Application Design Using Java

Lecture 20

Database Systems

- *DBMS (Database Management System)* is a software tool for storing and managing large amounts of data
- *Database* is a collection of data organized for a specific application, often stored in a DBMS
- *Database application* is a software product that uses DBMSs to store one or more databases to accomplish a specific purpose

What *Is* a Relational Database Management System?

Database Management System = DBMS

Relational DBMS = RDBMS

- A collection of files that store the data

- A big C program written by someone else that accesses and updates those files for you

Where are RDBMS used?

- Backend for traditional “database” applications
- Backend for large Websites
- Backend for Web services

Example of a Traditional Database Application

Suppose we are building a system

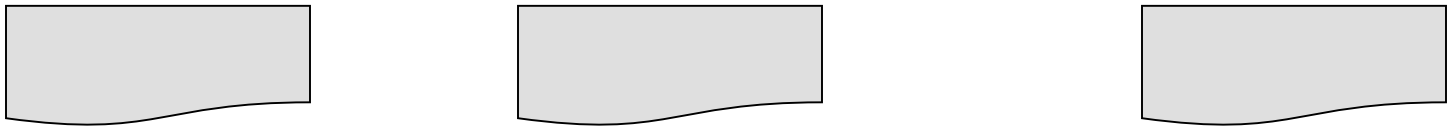
to store the information about:

- students
- courses
- professors
- who takes what, who teaches what

Can we do it without a DBMS ?

Sure we can! Start by storing the data in files:

students.txt courses.txt professors.txt



Now write C or Java programs to implement specific tasks

Doing it without a DBMS...

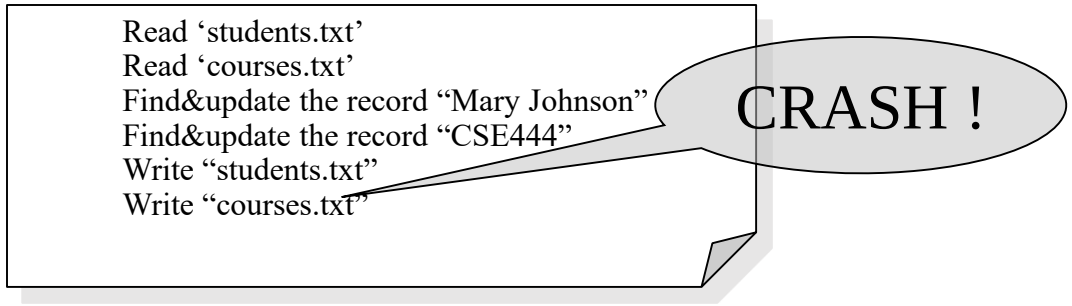
- Enroll “Mary Johnson” in “CSE444”:

Write a C program to do the following:

Read ‘students.txt’
Read ‘courses.txt’
Find&update the record “Mary Johnson”
Find&update the record “CSE444”
Write “students.txt”
Write “courses.txt”

