CSE 544
Data Models and Views

Lecture #4

Wednesday, January 19, 2011

Announcements

- Projects: please sign up to meet with me on Friday, between 11-1pm (need about 15’). Before that do this:
  - Form team
  - Choose project
  - Think, so we can have a meaningful discussion
- Homework 1: due on Monday, 12pm (before the lecture)

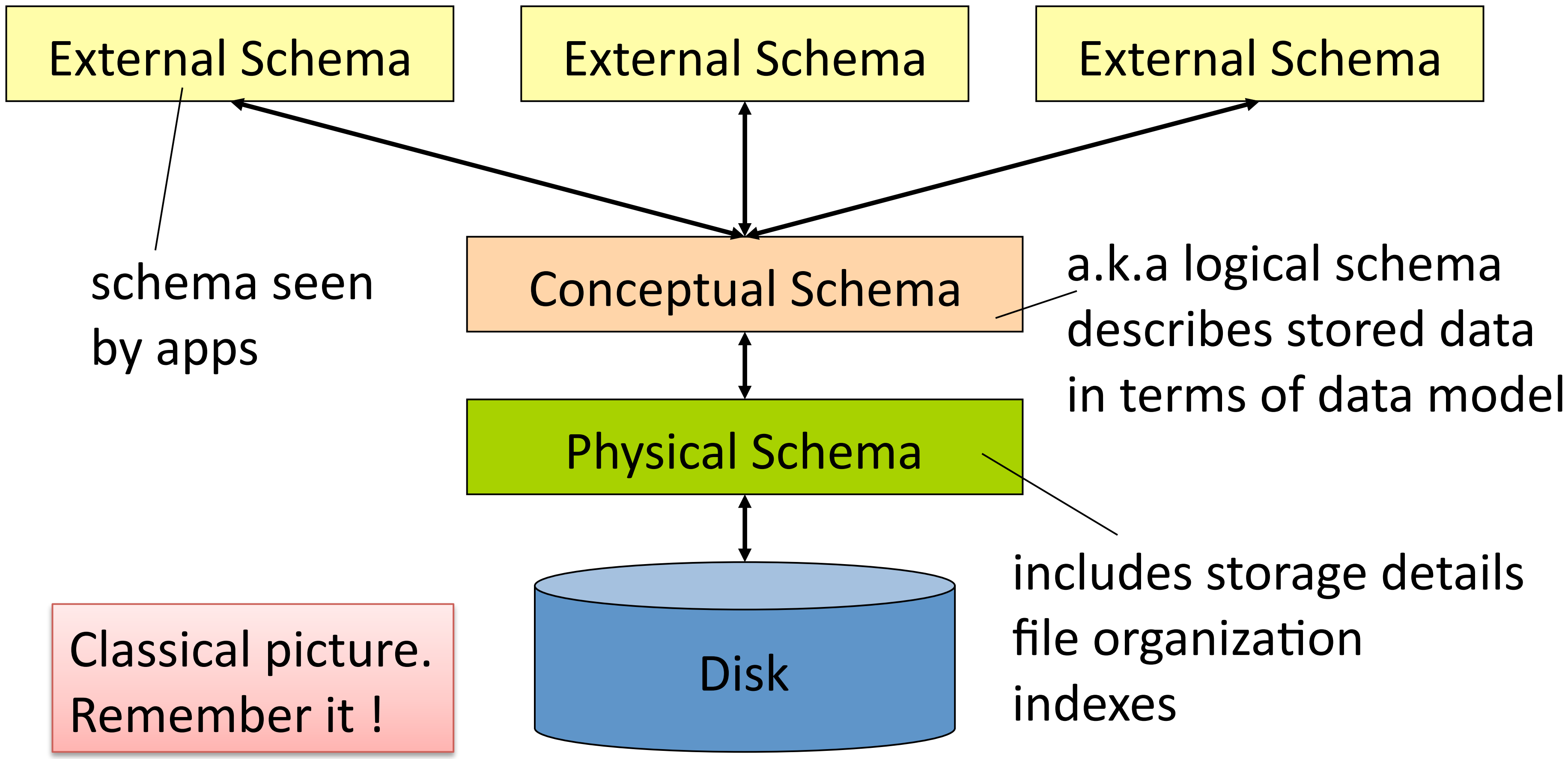
References

- M. Stonebraker and J. Hellerstein. What Goes Around Comes Around. In "Readings in Database Systems" (aka the Red Book). 4th ed.

