

CECS 323



Mimi Opkins

Introduction
Basic UML & SQL
Models
Classes & Schemes
Rows & Tables
Associations
Keys
SQL technique
SQL and RA
DDL & DML
Join
Multiple Joins
Join Types
Functions
Subqueries
Union & Minus
Views & Indexes

UML design
Many-to-Many
Many-to-Many 2
Subkeys
Repeated Attributes
Multi-Valued Attributes
Domains
Enumerated Domains
Subclasses
Aggregation
Recursive Associations
Normalization
BCNF
Transactions
JDBC

Introduction

- What is a database?
- Why do we need one?
 - Avoid redundancy
 - duplication of information in multiple tables within a database
 - Data integrity
 - Refers to the validity of data
 - Referential Integrity
 - ensures that relationships between tables remain consistent
 - Deletion Anomalies
 - Deletion of one row of a table results in the deletion of unintended information

Different Types of Databases

- Paper
- Flatfile
- Hierarchical
- Network
- Relational
- Object

Introduction – Redundancy

- The duplication of information in multiple tables within a database
- Ex. School Records may all have info about you
 - Admissions
 - Enrollment Services
 - Department
 - Student Union

Introduction – Data Integrity

- Refers to the validity of data.
- Data integrity can be compromised in a number of ways:
 - Human errors when data is entered
 - Errors that occur when data is transmitted from one computer to another
 - Software bugs or viruses
 - Hardware malfunctions, such as disk crashes
 - Natural disasters, such as fires and floods

Introduction – Referential Integrity

- ❑ Referential integrity ensures that relationships between tables remain consistent.
- ❑ When one table has a foreign key to another table, the concept of referential integrity states that you may not add a record to the table that contains the foreign key unless there is a corresponding record in the linked table.
- ❑ It also includes the techniques known as cascading update and cascading delete, which ensure that changes made to the linked table are reflected in the primary table.

Introduction – Deletion Anomalies

- ❑ A situation, usually caused by redundancy in the database design, in which the deletion of one row of a table results in the deletion of unintended information
- ❑ Ex. You're the only student enrolled in CECS 323 and you drop the course. As a result all the info about CECS 323 is removed from the catalog

Introduction – Relational Database

What is a relational database?

- Based on Relational Algebra
- Tables and their relationships to each other
- Easily navigated
- Popular
- Standard query language
- Familiar Concepts

Models and Languages

- ❑ Database design is a process of modeling an enterprise in the real world.
- ❑ A database itself is a model of the real world that contains selected information needed by the enterprise.
- ❑ There are many models and languages used for this modeling. Some of these are mathematically based. Others are less formal and more intuitive.

Unified Modeling Language (UML)

- ❑ The **Unified Modeling Language** (UML) was designed for software engineering of large systems using object-oriented (OO) programming languages.
- ❑ UML is a very large language; we will use only a small portion of it here, to model those portions of an enterprise that will be represented in the database.
- ❑ It is our tool for communicating with the client in terms that are used in the enterprise.
- ❑ Used to describe the conceptual view of a database.

Entity-Relationship (ER)

- ❑ The **Entity-Relationship** (ER) model is used in many database development systems.
- ❑ There are many different graphic standards that can represent the ER model. Some of the most modern of these look very similar to the UML class diagram, but may also include elements of the relational model.

Relational Model (RM)

- ❑ The **Relational Model** (RM) is the formal model of a database that was developed for IBM in the early 1970s by Dr. E.F. Codd.
- ❑ It is largely based on set theory, which makes it both powerful and easy to implement in computers.
- ❑ All modern relational databases are based on this model.
- ❑ We will use it to represent information that does not (and should not) appear in the UML model but is needed for us to build functioning databases.

Relational Algebra (RA)

- **Relational Algebra** (RA) is a formal language used to symbolically manipulate objects of the relational model.
- Terms used to refer to the logical view of the database.

Table Model

- ❑ The **table model** is an informal set of terms for relational model objects. These are the terms used most often by database developers.
- ❑ Terms used to refer to the physical view of the database.

