Lecture 03
Views, Constraints
Tuesday, April 14, 2009
Announcements

• Homework 1 was due a few minutes ago…

• Homework 2: due next week

• Homework 3: to be posted by tomorrow, due in two weeks
Outline

• Database modifications, Integrity constraints, triggers (Chapter 5)

• Views: (Chapters 3.6, 25.8, 25.9)
  – Some material discussed today is not in the book
Modifying the Database

Three kinds of modifications

- Insertions
- Deletions
- Updates

Sometimes they are all called “updates”
Inserting One Record

General form:

```
INSERT INTO R(A1, ..., An) VALUES (v1, ..., vn)
```

Example: Insert a new purchase to the database:

```
INSERT INTO Purchase(buyer, seller, product, store) VALUES ('Joe', 'Fred', 'wakeup-clock-espresso-machine', 'The Sharper Image')
```

Missing attribute → NULL.
Bulk Insertions

Purchase(buyer, seller, product, store)
Product(name, price)

```
INSERT INTO Product(name)
SELECT DISTINCT Purchase.product
FROM Purchase
WHERE Purchase.store = 'Joe'
```
Deletions

Purchase(buyer, seller, product, store)
Product(name, price)

```
DELETE FROM Purchase
WHERE seller = 'Joe' AND product = 'Brooklyn Bridge'
```

**SQL Fact**: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.
Updates

Purchase(buyer, seller, product, store)
Product(name, price)

```
UPDATE  Product
SET     price = 29.95
WHERE   name = ‘gizmo’
```

```
UPDATE  Product
SET     price = price/2
WHERE   name IN
        (SELECT product
         FROM    Purchase
         WHERE   store=‘Joe’);
```
Data Definition in SQL

• Data Manipulation Language: DML
  – Query and modify the database
  – What we have seen so far

• Data Definition Language: DDL
  – Create, delete, modify tables
  – Constraints
Creating Tables

CREATE TABLE Purchase(
    buyer VARCHAR(50),
    seller VARCHAR(50),
    product CHAR(20),
    store VARCHAR(30)
);

CREATE TABLE Product(
    name CHAR(20),
    price INT
);

Purchase(buyer, seller, product, store)
Product(name, price)

INT, SHORTINT, BIT(1), BIT(5), DATETIME, etc, etc
Deleting or Modifying a Table

```
DROP Product;
```

Exercise with care!!

```
ALTER TABLE Product
    ADD category VARCHAR(30);

ALTER TABLE Purchase
    DROP seller;
```

This changes the database *schema*. What happens to the data?
Default Values

Specifying default values:

```sql
CREATE TABLE Purchase(
    buyer VARCHAR(50),
    seller VARCHAR(50) DEFAULT 'Johnny',
    product CHAR(20),
    store VARCHAR(30) DEFAULT 'Wal-Mart'
);
```

The default of defaults: NULL
Indexes

**REALLY** important to speed up query processing time.

Person (name, age, city)

```sql
SELECT * 
FROM   Person 
WHERE  name = 'Smith'
```

May take too long to scan the entire Person table

```sql
CREATE INDEX myindex05 ON Person(name)
```

Now, when we rerun the query it will be much faster
B+ Tree Index

We will discuss them in detail in a later lecture.
Creating Indexes

Indexes can be created on more than one attribute:

Example:

```
CREATE INDEX doubleindex ON Person (age, city)
```

Helps in:

```
SELECT * FROM Person WHERE age = 55 AND city = 'Seattle'
```

and even in:

```
SELECT * FROM Person WHERE age = 55
```

But not in:

```
SELECT * FROM Person WHERE city = 'Seattle'
```
Constraints in SQL

• A constraint = a property that we’d like our database to hold

• The system will enforce the constraint by taking some actions:
  – forbid an update
  – or perform compensating updates
Constraints in SQL:
• Keys, foreign keys
• Attribute-level constraints
• Tuple-level constraints
• Global constraints: assertions

The more complex the constraint, the harder it is to check and to enforce
Keys

CREATE TABLE Product (  
    name CHAR(30) PRIMARY KEY,  
    price INT)  

OR:  

