Introduction to Database Systems CSE 444

Lecture 4: Views and Constraints

Review from Friday

Product(name, category)
Purchase(prodName, month, store)

- Compute, for each product, the total number of sales in 'September'

```
SELECT    Product.name, count(store)
FROM      Product LEFT OUTER JOIN Purchase ON
          Product.name = Purchase.prodName
          and Purchase.month = 'September'
GROUP BY  Product.name
```

Product

Name	Category
Gizmo	Gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Month	Store
Gizmo	Nov	Wiz
Camera	Sept	NULL
Camera	Sept	Wiz

Views vs Tables

- Views are relations except that they may not be physically stored.
- Why do we need views?

- Example:
  - Employee(ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
    SELECT     name, project
    FROM       Employee
    WHERE      department = 'Development'
```

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
```

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)

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
```

Applications of Virtual Views

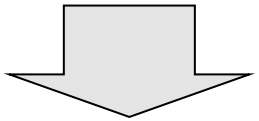
- Physical data independence. E.g.
  - Vertical data partitioning
  - Horizontal data partitioning

- Security
  - Handle different access rights
  - The view reveals only what the users are allowed to know

Vertical Partitioning

Resumes

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



T1

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

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

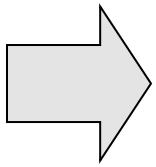
When to do this:

- 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	Huston	USA
345345	Sue	Seattle	USA
345343	Joan	Seattle	USA
234234	Ann	Portland	USA
--	Frank	Calgary	Canada
--	Jean	Montreal	Canada



CustomersInHuston

SSN	Name	City	Country
234234	Mary	Huston	USA

CustomersInSeattle

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

CustomersInCanada

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

Horizontal Partitioning

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

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

Which tables are inspected by the system ?

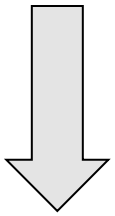
Horizontal Partitioning

Better:

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

Horizontal Partitioning

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



```
SELECT name
FROM   CustomersInSeattle
```

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	Seattle	-240
Joan	Seattle	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	Seattle	-240
Joan	Seattle	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

- Food for thought:
  - What happens when we insert a tuple to a view?
  - Update a tuple from a view?
  - Can we always/ever do this?

Constraints in SQL

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

Product(name, category)

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

OR:

```
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))
```

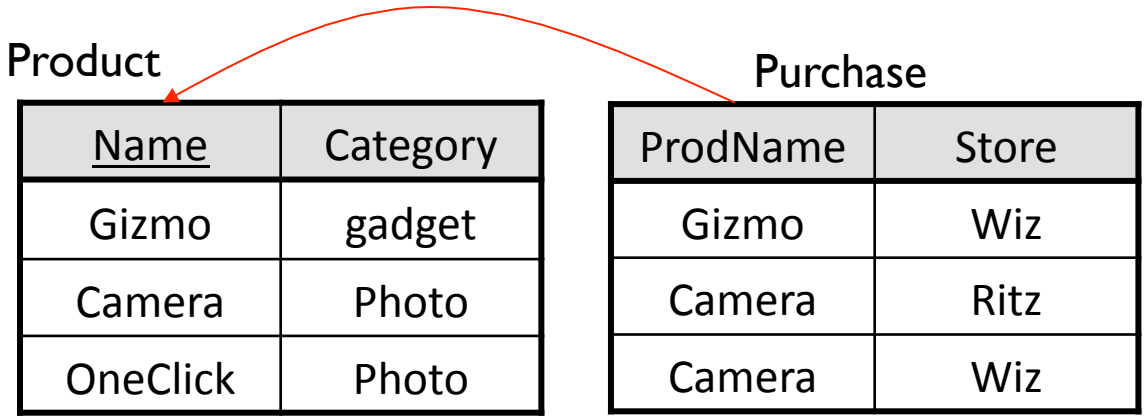
There is at most one PRIMARY KEY;
there can be many UNIQUE

Foreign Key Constraints

```
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



Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

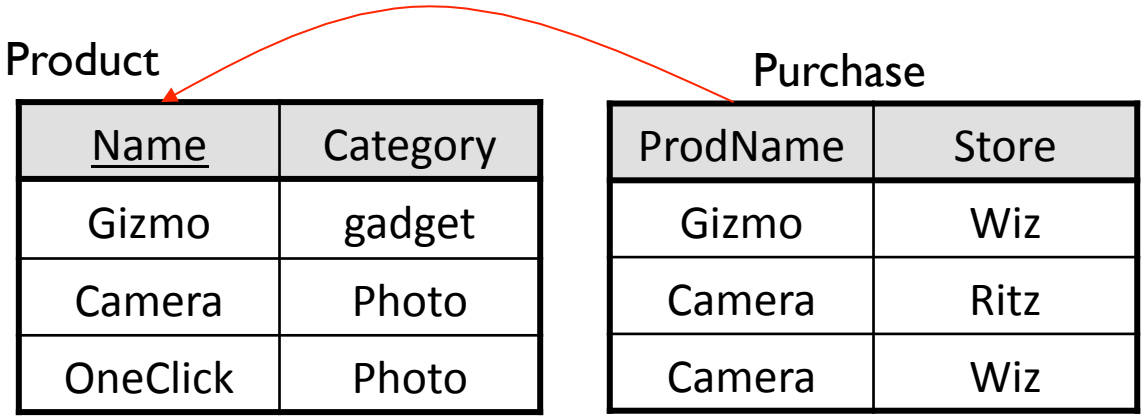
Foreign Key Constraints

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

What happens during updates ?

Types of updates:

- In Purchase: insert/update
- In Product: delete/update



Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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 !

- Constraints on tuples

  CHECK condition

CHECK condition

How is this different from a foreign key constraint?

```
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)
```