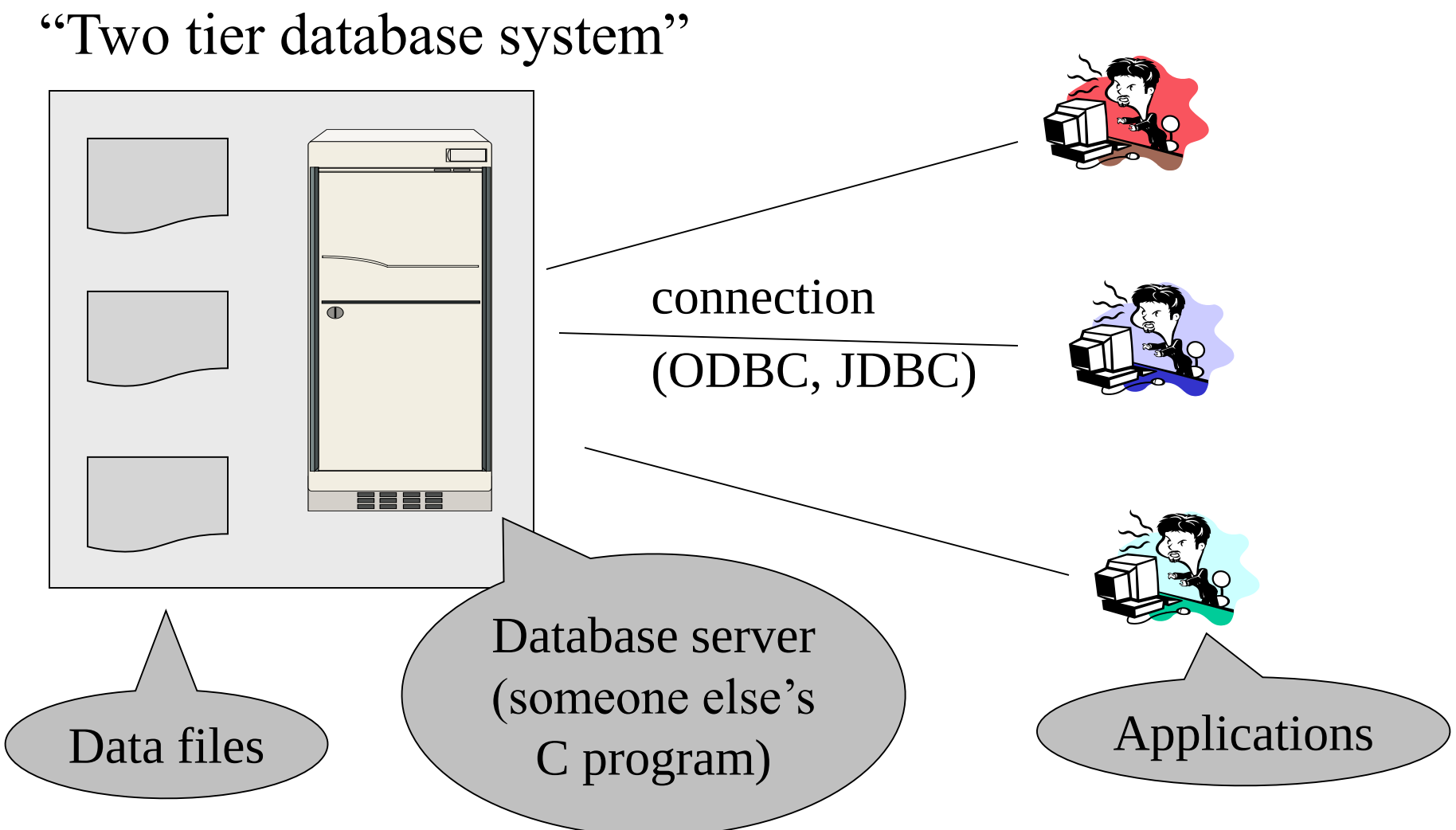
Problems without an DBMS...

- System crashes:



  - What is the problem ?
- Large data sets (say, 50GB)
  - What is the problem ?
- Simultaneous access by many users
  - Need locks: we know them from OS, but now data on disk; and is there any fun to re-implement them ?

Enters a DMBS



Functionality of a DBMS

The programmer sees SQL, which has two components:

- Data Definition Language - DDL
- Data Manipulation Language - DML
  - query language

Behind the scenes the DBMS has:

- Query optimizer
- Query engine
- Storage management
- Transaction Management (concurrency, recovery)

Functionality of a DBMS

Two things to remember:

- Client-server architecture
  - Slow, cumbersome connection
  - But good for the data
- It is just someone else's C program
  - In the beginning we may be impressed by its speed
  - But later we discover that it can be frustratingly slow
  - We can do any particular task faster outside the DBMS
  - But the DBMS is *general* and *convenient*

How the Programmer Sees the DBMS

- Start with DDL to *create tables*:

```
CREATE TABLE Students (
        Name CHAR(30)
        SSN CHAR(9) PRIMARY KEY NOT NULL,
        Category CHAR(20)
)   . . .
```

- Continue with DML to *populate tables:*

```
INSERT INTO Students
VALUES('Charles', '123456789', 'undergraduate')
. . . .
```

How the Programmer Sees the DBMS

- Tables:

Students:

SSN	Name	Category
123-45-6789	Charles	undergrad
234-56-7890	Dan	grad
	…	…

Takes:

SSN	CID
123-45-6789	CSE444
123-45-6789	CSE444
234-56-7890	CSE142
	…

Courses:

CID	Name	Quarter
CSE444	Databases	fall
CSE541	Operating systems	winter

- Still implemented as files, but behind the scenes can be quite complex

"*data independence*" = separate *logical* view from *physical implementation*

Transactions

- Enroll “Mary Johnson” in “CSE444”:

```
BEGIN TRANSACTION;

INSERT INTO Takes
    SELECT Students.SSN, Courses.CID
    FROM Students, Courses
    WHERE Students.name = ‘Mary Johnson’ and
                Courses.name = ‘CSE444’

-- More updates here....

IF everything-went-OK
     THEN COMMIT;
ELSE ROLLBACK
```

If system crashes, the transaction is still either committed or aborted

Transactions

- A *transaction* = sequence of statements that either all succeed, or all fail
- Transactions have the ACID properties:

  A = atomicity

  C = consistency

  I = independence

  D = durability

Queries

- Find all courses that “Mary” takes

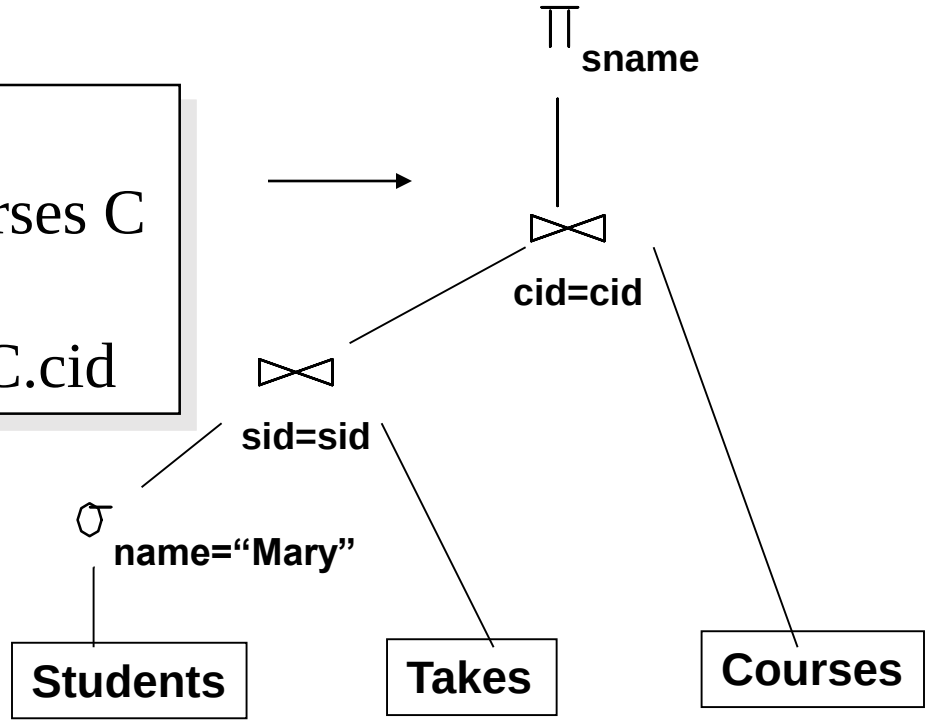
```
SELECT  C.name
FROM    Students S, Takes T, Courses C
WHERE   S.name=“Mary” and
        S.ssn = T.ssn and T.cid = C.cid
```

- What happens behind the scene ?
  - Query processor figures out how to answer the query efficiently.

Queries, behind the scene

Declarative SQL query → *Imperative query execution plan:*

```
SELECT  C.name
FROM Students S, Takes T, Courses C
WHERE S.name="Mary" and
        S.ssn = T.ssn and T.cid = C.cid
```



The **optimizer** chooses the best execution plan for a query

Database Systems

- The big commercial database vendors:
  - Oracle
  - IBM (with DB2) bought Informix recently
  - Microsoft (SQL Server)
  - Sybase
- Some free database systems (Unix) :
  - Postgres
  - MySQL
  - Predator
- We will use MySQL

New Trends in Databases

- Object-relational databases
- Main memory database systems
- XML XML XML !
  - Relational databases with XML support
  - Middleware between XML and relational databases
  - Native XML database systems
  - Lots of research here at UW on XML and databases
- Data integration
- Peer to peer, stream data management – still research

SQL Introduction

- Standard language for querying and manipulating data

- **S**tructured **Q**uery **L**anguage

- Many standards out there:
- ANSI SQL
- SQL92 (a.k.a. SQL2)
- SQL99 (a.k.a. SQL3)
- Vendors support various subsets of these
- What we discuss is common to all of them

SQL

- Data Definition Language (DDL)
  - Create/alter/delete tables and their attributes
- Data Manipulation Language (DML)
  - Query one or more tables
  - Insert/delete/modify tuples in tables

Data in SQL

1. Atomic types, a.k.a. data types
2. Tables built from atomic types

Unlike XML, no nested tables, only flat tables are allowed!

- We will see later how to decompose complex structures into multiple flat tables

Data Types in SQL

- Characters:
  - CHAR(20) -- fixed length
  - VARCHAR(40) -- variable length
- Numbers:
  - BIGINT, INT, SMALLINT, TINYINT
  - REAL, FLOAT -- differ in precision
  - MONEY
- Times and dates:
  - DATE
  - DATETIME -- SQL Server
- Others... All are simple

Tables in SQL

•Attribute names

•Product

•Table name

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

•Tuples or rows

Tables Explained

- A tuple = a record
  - Restriction: all attributes are of atomic type
- A table = a set of tuples
  - Like a list…
  - …but it is unordered: no **first()**, no **next()**, no **last()**.