CREATE TABLE Product (  
    name CHAR(30),  
    price INT,  
    PRIMARY KEY (name))
Keys with Multiple Attributes

```
CREATE TABLE Product ( 
    name CHAR(30), 
    category VARCHAR(20), 
    price INT, 
    PRIMARY KEY (name, category))
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Gadget</td>
<td>10</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>20</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Photo</td>
<td>30</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Gadget</td>
<td>40</td>
</tr>
</tbody>
</table>

Product(name, category, price)
Other Keys

CREATE TABLE Product (  
    productID CHAR(10),  
    name CHAR(30),  
    category VARCHAR(20),  
    price INT,  
    PRIMARY KEY (productID),  
    UNIQUE (name, category))

There is at most one PRIMARY KEY;  
there can be many UNIQUE
Foreign Key Constraints

CREATE TABLE Purchase (  
    buyer CHAR(30),  
seller CHAR(30),  
product CHAR(30) REFERENCES Product(name),  
store VARCHAR(30))
## Product

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

## Purchase

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>
Foreign Key Constraints

```
CREATE TABLE Purchase(
  buyer VARCHAR(50),
  seller VARCHAR(50),
  product CHAR(20),
  category VARCHAR(20),
  store VARCHAR(30),
  FOREIGN KEY (product, category)
    REFERENCES Product(name, category)
);
```

Purchase(buyer, seller, product, category, store)
Product(name, category, price)
What happens during updates?

Types of updates:
- In Purchase: insert/update
- In Product: delete/update
What happens during updates?

- SQL has three policies for maintaining referential integrity:
  - **Reject** violating modifications (default)
  - **Cascade**: after a delete/update do a delete/update
  - **Set-null** set foreign-key field to NULL
Constraints on Attributes and Tuples

Attribute level constraints:

```sql
CREATE TABLE Purchase ( . . .
store VARCHAR(30) NOT NULL, . . . )
```

```sql
CREATE TABLE Product ( . . .
price INT CHECK (price >0 and price < 999))
```

Tuple level constraints:

```sql
. . . CHECK (price * quantity < 10000) . . .
```
Comments on Constraints

• Can give them names, and alter later

• We need to understand exactly when they are checked

• We need to understand exactly what actions are taken if they fail
Semantic Optimization using Constraints

Purchase(buyer, seller, product, store)
Product(name, price)

```
SELECT Purchase.store 
FROM   Product, Purchase
WHERE  Product.name=Purchase.product
```

Why?
Triggers

Trigger = a procedure invoked by the DBMS in response to an update to the database

Trigger = Event + Condition + Action
Triggers in SQL

• Event = INSERT, DELETE, UPDATE

• Condition = any WHERE condition
  – Refers to the old and the new values

• Action = more inserts, deletes, updates
  – May result in cascading effects!
CREATE TRIGGER InsertPromotions AFTER UPDATE OF price ON Product

REFERENCING
   OLD AS x
   NEW AS y

FOR EACH ROW
WHEN (x.price > y.price)
INSERT INTO Promotions(name, discount)
VALUES x.name,
   (x.price-y.price)*100/x.price
EVENTS

INSERT, DELETE, UPDATE

• Trigger can be:
  – AFTER event
  – INSTEAD of event
Scope

- **FOR EACH ROW** = trigger executed for every row affected by update
  - OLD ROW
  - NEW ROW

- **FOR EACH STATEMENT** = trigger executed once for the entire statement
  - OLD TABLE
  - NEW TABLE
Statement Level Trigger

CREATE TRIGGER avg-price INSTEAD OF UPDATE OF price ON Product

REFERENCING
   OLD_TABLE AS OldStuff
   NEW_TABLE AS NewStuff

FOR EACH STATEMENT
WHEN (1000 < (SELECT AVG (price)
   FROM ((Product EXCEPT OldStuff) UNION NewStuff))
DELETE FROM Product
   WHERE (name, price, company) IN OldStuff;
INSERT INTO Product
   (SELECT * FROM NewStuff)
Triggers v.s. Integrity Constraints

Active database = a database with triggers

- Triggers can be used to enforce ICs
- Triggers are more general: alerts, log events
- But hard to understand: recursive triggers
- Syntax is vendor specific, and may vary significantly
  - Postgres has *rules* in addition to *triggers*
Views

• A view = a relation computed from other relations using a query

• May be stored (*materialized*), or computed on demand (*virtual*)

• Views have many kinds of applications
Example

Purchase(customer, product, store)
Product(pname, price)

