Principles of Database Systems
CSE 544

Lecture #4
Views and Constraints
Reading Material

• **Views:**
  – *Query answering using views*, by Halevy (due on Monday)
  – Book: 3.6

• **Constraints:**
  – Book 3.2, 3.3, 5.8
Views

• A view in SQL =
  – A table computed from other tables, s.t., whenever the base tables are updated, the view is updated too

• More generally:
  – A view is derived data that keeps track of changes in the original data

• Compare:
  – A function computes a value from other values, but does not keep track of changes to the inputs
CREATE VIEW StorePrice AS
SELECT DISTINCT x.store, y.price
FROM Purchase x, Product y
WHERE x.product = y.pname

This is like a new table StorePrice(store, price)

A Simple View

Create a view that returns for each store the prices of products purchased at that store
We Use a View Like Any Table

• A "high end" store is a store that sells some products over 1000.
• For each customer, return all the high end stores that they visit.

```
SELECT DISTINCT u.customer, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
  AND v.price > 1000
```
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 (must recomputes or update)
  – Indexes are materialized views

Indexes are materialized views
Query Modification

For each customer, find all the high end stores that they visit.

CREATE VIEW StorePrice AS
    SELECT DISTINCT x.store, y.price
    FROM Purchase x, Product y
    WHERE x.product = y.pname

SELECT DISTINCT u.customer, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
    AND v.price > 1000
For each customer, find all the high end stores that they visit.

CREATE VIEW StorePrice AS
    SELECT DISTINCT x.store, y.price
    FROM Purchase x, Product y
    WHERE x.product = y.pname

SELECT DISTINCT u.customer, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
    AND v.price > 1000

Modified query:

SELECT DISTINCT u.customer, u.store
FROM Purchase u,
    (SELECT DISTINCT x.store, y.price
     FROM Purchase x, Product y
     WHERE x.product = y.pname) v
WHERE u.store = v.store
    AND v.price > 1000
For each customer, find all the high end stores that they visit.

Modified query:

\[
\text{SELECT DISTINCT } u.\text{customer}, u.\text{store} \\
\text{FROM Purchase } u, \text{ Purchase } x, \text{ Product } y \\
\text{WHERE } u.\text{store} = x.\text{store} \\
\text{AND } y.\text{price} > 1000 \\
\text{AND } x.\text{product} = y.\text{pname}
\]

Modified and unnested query:

\[
\text{SELECT DISTINCT } u.\text{customer}, u.\text{store} \\
\text{FROM Purchase } u, \\
(\text{SELECT DISTINCT } x.\text{store}, y.\text{price} \\
\text{FROM Purchase } x, \text{ Product } y \\
\text{WHERE } x.\text{product} = y.\text{pname}) \ v \\
\text{WHERE } u.\text{store} = v.\text{store} \\
\text{AND } v.\text{price} > 1000
\]

Notice that Purchase occurs twice. Why?
Further Virtual View Optimization

Retrieve all stores whose name contains ACME

CREATE VIEW StorePrice AS
  SELECT DISTINCT x.store, y.price
  FROM Purchase x, Product y
  WHERE x.product = y.pname

SELECT DISTINCT v.store
FROM StorePrice v
WHERE v.store like '%ACME%'

Purchase(customer, product, store)
Product(pname, price)
Retrieve all stores whose name contains ACME

CREATE VIEW StorePrice AS
SELECT DISTINCT x.store, y.price
FROM Purchase x, Product y
WHERE x.product = y.pname

SELECT DISTINCT v.store
FROM StorePrice v
WHERE v.store like ‘%ACME%’

Modified query:

SELECT DISTINCT v.store
FROM
(SELECT DISTINCT x.store, y.price
FROM Purchase x, Product y
WHERE x.product = y.pname) v
WHERE v.store like ‘%ACME%’
Further Virtual View Optimization

Retrieve all stores whose name contains ACME

\[
\text{SELECT DISTINCT x.store} \\
\text{FROM Purchase x, Product y} \\
\text{WHERE x.product = y.pname} \\
\text{AND x.store like ‘%ACME%’}
\]

Modified query:

\[
\text{SELECT DISTINCT v.store} \\
\text{FROM} \\
(\text{SELECT DISTINCT x.store, y.price} \\
\text{FROM Purchase x, Product y} \\
\text{WHERE x.product = y.pname}) \ v \\
\text{WHERE v.store like ‘%ACME%’}
\]
Further Virtual View Optimization

Retrieve all stores whose name contains ACME

```
SELECT DISTINCT x.store
FROM Purchase x, Product y
WHERE x.product = y.pname
   AND x.store like '%%ACME%%'
```

Modified and unnested query:

```
SELECT DISTINCT x.store
FROM Purchase x
WHERE x.store like '%%ACME%%'
```

Assuming Product.pname is a key

and Purchase.product is a foreign key

Final Query
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
Example: Finding Witnesses

To find witnesses, create a view with the maximum price

```
CREATE TEMPORARY 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
```
Example: Finding Witnesses

To find witnesses, create a view with the maximum price

```sql
CREATE TEMPORARY 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

```sql
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
```
Example: Finding Witnesses

For one-time use, use temporary view, or:

```
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
```
Example: Finding Witnesses

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

```sql
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
```

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

**REALLY** important to speed up query processing time.

Person \((\text{pid}, \text{name}, \text{age}, \text{city})\)

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

May take too long to scan the entire Person table

```sql
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.
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'
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
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?
Indexes are Materialized Views

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

Indexes as LAV:

CREATE VIEW W AS
SELECT age, pid
FROM Person y

CREATE VIEW P AS
SELECT city, pid
FROM Person y

Each index is a relation:
(index value, record id)
Some DBMS make very advanced use…
Indexes are Materialized Views

Indexes as LAV:

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

“Covering indexes”: When the query uses only the indexes

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

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

SELECT age, city
FROM Person
WHERE age > 22
and city LIKE ‘S%’

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

CSE544 - Spring, 2013
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

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))

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

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

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)
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