Database Design and Implementation

CS 645

Views and constraints

Example

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

"virtual table"

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

Example

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

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

Queries over views: query modification

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

View:

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

Query:

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

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

Modified query:

```
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 Views: Query Modification

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
```

Types of Views

- Virtual views:
  - Pros/Cons?

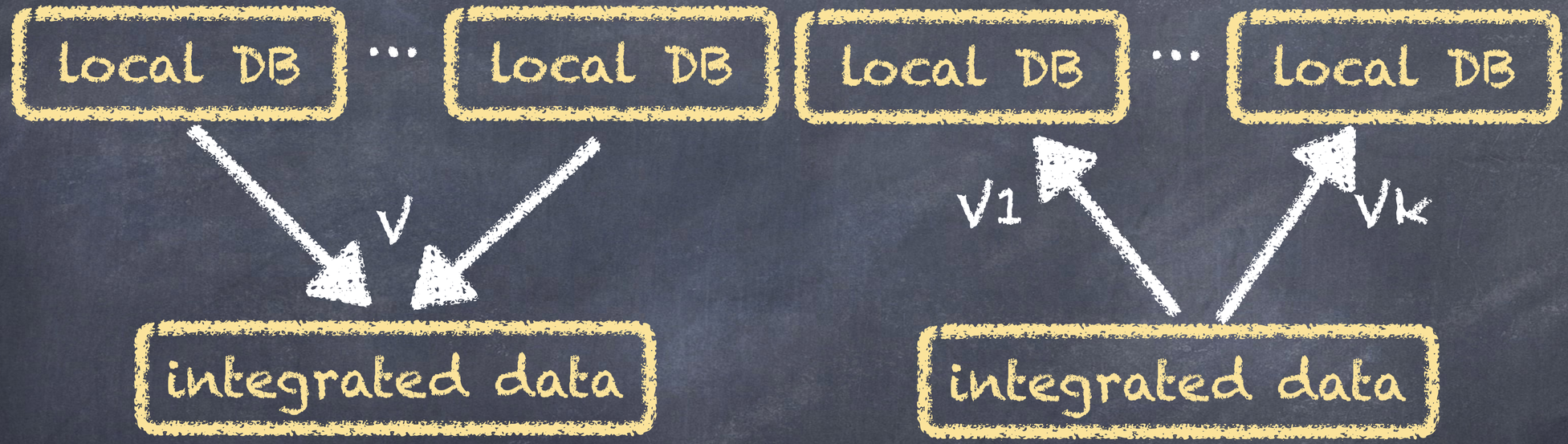
- Materialized views
  - Pros/Cons?

Types of Views

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

- Materialized views
  - Used in data warehouses
  - Pre-computed offline – fast at runtime
  - May have stale data
  - Indexes are materialized views

Data Integration



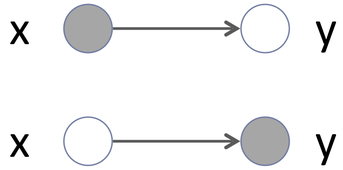
global as view

local as view

Query rewriting using views

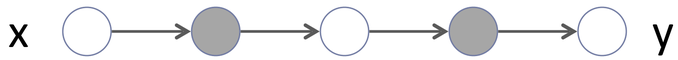
Suppose you have these views:

```
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)
```



= any color

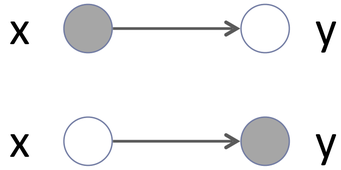
= black

Example from [Duschka&Genesereth'97]

Query rewriting using views

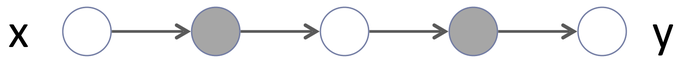
Suppose you have these views:

```
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)
```

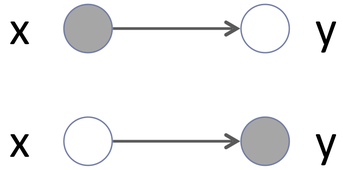
= any color

= black

Query rewriting using views

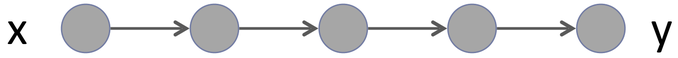
Suppose you have these views:

```
V1(x,y) :- black(x), edge(x,y)
V2(x,y) :- edge(x,y), black(y)
```



What about this query?