```sql
CREATE VIEW CustomerPrice AS
    SELECT x.customer, y.price
    FROM Purchase x, Product y
    WHERE x.product = y.pname
```

CustomerPrice(customer, price) “virtual table”
SELECT u.customer, v.store
FROM CustomerPrice u, Purchase v
WHERE u.customer = v.customer AND u.price > 100
Types of Views

• **Virtual views:**
  – Used in databases
  – Computed only on-demand – slow at runtime
  – Always up to date

• **Materialized views**
  – Used in data warehouses
  – Pre-computed offline – fast at runtime
  – May have stale data
  – Indexes *are* materialized views (read book)
Querying Virtual Views

• Have views $V_1, V_2, \ldots, V_n$

• Query $Q$ refers to these views

• Need to inline view definitions in the query

• Then need to simplify the expression
Queries Over Virtual Views

Purchase(customer, product, store)
Product(pname, price)
CustomerPrice(customer, price)

Query:

SELECT u.customer, v.store
FROM CustomerPrice u, Purchase v
WHERE u.customer = v.customer AND u.price > 100
Queries Over Virtual Views

Purchase(customer, product, store)
Product(pname, price)
CustomerPrice(customer, price)

Modified query:

```sql
SELECT u.customer, v.store
FROM (SELECT x.customer, y.price
     FROM Purchase x, Product y
     WHERE x.product = y.pname) u, Purchase v
WHERE u.customer = v.customer AND
    u.price > 100
```
Queries Over Virtual Views

Purchase(customer, product, store)
Product(pname, price)
CustomerPrice(customer, price)

Modified and unnested query:

```
SELECT x.customer, v.store
FROM Purchase x, Product y, Purchase v,
WHERE x.customer = v.customer AND
    y.price > 100 AND
    x.product = y.pname
```
Another Example

Purchase(customer, product, store)
Product(pname, price)
CustomerPrice(customer, price)

```
SELECT DISTINCT u.customer, v.store
FROM CustomerPrice u, Purchase v
WHERE u.customer = v.customer AND u.price > 100
```
Answer

Purchase(customer, product, store)
Product(pname, price)
CustomerPrice(customer, price)

```
SELECT DISTINCT x.customer, v.store
FROM     Purchase x, Product y, Purchase v,
WHERE    x.customer = v.customer  AND
         y.price > 100 AND
         x.product = y.pname
```
Set v.s. Bag Semantics

SELECT DISTINCT a,b,c
FROM    R, S, T
WHERE   . . .

SELECT  a,b,c
FROM    R, S, T
WHERE   . . .

Set semantics

Bag semantics
Inlining Queries: Sets/Sets

```
SELECT DISTINCT a,b,c
FROM (SELECT DISTINCT u,v
     FROM R,S
     WHERE ...), T
WHERE ...
```

```
SELECT DISTINCT a,b,c
FROM R, S, T
WHERE ...
```
Inlining Queries: Sets/Bags

```
SELECT DISTINCT a,b,c
FROM      (SELECT u,v
            FROM R,S
            WHERE ...),  T
WHERE    ...
```

```
SELECT DISTINCT a,b,c
FROM      R, S, T
WHERE    ...
```
Inlining Queries: Bags/Bags