Tables Explained

- The *schema* of a table is the table name and its attributes:

Product(PName, Price, Category, Manfacturer)

- A *key* is an attribute whose values are unique; we underline a key

Product(PName, Price, Category, Manfacturer)

SQL Query

•Basic form: (plus many many more bells and whistles)

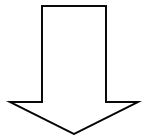
- SELECT attributes
- FROM relations (possibly multiple)
- WHERE conditions (selections)

Simple SQL Query

•Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
•SELECT  *
FROM     Product
WHERE    Category='Gadgets'
```



PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

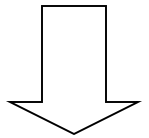
•"selection"

Simple SQL Query

•Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

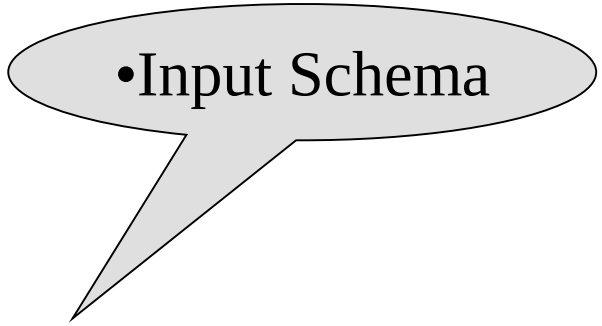
```
•SELECT  PName, Price, Manufacturer
FROM     Product
WHERE    Price > 100
```



•"selection" and
•"projection"

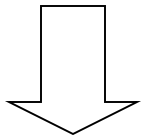
PName	Price	Manufacturer
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A Notation for SQL Queries

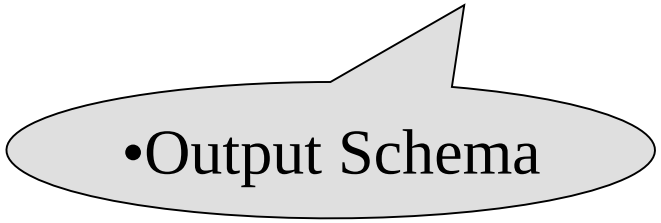


•Product(PName, Price, Category, Manfacturer)

```
•SELECT  PName, Price, Manufacturer
FROM     Product
WHERE    Price > 100
```



•Answer(PName, Price, Manfacturer)



Selections

What goes in the WHERE clause:

- x = y, x < y, x <= y, etc
  - For number, they have the usual meanings
  - For CHAR and VARCHAR: lexicographic ordering
    - Expected conversion between CHAR and VARCHAR
  - For dates and times, what you expect...
- Pattern matching on strings...

The **LIKE** operator

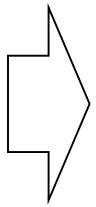
- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

Product(PName, Price, Category, Manufacturer)
Find all products whose name mentions ‘gizmo’:

```
•SELECT  *
FROM     Products
WHERE    PName LIKE ‘%gizmo%’
```

Eliminating Duplicates

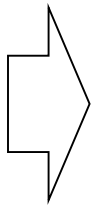
```
•SELECT  DISTINCT Category
•FROM    Product
```



Category
Gadgets
Photography
Household

•Compare to:

```
•SELECT  Category
•FROM    Product
```



Category
Gadgets
Gadgets
Photography
Household

Ordering the Results

```
•SELECT  PName, Price, Manufacturer
FROM     Product
WHERE    Category='gizmo' AND Price > 50
ORDER BY  Price, PName
```

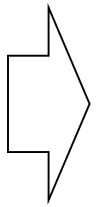
- Ordering is ascending, unless you specify the DESC keyword.

- Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering the Results

- SELECT Category
- FROM Product
- ORDER BY PName

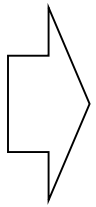
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi



•?

Ordering the Results

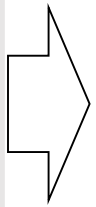
```
•SELECT   DISTINCT Category
FROM      Product
ORDER BY Category
```



Category
Gadgets
Household
Photography

•Compare to:

```
•SELECT   DISTINCT Category
FROM      Product
ORDER BY PName
```



•?

Joins in SQL

- Connect two or more tables:

•Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

•Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

•What is the connection between them ?

Joins

- Product (PName, Price, Category, Manufacturer)
- Company (CName, StockPrice, Country)

- Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT  PName, Price
FROM    Product, Company
WHERE   Manufacturer=CName AND Country='Japan'
        AND price <= 200
```

- Join between Product and Company

Joins in SQL

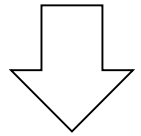
•Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

•Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
•SELECT  PName, Price
FROM     Product, Company
WHERE    Manufacturer=CName AND country='Japan'
         AND price <= 200
```



PName	Price
SingleTouch	$149.99

Joins

- Product (PName, Price, Category, Manufacturer)
- Company (CName, StockPrice, Country)
-

- Find all countries that manufacture some product in the 'Gadgets' category.

```
SELECT  Country
FROM    Product, Company
WHERE   Manufacturer=CName AND Category='Gadgets'
```

Joins in SQL

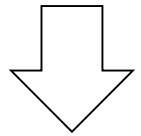
•Product

Name	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

•Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
•SELECT Country
FROM    Product, Company
WHERE   Manufacturer=CName AND Category='Gadgets'
```



Country
??
??