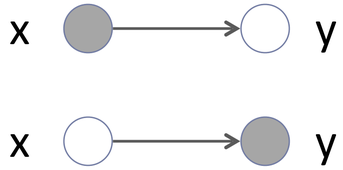
```
Q(x,y) :- black(x), edge(x,z1),
          black(z1), edge(z1,z2),
          black(z2),edge(z2,z3),
          black(z3), edge(z3,y), black(y)
```



Query rewriting using views

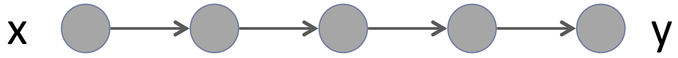
Suppose you have these views:

```
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)
```



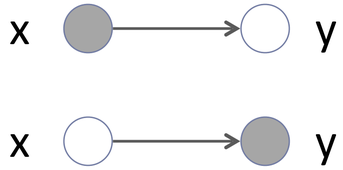
Answer:

```
Q(x,y) :- V1(x,z1), V1(z1,z2), V1(z2,z3), V1(z3,y), V2(z3,y)
```

Query rewriting using views

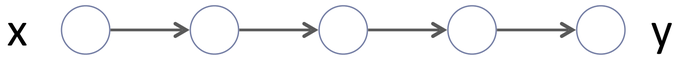
Suppose you have these views:

```
V1(x,y) :- black(x), edge(x,y)
V2(x,y) :- edge(x,y), black(y)
```



Can you rewrite this?

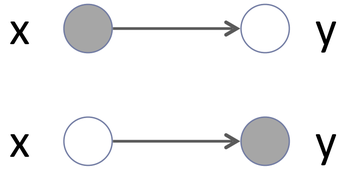
```
Q(x,y) :- edge(x,z1), edge(z1,z2),
          edge(z2,z3), edge(z3,y)
```



Query rewriting using views

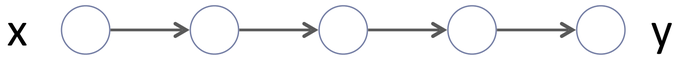
Suppose you have these views:

```
V1(x,y) :- black(x), edge(x,y)
V2(x,y) :- edge(x,y), black(y)
```



Can you rewrite this?

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



No! Maximally contained rewrite is:

```
Q(x,y) :- V1(x,z1), V2(z1,z2), V1(z2,z3), V2(z3,y)
Q(x,y) :- V2(x,z1), V2(z1,z2), V2(z2,z3), V2(z3,y)
Q(x,y) :- V2(x,z1), V1(z1,z2), V1(z2,z3), V2(z3,y)
…etc.
```

Vertical partitioning

Resumes

SSN	Name	Address	Resume	Picture
234234	Mary	Huston	Clob1…	Blob1…
345345	Sue	Amherst	Clob2…	Blob2…
345343	Joan	Amherst	Clob3…	Blob3…
234234	Ann	Portland	Clob4…	Blob4…

T1

SSN	Name	Address
234234	Mary	Huston
345345	Sue	Amherst
. . .		

T2

SSN	Resume
234234	Clob1…
345345	Clob2…

T3

SSN	Picture
234234	Blob1…
345345	Blob2…

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
```

Why use 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

SSN	Name	City	Country
234234	Mary	Houston	USA
345345	Sue	Amherst	USA
345343	Joan	Amherst	USA
234234	Ann	Portland	USA
--	Frank	Calgary	Canada
--	Jean	Montreal	Canada

CustomersInHouston

SSN	Name	City	Country
234234	Mary	Huston	USA

CustomersInAmherst

SSN	Name	City	Country
345345	Sue	Amherst	USA
345343	Joan	Amherst	USA

CustomersInCanada

SSN	Name	City	Country
--	Frank	Calgary	Canada
--	Jean	Montreal	Canada

Horizontal partitioning

```
CREATE VIEW Customers AS
      CustomersInHouston
          UNION ALL
      CustomersInAmherst
          UNION ALL
        . . .
```

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

Which tables are inspected by the system ?

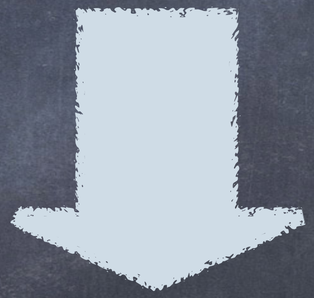
Horizontal partitioning

Better:

```
CREATE VIEW Customers AS
      (SELECT * FROM CustomersInHuston
       WHERE  city = 'Huston')
          UNION ALL
      (SELECT * FROM CustomersInAmherst
       WHERE  city = 'Amherst')
          UNION ALL
       . . .
```

Horizontal partitioning

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



```
SELECT name
FROM   CustomersInAmherst
```

Horizontal partitioning

- Optimizations:
  - E.g., archived applications and active applications
- Distributed databases
- Data integration

Views and security

Customers:

Name	Address	Balance
Mary	Huston	450.99
Sue	Amherst	-240
Joan	Amherst	333.25
Ann	Portland	-520

Fred is not allowed to see this

```
CREATE VIEW PublicCustomers
       SELECT Name, Address
       FROM   Customers
```

Fred is allowed to see this

Views and security

Customers:

Name	Address	Balance
Mary	Huston	450.99
Sue	Amherst	-240
Joan	Amherst	333.25
Ann	Portland	-520

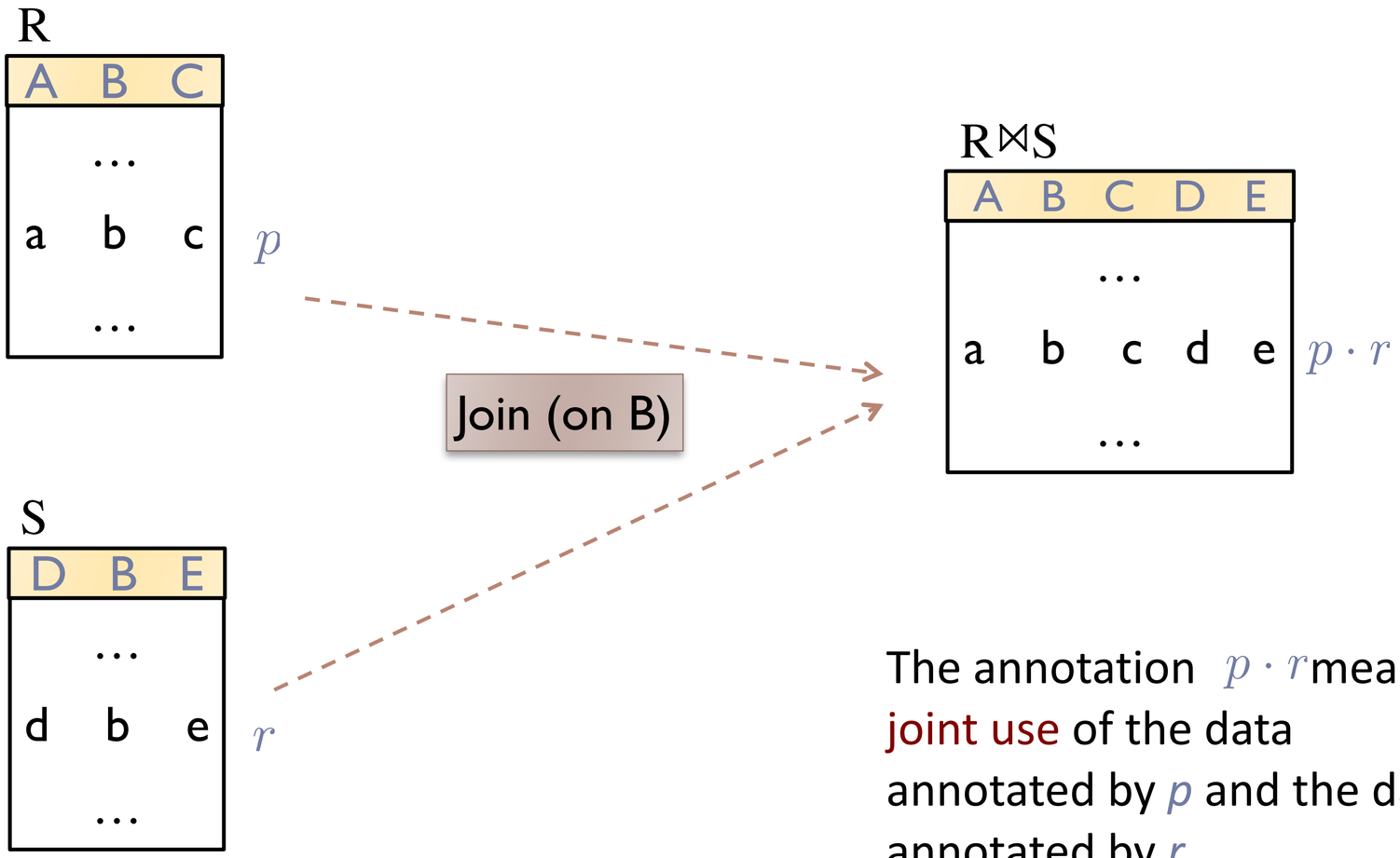
John is not allowed to see >0 balances

