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

simple

- Keys, foreign keys
- Attribute-level constraints
- Tuple-level constraints
- Global constraints: assertions

complex

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:

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

```
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))
```

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

```
CREATE TABLE Purchase (
  buyer CHAR(30),
  seller CHAR(30),
  product CHAR(30) REFERENCES Product(name),
  store VARCHAR(30))
```

Foreign key

Purchase(buyer, seller, product, store)
Product(<u>name</u>, price)

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Foreign Key Constraints

```
CREATE   TABLE Purchase(
   buyer VARCHAR(50),
   seller VARCHAR(50),
   product CHAR(20),
   category VAVRCHAR(20),
   store VARCHAR(30),
   FOREIGN KEY (product, category)
       REFERENCES Product(name, category)
  );
```

Purchase(buyer, seller, product, category, store)
Product(<u>name, category</u>, price)

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

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) . . .
```

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 using Constraints

Purchase(buyer, seller, product, store)
Product(<u>name</u>, price)

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

⇩

Why ? and When ?

```
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
```

Event

Condition

Action

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)
```

Trigers 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(<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

- 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

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

CustomerPrice(customer, price)

Queries Over Virtual 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
```

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

CustomerPrice(customer, price)

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
```

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

CustomerPrice(customer, price)

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
```

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

CustomerPrice(customer, price)

Another Example

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

↓

??

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

CustomerPrice(customer, price)

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

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
```

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

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

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

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

```
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   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 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

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

Set semantics

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

Bag semantics

Unnesting: 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    . . .
```

Unnesting: 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    . . .
```

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    . . .
```

⇒

NO

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(<u>cid, pid</u>, weight, date)
Product(<u>pid</u>, 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

What constraints do we need to have for this optimization ?

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
```

```
SELECT  x.cid, x.date
  FROM     Order x
  WHERE   x.weight > 20
```

Q: Where do we encounter non-minimized queries ?

Query Minimization under Bag Semantics

Order(<u>cid, pid</u>, weight, date)
Product(<u>pid</u>, name, price)

A: in queries resulting from view inlining

```
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

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

Q

Q'

```
SELECT A
FROM R1 x1, …, Rk xk
WHERE C
```

```
SELECT A'
FROM R1' y1, …, Rm' ym
WHERE C'
```

A <u>*homomorphism*</u> from Q' to Q
is a mapping h : {y1, …, ym} → {x1, …, xk}
such that:
(a) If h(yi) = xj, then Ri' = Rj
(b) C logically implies h(C') and
(c) h(A') = A

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

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

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.pid = z.pid
    and y.price < 99
    and x.weight > 150
    and z.weight > 100
```

Order(<u>cid, pid</u>, weight, date)
Product(<u>pid</u>, 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'

```
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'

```
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
```

Q and Q' are equivalent !

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

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'

Find a homomorphism h: Q → Q'

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

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)

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

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

Updateable view

Updatable Views

- Have a virtual view V(A1, A2, …) over tables R1, R2, …
- User wants to *update* a tuple in V
  - Insert/modify/delete
- Can we translate this into updates to R1, R2, … ?
- If yes: V = “an updateable view”
- If not: V = “a non-updateable view”

Updating Views

Purchase(customer, product, store)

Product(<u>pname</u>, 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 AcmePurchase
VALUES('Joe', 'Gizmo')
```

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

Updateable view

Updating Views

Purchase(customer, product, store)

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

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

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

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

Updateable view

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)

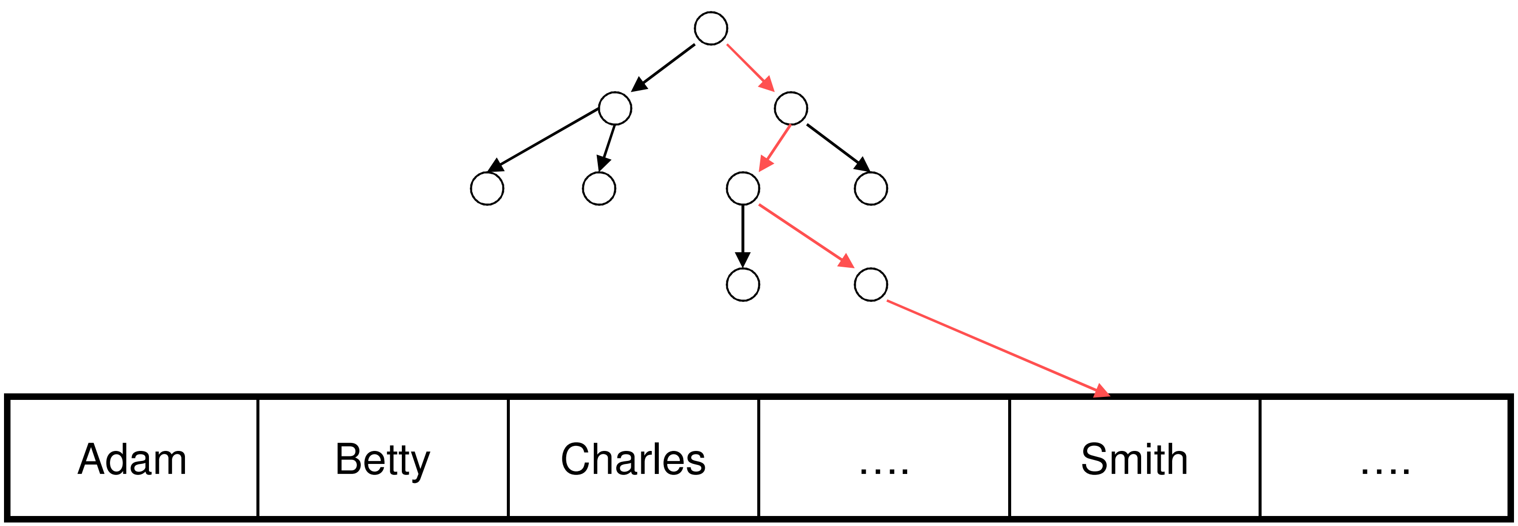
```
SELECT *
FROM     Person
WHERE   name = 'Smith'
```

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:

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

Helps in:

```
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, …)

(big)

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

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

(smaller)

```
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 ?

Purchase(customer, product, store)
Product(<u>pname</u>, price)

```
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 ?

Incremental View Update

Order(<u>cid, pid</u>, date)
Product(<u>pid</u>, 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 ?

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(<u>pname</u>, city)

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
```

Goal: rewrite this query in terms of the view

```
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’
```