Data Model Motivation

- User is concerned with real-world data
  - Data represents different aspects of user's business
  - Data typically includes entities and relationships between them
  - Example entities are students, courses, products, clients
  - Example relationships are course registrations, product purchases
- User somehow needs to define data to be stored in DBMS
- Data model enables a user to define the data using high-level constructs without worrying about many low-level details of how data will be stored on disk

Levels of Abstraction



Outline

- Different types of data
- Early data models
  - IMS
  - CODASYL
- Physical and logical independence in the relational model
- Other data models

Different Types of Data

- **Structured data**
  - What is this ? Examples ?
- **Semistructured data**
  - What is this ?
  - Examples ?
- **Unstructured data**
  - What is this ? Examples ?

Different Types of Data

- **Structured data**
  - All data conforms to a schema. Ex: business data
- **Semistructured data**
  - Some structure in the data but implicit and irregular
  - Ex: resume, ads
- **Unstructured data**
  - No structure in data. Ex: text, sound, video, images

- Our focus: structured data & relational DBMSs

Outline

- Different types of data
- Early data models
  - IMS – late 1960’s and 1970’s
  - CODASYL – 1970’s
- Physical and logical independence in the relational model
- Other data models

Early Proposal 1: IMS

- What is it ?

Early Proposal 1: IMS

- **Hierarchical data model**

- **Record**
  - **Type**: collection of named fields with data types (+)
  - **Instance**: must match type definition (+)
  - Each instance must have a **key** (+)
  - Record types must be arranged in a **tree** (-)

- **IMS database** is collection of instances of record types organized in a tree

IMS Example

- See Figure 2 in paper “What goes around comes around”

Data Manipulation Language: DL/1

- How does a programmer retrieve data in IMS ?

Data Manipulation Language: DL/1

- Each record has a hierarchical sequence key (HSK)
  - Records are totally ordered: depth-first and left-to-right

- HSK defines semantics of commands:
  - get_next
  - get_next_within_parent

- **DL/1 is a record-at-a-time language**
  - Programmer constructs an algorithm for solving the query
  - Programmer must worry about query optimization

Data storage

- How is the data physically stored in IMS ?

Data storage

- Root records
  - Stored sequentially (sorted on key)
  - Indexed in a B-tree using the key of the record
  - Hashed using the key of the record
- Dependent records
  - Physically sequential
  - Various forms of pointers
- Selected organizations restrict DL/1 commands
  - No updates allowed with sequential organization
  - No "get-next" for hashed organization

Data Independence

- What is it ?

Data Independence

- **Physical data independence**: Applications are insulated from changes in **physical storage details**

- **Logical data independence**: Applications are insulated from changes to **logical structure of the data**

- **Why are these properties important?**
  - Reduce program maintenance as
  - Logical database design changes over time
  - Physical database design tuned for performance

IMS Limitations

- **Tree-structured data model**
  - **Redundant** data, existence **depends on parent, artificial structure**
- **Record-at-a-time** user interface
  - User must specify **algorithm** to access data
- **Very limited physical independence**
  - Phys. organization limits possible operations
  - Application programs break if organization changes
- Provides **some logical independence**
  - DL/1 program runs on logical database
  - Difficult to achieve good logical data independence with a tree model

Early Proposal 2: CODASYL

- What is it ?

Early Proposal 2: CODASYL

- **Networked data model**

- Primitives are also **record types** with **keys** (+)
- Network model is **more flexible than hierarchy(+)**
  - Ex: no existence dependence
- Record types are organized into **network** (-)
  - A record can have multiple parents
  - Arcs between records are named
  - At least one entry point to the network
- **Record-at-a-time** data manipulation language (-)

CODASYL Example

- See Figure 5 in paper “What goes around comes around”

CODASYL Limitations

- **No physical data independence**
  – Application programs break if organization changes
- **No logical data independence**
  – Application programs break if organization changes
- Very **complex**
- Programs must “**navigate** the hyperspace”
- Load and recover as **one gigantic object**

Outline

- Different types of data
- Early data models
  - IMS
  - CODASYL
- Physical and logical independence in the relational model
- Other data models

Relational Model Overview

- Proposed by Ted Codd in 1970
- Motivation: better logical and physical data independence

Relational Model Overview

- Defines logical schema only
  - No physical schema

- Set-at-a-time query language

Physical Independence

- Definition: **Applications are insulated from changes in physical storage details**
- Early models (IMS and CODASYL): No
- Relational model: Yes
  - Yes through set-at-a-time language: algebra or calculus
  - No specification of what storage looks like
  - Administrator can optimize physical layout

Physical Independence

- Definition: **Applications are insulated from changes in physical storage details**

- Early models (IMS and CODASYL): No

- Relational model: Yes
  - Yes through set-at-a-time language: algebra or calculus
  - No specification of what storage looks like
  - Administrator can optimize physical layout

Logical Independence

- Definition: **Applications are insulated from changes to logical structure of the data**
- Early models
  - IMS: some logical independence
  - CODASYL: no logical independence
- Relational model
  - Yes through views

Great Debate

- Pro relational
  - What where the arguments ?