Structured Query Language (SQL)

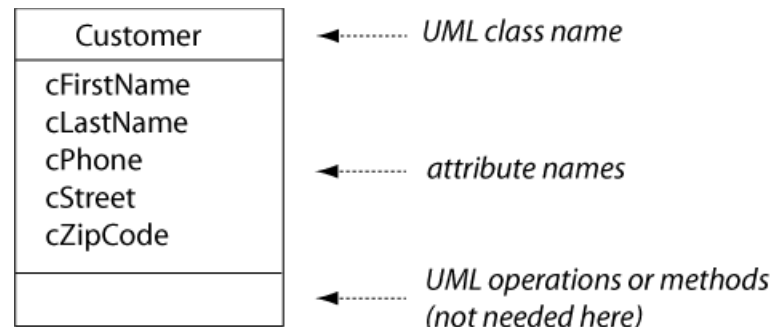
- ❑ The **Structured Query Language** (SQL, pronounced sequel or ess-que-ell) is used to build and manipulate relational databases.
- ❑ It is based on relational algebra, but provides additional capabilities that are needed in commercial systems.
- ❑ It is a declarative, rather than a procedural, programming language.
- ❑ There is a standard for this language, but products vary in how closely they implement it.

Basic Structures: Classes and Schemes – UML Class

- ❑ **UML class (ER term: entity)** is anything in the enterprise that is to be represented in our database
- ❑ The first step in modeling a class is to describe it in natural language.
Example: build a sales database. Let's start by defining customer using natural language.
- ❑ **Attribute (properties)** is a piece of information that characterizes each member of a class
- ❑ **Descriptive attributes (natural attribute)** are those which actually provide real-world information about the class.
 - UML only uses descriptive attributes
 - ID number are not descriptive attributes

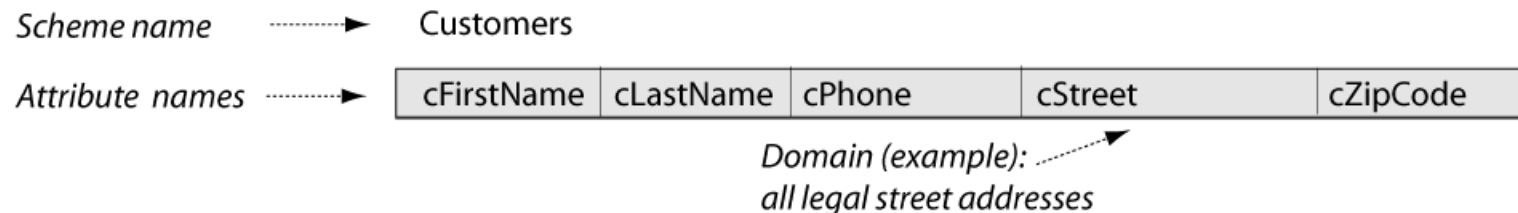
Basic Structures: Classes and Schemes – Class Diagram

- ❑ A class diagram shows the class name and list of attributes that identify data elements we need to know about each **member (instance)**, occurrence, of a class)
- ❑ The Customer class represents any person who has done business with us or who we think might do business with us in the future. Its attributes are:
 - Customer first name.
 - Customer last name.
 - Customer phone.
 - Customer street.
 - Customer zip code.



Basic Structures: Classes and Schemes – Relation Scheme

- In an OO programming language, each class is **instantiated** with **objects** of that class. In building a relational database, each class is first translated into a relation model **scheme**. The scheme starts with all of the attributes from the class diagram.



- The Customers relation scheme attributes are:
 - Customer first name, a person's first name.
 - Customer last name, a person's last name.
 - Customer phone, a valid telephone number.
 - Customer street, a street address.
 - Customer zip code, a zip code designated by the United States Postal Service.

Basic Structures: Classes and Schemes – Sets

- In the relational model, a scheme is defined as a set of attributes, together with an **assignment rule** that associates each attribute with a set of legal values that may be assigned to it. These values are called the **domain** of the attribute.
Customers Scheme = {cFirstname, cLastname, cPhone, cStreet, cZipCode}.
- It's important to recognize that defining schemes or domains as *sets* of something automatically tells us a lot more about them:
 - They cannot contain duplicate elements.
 - The elements in them are unordered.
 - We can develop rules for what can be included in them and what is excluded from them. For example, zip codes don't belong in the domain (set) of phone numbers, and vice-versa.
 - We can define subsets of them—for example, we can display only a selected set of attributes from a scheme, or we can limit the domain of an attribute to a specific range of values.
 - They may be manipulated with the usual set operators (union, intersection).

