Outline

• Integrity constraints: Chapter 5.7

• Triggers: Chapter 5.8;
  Also recommended: the other textbook

• Views: Chapters 3.6, 25.8, 25.9
  We discuss here material that is NOT covered in ANY books
Constraints in SQL

• A constraint = a property that we’d like our database to hold

• The system will enforce the constraint by taking some actions:
  – forbid an update
  – or perform compensating updates
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

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

```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>

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
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

Referential integrity constraints
May write just Product (why ?)
<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>
Foreign Key Constraints

- OR

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
What happens during updates?

Types of updates:

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

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>ProdName</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Camera</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

READING ASSIGNMENT: 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
CREATE TABLE Purchase (  
prodName CHAR(30),  
date DATETIME NOT NULL)
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)
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

• Apply constraints to rewrite the query
• Simple example:

\[
\text{SELECT } x.a \text{ FROM } R \times, S \ y \text{ WHERE } x.fk=y.key \\
\text{same as} \\
\text{SELECT } x.a \text{ FROM } R.x
\]

• More advanced optimizations possible using complex constraints
Triggers

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

Trigger = Event + Condition + Action

Recommended reading: Chapt. 7 from The Complete Book
Triggers in SQL

• A trigger contains an event, a condition, an action.
• Event = INSERT, DELETE, UPDATE
• Condition = any WHERE condition (may refer to the old and the new values)
• Action = more inserts, deletes, updates
• Many, many more bells and whistles...
• Read in the book (it only scratches the surface...)
Triggers

Enable the database programmer to specify:
• when to check a constraint,
• what exactly to do.

A trigger has 3 parts:

• An event (e.g., update to an attribute)
• A condition (e.g., a query to check)
• An action (deletion, update, insertion)

When the event happens, the system will check the constraint, and if satisfied, will perform the action.

NOTE: triggers may cause cascading effects.
Database vendors did not wait for standards with triggers!
Elements of Triggers (in SQL3)

• Timing of action execution: before, after or instead of triggering event

• The action can refer to both the old and new state of the database.

• Update events may specify a particular column or set of columns.

• A condition is specified with a WHEN clause.

• The action can be performed either for
  • once for every tuple, or
  • once for all the tuples that are changed by the database operation.
Example: Row Level Trigger

CREATE TRIGGER InsertPromotions

AFTER UPDATE OF price ON Product
REFERENCING
  OLD AS OldTuple
  NEW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)

INSERT INTO Promotions(name, discount)
VALUES OldTuple.name,
  (OldTuple.price-NewTuple.price)*100/OldTuple.price
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 average-price-preserve 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)
Bad Things Can Happen

CREATE TRIGGER Bad-trigger

AFTER UPDATE OF price IN Product
REFERENCING OLD AS OldTuple
    NEW AS NewTuple
FOR EACH ROW WHEN (NewTuple.price > 50)

UPDATE Product
SET price = NewTuple.price * 2
WHERE name = NewTuple.name
Triggers v.s. Integrity Constraints

• Triggers can be used to enforce ICs

• More versatile:
  – Your project: ORDER should always “get” the address from CUSTOMER

• May have other usages:
  – User alerts, generate log events for auditing

• Hard to understand
  – E.g. recursive triggers
Views

Views are relations, except that they are not physically stored.

For presenting different information to different users,

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

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

Payroll has access to **Employee**, others only to **Developers**
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
Issues in Virtual Views

• Query Modification

• Applications

• Updating views

• Query minimization
Queries Over Views: Query Modification

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

<table>
<thead>
<tr>
<th>Query:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT u.customer, v.store</td>
</tr>
<tr>
<td>FROM CustomerPrice u, Purchase v</td>
</tr>
<tr>
<td>WHERE u.customer = v.customer AND u.price &gt; 100</td>
</tr>
</tbody>
</table>
Queries Over Views: Query Modification

**Modified query:**

```sql
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 rewritten 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
```
But What About This?