- Against relational
  - What where the arguments ?

- How was it settled ?

Great Debate

- Pro relational
  - CODASYL is too complex
  - CODASYL does not provide sufficient data independence
  - Record-at-a-time languages are too hard to optimize
  - Trees/networks not flexible enough to represent common cases
- Against relational
  - COBOL programmers cannot understand relational languages
  - Impossible to represent the relational model efficiently
  - CODASYL can represent tables
- Ultimately settled by the market place

Outline

- Different types of data
- Early data models
  - IMS
  - CODASYL
- Physical and logical independence in the relational model
- Other data models

Other Data Models

- Entity-Relationship: 1970's
  – Successful in logical database design (next lecture)
- Extended Relational: 1980's
- Semantic: late 1970's and 1980's
- Object-oriented: late 1980's and early 1990's
  – Address impedance mismatch: relational dbs ←→ OO languages
  – Interesting but ultimately failed (several reasons, see paper)
- Object-relational: late 1980's and early 1990's
  – User-defined types, ops, functions, and access methods
- Semi-structured: late 1990's to the present

Summary

- Data independence is desirable
  - Both physical and logical
  - Early data models provided very limited data independence
  - Relational model facilitates data independence
    - Set-at-a-time languages facilitate phys. indep. [more next lecture]
    - Simple data models facilitate logical indep. [more next lecture]
- Flat models are also simpler, more flexible
- User should specify what they want not how to get it
  - Query optimizer does better job than human
- New data model proposals must
  - Solve a "major pain" or provide significant performance gains

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”

Purchase(customer, product, store)
Product(pname, 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:
  - Pros/cons ???

- <u>Materialized</u> views
  - Pros/cons ??

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)

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)

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)

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, rarely accessed
  - E.g. Picture
- In distributed databases
  - Customer info site 1, customer orders at site 2
- In data integration
  - T1 comes from one source
  - T2 comes from a different source

Horizontal Partitioning

Customers

SSN	Name	City
234234	Mary	Huston
345345	Sue	Seattle
345343	Joan	Seattle
234234	Ann	Portland
--	Frank	Spokane
--	Jean	Spokane

CustomersInHuston

SSN	Name	City
234234	Mary	Huston

CustomersInSeattle

SSN	Name	City
345345	Sue	Seattle
345343	Joan	Seattle

CustomersInCanada

SSN	Name	City
--	Frank	Spokane
--	Jean	Spokane

Horizontal Partitioning

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

CustomersInHuston

SSN	Name	City
234234	Mary	Huston

CustomersInSeattle

SSN	Name	City
345345	Sue	Seattle
345343	Joan	Seattle

CustomersInCanada

SSN	Name	City
--	Frank	Spokane
--	Jean	Spokane

Horizontal Partitioning

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

Which tables are inspected by the system ?

CustomersInHuston

SSN	Name	City
234234	Mary	Huston

CustomersInSeattle

SSN	Name	City
345345	Sue	Seattle
345343	Joan	Seattle

CustomersInCanada

SSN	Name	City
--	Frank	Spokane
--	Jean	Spokane

Horizontal Partitioning

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

Which tables are inspected by the system ?

All ! The system doesn't know where 'Seattle' is

CustomersInHuston

SSN	Name	City
234234	Mary	Huston

CustomersInSeattle

SSN	Name	City
345345	Sue	Seattle
345343	Joan	Seattle

CustomersInCanada

SSN	Name	City
--	Frank	Spokane
--	Jean	Spokane

Better

```
CREATE VIEW Customers AS
    SELECT *, 'Huston' AS City
    FROM CustomersInHuston
        UNION ALL
    SELECT *, 'Seattle' AS City
    FROM CustomersInSeattle
        UNION ALL
    . . .
```

