Principles of Database Systems CSE 544

Lecture #3

Views and Constraints

Announcements

- Regular lecture:
  - Monday, April 2nd
  - 2nd Paper review due (What Goes UP, skip 5-7)
- Cancelled:
  - Lecture on Wednesday, April 4

- Project:
  - Form teams by April 1st (Sunday)
  - Send email to Paris and me: team members (cc them), team name, a tentative project (or several)

Reading Material

- Views:
  - *Query answering using views*, by Halevy
  - Book: 3.6
- Constraints:
  - Book 3.2, 3.3, 5.8

Views

Outline:

- View basics, including examples
- Paper and more
  - Applications
  - Query rewriting v.s. query answering
  - Maximal contained rewriting

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

CustomerPrice(customer, price)

View Basics

Views are named relations, defined by a query

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

CustomerPrice(customer, price) = a "virtual table"

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

CustomerPrice(customer, price)

View Basics

"Find all stores visited by customers who bought some product over $100"

We can later use the view:

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

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

CustomerPrice(customer, price)

View Basics

"Find all stores visited by customers who bought some product over $100"

View:

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

Query:

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

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

CustomerPrice(customer, price)

View Basics

"Find all stores visited by customers who bought some product over $100"

Modified query:

```
SELECT  DISTINCT u.customer, v.store
FROM  (SELECT  DISTINCT 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
```

Next, unnest the query…

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

CustomerPrice(customer, price)

View Basics

"Find all stores visited by customers who bought some product over $100"

Modified and unnested query:

```
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
```

Note: Purchase occurs twice (why?)

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

CustomerPrice(customer, price)

Practice at Home…

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

??

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

CustomerPrice(customer, price)

Answer

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

```
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
```

Types of Views

- Virtual views:
  - Pros/cons ?
- Materialized views
  - Pros/cons ?

Types of Views

- Virtual views:
  - Used in databases
  - Computed only on-demand – slow at runtime
  - Always up to date
- Materialized views
  - Used in databases and data warehouses
  - Pre-computed offline – fast at runtime
  - May have stale data *or* expensive synchronization

Basic Usage of a View

- Virtual view:
  - View inlining, or query modification
  - Here the view acts like a macro for a query

- Materialized view:
  - Use the view as derived data
  - Save the cost of computing it

Example: Finding Witnesses

Product (pname, price, category, manufacturer)

Company (cname, country)

For each country, find its most expensive product(s)

Example: Finding Witnesses

Product (pname, price, category, manufacturer)

Company (cname, country)

For each country, find its most expensive product(s)

Finding the maximum price is easy…

```
SELECT x.country, max(y.price)
FROM   Company x, Product y
WHERE  x.cname = y.manufacturer
GROUP BY  x.country
```

But we need the *witnesses*, i.e. the products with max price

Product (pname, price, category, manufacturer)
Company (cname, country)

Example: Finding Witnesses

To find witnesses, create a view with the maximum price

```
CREATE VIEW CountryMaxPrice AS
  SELECT x.country, max(y.price) as mprice
  FROM   Company x, Product y
  WHERE  x.cname = y.manufacturer
  GROUP BY  x.country
```

Is this virtual or materialized?

Product (pname, price, category, manufacturer)
Company (cname, country)

Example: Finding Witnesses

To find witnesses, create a view with the maximum price

```
CREATE VIEW CountryMaxPrice AS
  SELECT x.country, max(y.price) as mprice
  FROM   Company x, Product y
  WHERE  x.cname = y.manufacturer
  GROUP BY  x.country
```

Is this virtual or materialized?

Next, use it to find the product that matches that price

```
SELECT u.country, v.pname, v.price
FROM Company u, Product v, CountryMaxPrice AS p
WHERE  u.country = p.country and v.price = p.mprice
```

Product (pname, price, category, manufacturer)
Company (cname, country)

Example: Finding Witnesses

If the view is reused, *and* performance is an issue, then:

```
CREATE TABLE CountryMaxPrice AS
  SELECT x.country, max(y.price) as mprice
  FROM   Company x, Product y
  WHERE  x.cname = y.manufacturer
  GROUP BY  x.country
```

Is this virtual or materialized?

```
SELECT u.country, v.pname, v.price
FROM Company u, Product v, CountryMaxPrice p
WHERE  u.country = p.country and v.price = p.mprice
```

