Lecture 03:
Views and Constraints

Wednesday, October 13, 2010
Announcements

• HW1: was due yesterday

• HW2: due next Tuesday
Outline and Reading Material

• Constraints and triggers
  – Book: 3.2, 3.3, 5.8

• Views
  – Book: 3.6
  – *Answering queries using views: A survey*, A.Y. Halevy: Sections 1 and 2 (Section 3 is optional)

Most of today’s material is NOT covered in the book. Read the slides carefully
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>

Product(name, category, price)
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**
CREATE TABLE Purchase (  
buyer CHAR(30),  
seller CHAR(30),  
product CHAR(30) REFERENCES Product(name),  
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),
  product CHAR(20),
  category VARCHAR(20),
  store VARCHAR(30),
  FOREIGN KEY (product, category)
    REFERENCES Product(name, category)
);
What happens during updates?

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

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Category</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td></td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Purchase</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProdName</td>
<td>Store</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
<td></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)
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
Semantic Optimization using Constraints

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

```sql
SELECT Purchase.store
FROM Product, Purchase
WHERE Product.name=Purchase.product
```

Why? and When?

```sql
SELECT Purchase.store
FROM Purchase
```
Triggers

Trigger = a procedure invoked by the DBMS in response to an update to the database

Some applications use triggers to enforce integrity constraints

Trigger = Event + Condition + Action
Triggers in SQL

• Event = INSERT, DELETE, UPDATE

• Condition = any WHERE condition
  – Refers to the old and the new values

• Action = more inserts, deletes, updates
  – May result in cascading effects!
Example: Row Level Trigger

CREATE TRIGGER InsertPromotions AFTER UPDATE OF price ON Product
REFERENCING
  OLD AS x
  NEW AS y
FOR EACH ROW WHEN (x.price > y.price)
INSERT INTO Promotions(name, discount)
VALUES x.name,
  (x.price-y.price)*100/x.price

Warning: complex syntax and vendor specific. Take away from the slides the main ideas, not the syntactic details.
EVENTS

INSERT, DELETE, UPDATE

• Trigger can be:
  – AFTER event
  – INSTEAD of event
Scope

• **FOR EACH ROW** = trigger executed for every row affected by update
  – OLD ROW
  – NEW ROW

• **FOR EACH STATEMENT** = trigger executed once for the entire statement
  – OLD TABLE
  – NEW TABLE
Statement Level Trigger

```
CREATE TRIGGER avg-price INSTEAD OF UPDATE OF price ON Product

REFERENCING
   OLD_TABLE AS OldStuff
   NEW_TABLE AS NewStuff

FOR EACH STATEMENT
WHEN (1000 < (SELECT AVG (price)
               FROM ((Product EXCEPT OldStuff) UNION NewStuff))
DELETE FROM Product
       WHERE (name, price, company) IN OldStuff;
INSERT INTO Product
    (SELECT * FROM NewStuff)
```
Triggers v.s. Constraints

Active database = a database with triggers

- Triggers can be used to enforce ICs
- Triggers are more general: alerts, log events
- But hard to understand: recursive triggers
- Syntax is vendor specific, and may vary significantly
  - Postgres has *rules* in addition to *triggers*
Views: Overview

• Virtual views
  – Applications
  – Technical challenges

• Materialized views
  – Applications
  – Technical challenges
Views

Views are relations, but 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)

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

CustomerPrice(customer, price) "virtual table"
We can later use the view:

```
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 *or* expensive synchronization
Queries Over Virtual Views: Query Modification

**View:**

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

**Query:**

```sql
SELECT u.customer, v.store
FROM CustomerPrice u, Purchase v
WHERE u.customer = v.customer AND u.price > 100
```
Queries Over Virtual Views: Query Modification

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 Virtual 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
```
Another Example

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

SELECT DISTINCT u.customer, v.store
FROM CustomerPrice u, Purchase v
WHERE u.customer = v.customer AND u.price > 100

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

• Physical data independence. E.g.
  – Vertical data partitioning
  – Horizontal data partitioning

• Security
  – The 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?
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
...
Horizontal Partitioning

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

Which tables are inspected by the system?

