Introduction to Database Systems
CSE 444

Lecture 4: Views and Constraints
Outline

• **Views**: Sections 8.1, 8.2, 8.3
  – [Old edition, Sections 6.6 and 6.7]

• **Constraints**: Sections 2.3, 7.1, 7.2
  – [Old edition: Sections 7.1 and 7.2 only]

• Won’t discuss updates! In sections…
Views

Views are relations, except that they may not be physically stored

For presenting different information to different users

Employee(ssn, name, department, project, salary)

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

Payroll has access to Employee, others only to Developers
Example

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

<table>
<thead>
<tr>
<th>CREATE VIEW</th>
<th>CustomerPrice AS</th>
<th>SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x.customer, y.price</td>
</tr>
<tr>
<td>FROM</td>
<td>Purchase x, Product y</td>
<td></td>
</tr>
<tr>
<td>WHERE</td>
<td>x.product = y.pname</td>
<td></td>
</tr>
</tbody>
</table>

CustomerPrice(customer, price)  “virtual table”
Example

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

CustomerPrice(customer, price)

We can later use the view just like any other relation:

```
SELECT DISTINCT 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

View:

```
CREATE VIEW CustomerPrice AS
    SELECT 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
```
Queries Over Views: Query Modification

Modified query:

```
SELECT DISTINCT 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 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
```
Applications of Virtual Views

• **Increased physical data independence.** E.g.
  – Vertical data partitioning
  – Horizontal data partitioning

• **Logical data independence.** E.g.
  – Change schemas of base relations (i.e., stored tables)

• **Security**
  – View reveals only what the users are allowed to know
### Vertical Partitioning

#### Resumes

<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
Vertical Partitioning

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

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

When do we use vertical partitioning?
Vertical Partitioning Applications

1. Can improve performance of some queries
   - When queries touch small fraction of columns
   - Only need to read desired columns from disk
   - Can produce big I/O savings for wide tables
   - Potential benefit in data warehousing applications

• But
  - Repeated key columns add a lot of overhead
  - Need expensive joins to reconstruct tuples
Vertical Partitioning Applications

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

3. In distributed databases
   – Customer personal info at one site, profile at another

4. 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>Houston</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>

### CustomersInHouston

<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>Houston</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
    CustomersInHouston
    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 CustomersInHouston
 WHERE city = 'Houston')
UNION ALL
(SELECT * FROM CustomersInSeattle
 WHERE city = 'Seattle')
UNION ALL
...
```

Other techniques exist: read DBMS documentation
Horizontal Partitioning

\[
\text{SELECT name FROM Customers WHERE city = 'Seattle'}
\]

\[
\text{SELECT name FROM CustomersInSeattle}
\]
Horizontal Partitioning Applications

- **Performance optimization**
  - Especially for data warehousing
  - E.g. one partition per month
  - E.g. archived applications and active applications

- **Distributed and parallel 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>Houston</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

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>

Wilma is not allowed to see >0 balances

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

• **Views**: Sections 8.1, 8.2, 8.3
  – [Old edition, Sections 6.6 and 6.7]

• **Constraints**: Sections 2.3, 7.1, 7.2
  – [Old edition: Sections 7.1 and 7.2 only]
Integrity Constraints Motivation

An integrity constraint is a condition specified on a database schema that restricts the data that can be stored in an instance of the database.

- ICs help prevent entry of incorrect information
- DBMS enforces integrity constraints
  - Allows only legal database instances (i.e., those that satisfy all constraints) to exist
  - Ensures that all necessary checks are always performed and avoids duplicating the verification logic in each application
Types of 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
Key Constraints

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)

```sql
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

**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

• Example with multi-attribute primary key

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

• (name, category) must be a PRIMARY KEY in Product
What happens during updates?

Types of updates:
- In Purchase: insert/update
- In Product: delete/update

<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>

<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>
What happens during updates?

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

**READING ASSIGNMENT:** 7.1.2 and 7.1.3
- [Old edition: 7.1.5, 7.1.6]
Constraints on Attributes and Tuples

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

- Constraints on tuples
  CHECK condition
Constraints on Attributes and Tuples

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

What is the difference from Foreign-Key?
CREATE ASSERTION myAssert CHECK
NOT EXISTS(
    SELECT Product.name
    FROM Product, Purchase
    WHERE Product.name = Purchase.prodName
    GROUP BY Product.name
    HAVING count(*) > 200)

But most DBMSs do not implement assertions
Instead, they provide triggers
To learn more, read the rest of Chapter 7