```
SELECT  a,b,c
FROM    (SELECT u,v
          FROM R,S
          WHERE ...), T
WHERE   ...
```

```
SELECT  a,b,c
FROM    R, S, T
WHERE   ...
```
Inlining Queries: Bags/Sets

SELECT a,b,c
FROM (SELECT DISTINCT u,v
      FROM R,S
      WHERE ...), T
WHERE ...

NO
Applications of Virtual Views

• Physical data independence
  – Vertical data partitioning
  – Horizontal data partitioning

• Security
  – The view reveals only what the users are allowed to know

• Materialized views for query speedup
  – Indexes, denormalization, semantic caching
# Vertical Partitioning

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Resume</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Huston</td>
<td>Clob1…</td>
<td>Blob1…</td>
</tr>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
<td>Clob2…</td>
<td>Blob2…</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
<td>Clob3…</td>
<td>Blob3…</td>
</tr>
<tr>
<td>234234</td>
<td>Ann</td>
<td>Portland</td>
<td>Clob4…</td>
<td>Blob4…</td>
</tr>
</tbody>
</table>

### T1

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Huston</td>
</tr>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### T2

<table>
<thead>
<tr>
<th>SSN</th>
<th>Resume</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Clob1…</td>
</tr>
<tr>
<td>345345</td>
<td>Clob2…</td>
</tr>
</tbody>
</table>

### T3

<table>
<thead>
<tr>
<th>SSN</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Blob1…</td>
</tr>
<tr>
<td>345345</td>
<td>Blob2…</td>
</tr>
</tbody>
</table>
Vertical Partitioning

CREATE VIEW Resumes AS
SELECT T1.ssn, T1.name, T1.address,
    T2.resume, T3.picture
FROM T1,T2,T3
WHERE T1.ssn=T2.ssn and T2.ssn=T3.ssn

When do we use vertical partitioning?
Vertical Partitioning

```
SELECT address
FROM    Resumes
WHERE   name = 'Sue'
```

Which of the tables T1, T2, T3 will be queried by the system?
Vertical Partitioning

When to do this:

• When some fields are large, and rarely accessed
  – E.g. Picture

• In distributed databases
  – Customer personal info at one site, customer profile at another

• In data integration
  – T1 comes from one source
  – T2 comes from a different source
### Horizontal Partitioning

**Customers**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Huston</td>
<td>USA</td>
</tr>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
<td>USA</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
<td>USA</td>
</tr>
<tr>
<td>234234</td>
<td>Ann</td>
<td>Portland</td>
<td>USA</td>
</tr>
<tr>
<td>--</td>
<td>Frank</td>
<td>Calgary</td>
<td>Canada</td>
</tr>
<tr>
<td>--</td>
<td>Jean</td>
<td>Montreal</td>
<td>Canada</td>
</tr>
</tbody>
</table>

**CustomersInHuston**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Huston</td>
<td>USA</td>
</tr>
</tbody>
</table>

**CustomersInSeattle**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
<td>USA</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
<td>USA</td>
</tr>
</tbody>
</table>

**CustomersInCanada**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>Frank</td>
<td>Calgary</td>
<td>Canada</td>
</tr>
<tr>
<td>--</td>
<td>Jean</td>
<td>Montreal</td>
<td>Canada</td>
</tr>
</tbody>
</table>
Horizontal Partitioning

CREATE VIEW Customers AS
    CustomersInHuston
    UNION ALL
    CustomersInSeattle
    UNION ALL
    ...

Horizontal Partitioning

```
SELECT name
FROM Customers
WHERE city = 'Seattle'
```

Which tables are inspected by the system?

WHY ???
Horizontal Partitioning

Better:

```
CREATE VIEW Customers AS
 (SELECT * FROM CustomersInHuston
  WHERE city = 'Huston')
 UNION ALL
 (SELECT * FROM CustomersInSeattle
  WHERE city = 'Seattle')
 UNION ALL
 . . .
```
Horizontal Partitioning

```sql
SELECT name
FROM Customers
WHERE city = 'Seattle'
```

```sql
SELECT name
FROM CustomersInSeattle
```
Horizontal Partitioning

Applications:

• Optimizations:
  – E.g. archived applications and active applications

• Distributed databases

• Data integration
Views and Security

Customers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW PublicCustomers
SELECT Name, Address
FROM Customers

Fred is allowed to see this

Fred is not allowed to see this
Views and Security

Customers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

John is not allowed to see balances >0

CREATE VIEW BadCreditCustomers
SELECT *
FROM Customers
WHERE Balance < 0
Materialized Views for Query Speedup

Examples:

• Indexes
  – Rule of thumb: an index is a view!

• Denormalization
  – E.g. Join indexes
Indexes are Materialized Views

Product(pid, name, weight, price, …)  (big)

CREATE INDEX W ON Product(weight)
CREATE INDEX P ON Product(price)

W(pid, weight)  (smaller)
P(pid, price)

SELECT weight, price
FROM Product
WHERE weight > 10
and price < 100

SELECT x.weight, y.price
FROM W x, P y
WHERE x.weight > 10
   and y.price < 100
   and x.pid = y.pid
Denormalization

Real example from Graduate Admissions

<table>
<thead>
<tr>
<th>Application(id, name, school)</th>
<th>GRE(id, score, year)</th>
</tr>
</thead>
</table>

Common query