WHY ???
Horizontal Partitioning

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

Now even humans can’t tell which table contains customers in Seattle

```sql
CREATE VIEW Customers AS
CustomersInXXX
UNION ALL
CustomersInYYY
UNION ALL

...
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

Applications:

• 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

### 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>

**John is not allowed to see >0 balances**

CREATE VIEW BadCreditCustomers
SELECT *
FROM Customers
WHERE Balance < 0
Technical Challenges in Virtual Views

- Simplifying queries over virtual views
- Updating virtual views
Simplifying Queries over Virtual Views

- Query un-nesting
- Query minimization
Set v.s. Bag Semantics

**Set semantics**

```
SELECT DISTINCT a,b,c
FROM R, S, T
WHERE . . .
```

**Bag semantics**

```
SELECT a,b,c
FROM R, S, T
WHERE . . .
```
Unnesting: Sets/Sets

\[
\begin{aligned}
\text{SELECT} &\quad \text{DISTINCT } a, b, c \\
\text{FROM} &\quad (\text{SELECT DISTINCT } u, v \\
&\quad \text{FROM } R, S \\
&\quad \text{WHERE } \ldots), \ T \\
\text{WHERE} &\quad \ldots
\end{aligned}
\]

\[
\begin{aligned}
\text{SELECT} &\quad \text{DISTINCT } a, b, c \\
\text{FROM} &\quad R, S, T \\
\text{WHERE} &\quad \ldots
\end{aligned}
\]
Unnesting: Sets/Bags

SELECT DISTINCT a, b, c
FROM (SELECT u, v
       FROM R, S
       WHERE ...), T
WHERE ...

\[\rightarrow\]

SELECT DISTINCT a, b, c
FROM R, S, T
WHERE ...
Unnesting: Bags/Bags