Basic Structures: Classes and Schemes – Table Structure

When we actually build the database, each relation scheme becomes the structure of one table.

```
CREATE TABLE customers (  
    cfirstname VARCHAR(20) NOT NULL,  
    clastname VARCHAR(20) NOT NULL,  
    cphone VARCHAR(20) NOT NULL,  
    cstreet VARCHAR(50),  
    czipcode VARCHAR(5));
```

Basic Structures: Classes and Schemes – Data Models

- Conceptual
- Logical
- Physical

Exercise: Designing Classes

- ❑ Design classes to represent the following "things" in the given enterprises. For each one, describe the class in English, then draw the class diagram.
 - A student at a university.
 - A faculty member at a university.
 - A work of art that is displayed in a gallery or museum.
 - An automobile that is registered with the Motor Vehicle Department.
 - A pizza that is on the menu at a restaurant.
- ❑ The solution to this exercise will be discussed in class

Exercise: More Designing Classes

- Design classes to represent the any three different enterprises that you find.
 - For each one, describe the class in English, then draw the class diagram.
- We'll discuss your examples in class.

Basic Structures: Rows and Tables

- Definitions

- Each real-world individual of a class is represented by a **row** of information in a database table.
- The row is defined in the relational model as a **tuple** that is constructed over a given scheme.
- Mathematically, the **tuple** is a function that assigns a constant value from the attribute domain to each attribute of the scheme. Since the scheme is really a set of attributes, order is not important.

Basic structures: Rows and Tables

– Assignment

Scheme name ➔ Customers

Attribute names ➔

Row (tuple) ➔

cFirstName	cLastName	cPhone	cStreet	cZipCode
Tom	Jewett	714-555-1212	10200 Slater	92708

Data cell ➔
(domain value assigned to attribute name)

- Each attribute of the Customers scheme is assigned a value from its domain:
 - Customer first name is assigned the value "Tom", from the domain of people's first names.
 - Customer last name is assigned the value "Jewett", from the domain of people's last names.
 - Customer phone is assigned the value "714-555-1212", from the domain valid telephone numbers.
 - Customer street is assigned the value "10200 Slater", from the domain of street addresses.
 - Customer zip code is assigned the value "92708", from the domain of zip codes designated by the United States Postal Service.
- Each of the assignments results in a data cell of the row or tuple.

Basic Structures: Rows and Tables

– Formal Notation

In formal notation, we could show the assignments explicitly, where t represents a tuple:

```
 $t_{TJ} = \langle \text{cfirstname} := \text{'Tom'}, \text{clastname} := \text{'Jewett'},$   
 $\text{cphone} := \text{'714-555-1212'}, \text{cstreet} := \text{'10200}$   
 $\text{Slater'}, \text{czipcode} := \text{'92708'} \rangle$ 
```

Database:

```
INSERT INTO customers (cfirstname,  
    clastname, cphone, cstreet, czipcode)  
VALUES ('Tom', 'Jewett', '714-555-1212',  
    '10200 Slater', '92708');
```

```
UPDATE customers SET cphone = '714-555-  
2323' WHERE cphone = '714-555-1212';
```

Basic Structures: Rows and Tables

- Database

- A database **table** is simply a collection of zero or more rows. This follows from the relational model definition of a **relation** as a set of tuples over the same scheme. Order is not important.

Table (relation) name	----->	Customers																				
Primary key attributes	----->	<table><tr><td colspan="5">PK</td></tr></table>	PK																			
PK																						
Column (attribute) names	----->	<table><tr><td>cFirstName</td><td>cLastName</td><td>cPhone</td><td>cStreet</td><td>cZipCode</td></tr><tr><td>Tom</td><td>Jewett</td><td>714-555-1212</td><td>10200 Slater</td><td>92708</td></tr><tr><td>Alvaro</td><td>Monge</td><td>562-333-4141</td><td>2145 Main</td><td>90840</td></tr><tr><td>Wayne</td><td>Dick</td><td>562-777-3030</td><td>1250 Bellflower</td><td>90840</td></tr></table>	cFirstName	cLastName	cPhone	cStreet	cZipCode	Tom	Jewett	714-555-1212	10200 Slater	92708	Alvaro	Monge	562-333-4141	2145 Main	90840	Wayne	Dick	562-777-3030	1250 Bellflower	90840
cFirstName	cLastName	cPhone	cStreet	cZipCode																		
Tom	Jewett	714-555-1212	10200 Slater	92708																		
Alvaro	Monge	562-333-4141	2145 Main	90840																		
Wayne	Dick	562-777-3030	1250 Bellflower	90840																		
Rows (tuples)	----->																					