Joins

- Product (PName, Price, Category, Manufacturer)
- Purchase (Buyer, Seller, Store, Product)
- Person(Persname, PhoneNumber, City)

- Find names of people living in Seattle that bought some product in the 'Gadgets' category, and the names of the stores they bought such product from

```
SELECT  DISTINCT Persname, Store
FROM    Person, Purchase, Product
WHERE   Persname=Buyer AND Product = Pname AND
        City='Seattle'  AND Category='Gadgets'
```

When are two tables related?

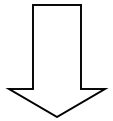
- You guess they are
- I tell you so
- Foreign keys are a method for schema designers to tell you so
  - A foreign key states that a column is a reference to the key of another table ex: Product.Manufacturer is foreign key of Company
  - Gives information and enforces constraint

Disambiguating Attributes

- Sometimes two relations have the same attribute name:
  Person(PName, Address, Worksfor)
  Company(CName, Address)

```
•SELECT   DISTINCT PName, Address
FROM      Person, Company
WHERE     Worksfor = CName
```

•Which address ?



```
•SELECT   DISTINCT Person.PName, Company.Address
FROM      Person, Company
WHERE     Person.Worksfor = Company.CName
```

Tuple Variables

- Product (PName, Price, Category, Manufacturer)
- Purchase (Buyer, Seller, Store, Product)
- Person(Persname, PhoneNumber, City)

- Find all stores that sold at least one product that the store 'BestBuy' also sold:

```
SELECT DISTINCT x.store
FROM   Purchase AS x, Purchase AS y
WHERE  x.product = y.product AND y.store = 'BestBuy'
```

- Answer (store)

Tuple Variables

•General rule:

tuple variables introduced automatically by the system:

• Product (name, price, category, manufacturer)

```
• SELECT  Name
FROM      Product
WHERE   Price > 100
```

•Becomes:

```
• SELECT Product.Name
FROM    Product AS Product
WHERE Product.Price > 100
```

•Doesn’t work when Product occurs more than once:

•In that case the user needs to define variables explicitly.

Meaning (Semantics) of SQL Queries

SELECT a1, a2, …, ak
FROM R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE Conditions

1. Nested loops:

```
Answer = {}
for x1 in R1 do
    for x2 in R2 do
        …..
            for xn in Rn do
                if Conditions
                    then Answer = Answer ∪ {(a1,…,ak)}
return Answer
```

Meaning (Semantics) of SQL Queries

SELECT a1, a2, …, ak
FROM R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE Conditions

2. Parallel assignment

- Answer = {}
- **for** all assignments x1 **in** R1, …, xn **in** Rn **do**
- **if** Conditions **then** Answer = Answer ∪ {(a1,…,ak)}
- **return** Answer

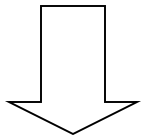
Doesn't impose any order!

Renaming Columns

•Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
•SELECT   Pname AS prodName, Price AS askPrice
FROM      Product
WHERE     Price > 100
```



•Query with renaming

prodName	askPrice
SingleTouch	$149.99
MultiTouch	$203.99

Union, Intersection, Difference

```
(SELECT  name
FROM     Person
WHERE    City="Seattle")

UNION

(SELECT  name
FROM     Person, Purchase
WHERE    buyer=name AND store="The Bon")
```

- Similarly, you can use INTERSECT and EXCEPT.
- You must have the same attribute names (otherwise: rename).

```
•(SELECT DISTINCT R.A
FROM R)
INTERSECT ( (SELECT S.A FROM S)
                    UNION
                    (SELECT T.A FROM T))
```

Conserving Duplicates

```
(SELECT  Name
FROM     Person
WHERE    City="Seattle")

UNION  ALL

(SELECT  Name
FROM     Person, Purchase
WHERE    Buyer=Name AND Store="The Bon")
```

Removing Duplicates

```
SELECT  Company.Name
FROM      Company, Product, Purchase
WHERE   Company.Name= Product.Maker
          AND  Product.Name  = Purchase.Product
          AND Purchase.Buyer = ‘Joe Blow’
```

- Multiple copies

```
SELECT DISTINCT Company.Name
FROM      Company, Product, Purchase
WHERE   Company.Name= Product.Maker
          AND  Product.Name  = Purchase.product
          AND Purchase.Buyer = ‘Joe Blow’
```

- Single copies

Removing Duplicates

```
• SELECT DISTINCT Company.name
• FROM      Company, Product
• WHERE   Company.Name= Product.Maker
•           AND  Product.Name  IN
•                 (SELECT Purchase.Product
•                   FROM    Purchase
•                   WHERE Purchase.Buyer = 'Joe Blow')
```