```
SELECT DISTINCT u.customer, v.store
FROM CustomerPrice u, Purchase v
WHERE u.customer = v.customer AND u.price > 100
```
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
Set v.s. Bag Semantics

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

Set semantics

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

Bag semantics
Inlining Queries: Sets/Sets

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

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

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

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

Inlining Queries: Bags/Sets

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

• Logical data independence
  Typical examples:
  – Vertical data partitioning
  – Horizontal data partitioning

• Security
  – Table V 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>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Name</td>
<td>Address</td>
</tr>
<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>

<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>
<tr>
<td>345345</td>
<td>Clob3…</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

When do we use vertical partitioning?
Vertical Partitioning

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

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

Applications:

• 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

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

Which tables are inspected by the system?

WHY ???
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 allowed to see only <0 balances

CREATE VIEW BadCreditCustomers
SELECT *
FROM Customers
WHERE Balance < 0
Updating Views

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

**Updateable view**

```
INSERT INTO Expensive-Product VALUES('Gizmo')
```

```
INSERT INTO Product VALUES('Gizmo', NULL)
```
Updating Views

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

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

INSERT INTO Toy-Product
VALUES(‘Joe’, ‘Gizmo’)

INSERT INTO Product
VALUES(‘Joe’, ’Gizmo’,NULL)

Note this

Updateable view
Updating Views

**Purchase**(
  customer, product, store)

**Product**(
  **pname**, price)

```sql
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**
Query Minimization

Order(cid, pid, date)
Product(pid, name, weight, 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 < 100

CREATE VIEW LightOrders AS
SELECT a.cid, a.pid, a.date, b.name, b.price
FROM Order a, Product b
WHERE a.pid = b.pid and b.weight < 100

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

Customers who bought a cheap, light product
Query Minimization

Order(cid, pid, date)
Product(pid, name, weight, 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 < 100

CREATE VIEW LightOrders AS
  SELECT a.cid, a.pid, a.date, b.name, b.price
  FROM Order a, Product b
  WHERE a.pid = b.pid and b.weight < 100

SELECT u.cid
FROM CheapOrders u,
    LightOrders 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
Query Minimization under Bag Semantics

Rule 1: If $x, y$ are tuple variables over the same table and $x.id = y.id$, then combine $x, y$ into a single variable

Rule 2: If $x$ ranges over $S$, $y$ ranges over $T$, and the only condition on $y$ is $x.fk = y.key$, then remove $T$ from the query
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 < 100 and b.weight < 10
  and x.cid = a.cid and x.pid = a.pid

x = a

SELECT a.cid
FROM Order x, Product y, Product b
WHERE x.pid = y.pid and x.pid = b.pid
  and y.price < 100 and b.weight < 10

y = b

SELECT a.cid
FROM Order x, Product y
WHERE x.pid = y.pid and
  y.price < 100 and x.weight < 10
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 there exists a homomorphism from Q to Q’ then Q’ is equivalent to Q, hence one can safely remove x.

Definition. A homomorphism from Q to Q’ is mapping h from the tuple variables of Q to those of Q’ s.t. for every predicate P in the WHERE clause of Q, the predicate h(P) is logically implied by the WHERE clause in Q’
Homomorphism

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

\[H(x) = x', \quad H(y) = H(z) = z'
\]

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

Examples:

- Indexes
- Join indexes
- Views in data warehouses
- Distribution/replication
Issues with Materialized Views

• Synchronization
  – View becomes stale when base tables get updated

• Query rewriting using views
  – Much harder than query modification

• View selection
  – Given a choice, which views should we materialize?
View Synchronization

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

Which one is best for: indexes, data warehouses, replication?
Denormalization:
Story From the Trenches

Graduate Admissions:

• Application(id, name, school)
  GRE(id, score, year)  /* normalization ! */
• Very common query:
  List(id, name, school,
      GRE-some-average-or-last-score)
• VERY SLOW !
• Solution: Application(id,name,school,GRE)
• De-normalized; computed field; materialized view
• Synchronized periodically (once per night).
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 recompute the entire view!
Incremental View Update