```
CREATE VIEW BadCreditCustomers
       SELECT *
       FROM   Customers
       WHERE  Balance < 0
```

Views and updates

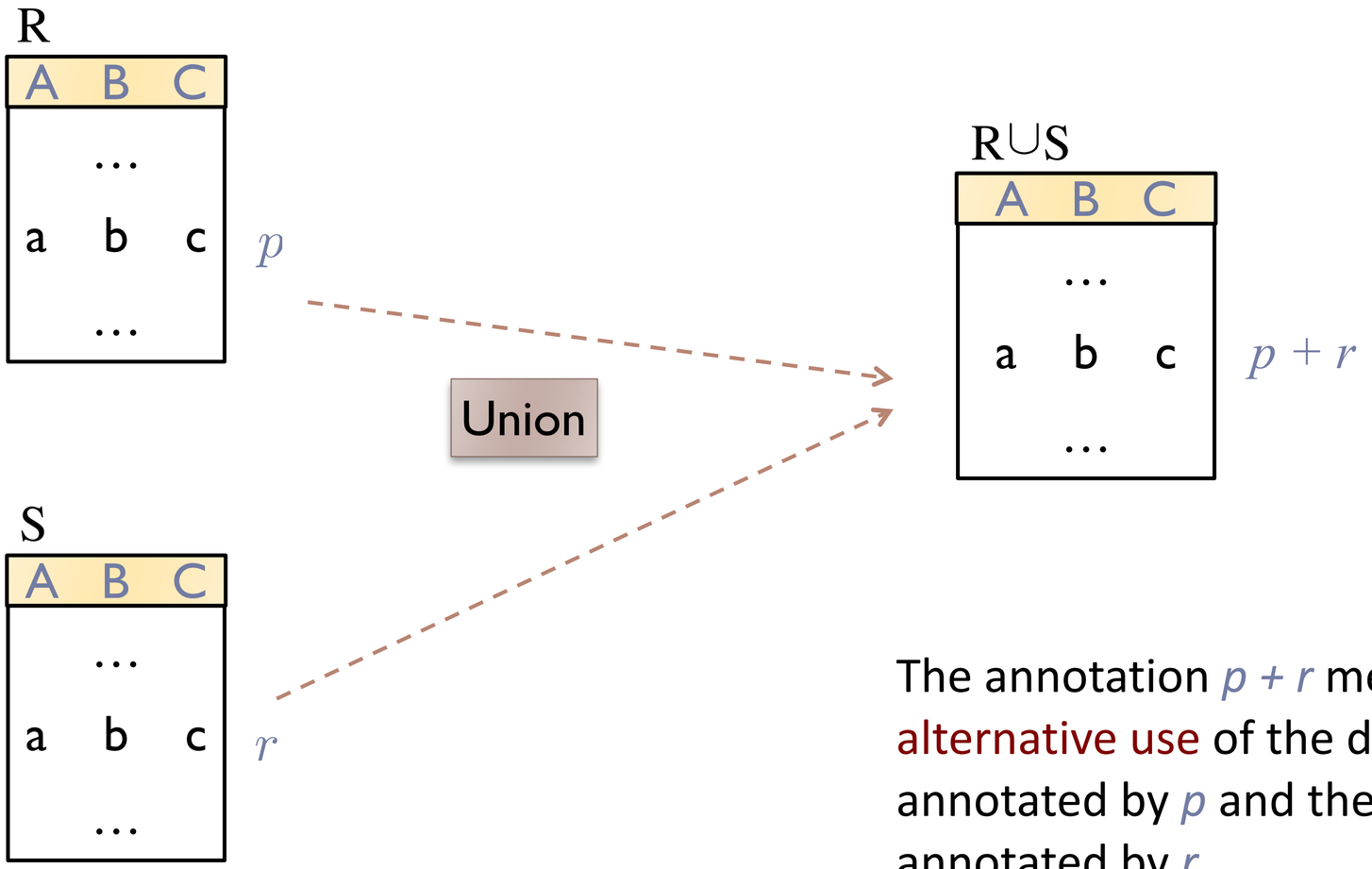
- Discussion:
  - What happens when we insert a tuple to a view?
  - Update a tuple from a view?
  - Delete a tuple from a view?

Propagating annotations



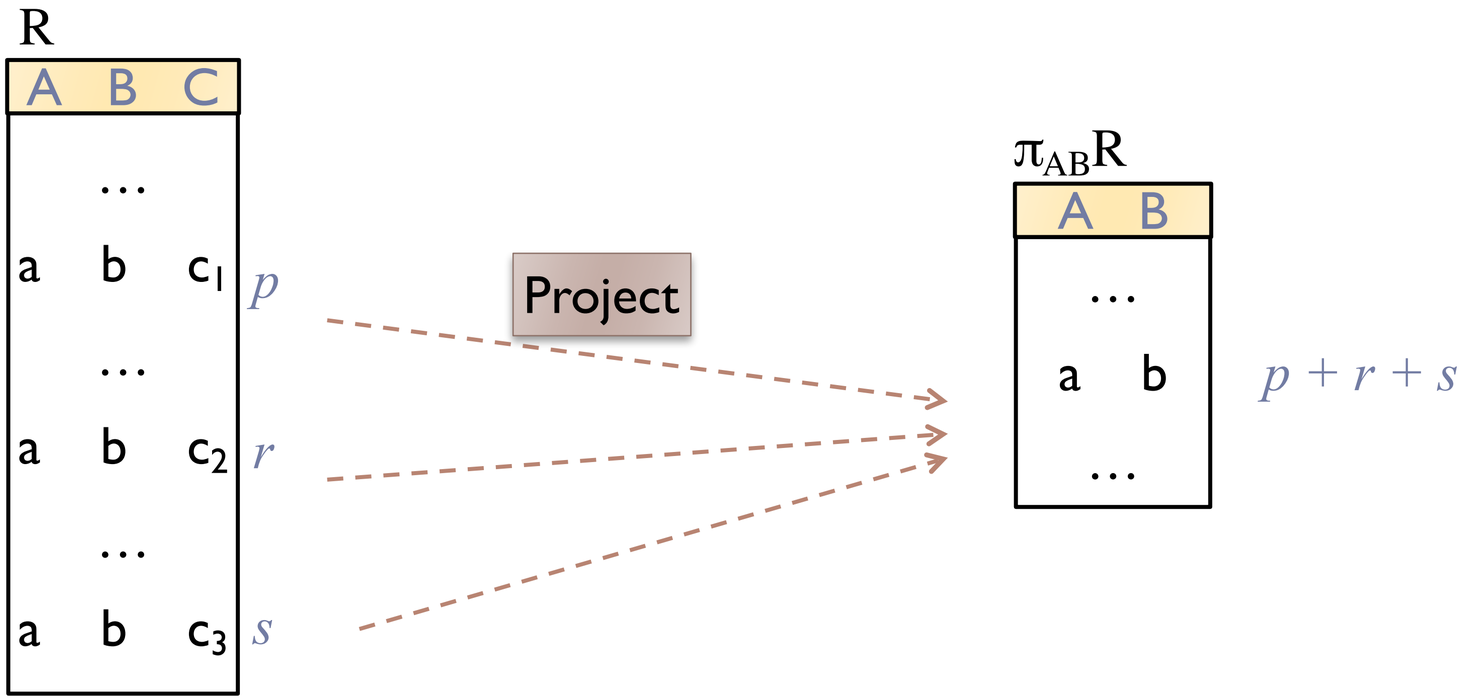
The annotation $p \cdot r$ means joint use of the data annotated by *p* and the data annotated by *r*

Propagating annotations (2)



The annotation $p + r$ means alternative use of the data annotated by p and the data annotated by r

Propagating Annotations (3)



+ denotes alternative use of data

The view deletion problem

- *D* a database instance and *V=Q(D)* a view defined over *D*.
  - *Find a set of tuples ΔD to remove from D so that a specific tuple t is removed from the view*
  - *Minimize the number of side-effects in the view*
    - *View side-effect problem*
      - *Hard: queries with joins and projection or union*