You may also want to create indexes on CountryMaxPrice

Product (pname, price, category, manufacturer)
Company (cname, country)

Example: Finding Witnesses

For one-time use, don't create a view, but instead:

```
SELECT u.country, v.pname, v.price
FROM Company u, Product v,
          (SELECT x.country, max(y.price) as mprice
           FROM    Company x, Product y
           WHERE  x.cname = y.manufacturer
           GROUP BY  x.country) AS p
WHERE  u.country = p.country and v.price = p.mprice
```

Or:

```
WITH CountryMaxPrice AS
          (SELECT x.country, max(y.price) as mprice
           FROM    Company x, Product y
           WHERE  x.cname = y.manufacturer
           GROUP BY  x.country)
SELECT u.country, v.pname, v.price
FROM Company u, Product v,  CountryMaxPrice p
WHERE  u.country = p.country and v.price = p.mprice
```

Product (pname, price, category, manufacturer)
Company (cname, country)

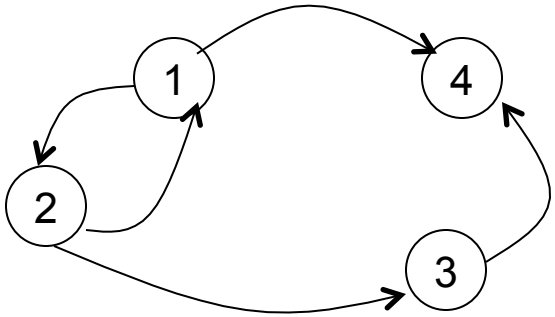
Example: Finding Witnesses

Finally, here's a totally different solution:

```
SELECT x.country, y.pname, y.price
FROM   Company x, Product y
WHERE  x.cname = y.manufacturer
       and  y.price >=
                ALL (SELECT z.price
                     FROM Product z
                     WHERE x.cname = z.manufacturer)
```

Closer Look at Query Modification

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4

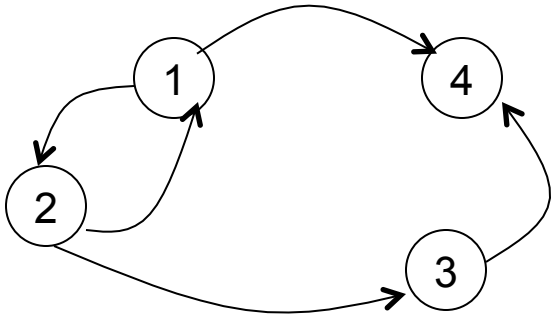
What do these queries return ?

q(x) :- R(x,y)

q(x) :- R(x,y) ∧ R(y,z) ∧ R(z,u)

Closer Look at Query Modification

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4

What do these queries return ?

q(x) :- R(x,y)

Nodes that have at least one child: {1,2,3}

q(x) :- R(x,y) ∧ R(y,z) ∧ R(z,u)

Nodes that have a great-grand-child: {1,2}

Closer Look at Query Modification

R encodes a graph

Consider the views:

```
V1(x,y) :- R(x,z),R(z,y)
```

Expand this query:

```
Q(x,y) :- V1(x,z),V1(z,y)
```

Closer Look at Query Modification

R encodes a graph

Consider the views:

```
V1(x,y) :- R(x,z),R(z,y)
```

Answer:

```
Q(x,y) :-
    R(x,z1),R(z1,z2),R(z2,z3),R(z3,y)
```

Expand this query:

```
Q(x,y) :- V1(x,z),V1(z,y)
```

Closer Look at Query Modification

R encodes a graph

Now consider the following views:

```
V1(x,y) :- R(x,z),R(z,y)
V2(x,y) :- V1(x,z),V1(z,y)
V3(x,y) :- V2(x,z),V2(z,y)
```

Expand this query:

```
Q(x,y) :- V3(x,z),V3(z,y)
```

Closer Look at Query Modification

R encodes a graph

Now consider the following views:

```
V1(x,y) :- R(x,z),R(z,y)
V2(x,y) :- V1(x,z),V1(z,y)
V3(x,y) :- V2(x,z),V2(z,y)
```

Expand this query:

```
Q(x,y) :- V3(x,z),V3(z,y)
```

Answer:

```
Q(x,y) :-
    R(x,z1),R(z1,z2),R(z2,z3),R(z3,z4),
    R(z4,z5),R(z5,z6),R(z6,z7),R(z7,z8),
    R(z8,z9),R(z9,z10),R(z10,z11),R(z11,z12),
    R(z12,z13),R(z13,z14),R(z14,z15),R(z15,y)
```

Lesson: expanding multiple levels of views → exponential size increase

Applications of Views

What applications does the paper describe?

