Lecture 03
Views, Constraints

Tuesday, January 23, 2007

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

simplest

Most complex

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:

Product(<u>name</u>, category)

```
CREATE TABLE Product (
        name CHAR(30),
        category VARCHAR(20)
PRIMARY KEY (name))
```

Keys with Multiple Attributes

```
CREATE TABLE Product (
        name CHAR(30),
        category VARCHAR(20),
        price INT,
    PRIMARY KEY (name, category))
```

Name	Category	Price
Gizmo	Gadget	10
Camera	Photo	20
Gizmo	Photo	30
~~Gizmo~~	~~Gadget~~	~~40~~

Product(<u>name, category</u>, 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

Referential integrity constraints

```
CREATE TABLE Purchase (
      prodName CHAR(30)
            REFERENCES Product(name),
      date DATETIME)
```

May write just Product (why ?)

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

Product

<u>Name</u>	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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 ASSIGNEMNT: 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)
```

What
is the difference from
Foreign-Key ?

```
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:

  SELET x.a FROM R x, S y WHERE x.fk=y.key
  same as
  SELECT x.a 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
```

Event

Condition

Action

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
```

Trigers 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)

```
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(<u>pname</u>, 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"

Purchase(customer, product, store)

Product(<u>pname</u>, price)

CustomerPrice(customer, price)

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

- <u>Virtual</u> views:
  - Used in databases
  - Computed only on-demand – slow at runtime
  - Always up to date
- <u>Materialized</u> 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

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
```

Queries Over 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 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
```

↓

??

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
```

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    . . .
```

⇒ **NO**

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

Resumes

SSN	Name	Address	Resume	Picture
234234	Mary	Huston	Clob1…	Blob1…
345345	Sue	Seattle	Clob2…	Blob2…
345343	Joan	Seattle	Clob3…	Blob3…
234234	Ann	Portland	Clob4…	Blob4…

T1

SSN	Name	Address
234234	Mary	Huston
345345	Sue	Seattle
. . .		

T2

SSN	Resume
234234	Clob1…
345345	Clob2…

T3

SSN	Picture
234234	Blob1…
345345	Blob2…

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

SSN	Name	City	Country
234234	Mary	Huston	USA
345345	Sue	Seattle	USA
345343	Joan	Seattle	USA
234234	Ann	Portland	USA
--	Frank	Calgary	Canada
--	Jean	Montreal	Canada

CustomersInHuston

SSN	Name	City	Country
234234	Mary	Huston	USA

CustomersInSeattle

SSN	Name	City	Country
345345	Sue	Seattle	USA
345343	Joan	Seattle	USA

CustomersInCanada

SSN	Name	City	Country
--	Frank	Calgary	Canada
--	Jean	Montreal	Canada

Horizontal Partitioning

```
CREATE VIEW Customers AS
    CustomersInHuston
        UNION ALL
    CustomersInSeattle
        UNION ALL
     . . .
```

Horizontal Partitioning

```
SELECT name
FROM   Cusotmers
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   Cusotmers
WHERE  city = ‘Seattle’
```

```
SELECT name
FROM   CusotmersInSeattle
```

Horizontal Partitioning

Applications:

- Optimizations:
  - E.g. archived applications and active applications
- Distributed databases
- Data integration

Views and Security

Customers:

Name	**Address**	**Balance**
Mary	Huston	450.99
Sue	Seattle	-240
Joan	Seattle	333.25
Ann	Portland	-520

Fred is not allowed to see this

Fred is allowed to see this

```
CREATE VIEW PublicCustomers
        SELECT Name, Address
        FROM Customers
```

Views and Security

Customers:

Name	**Address**	**Balance**
Mary	Huston	450.99
Sue	Seattle	-240
Joan	Seattle	333.25
Ann	Portland	-520

John is allowed to see only <0 balances

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

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)
```

Updateable view

Updating Views

Purchase(customer, product, store)

Product(<u>pname</u>, price)

```
INSERT
INTO Toy-Product
VALUES('Joe', 'Gizmo')
```

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

Updateable view

```
INSERT
INTO Product
VALUES('Joe','Gizmo',NULL)
```

Note this

Updating Views

Purchase(customer, product, store)

Product(<u>pname</u>, 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

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
```

Customers who bought a cheap, light product

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

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

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
```

H(x) = x', H(y) = H(z) = z'

Q'

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

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).

Incremental View Update

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

```
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:

```
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

```
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:

```
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

```
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 V1,…,Vn, and a query Q, can we answer Q using only the answers to V1,…,Vn?

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(<u>pid</u>, name, weight, price, …many other attributes)

```
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 ?)