- Additional rows are built on the Customers scheme as before. The table or relation consists of all rows.
- Three of the attributes in the Customers scheme are now identified as the primary key, which is explained later on.

Basic Structures: Rows and Tables

- Tuples

Knowing that the relation (table) is a set of tuples (rows) tells us more about this structure, as we saw with schemes and domains.

- Each tuple/row is unique; there are no duplicates
- Tuples/rows are unordered; we can display them in any way we like and the meaning doesn't change. (SQL gives us the capability to control the display order.)
- Tuples/rows may be included in a relation/table set if they are constructed on the scheme of that relation; they are excluded otherwise. (It would make no sense to have an Order row in the Customers table.)
- We can define subsets of the rows by specifying criteria for inclusion in the subset. (Again, this is part of a SQL query.)
- We can find the union, intersection, or difference of the rows in two or more tables, as long as they are constructed over the same scheme.

Basic Structures: Rows and Tables

– Insuring Unique Rows

- Each row in a table must be distinct. So there must be a set of attributes in each relation that guarantee uniqueness. Any set of attributes that can do this is called a ***super key*** of the relation.
- The database designer picks of the possible super key sets to serve as the ***primary key*** or unique identifier of each row.

Basic Structures: Rows and Tables

– Super Keys

cfirstname, clastname, cphone, cstreet, czipcode

cfirstname, clastname, cphone, cstreet

cfirstname, clastname, cphone, czipcode

cfirstname, clastname, cstreet, czipcode

cfirstname, clastname, cphone,

cfirstname, clastname, cstreet

Basic Structures: Rows and Tables

– Insuring Unique Rows - SQL

```
ALTER TABLE customers
  ADD CONSTRAINT customers_pk
    PRIMARY KEY (cfirstname, clastname, cphone);

CREATE TABLE customers (
  cfirstname VARCHAR(20) NOT NULL,
  clastname VARCHAR(20) NOT NULL,
  cphone VARCHAR(20) NOT NULL,
  cstreet VARCHAR(50),
  czipcode VARCHAR(5)),
  CONSTRAINT customers_pk
    PRIMARY KEY (cfirstname, clastname, cphone);
```