Applications of Views

What applications does the paper describe?

- Query optimization
  - E.g. Indexes
- Physical and logical data independence
  - E.g. de-normalization, data partitioning
- Semantic caching
- Data integration

Indexes

REALLY important to speed up query processing time.

Person (pid, name, age, city)

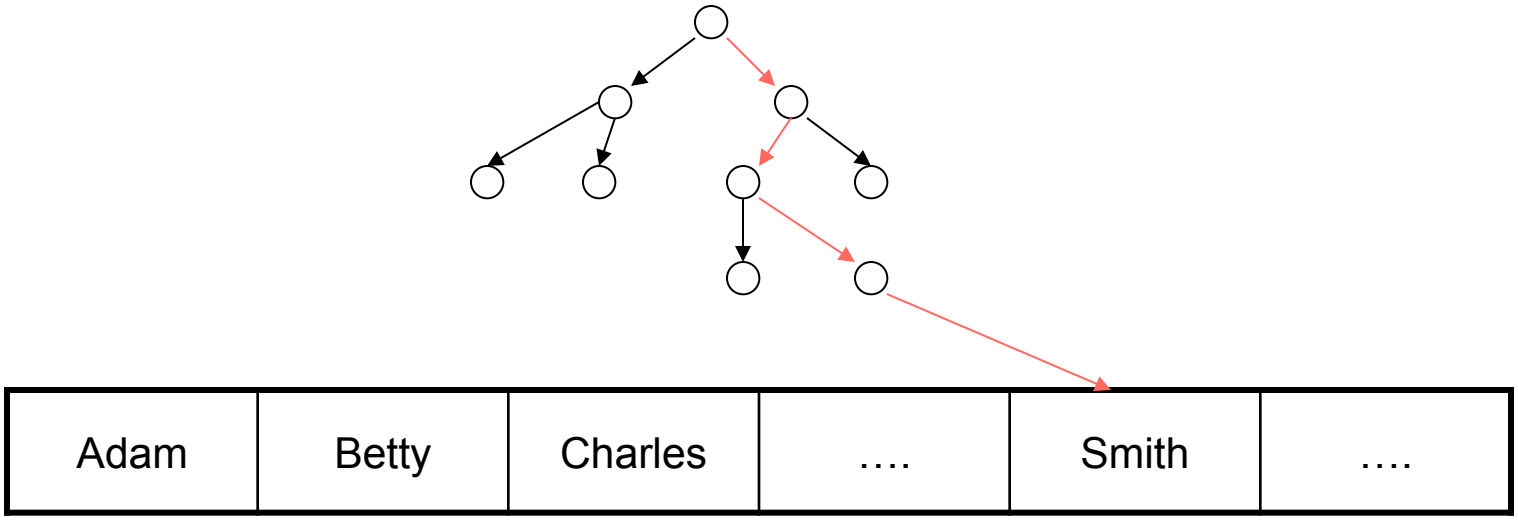
```
SELECT *
FROM   Person
WHERE  name = 'Smith'
```

May take too long to scan the entire Person table

```
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.

Person(pid, name, age, city)

Creating Indexes

Indexes can be created on more than one attribute:

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

For which of the queries below is this index helpful?

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

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

```
SELECT *
FROM   Person
WHERE city = 'Seattle'
```

Person(pid, name, age, city)

Creating Indexes

Indexes can be created on more than one attribute:

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

For which of the queries below is this index helpful?

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

YES

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

YES

```
SELECT *
FROM   Person
WHERE city = 'Seattle'
```

NO

Person(pid, name, age, city)

Indexes are Materialized Views

```
CREATE INDEX  W
    ON Person(age)
CREATE INDEX  P
    ON Person(city)
```

If W and P are “views”, what is their schema?
Which query defines them?

Person(pid, name, age, city)

Indexes are Materialized Views

```
CREATE INDEX  W
    ON Person(age)
CREATE INDEX  P
    ON Person(city)
```

Each index is a relation:
(index value, record id)

Some DBMS make very advanced use…

Indexes as LAV:

```
CREATE VIEW  W  AS
       SELECT  age, pid
       FROM    Person y
CREATE VIEW  P  AS
       SELECT  city, pid
       FROM    Person y
```

Person(pid, name, age, city)

Indexes are Materialized Views

```
CREATE INDEX  W
    ON Person(age)
CREATE INDEX  P
    ON Person(city)
```

```
SELECT age, city
FROM Person
WHERE age > 22
      and city LIKE 'S%'
```

"Covering indexes": When the query uses *only* the indexes

Indexes as LAV:

```
CREATE VIEW  W  AS
      SELECT  age, pid
      FROM    Person y
CREATE VIEW  P  AS
      SELECT  city, pid
      FROM    Person y
```

```
SELECT x.age, y.city
FROM W x, P y
WHERE x.age > 22
      and y.city LIKE 'S%
      and x.pid = y.pid
```

Denormalization

- Scenario: we have a relational schema that is in BCNF (recall: this means only the key implies any other attribute(s))

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

- But we often need to join these two relations, so we compute their join

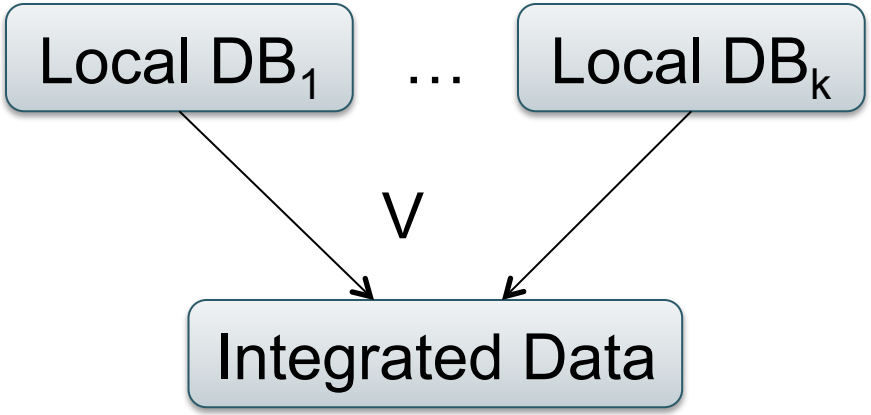
Denormalization

```
CREATE Table CustomerPurchase  AS
        SELECT  x.pid, x.customer, x.store, y.pname, y.price
        FROM    Purchase x, Product y
        WHERE   x.product = y.pname
```

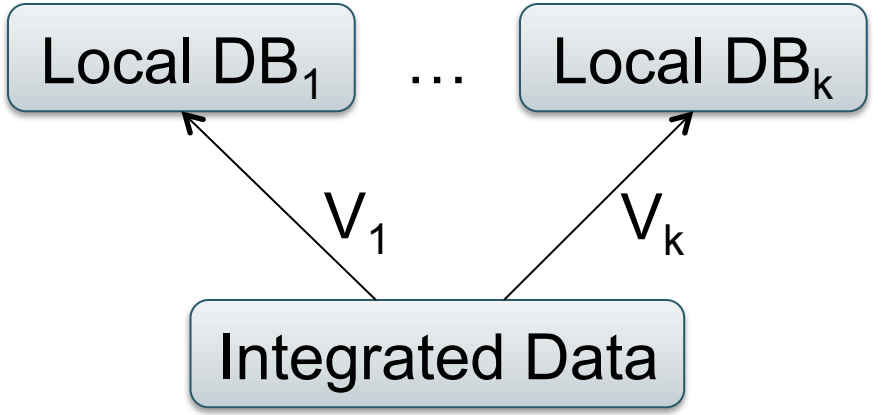
- This table is not in BCNF (why not?)
- But that's OK, the application still sees the original two relations. How?

Purchase(pid, customer, product, store) – a view…
Product(pname, price) – a view…

Data Integration Terminology



Global as View



Local as View

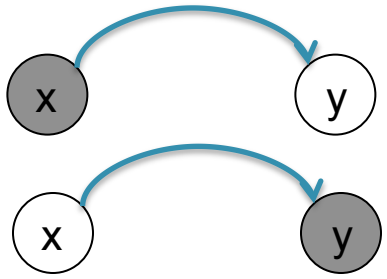
Which one needs query expansion,
which one needs query answering using views ?

[Duschka&Genesereth'97]

Query Rewriting Using Views

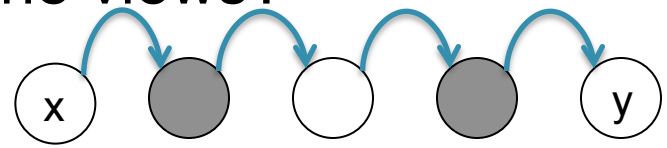
Suppose you only have these two views:

```
v1(x,y) :- black(x), edge(x,y)
v2(x,y) :- edge(x,y), black(y)
```



Can you rewrite this query in terms of the views?

