Introduction to Database Systems
CSE 444

Lecture 4: Views and Constraints
Review from Friday

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

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

<table>
<thead>
<tr>
<th>Product</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>

<table>
<thead>
<tr>
<th>Purchase</th>
<th>ProdName</th>
<th>Month</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Nov</td>
<td>Wiz</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Sept</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Sept</td>
<td>Wiz</td>
<td></td>
</tr>
</tbody>
</table>
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'
```
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)
Example

\[
\text{SELECT } u\.\text{customer}, v\.\text{store} \\
\text{FROM } \text{CustomerPrice} u, \text{Purchase} v \\
\text{WHERE } u\.\text{customer} = v\.\text{customer} \text{ and } u\.\text{price} > 100
\]

Purchase(\text{customer, product, store}) \\
Product(\text{pname, price}) \\
CustomerPrice(\text{customer, price})
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

**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:

```sql
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

<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

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

<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
.
.
.

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:

<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 not allowed to see this
Fred is allowed to see this
Views and Security

<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** BadCreditCustomers

```
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))

<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>
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)

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

```sql
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

CREATE TABLE Purchase (  
  prodName CHAR(30)  
  CHECK (prodName IN  
          SELECT Product.name  
          FROM Product),  
  date DATETIME NOT NULL)
CREATE ASSERTION myAssert CHECK NOT EXISTS (
  SELECT Product.name 
  FROM Product, Purchase 
  WHERE Product.name = Purchase.prodName 
  GROUP BY Product.name 
  HAVING count(*) > 200)