```
• SELECT DISTINCT Company.Name
• FROM      Company, Product, Purchase
• WHERE   Company.Name= Product.Maker
•           AND  Product.Name  = Purchase.Product
•           AND  Purchase.Buyer = 'Joe Blow'
```

- Now
- they are
- equivalent

Aggregation

```
SELECT  AVG(Price)
FROM    Product
WHERE   Maker="Toyota"
```

•SQL supports several aggregation operations:

- SUM, MIN, MAX, AVG, COUNT

Aggregation: Count

```
•SELECT  COUNT(*)
•FROM    Product
•WHERE   Year > 1995
```

•Except COUNT, all aggregations apply to a single attribute

Aggregation: Count

•COUNT applies to duplicates, unless otherwise stated:

•SELECT Count(Category) same as Count(*)
•FROM Product
•WHERE Year > 1995

•Better:

•SELECT Count(DISTINCT Category)
•FROM Product
•WHERE Year > 1995

Simple Aggregation

•Purchase(product, date, price, quantity)

•Example 1: find total sales for the entire database

```
SELECT  SUM(price * quantity)
FROM    Purchase
```

•Example 1: find total sales of bagels

```
SELECT  SUM(price * quantity)
FROM    Purchase
WHERE   product = 'bagel'
```

Grouping and Aggregation

•Usually, we want aggregations on certain parts of the relation.

•Purchase(product, date, price, quantity)

•Example 2: **find total sales after 10/1 per product.**

```
SELECT   product, SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > "10/1"
GROUPBY  product
```

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.
2. Group by the attributes in the GROUPBY
3. Select one tuple for every group (and apply aggregation)

•SELECT can have (1) grouped attributes or (2) aggregates.

Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match
- Right outer join:
  - Include the right tuple even if there’s no match
- Full outer join:
  - Include the both left and right tuples even if there’s no match

Modifying the Database

Three kinds of modifications

- Insertions
- Deletions
- Updates

Sometimes they are all called “updates”

Insertions

•General form:

```
INSERT INTO R(A1,…., An) VALUES (v1,…., vn)
```

•Example: Insert a new purchase to the database:

```
INSERT INTO Purchase(buyer, seller, product, store)
VALUES ('Joe', 'Fred', 'wakeup-clock-espresso-machine',
                                    'The Sharper Image')
```

- Missing attribute → NULL.
- May drop attribute names if give them in order.

Insertions

```
INSERT   INTO   PRODUCT(name)
SELECT  DISTINCT  Purchase.product
FROM      Purchase
WHERE   Purchase.date > “10/26/01”
```

- The query replaces the VALUES keyword.
- Here we insert *many* tuples into PRODUCT

Insertion: an Example

- Product(name, listPrice, category)
- Purchase(prodName, buyerName, price)

- prodName is foreign key in Product.name

- Suppose database got corrupted and we need to fix it:

- Product

name	listPrice	category
gizmo	100	gadgets

- Purchase

prodName	buyerName	price
camera	John	200
gizmo	Smith	80
camera	Smith	225

- Task: insert in Product all prodNames from Purchase

Insertion: an Example

```
INSERT  INTO  Product(name)
SELECT  DISTINCT  prodName
FROM    Purchase
WHERE   prodName  NOT IN (SELECT  name FROM  Product)
```

name	listPrice	category
gizmo	100	Gadgets
camera	-	-

Insertion: an Example

```
•INSERT  INTO  Product(name, listPrice)
• SELECT  DISTINCT  prodName, price
• FROM  Purchase
• WHERE  prodName  NOT IN (SELECT  name FROM  Product)
```

name	listPrice	category
gizmo	100	Gadgets
camera	200	-
camera ??	225 ??	-

← Depends on the implementation

Deletions

•Example:

```
•DELETE    FROM    PURCHASE
•WHERE     seller = 'Joe'   AND
•                  product = 'Brooklyn Bridge'
```

•Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.

Updates

•Example:

```
UPDATE   PRODUCT
SET    price = price/2
WHERE  Product.name  IN
                (SELECT product
                  FROM    Purchase
                  WHERE  Date ='Oct, 25, 1999');
```

Data Definition in SQL

- So far we have see the ***Data Manipulation Language***, DML
- Next: ***Data Definition Language*** (DDL)

- Data types:
  - Defines the types.

- Data definition: defining the schema.
  - Create tables
  - Delete tables
  - Modify table schema

- Indices: to improve performance

Data Types in SQL

- Characters:
  - CHAR(20) -- fixed length
  - VARCHAR(40) -- variable length
- Numbers:
  - INT, REAL plus variations
- Times and dates:
  - DATE, DATETIME (SQL Server only)
- To reuse domains:
  CREATE DOMAIN address AS VARCHAR(55)

Creating Tables

- Example:

```
CREATE    TABLE Person(
            name                    VARCHAR(30),
            social-security-number  INT,
            age                     SHORTINT,
            city                    VARCHAR(30),
            gender                  BIT(1),
            Birthdate               DATE