CustomersInHuston

SSN	Name	~~City~~
234234	Mary	~~Huston~~

CustomersInSeattle

SSN	Name	~~City~~
345345	Sue	~~Seattle~~
345343	Joan	~~Seattle~~

CustomersInCanada

SSN	Name	~~City~~
--	Frank	~~Spokane~~
--	Jean	~~Spokane~~

Better

```
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

SQL Security Model

- Discretionary access control:
  - Users × Tables × {SELECT, INSERT, UPDATE, …}
  - GRANT and REVOKE commands
- Coarse grained ! Now row-level access control:
  - Each customer is allowed to see his/her own records
- Views are quick fix to that

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

How do we grant Fred access only to Name/Address ?

Views and Security

Fred is not allowed to see this

Customers:

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

Grant **Fred** access to 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

How do we grant John access only to delinquent accounts ?

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 Problems in Virtual Views

- Simplifying queries over virtual views
- Updating virtual views

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 Queries

- Inner query: set/bag semantics
- Outer query: set/bag semantics
- When can we unnest ?

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

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

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

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

```
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      . . .
```

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

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

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

```
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    . . .
```

```
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    . . .
```

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

⇒

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

⇒

```
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    . . .
```

```
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    . . .
```

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

NO

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

```
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     . . .
```

```
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     . . .
```

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

⇒

NO

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

⇒

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

Updating Virtual Views

- V(A1, A2, ..) = view over R1, R2, …
- Insert/modify/delete in/from V
- Can we push this to R1, R2, … ?
  - Updatable view = yes.
  - Non-updatable view = no.

Updatable View

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

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

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

Updatable View

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
```

↓

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

Updatable View

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')
```

Updatable View

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

Nonupdatable 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
```

⇩

? ? ? ? ?

Most views are non-updateable

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)

```
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
```

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)

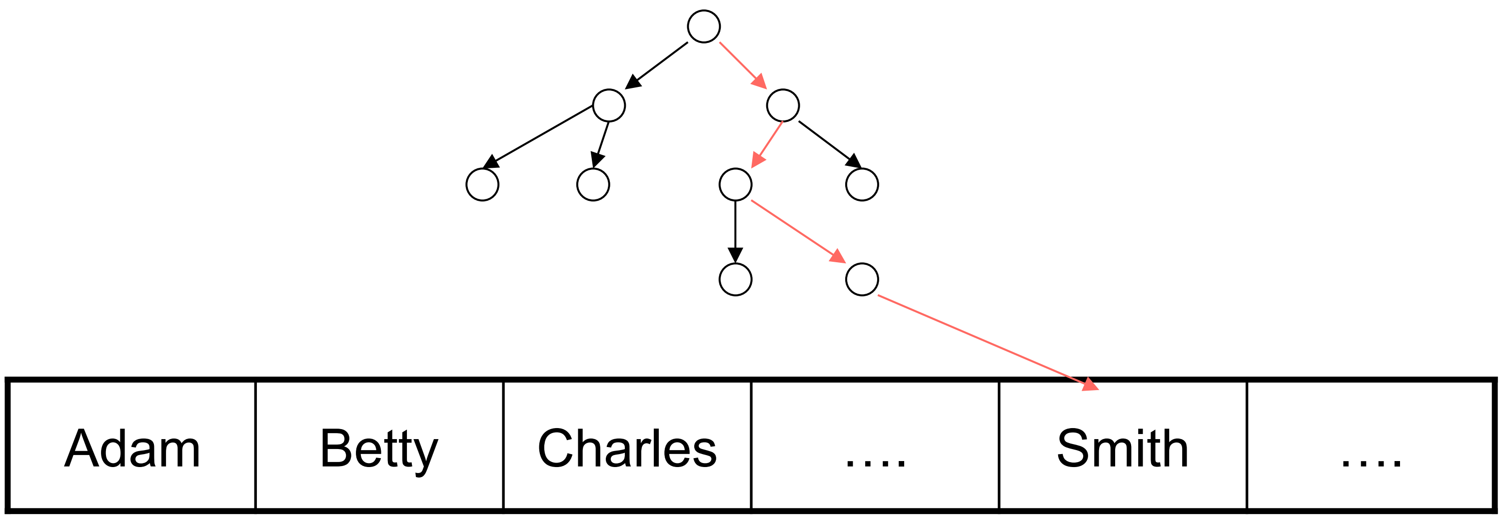
```
SELECT *
FROM    Person
WHERE   name = 'Smith'
```

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)
```

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 ?

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

Incremental View Update

Product(<u>pid</u>, 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'
```