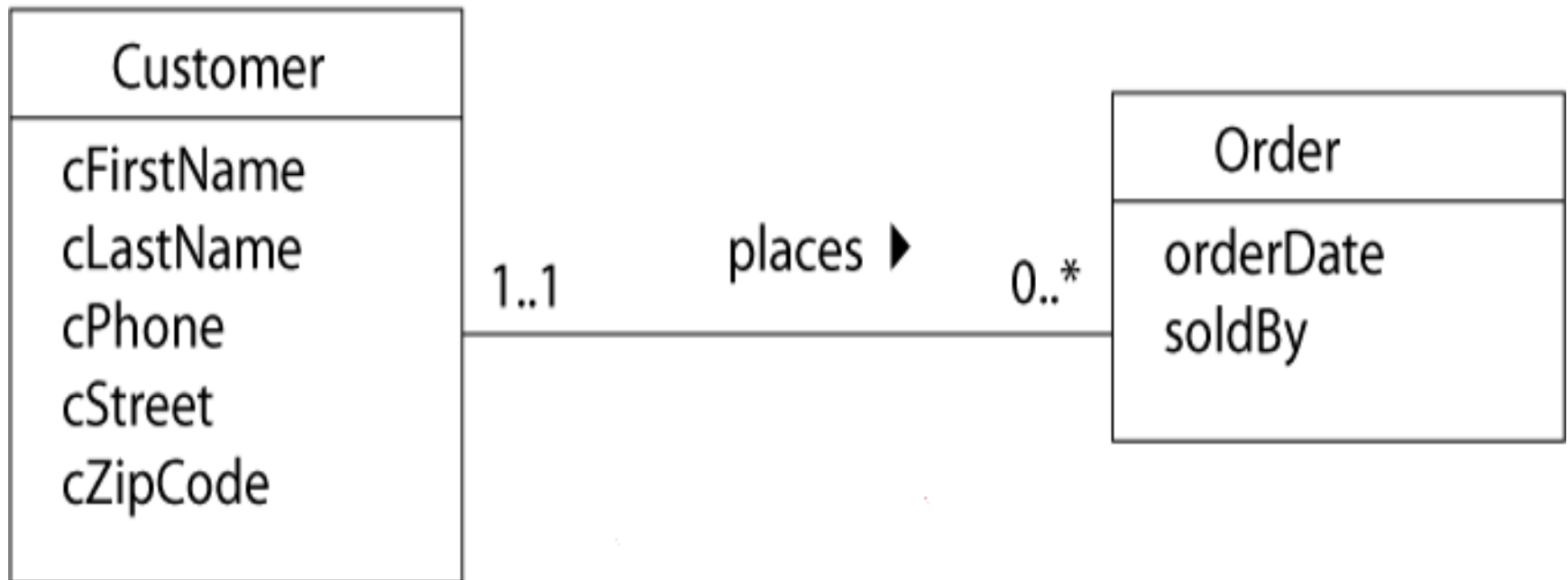

Basic Structures: Associations – The UML Association

- ❑ **UML association (ER term: relationship)** is the way that two classes are functionally connected to each other.
- ❑ Example: relationship between customers and orders in a sales database. First define orders then define the relationship.
- ❑ **UML multiplicity (ER term: cardinality)** how few (at minimum) and how many (at maximum) individuals of one class may be connected to a *single* individual of the other class.

Basic structures: Associations – Class Diagram

One customer places zero to many orders

One order is placed by one and only one customer



Basic structures: Associations – Class Diagram Data Dictionary

- The Customer class represents any person who has done business with us or who we think might do business with us in the future. Its attributes are:
 - Customer first name.
 - Customer last name.
 - Customer phone.
 - Customer street.
 - Customer zip code.
- The Order class represents an event that happens when a customer decides to buy one or more of our products. Its attributes are:
 - Order date.
 - Sold by, which identifies the sales person.
- The association between customer and order classes is:
 - Each customer places zero or more orders.
 - Each order is placed by one and only one customer.

Basic Structures: Associations – Class Diagram Description

- ❑ Looking at the *maximum* multiplicity at each end of the line (1 and * here), we call this a **one-to-many** association.
- ❑ The UML representation of the Order class contains only its own descriptive attributes. The UML **association** tells which customer placed an order.
- ❑ In the database, we will need a different way to identify the customer; that will be part of the relation scheme.

Basic Structures: Associations – Relation Scheme

- The relation scheme for the new Orders table contains all of the attributes from the class diagram as before. But we also need to represent the association in the database – which customer placed each order. This is done by copying the PK attributes of the Customer into the Orders scheme.
- The copied attributes are called a ***foreign key***.

Basic Structures: Associations – Relation Scheme Diagram

Customers

cFirstName	cLastName	cPhone	cStreet	cZipCode
Primary Key				

1..1 (*parent*)

Orders

0..* (*child*)

Foreign Key				
cFirstName	cLastName	cPhone	orderDate	soldBy
Primary Key				

Basic structures: Associations – Relation Scheme Data Dictionary

- The Customers relation scheme attributes are:
 - Customer first name, a person's first name.
 - Customer last name, a person's last name.
 - Customer phone, a valid telephone number.
 - Customer street, a street address.
 - Customer zip code, a zip code designated by the United States Postal Service.
- The primary key attributes of the Customers relation are the first name, last name, and phone.
- The Orders relation scheme attributes are:
 - Customer first name, foreign key from Customers.
 - Customer last name, foreign key from Customers.
 - Customer phone, foreign key from Customers.
 - Order date, a calendar date possibly with the clock time.
 - Sold by, the first name of the sales person taking the order.
- The primary key attributes of the Orders relation are the three foreign key attributes plus the order date.