```sql
SELECT x.id, max(y.score) 
FROM Application x, GRE y 
WHERE x.id=y.id 
GROUP BY x.id
```

```
CREATE VIEW AppWithGRE AS 
SELECT x.id,x.name, x.school, y.score, y.year 
FROM Application x, GRE y 
WHERE x.id=y.id
```

```
SELECT x.id, max(y.score) 
FROM Application x, GRE y 
WHERE x.id=y.id
```

VERY SLOW!

Synchronize once per night
Semantic Caching

• Queries Q1, Q2, … have been executed, and their results are stored in main memory
• Now we need to compute a new query Q
• Sometimes we can use the prior results in answering Q
• This, too, is a form of query rewriting using views (why ?)
Technical Challenges in Managing Views

• Updating views
• Simplifying queries over virtual views
• Synchronizing materialized views
• Query answering using views
CREATE VIEW Expensive-Product AS
SELECT pname
FROM Product
WHERE price > 100

INSERT INTO Expensive-Product VALUES(‘Gizmo’)

INSERT INTO Product VALUES(‘Gizmo’, NULL)
Updating Views

Purchase(customer, product, store)
Product(pname, price)

CREATE VIEW AcmePurchase AS
SELECT customer, product
FROM Purchase
WHERE store = 'AcmeStore'

INSERT INTO Toy-Product VALUES('Joe', 'Gizmo')
INSERT INTO Product VALUES('Joe', 'Gizmo', NULL)

Note this
Updateable view
Updating Views

Non-updateable view

Most views are non-updateable

INSERT INTO CustomerPrice
VALUES(‘Joe’, 200)

CREATE VIEW CustomerPrice AS
SELECT x.customer, y.price
FROM Purchase x, Product y
WHERE x.product = y.pname

Purchase(customer, product, store)
Product(pname, price)
Simplifying Queries over Virtual Views

• After the views are expanded in the query’s body, the resulting expression is often redundant and inefficient.

• Query minimization = the problem of rewriting a query into an equivalent query that is smaller (and, hence, more efficient).
Customers who ordered a cheap, lightweight product

Order(cid, pid, date)
Product(pid, name, weight, price)

CREATE VIEW CheapOrders AS
    SELECT x.cid, x.pid, x.date, y.name, y.price
    FROM Order x, Product y
    WHERE x.pid = y.pid and y.price < 99

CREATE VIEW LightOrders AS
    SELECT a.cid, a.pid, a.date, b.name, b.price
    FROM Order a, Product b
    WHERE a.pid = b.pid and b.weight < 15

SELECT u.cid
FROM CheapOrders u, LightOrders v
WHERE u.pid = v.pid
    and u.cid = v.cid
Query Minimization

**Order**(*cid*, *pid*, *date*)
**Product**(*pid*, *name*, *weight*, *price*)

CREATE VIEW CheapOrders AS
SELECT x.cid, x.pid, x.date, y.name, y.price
FROM Order x, Product y
WHERE x.pid = y.pid and y.price < 99

CREATE VIEW LightOrders AS
SELECT a.cid, a.pid, a.date, b.name, b.price
FROM Order a, Product b
WHERE a.pid = b.pid and b.weight < 15

SELECT u.cid
FROM CheapOrders u,
    LightOrders v
WHERE u.pid = v.pid
    and u.cid = v.cid

SELECT a.cid
FROM Order x, Product y
WHERE Order a, Product b
WHERE . . . .
Query Minimization under Bag Semantics

Rule 1: If $x, y$ are tuple variables over the same table and $x.id = y.id$, then combine $x, y$ into a single variable

Rule 2: If $x$ ranges over $S$, $y$ ranges over $T$, the only condition on $y$ is $x.fk = y.key$, and $y$ is not used anywhere else, then remove $T$ (and $y$) from the query
SELECT a.cid
FROM Order x, Product y, Order a, Product b
WHERE x.pid = y.pid and a.pid = b.pid
and y.price < 99 and b.weight < 15
and x.cid = a.cid and x.pid = a.pid

x = a

SELECT a.cid
FROM Order x, Product y, Product b
WHERE x.pid = y.pid and x.pid = b.pid
and y.price < 99 and b.weight < 15

y = b

SELECT a.cid
FROM Order x, Product y
WHERE x.pid = y.pid and
      y.price < 99 and x.weight < 15
Query Minimization under Set Semantics