      - *PTIME: the rest*
  - *Minimize the number of tuples deleted from D*
    - *Source side-effect problem*
      - *Same dichotomy*

[Buneman, Khanna, Tan. PODS 2002]

view side effects

Query class	Deciding whether there is a side-effect-free deletion
Queries involving PJ	NP-hard
Queries involving JU	NP-hard
SPU	P
SJ	P

source side effects

Query class	Finding the minimum source deletions
Queries involving PJ	NP-hard
Queries involving JU	NP-hard
SPU	P
SJ	P

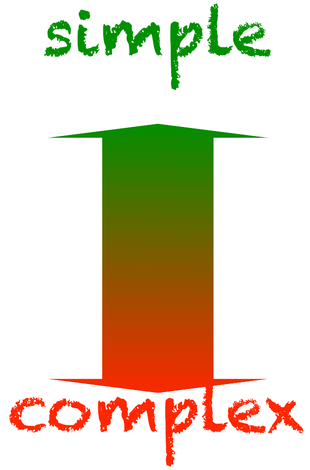
Constraints

- Constraint: a property that we want our data to satisfy
- Enforce by taking actions:
  - Forbid an update
  - Or perform compensating updates

- Two approaches:
  - Declarative constraints
  - Triggers

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 enforce

Keys

Product(<u>name</u>, category)

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

=

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

Keys with multiple attributes

Product(name, category, price)