 );
```

Deleting or Modifying a Table

- Deleting:
  - Example:
    ```
    DROP Person;
    ```
  - Exercise with care !!
- Altering: (adding or removing an attribute).
  - Example:
    ```
    ALTER TABLE Person
        ADD phone CHAR(16);

    ALTER TABLE Person
        DROP age;
    ```
- What happens when you make changes to the schema?

Default Values

•Specifying default values:

```
•CREATE  TABLE Person(
•       name                        VARCHAR(30),
•       social-security-number  INT,
•       age         SHORTINT   DEFAULT 100,
•       city     VARCHAR(30) DEFAULT   'Seattle',
•       gender        CHAR(1)   DEFAULT  '?',
•       Birthdate                 DATE
```

•The default of defaults: NULL

Defining 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

A Different View

- Person(name, city)
- Purchase(buyer, seller, product, store)
- Product(name, maker, category)

```
CREATE VIEW  Seattle-view  AS

    SELECT  buyer, seller, product, store
    FROM    Person, Purchase
    WHERE   Person.city = "Seattle"    AND
            Person.name = Purchase.buyer
```

- We have a new virtual table:
- Seattle-view(buyer, seller, product, store)

A Different View

- We can later use the view:

```
SELECT   name, store
FROM     Seattle-view, Product
WHERE    Seattle-view.product = Product.name  AND
         Product.category = “shoes”
```

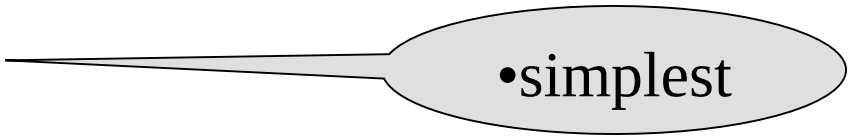
Constraints in SQL

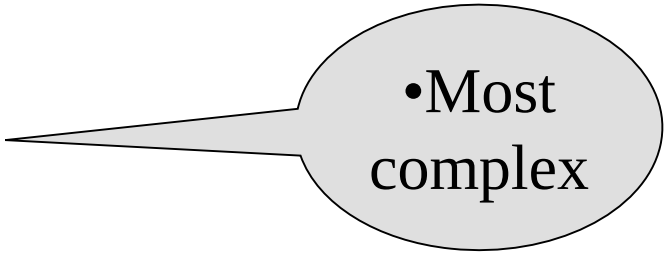
- A constraint = a property that we'd like our database to hold
- The system will enforce the constraint by taking some actions:
  - forbid an update
  - or perform compensating updates

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

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

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)
```

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

- Product

Name	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

Transactions

Address two issues:

- Access by multiple users
  - Remember the “client-server” architecture: one server with many clients
- Protection against crashes

Flight Reservation

- get values for :flight, :date, :seat

- EXEC SQL SELECT occupied INTO :occ
- FROM Flight
- WHERE fltNum = :flight AND fltdt= :date AND fltSeat=:seat
- if (!occ) {
- EXEC SQL UPDATE Flights
- SET occupied = ‘true’
- WHERE fltNum= :flight AND fltdt= :date AND fltSeat=:seat
- /* more code missing */
- }
- else /* notify customer that seat is not available */

Problem #1

- Customer 1 - finds a seat empty
- Customer 2 - finds the same seat empty
- Customer 1 - reserves the seat.
- Customer 2 - reserves the seat.
- Customer 1 will not be happy.
  - *serializability*

Bank Transfers

- Transfer :amount from :account1 to :account2
- EXEC SQL SELECT balance INTO :balance1
- FROM Accounts
- WHERE accNo = :account1
- if (balance1 >= amount)
- EXEC SQL UPDATE Accounts
- SET balance = balance + :amount
- WHERE acctNo = :account2;
- EXEC SQL UPDATE Accounts
- SET balance = balance - :amount
- WHERE acctNo = :account1;

Transactions

- The user/programmer can group a sequence of commands so that they are executed atomically and in a serializable fashion:
  - Transaction commit: all the operations should be done and recorded.
  - Transaction abort: none of the operations should be done.

- In SQL:
  - EXEC SQL COMMIT;
  - EXEC SQL ROLLBACK;

- *Easier said than done...*

ACID Properties

- *A*tomicity: **all** actions of a transaction happen, or **none** happen.

- *C*onsistency: if a transaction is consistent, and the database starts
- from a consistent state, then it will end in a consistent
- state.

- *I*solation: the execution of one transaction is isolated from other
- transactions.

- *D*urability: if a transaction commits, its effects persist in the
- database.

How Do We Assure ACID?

- Concurrency control:
  - Guarantees consistency and isolation, given atomicity.
- Logging and Recovery:
  - Guarantees atomicity and durability.