```
SELECT DISTINCT x.pid
FROM   Product x, Product y, Product z
WHERE  x.category = y.category and y.price > 100
       and x.category = z.category and z.price > 500
       and z.weight > 10
```

Same as:

```
SELECT DISTINCT x.pid
FROM   Product x, Product z
WHERE  x.category = z.category and z.price > 500
       and z.weight > 10
```
Query Minimization under Set Semantics

Rule 3: Let $Q'$ be the query obtained by removing the tuple variable $x$ from $Q$. If there exists a homomorphism from $Q$ to $Q'$ and both $Q$, $Q'$ have set semantics, then $Q'$ is equivalent to $Q$. Hence one can safely remove $x$. 
Definition of a Homomorphism

A **homomorphism** from Q to Q’ is a mapping h from the tuple variables of Q to the tuple variables of Q’ such that:

For every predicate P in the WHERE clause of Q, the predicate \( h(P) \) is logically implied by the WHERE clause of Q’

**Theorem** If there exists a homomorphism from Q’ to Q, then Q is contained in Q’.

If there exists homomorphisms both from Q’ to Q and from Q to Q’, then Q and Q’ are logically equivalent.
Homomorphism

Q

SELECT DISTINCT x.pid
FROM Product x, Product y, Product z
WHERE x.category = y.category and y.price > 100
    and x.category = z.category and z.price > 500
    and z.weight > 10

H(x) = x’,   H(y) = H(z) = z’

Q’

SELECT DISTINCT x’.pid
FROM Product x’, Product z’
WHERE x’.category = z’.category and z’.price > 500
    and z’.weight > 10
Synchronizing Materialized Views

• Immediate synchronization = after each update
• Deferred synchronization
  – Lazy = at query time
  – Periodic
  – Forced = manual

Which one is best for: indexes, data warehouses, replication?
CREATE VIEW FullOrder AS
  SELECT x.cid, x.pid, x.date, y.name, y.price
  FROM Order x, Product y
  WHERE x.pid = y.pid

UPDATE Product
  SET price = price / 2
  WHERE pid = '12345'

UPDATE FullOrder
  SET price = price / 2
  WHERE pid = '12345'

No need to recompute the entire view!
Incremental View Update

Product(pid, name, category, price)

CREATE VIEW Categories AS
SELECT DISTINCT category
FROM Product

DELETE Product
WHERE pid = ‘12345’

DELETE Categories
WHERE category in
(SELECT category
FROM Product
WHERE pid = ‘12345’)

It doesn’t work! Why? How can we fix it?
Answering Queries Using Views

• We have several materialized views:
  – V1, V2, …, Vn

• Given a query Q
  – Answer it by using views instead of base tables

• Variation: *Query rewriting using views*
  – Answer it by rewriting it to another query first

• Example: if the views are indexes, then we rewrite the query to use indexes
Query Rewriting Using Views

Purchase(buyer, seller, product, store)
Person(pname, city)

CREATE VIEW SeattleView AS
SELECT y.buyer, y.seller, y.product, y.store
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND x.pname = y.buyer

Goal: rewrite this query in terms of the view

SELECT y.buyer, y.seller
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND x.pname = y.buyer AND y.product = 'gizmo'
Query Rewriting Using Views

```
SELECT y.buyer, y.seller
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND x..pname = y.buyer AND y.product='gizmo'
```

```
SELECT buyer, seller
FROM SeattleView
WHERE product= 'gizmo'
```
Rewriting is not always possible

CREATE VIEW DifferentView AS
SELECT y.buyer, y.seller, y.product, y.store
FROM Person x, Purchase y, Product z
WHERE x.city = 'Seattle' AND
  x.pname = y.buyer AND
  y.product = z.name AND
  z.price < 100

SELECT y.buyer, y.seller
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND
  x.pname = y.buyer AND
  y.product = 'gizmo'

SELECT buyer, seller
FROM DifferentView
WHERE product = 'gizmo'

“Maximally contained rewriting”