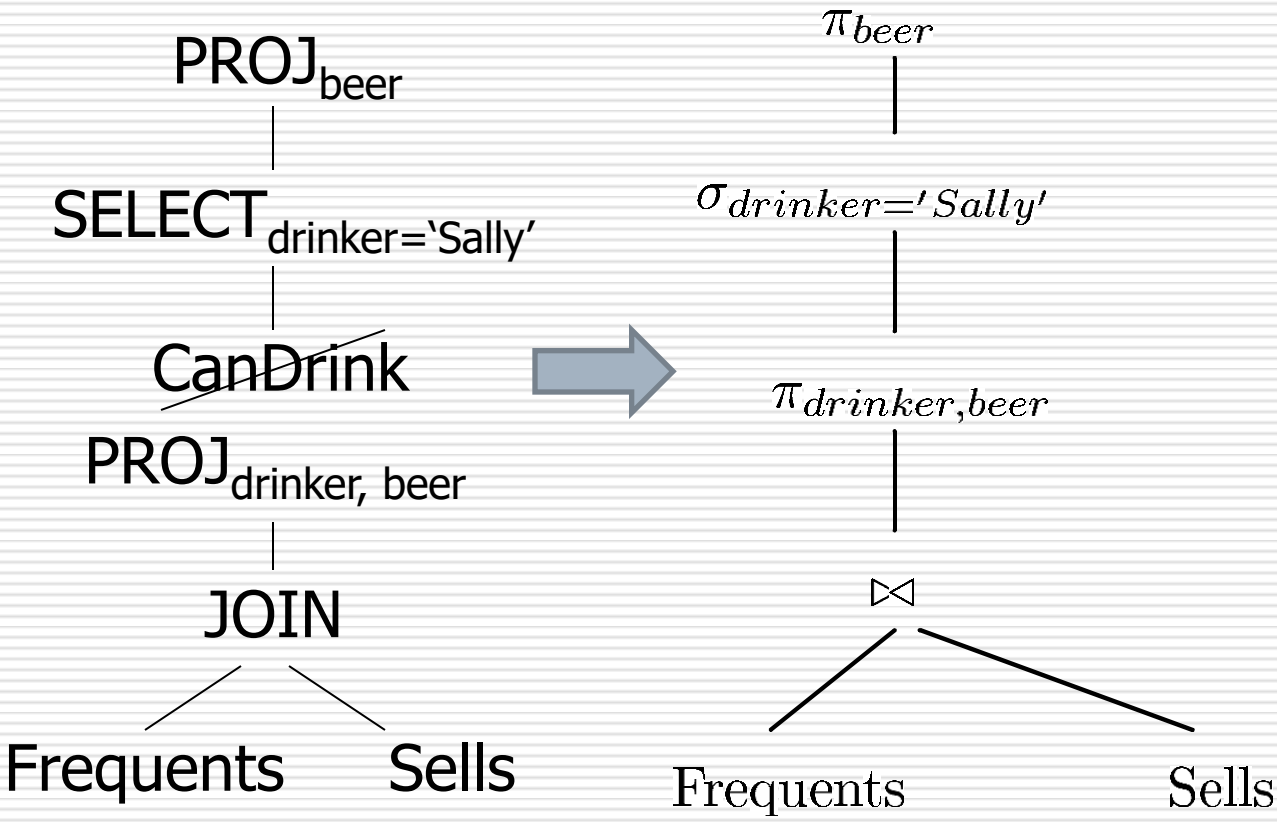
# What Happens When a View Is Used?

- 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;