Product(pid, name, category, price)

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?
Answering Queries Using Views

• What if we want to use a set of views to answer a query.
• Why?
  – The obvious reason…
Reusing a Materialized View

• Suppose I have only the result of SeattleView:

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

• and I want to answer the query

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

Then, I can rewrite the query using the view.
Query Rewriting Using Views

Rewritten query:

```
SELECT  buyer, seller
FROM    SeattleView
WHERE   product= 'gizmo'
```

Original query:

```
SELECT  y.buyer, y.seller
FROM    Person x, Purchase y
WHERE   x.city = ‘Seattle’  AND
        x..pname = y.buyer AND
        y.product=‘gizmo’.
```
Another Example

• I still have **only** the result of SeattleView:

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

• but I want to answer the query

  ```sql
  SELECT  y.buyer, y.seller
  FROM    Person x, Purchase y
  WHERE   x.city = 'Seattle' AND
          x.pname = y.buyer AND
          x.Phone LIKE '206 543 %'.
  ```
And Now?

• I still have **only** the result of SeattleOtherView:
  
  ```sql
  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
  ```

• but I want to answer the query
  
  ```sql
  SELECT y.buyer, y.seller
  FROM Person x, Purchase y
  WHERE x.city = 'Seattle' AND
  x.pname = y.buyer.
  ```
And Now?

- I still have **only** the result of:
  
  SELECT  seller, buyer, Sum(Price)
  FROM    Purchase
  WHERE   Purchase.store = ‘The Bon’
  Group By seller, buyer

- but I want to answer the query
  
  SELECT  seller, Sum(Price)
  FROM    Purchase
  WHERE   Person.store = ‘The Bon’
  Group By seller

And what if it’s the other way around?
Finally…

- I still have **only** the result of:
  - `SELECT` seller, buyer, `Count(*)`
  - `FROM` Purchase
  - `WHERE` Purchase.store = ‘The Bon’
  - `Group By` seller, buyer

- but I want to answer the query
  - `SELECT` seller, `Count(*)`
  - `FROM` Purchase
  - `WHERE` Person.store = ‘The Bon’
  - `Group By` seller
The General Problem

- Given a set of views $V_1, \ldots, V_n$, and a query $Q$, can we answer $Q$ using only the answers to $V_1, \ldots, V_n$?
Application 1: Horizontal Partition

CREATE VIEW CustomersInHuston AS
    SELECT *
    FROM Customers
    WHERE city=‘Huston’

CREATE VIEW CustomersInSeattle AS
    SELECT *
    FROM Customers
    WHERE city=‘Seattle’

....

No more unions!
Application 1: Horizontal Partition

SELECT name
FROM Customer
WHERE city = 'Seattle'

Rewrite using available views:

SELECT name
FROM CustomersInSeattle

This is query rewriting using views
Application 2: Aggressive Use of Indexes

Product(pid, name, weight, price, …many other attributes)

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

DMBS stores three files: Product (big) W P (smaller)

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

Which files are needed to answer the query?
Indexes ARE Views

Product(pid, name, weight, price, …many other attributes)

CREATE VIEW W AS
SELECT pid, weight
FROM Product

CREATE VIEW P AS
SELECT pid, weight
FROM Product

CREATE INDEX W ON Product(weight)

CREATE INDEX P ON Product(price)

CREATE VIEW W AS
SELECT pid, weight
FROM Product

CREATE VIEW P AS
SELECT pid, weight
FROM Product
Indexes ARE Views

Product(pid, name, weight, price, …many other attributes)

CREATE VIEW W AS
    SELECT pid, weight
    FROM Product

CREATE VIEW P AS
    SELECT pid, weight
    FROM Product

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

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

This, too, is query rewriting using views
Application 3: 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
• This, too, is a form of query rewriting using views (why ?)