```
SELECT a,b,c
FROM (SELECT u,v
      FROM R,S
      WHERE ...), T
WHERE ...  
```

```
SELECT a,b,c
FROM R, S, T
WHERE ...  
```
Unnesting: Bags/Sets

```
SELECT a,b,c
FROM (SELECT DISTINCT u,v
     FROM R,S
     WHERE ...), T
WHERE ...
```
Query Minimization

• Replace a query Q with Q’ having fewer tables in the FROM clause
• When Q has fewest number of tables in the FROM clause, then we say it is minimized
• Usually (but not always) users write queries that are already minimized
• But the result of rewriting a query over view is often not minimized
Query Minimization under Bag Semantics

Rule 1: If:

- x, y are tuple variables over the same table and:
- The condition x.key = y.key is in the WHERE clause

Then combine x, y into a single variable query
Query Minimization under Bag Semantics

Order(cid, pid, weight, date)
Product(pid, name, price)

What constraints do we need to have for this optimization?

SELECT y.name, x.date
FROM Order x, Product y, Order z
WHERE x.pid = y.pid and y.price < 99 and y.pid = z.pid
and x.cid = z.cid and z.weight > 150

SELECT y.name, x.date
FROM Order x, Product y
WHERE x.pid = y.pid and y.price < 99
and x.weight > 150
Query Minimization under Bag Semantics

Rule 2: If

- x ranges over S, y ranges over T, and
- The condition x.fk = y.key is in the WHERE clause, and
- there is a not null constraint on x.fk
- y is not used anywhere else, and

Then remove T (and y) from the query
Query Minimization under Bag Semantics

Order(cid, pid, weight, date)
Product(pid, name, price)

SELECT x.cid, x.date
FROM Order x, Product y
WHERE x.pid = y.pid and x.weight > 20

Q: Where do we encounter non-minimized queries?

What constraints do we need to have for this optimization?
Query Minimization under Bag Semantics

Order(cid, pid, weight, date)
Product(pid, name, price)

CREATE VIEW CheapOrders AS
SELECT x.cid, x.pid, x.date, y.name, y.price
FROM Order x, Product y
WHERE x.pid = y.pid and y.price < 99

CREATE VIEW HeavyOrders AS
SELECT a.cid, a.pid, a.date, b.name, b.price
FROM Order a, Product b
WHERE a.pid = b.pid and a.weight > 150

SELECT u.cid
FROM CheapOrders u,
    HeavyOrders v
WHERE u.pid = v.pid
    and u.cid = v.cid

Customers who ordered cheap, heavy products

A: in queries resulting from view inlining
Query Minimization

Order(cid, pid, weight, date)
Product(pid, name, price)

CREATE VIEW CheapOrders AS
    SELECT x.cid, x.pid, x.date, y.name, y.price
    FROM Order x, Product y
    WHERE x.pid = y.pid and y.price < 99

CREATE VIEW HeavyOrders AS
    SELECT a.cid, a.pid, a.date, b.name, b.price
    FROM Order a, Product b
    WHERE a.pid = b.pid and a.weight > 150

SELECT u.cid
FROM CheapOrders u,
    HeavyOrders v
WHERE u.pid = v.pid
    and u.cid = v.cid

SELECT a.cid
FROM Order x, Product y
    Order a, Product b
WHERE . . . .

Redundant Orders and Products
SELECT a.cid
FROM   Order x, Product y, Order a, Product b
WHERE  x.pid = y.pid and a.pid = b.pid
       and y.price < 99 and a.weight > 150
       and x.cid = a.cid and x.pid = a.pid

x = a

SELECT x.cid
FROM   Order x, Product y, Product b
WHERE  x.pid = y.pid and x.pid = b.pid
       and y.price < 99 and x.weight > 150

y = b

SELECT x.cid
FROM   Order x, Product y
WHERE  x.pid = y.pid and
       y.price < 99 and x.weight > 150
Query Minimization under Set Semantics

• Rules 1 and 2 still apply

• Rule 3 involves homomorphisms
Definition of a Homomorphism

A **homomorphism** from $Q'$ to $Q$ is a mapping $h : \{y_1, \ldots, y_m\} \rightarrow \{x_1, \ldots, x_k\}$ such that:

(a) If $h(y_i) = x_j$, then $R_{i'} = R_j$
(b) $C$ logically implies $h(C')$ and
(c) $h(A') = A$

SELECT A
FROM R1 x1, ..., Rk xk
WHERE C

SELECT A'
FROM R1' y1, ..., Rm' ym
WHERE C'
Definition of a Homomorphism

**Theorem** If there exists a homomorphism from Q’ to Q, then every answer returned by Q is also returned by Q’.

We say that Q is *contained* in Q’

If there exists a homomorphism from Q’ to Q, and a homomorphism from Q to Q’, then Q and Q’ are equivalent
Find Homomorphism

Q

```
SELECT x.cid
FROM Order x,
    Product y
WHERE x.pid = y.pid
    and y.price < 99
    and x.weight > 150
```

Q’

```
SELECT x.cid
FROM Order x,
    Product y,
    Order z
WHERE x.pid = y.pid
    and y.price < 99
    and x.weight > 150
    and z.weight > 100
```
Homomorphism \( Q \leftrightarrow Q' \)

**Q**

```
SELECT x.cid
FROM Order x, Product y
WHERE x.pid = y.pid
    and y.price < 99
    and x.weight > 150
```

**Q’**

```
SELECT x.cid
FROM Order x, Product y, Order z
WHERE x.pid = y.pid
    and y.pid = z.pid
    and y.price < 99
    and x.weight > 150
    and z.weight > 100
```

Every answer to **Q** is also an answer to **Q’**

**WHY?**
Order(cid, pid, weight, date)
Product(pid, name, price)

Homomorphism Q → Q'

Q

SELECT x.cid
FROM Order x, Product y
WHERE x.pid = y.pid
   and y.price < 99
   and x.weight > 150

Q' and Q are equivalent!

Q'

SELECT x.cid
FROM Order x, Product y,
     Order z
WHERE x.pid = y.pid
   and y.pid = z.pid
   and y.price < 99
   and x.weight > 150
   and z.weight > 100
Query Minimization under Set Semantics

```
SELECT DISTINCT x.pid
FROM   Product x, Product y, Product z
WHERE  x.category = y.category and y.price > 100
       and x.category = z.category and z.price > 500
       and z.weight > 10
```

Same as:

```
SELECT DISTINCT x.pid
FROM   Product x, Product z
WHERE  x.category = z.category and z.price > 500
       and z.weight > 10
```
Query Minimization under Set Semantics

Rule 3: Let Q’ be the query obtained by removing the tuple variable x from Q. If:

- Q has set semantics (and same for Q’)
- there exists a homomorphism from Q to Q’

Then Q’ is equivalent to Q. Hence one can safely remove x.
Example

\[
\begin{align*}
Q & \quad \text{SELECT DISTINCT } x\text{.pid} \\
  & \quad \text{FROM Product } x, \text{ Product } y, \text{ Product } z \\
  & \quad \text{WHERE } x\text{.category} = y\text{.category} \text{ and } y\text{.price} > 100 \\
  & \quad \quad \text{and } x\text{.category} = z\text{.category} \text{ and } z\text{.price} > 500 \\
  & \quad \quad \quad \text{and } z\text{.weight} > 10
\end{align*}
\]

\[
Q' \quad \text{Find a homomorphism } h: Q \rightarrow Q'
\]

\[
\begin{align*}
Q' & \quad \text{SELECT DISTINCT } x'\text{.pid} \\
  & \quad \text{FROM Product } x', \text{ Product } z' \\
  & \quad \text{WHERE } x'\text{.category} = z'\text{.category} \text{ and } z'\text{.price} > 500 \\
  & \quad \quad \text{and } z'\text{.weight} > 10
\end{align*}
\]
Example

Q

```
SELECT DISTINCT x.pid
FROM Product x, Product y, Product z
WHERE x.category = y.category and y.price > 100
    and x.category = z.category and z.price > 500
    and z.weight > 10
```

Q′

Answer: H(x) = x′, H(y) = H(z) = z′

```
SELECT DISTINCT x’.pid
FROM Product x’, Product z’
WHERE x’.category = z’.category and z’.price > 500
    and z’.weight > 10
```
Updating Views

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

CREATE VIEW Expensive-Product AS
SELECT pname
FROM Product
WHERE price > 100

INSERT INTO Expensive-Product
VALUES('Gizmo')
Updatable Views

- Have a virtual view $V(A_1, A_2, \ldots)$ over tables $R_1, R_2, \ldots$
- User wants to *update* a tuple in $V$
  - Insert/modify/delete
- Can we translate this into updates to $R_1, R_2, \ldots$?
- If yes: $V =$ “an updateable view”
- If not: $V =$ “a non-updateable view”
Updating Views

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

CREATE VIEW Expensive-Product AS
SELECT pname
FROM Product
WHERE price > 100

INSERT INTO Expensive-Product VALUES(‘Gizmo’)

INSERT INTO Product VALUES(‘Gizmo’, NULL)
Updating Views

CREATE VIEW AcmePurchase AS
SELECT customer, product
FROM Purchase
WHERE store = 'AcmeStore'

INSERT INTO AcmePurchase
VALUES('Joe', 'Gizmo')
Updating Views

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

CREATE VIEW AcmePurchase AS
SELECT customer, product
FROM Purchase
WHERE store = 'AcmeStore'

INSERT INTO AcmePurchase
VALUES ('Joe', 'Gizmo')

INSERT INTO Purchase
VALUES ('Joe', 'Gizmo', NULL)

Note this
Updating Views

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

INSERT INTO CustomerPrice VALUES(‘Joe’, 200)

CREATE VIEW CustomerPrice AS

SELECT x.customer, y.price
FROM Purchase x, Product y
WHERE x.product = y.pname

Non-updateable view

Most views are non-updateable
Materialized Views

• The result of the view is materialized

• May speed up query answering significantly

• But the materialized view needs to be synchronized with the base data
Applications of Materialized Views

- Indexes
- Denormalization
- Semantic caching
Indexes

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

Person (name, age, city)

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

May take too long to scan the entire Person table

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:

Example:

```
CREATE INDEX doubleindex ON Person (age, city)
```
Creating Indexes

Indexes can be created on more than one attribute:

Example:

```sql
CREATE INDEX doubleindex ON Person (age, city)
```

Helps in:

```sql
SELECT *
FROM Person
WHERE age = 55 AND city = 'Seattle'
```
Creating Indexes

Indexes can be created on more than one attribute:

Example:

```
CREATE INDEX doubleindex ON Person (age, city)
```

Helps in:

```
SELECT * FROM Person WHERE age = 55 AND city = 'Seattle'
```

and even in:

```
SELECT * FROM Person WHERE age = 55
```
Creating Indexes

Indexes can be created on more than one attribute:

Example:

```
CREATE INDEX doubleindex ON Person (age, city)
```

Helps in:

```
SELECT *
FROM   Person
WHERE  age = 55 AND city = 'Seattle'
```

and even in:

```
SELECT *
FROM   Person
WHERE  age = 55
```

But not in:

```
SELECT *
FROM   Person
WHERE  city = 'Seattle'
```
Indexes are Materialized Views

Product(pid, name, weight, price, …)

CREATE INDEX W ON Product(weight)
CREATE INDEX P ON Product(price)

W(pid, weight)
P(pid, price)

SELECT weight, price
FROM Product
WHERE weight > 10
    and price < 100

SELECT x.weight, y.price
FROM W x, P y
WHERE x.weight > 10
    and y.price < 100
    and x.pid = y.pid
Denormalization

- Compute a view that is the join of several tables
- The view is now a relation that is not in normal form WHY?

```sql
CREATE VIEW CustomerPrice AS
SELECT *
FROM Purchase x, Product y
WHERE x.product = y.pname
```
Semantic Caching

- Queries Q1, Q2, … have been executed, and their results are stored in main memory
- Now we need to compute a new query Q
- Sometimes we can use the prior results in answering Q
- These queries can be seen as materialized views
Technical Challenges in Managing Views

• Synchronizing materialized views
  – A.k.a. incremental view maintenance, or incremental view update

• Answering queries using views
Synchronizing Materialized Views

• Immediate synchronization = after each update
• Deferred synchronization
  – Lazy = at query time
  – Periodic
  – Forced = manual

Which one is best for: indexes, data warehouses, replication?
CREATE VIEW FullOrder AS
  SELECT x.cid, x.pid, x.date, y.name, y.price
  FROM Order x, Product y
  WHERE x.pid = y.pid

UPDATE Product
SET price = price / 2
WHERE pid = '12345'

UPDATE FullOrder
SET price = price / 2
WHERE pid = '12345'

No need to recomputed the entire view!
CREATE VIEW Categories AS
SELECT DISTINCT category
FROM Product

DELETE Product
WHERE pid = '12345'

DELETE Categories
WHERE category in
(SELECT category
FROM Product
WHERE pid = '12345')

It doesn’t work! Why? How can we fix it?
Incremental View Update

Product(pid, name, category, price)

CREATE VIEW Categories AS
    SELECT category, count(*) as c
    FROM Product
    GROUP BY category

DELETE Product
WHERE pid = '12345'

UPDATE Categories
SET c = c-1
WHERE category in
    (SELECT category
     FROM Product
     WHERE pid = '12345')
DELETE Categories
WHERE c = 0
Answering Queries Using Views

- We have several materialized views:
  - V1, V2, ..., Vn
- Given a query Q
  - Answer it by using views instead of base tables
- Variation: *Query rewriting using views*
  - Answer it by rewriting it to another query first
- Example: if the views are indexes, then we rewrite the query to use indexes
Rewriting Queries Using Views

Purchase(buyer, seller, product, store)
Person(pname, city)

Goal: rewrite this query in terms of the view

Have this materialized view:

```
CREATE VIEW SeattleView AS
SELECT y.buyer, y.seller, y.product, y.store
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND x.pname = y.buyer
```

SELECT y.buyer, y.seller
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND x.pname = y.buyer AND y.product = 'gizmo'
Rewriting Queries Using Views

```
SELECT  y.buyer, y.seller
FROM    Person x, Purchase y
WHERE   x.city = 'Seattle' AND
        x..pname = y.buyer AND
        y.product= 'gizmo'
```

```
SELECT  buyer, seller
FROM    SeattleView
WHERE   product= 'gizmo'
```
Rewriting is not always possible

CREATE VIEW DifferentView AS
SELECT y.buyer, y.seller, y.product, y.store
FROM Person x, Purchase y, Product z
WHERE x.city = 'Seattle' AND x.pname = y.buyer AND y.product = z.name AND z.price < 100

SELECT y.buyer, y.seller
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND x..pname = y.buyer AND y.product = 'gizmo'

“Maximally contained rewriting”

SELECT buyer, seller
FROM DifferentView
WHERE product = 'gizmo'