If you are going to be in the logging business, one of the things that you'll have to do is learn about heavy equipment.

-- Robert VanNatta

Logging History of Columbia County

Transactions in SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction

- In “embedded” SQL:

  BEGIN TRANSACTION
  [SQL statements]
  COMMIT or ROLLBACK (=ABORT)

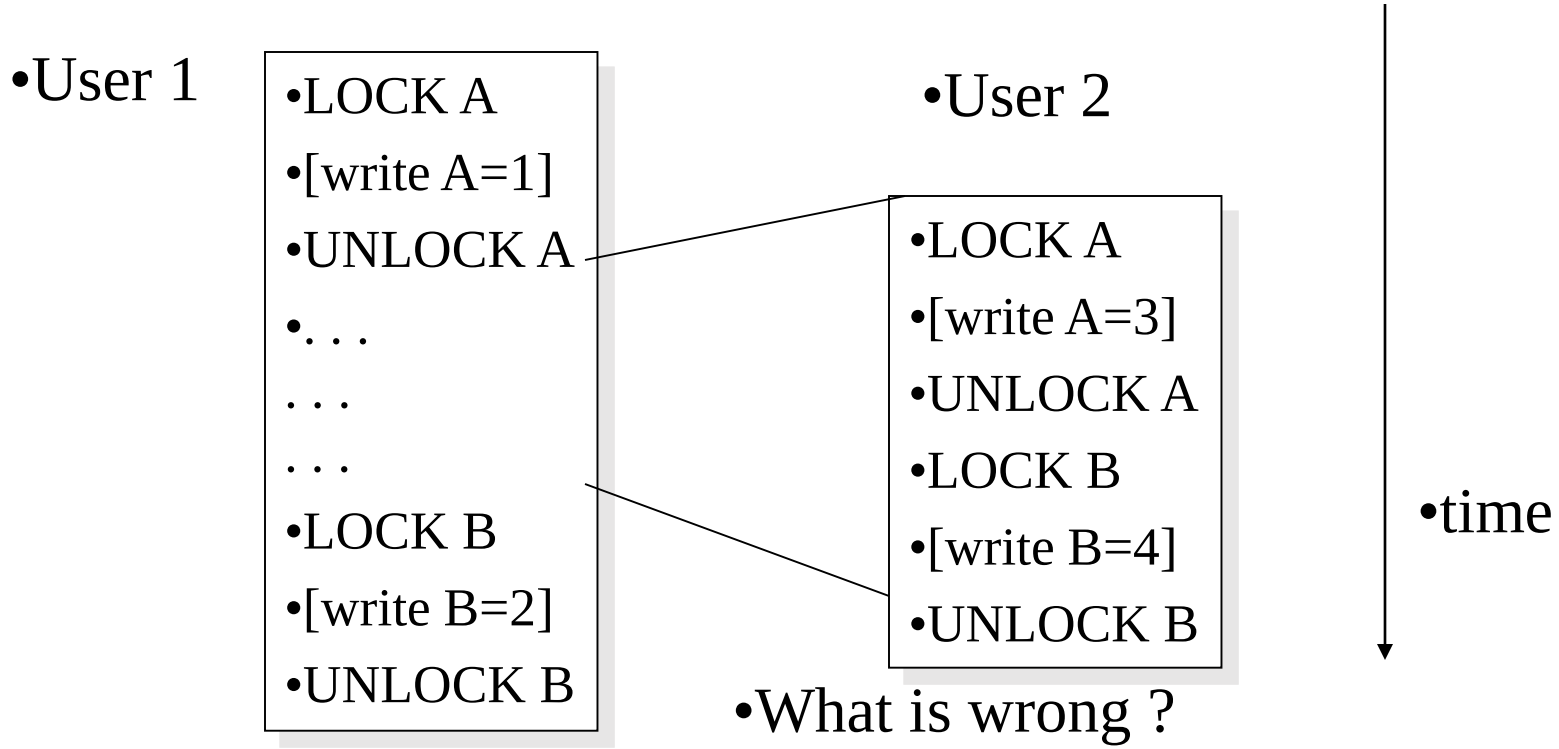
Transactions: Serializability

Serializability = the technical term for isolation

- An execution is ***serial*** if it is completely before or completely after any other transaction's execution
- An execution is ***serializable*** if it equivalent to one that is serial
- DBMS can offer serializability guarantees

Serializability

- Enforced with locks, like in Operating Systems!
- But this is not enough:



Serializability

- Solution: two-phase locking
  - Lock everything at the beginning
  - Unlock everything at the end

- Read locks: many simultaneous read locks allowed
- Write locks: only one write lock allowed
- Insert locks: one per table

Isolation Levels in SQL

1. “Dirty reads”

   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”

   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”

   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions (default):

   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

//TODO before next lecture:

- Homework 4 due on 4/16 at 11:59 pm EDT. Must be submitted on Submitty.