```
q(x,y) :- edge(x,z1), black(z1),
          edge(z1,z2),edge(z2,z3)
          black(z3), edge(z3,y)
```



NOTE:

○ means "any color"

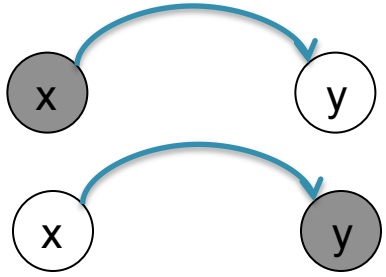
● means "black"

[Duschka&Genesereth'97]

Query Rewriting Using Views

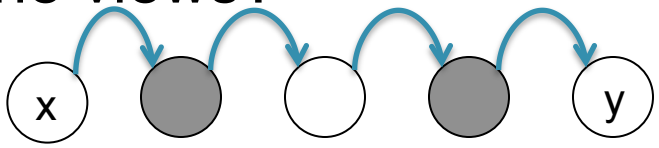
Suppose you only have these two views:

```
v1(x,y) :- black(x), edge(x,y)
v2(x,y) :- edge(x,y), black(y)
```



Can you rewrite this query in terms of the views?

```
q(x,y) :- edge(x,z1), black(z1),
          edge(z1,z2),edge(z2,z3)
          black(z3), edge(z3,y)
```



Answer:

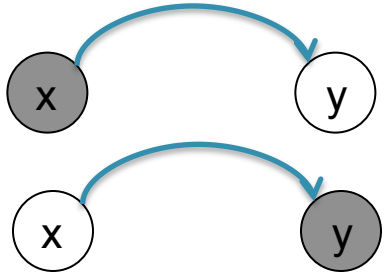
```
q(x,y) :- v2(x,z1),v1(z1,z2),v2(z2,z3),v1(z3,y)
```

[Duschka&Genesereth'97]

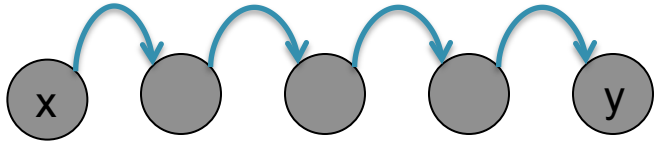
Query Rewriting Using Views

Suppose you only have these two views:

```
v1(x,y) :- black(x), edge(x,y)
v2(x,y) :- edge(x,y), black(y)
```



What about this query?



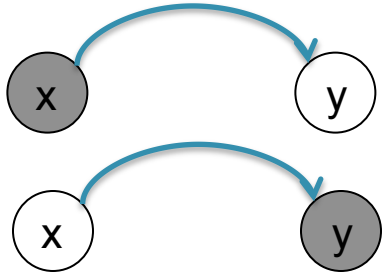

```
q(x,y) :- black(x),edge(x,z1), black(z1),
          edge(z1,z2),black(z2),edge(z2,z3)
          black(z3), edge(z3,y),black(y)
```

[Duschka&Genesereth'97]

Query Rewriting Using Views

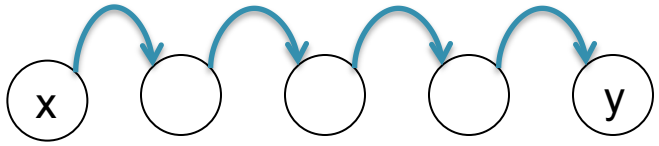
Suppose you only have these two views:

```
v1(x,y) :- black(x), edge(x,y)
v2(x,y) :- edge(x,y), black(y)
```



Can we rewrite this query?