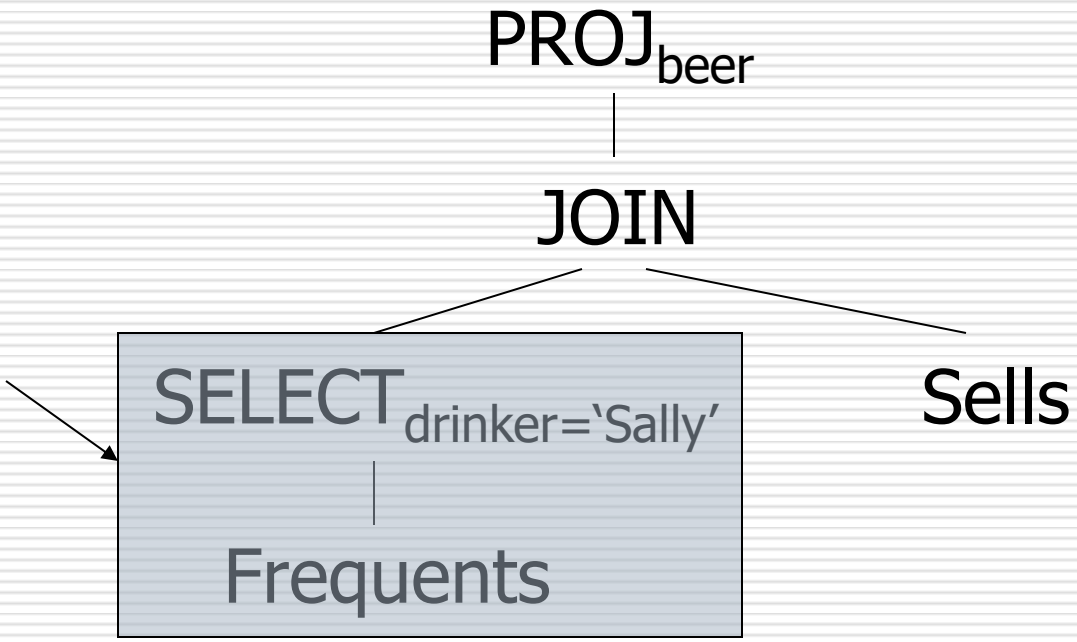


# DMBS Optimization

- 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.



# 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.

# Classroom Exercises

- Test View and Table

# Triggers on Views

- Generally, it is impossible to modify a virtual view, because it doesn't exist.
- 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

Stored like a base table.

- Disadvantage: each time a base table changes, the materialized view may change.
  - Cannot afford to recompute the view with each change.
- Advantage: speed up those queries which involve a join of many relations.

# Example

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

- **CanDrink** is then stored as a base table.

# How to update the CanDrink?

- Insert into **Sells** values('Joe's bar','Budlit', 2.3)

→

- Insert into Candrink select drink,'Budlit' from frequents where bar='Joes''s bar';
- **Incrementally update**
- Instead of recomputing the view

# How to update the materialized view? (cont.)

- Materialized views are for Data analysis, their data might be out of date.
- **Periodic reconstruction** of the materialized view is possible.

# Materialized view ---speed up the query

```
SELECT SUM(price)
FROM Sales NATURAL JOIN Frequents
   NATURAL JOIN Beers
WHERE drink = 'David Wu' AND
   manf = 'Anheuser-Busch'
GROUP BY bar, beer;
```

→ Select sum(price) from **canDrink natural join Beers** where drink='David Wu' and manf= 'Anheuser-Busch'

# Indexes

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

- If we are doing mostly insertion, very few queries, then we do not want an index

# Summary of chapter 8

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

# Classroom Demo

- Create a view who shows all CS students, called Csstudents.
- Create a view of top cs students.

- How to insert or delete this view (CSstudents) ?
- Use **INSTEAD OF** trigger.

# a view to show all CS students, called Csstudents.

```
Create view CSstudents as
select sc.sid,name,cid,cname,grade
from students,sc
where dept='cs' and
students.sid=sc.sid;
select * from csstudents;
insert into csstudents(sid,name)
values(11,'mary');
```

# A view of top cs students based on the view of CSstudents

```
create view CStopstudents as
select sid, name, avg(grade) as GPA
from CSstudents
group by sid,name
having avg(grade) > 70;

Drop view Cstopstudents;
```

# Instead of Trigger

```
create trigger CSstudentInsert
instead of insert on CSstudents
for each row
begin
insert into students values
(new.sid,new.name,'cs',null);
insert into sc values
(new.sid,new.cid,1,new.cname,new.grade
);
end;
```

# Instead of Trigger

```
create trigger CSstudentdelete
instead of delete on CSstudents
for each row
begin
 delete from students where sid=old.sid;
 delete from sc where sid=old.sid;
end;
```

# Test

- insert into csstudents(sid,name) values(11,'mary');
- select * from CSstudents;
- Select * from **students**;
- delete from CSstudents where sid=1;
- select * from CSstudents;
- Select * from **students**;