```
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 keys

```
CREATE TABLE Purchase (
      prodName CHAR(30)
            REFERENCES Product(name),
      date DATETIME)
```

may write just Product

prodName is a foreign key to Product(name)
name must be a key in Product (primary key or unique)

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Foreign keys

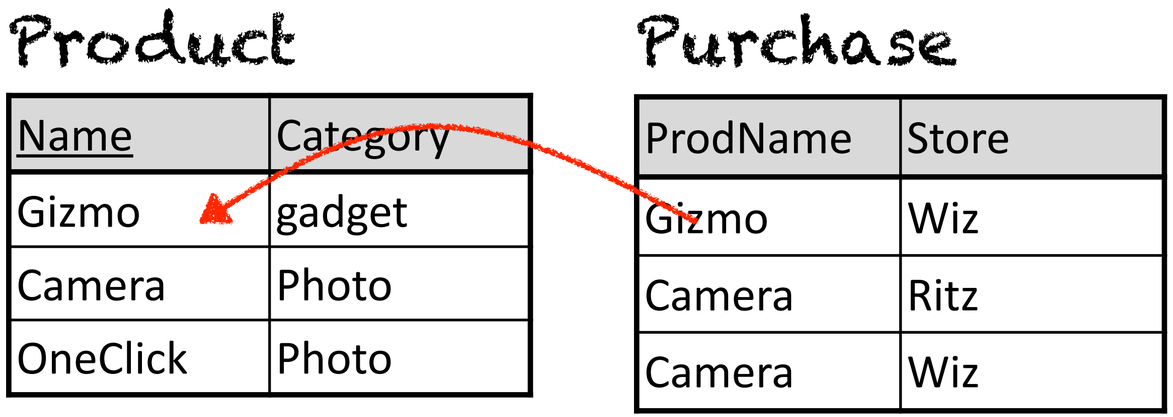
```
CREATE TABLE Purchase (
      prodName CHAR(30),
      category VARCHAR(20),
      date DATETIME,
      store VARCHAR(30),
      FOREIGN KEY (prodName, category)
        REFERENCES  Product(name, category)
```

Product(<u>name, category</u>, price)
Purchase(proName, category, date

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

```
CREATE TABLE Purchase (
      prodName CHAR(30)
            REFERENCES Product(name)
            ON DELETE SET NULL
            ON UPDATE CASCADE)
```

Constraints on attributes and tuples

- Constraints on attributes:
  - NOT NULL -- obvious meaning…
  - CHECK condition -- any condition !

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

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

- Constraints on tuples
  - CHECK condition

```
… CHECK(price*quantity < 1000)…
```

more complex CHECK constraints

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

Often not implemented in DBMSs!

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)
```

Often not implemented in DBMSs!

Semantic optimization with constraints

Product(<u>name</u>, price)
Purchase(buyer, seller, prodName, store)

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


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

Semantic optimization with constraints

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

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


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

Yes, if Purchase.prodName is a foreign key, and not null