```
q(x,y) :- edge(x,z1),edge(z1,z2),
          edge(z2,z3), edge(z3,y)
```

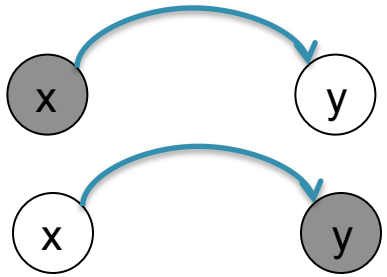


[Duschka&Genesereth'97]

Query Rewriting Using Views

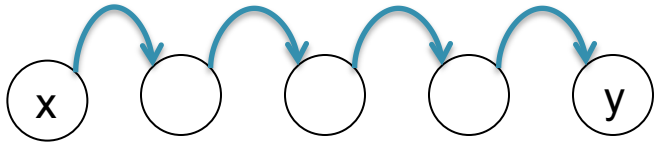
Suppose you only have these two views:

```
v1(x,y) :- black(x), edge(x,y)
v2(x,y) :- edge(x,y), black(y)
```



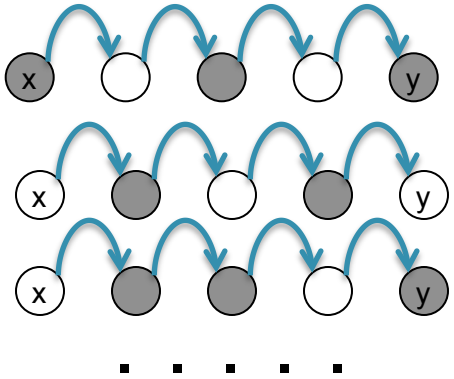
Can we rewrite this query?

```
q(x,y) :- edge(x,z1),edge(z1,z2),
          edge(z2,z3), edge(z3,y)
```



No! Maximally contained rewriting is:

```
q(x,y) :- v1(x,z1),v2(z1,z2),v1(z2,z3),v2(z3,y)
q(x,y) :- v2(x,z1),v1(z1,z2),v2(z2,z3),v1(z3,y)
q(x,y) :- v2(x,z1),v1(z1,z2),v1(z2,z3),v2(z3,y)
. . . .
```



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

Query Rewriting Using Views

Have this materialized view:

```
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'
```

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

Query Rewriting Using Views

```
SELECT  buyer, seller
FROM    SeattleView
WHERE   product= 'gizmo'
```

```
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

```
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
```

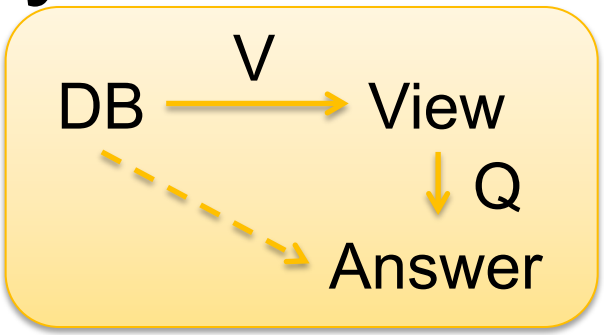
```
SELECT  y.buyer, y.seller
FROM    Person x, Purchase y
WHERE   x.city = 'Seattle'    AND
        x..pname = y.buyer AND
        y.product='gizmo'
```

"Maximally contained rewriting"

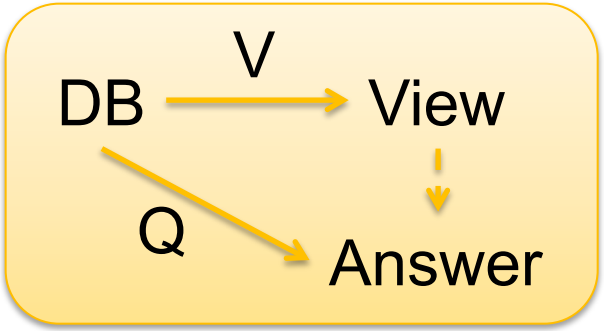
```
SELECT  buyer, seller
FROM    DifferentView
WHERE   product= 'gizmo'
```

Summary

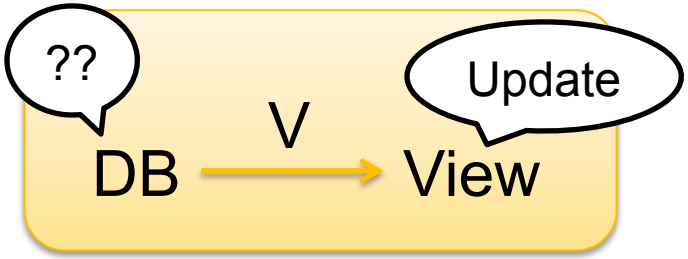
- View inlining, or query modification



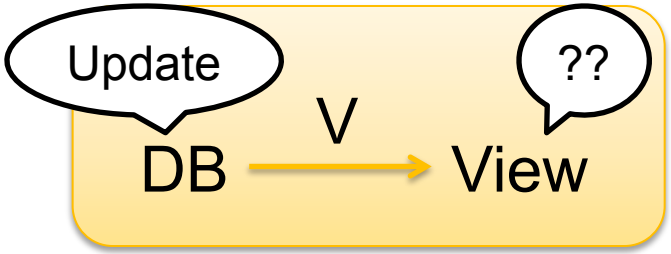
- Query answering/rewriting using views



- Updating views



- Incremental view update



Constraints

Constraints

- A constraint = a property that we'd like our database to hold
- Enforce it by taking some actions:
  - Forbid an update
  - Or perform compensating updates
- Two approaches:
  - Declarative integrity constraints
  - Triggers

Integrity Constraints in SQL

- Keys, foreign keys
- Attribute-level constraints
- Tuple-level constraints
- Global constraints: assertions

simple

complex

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))
```

name	category	price
Gizmo	Gadget	10
Camera	Photo	20
Gizmo	Photo	30
~~Gizmo~~	~~Gadget~~	~~40~~

Other Keys

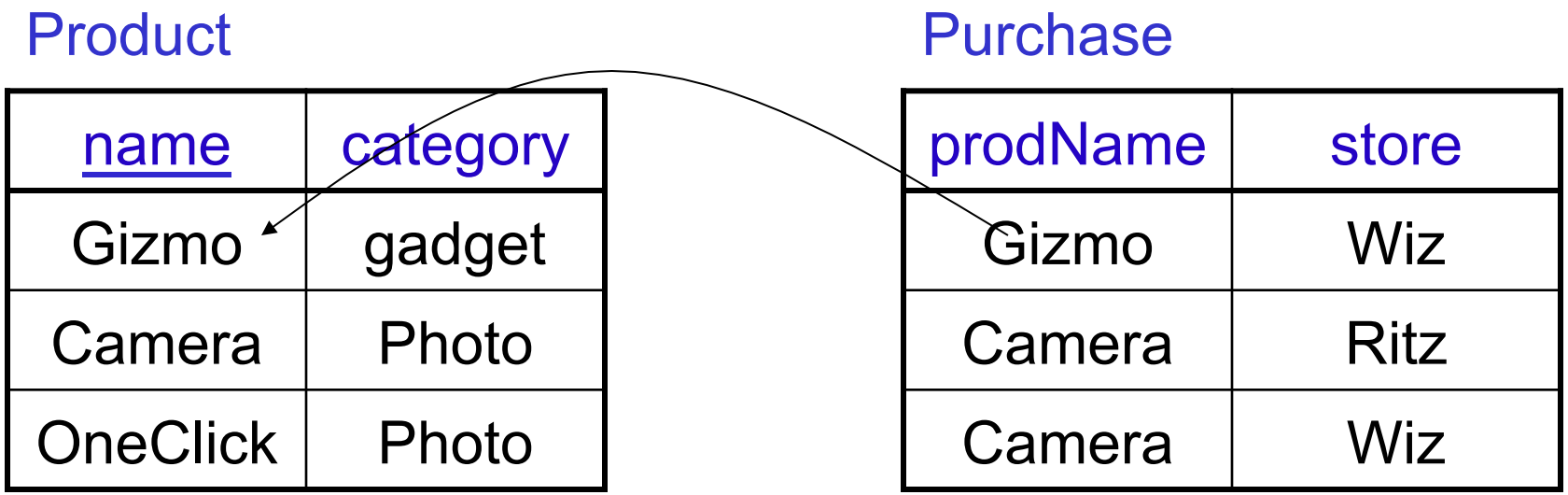
```
CREATE TABLE Product (
    productID  CHAR(10),
    name CHAR(30),
    category VARCHAR(20),
    price INT,
    PRIMARY KEY (productID),
    UNIQUE (name, category))
```

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

Foreign Key Constraints

```
CREATE TABLE Purchase (
   buyer CHAR(30),
   seller CHAR(30),
   prodName CHAR(30) REFERENCES Product,
   store VARCHAR(30))
```

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


Product
name
category
Gizmo
gadget
Camera
Photo
OneClick
Photo
Purchase
prodName
store
Gizmo
Wiz
Camera
Ritz
Camera
Wiz

Foreign Key Constraints

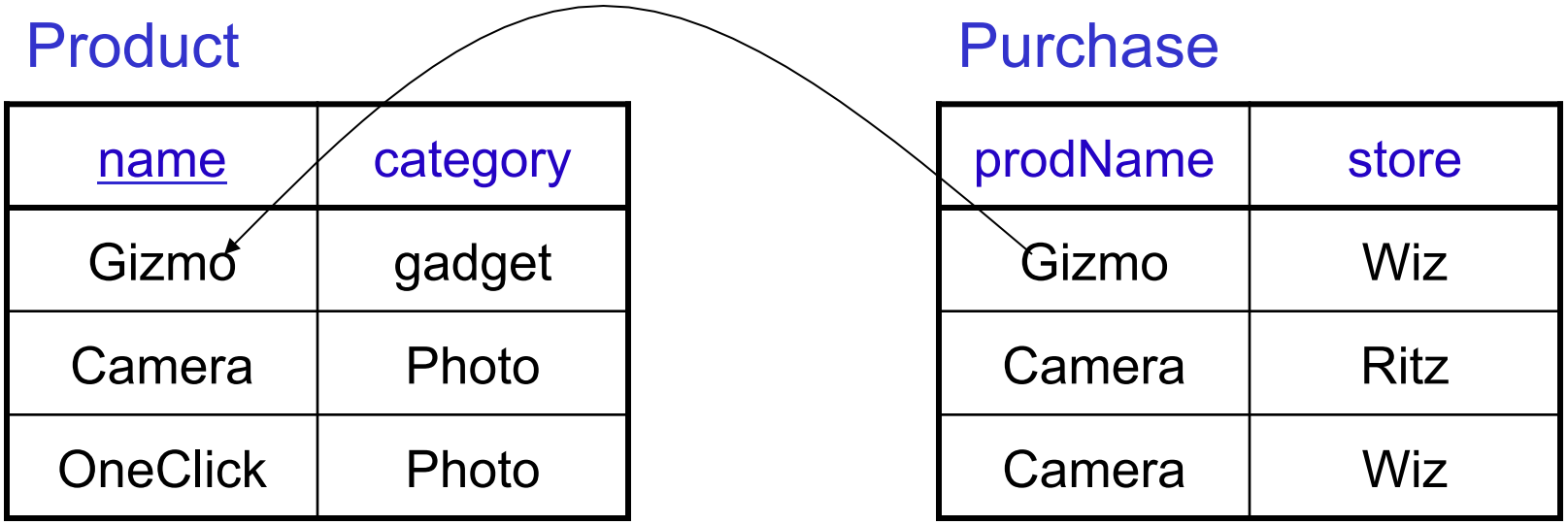
```
CREATE   TABLE Purchase(
    buyer VARCHAR(50),
    seller VARCHAR(50),
    prodName CHAR(20),
    category VAVRCHAR(20),
    store VARCHAR(30),
    FOREIGN KEY (prodName, category)
        REFERENCES Product);
```

Purchase(buyer, seller, product, category, store)
Product(<u>name, category</u>, 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:

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

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

Tuple level constraints:

```
. . . CHECK (price * quantity < 10000) . . .
```


```
CREATE TABLE Purchase (
        prodName CHAR(30)
                CHECK (prodName IN
                        SELECT Product.name
                        FROM Product),
        date DATETIME NOT NULL)
```

General Assertions

```
CREATE ASSERTION myAssert CHECK
  NOT EXISTS(
        SELECT Product.name
        FROM Product, Purchase
        WHERE Product.name = Purchase.prodName
        GROUP BY Product.name
        HAVING count(*) > 200)
```

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
```

When can we rewrite the query ?

```
SELECT Purchase.store
FROM   Purchase
```

Semantic Optimization using Constraints

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

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

```
SELECT Purchase.store
FROM   Purchase
```

Yes, provided that:

Purchase.product is foreign key AND not null