Basic Structures: Associations – Relation Scheme Description

- Since we can't have an order without a customer, we call Customers the ***parent*** and Orders the ***child*** scheme in this association.
- The “one” side of an association is always the parent, and provides the PK attributes to be copied.
- The “many” side of an association is always the child, into which the FK attributes are copied.
- Memorize it: one, parent, PK; many, child, FK.
- An FK might or might not become part of the PK of the child relation into which it is copied. In this case, it does, since we need to know both *who* placed an order and *when* the order was placed in order to identify it uniquely.

Basic Structures: Associations – The Child Table

```
CREATE TABLE orders (  
    cfirstname VARCHAR(20),  
    clastname VARCHAR(20),  
    cphone VARCHAR(20),  
    orderdate DATE,  
    soldby VARCHAR(20));
```

- ❑ The FK attributes must be exactly the same data type and size as in the PK table
- ❑ In some DBMS, DATE is both Date and Time

Basic Structures: Associations – Uniqueness

- To insure that every row of the Orders table is unique, we need to know both who the customer is and what day (and time) the order was placed.

- We specify all of these attributes as the pk:

```
ALTER TABLE orders
```

```
ADD CONSTRAINT orders_pk
```

```
PRIMARY KEY (cfirstname, clastname,  
cphone, orderdate);
```

Basic Structures: Associations – Referential Integrity

In addition, we need to identify which attributes make up the FK, and where they are found as a PK. The FK constraint will insure that every order contains a valid customer name and phone number—this is called maintaining the ***referential integrity*** of the database.

```
ALTER TABLE orders
```

```
ADD CONSTRAINT orders_customers_fk  
FOREIGN KEY(cfirstname, clastname,  
cphone)
```

```
REFERENCES customers (cfirstname,  
clastname, cphone);
```

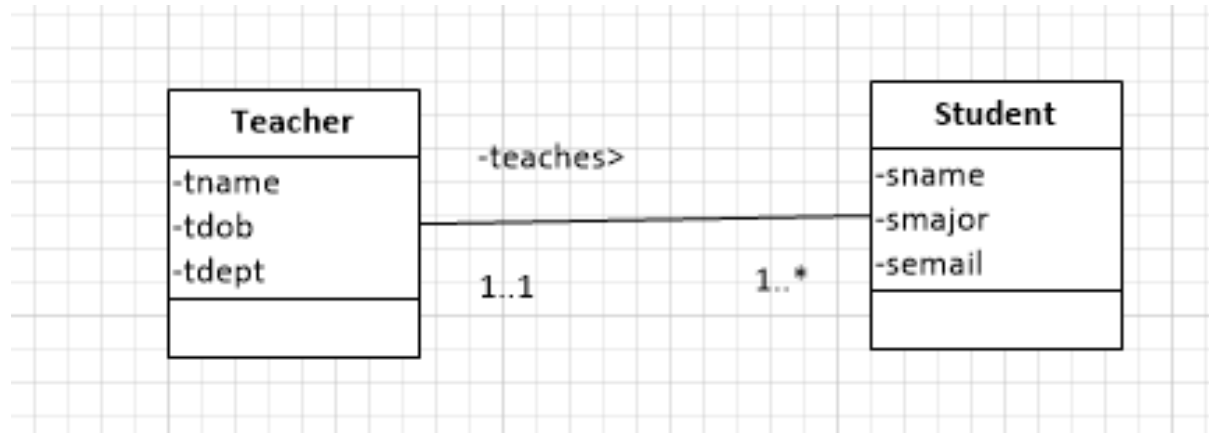
Basic Structures: Associations – The Orders Table

Orders

cfirstname	clastname	cphone	orderdate	soldby
Alvaro	Monge	562-333-4141	2003-07-14	Patrick
Wayne	Dick	562-777-3030	2003-07-14	Patrick
Alvaro	Monge	562-333-4141	2003-07-18	Kathleen
Alvaro	Monge	562-333-4141	2003-07-20	Kathleen

Another Example

- Let's limit this enterprise to consider the relationship between a University instructor and the students in one class



Class Definitions

- ❑ A teacher is a person that instructs students taking courses to learn.
- ❑ A student is someone enrolled in classes to receive an education.

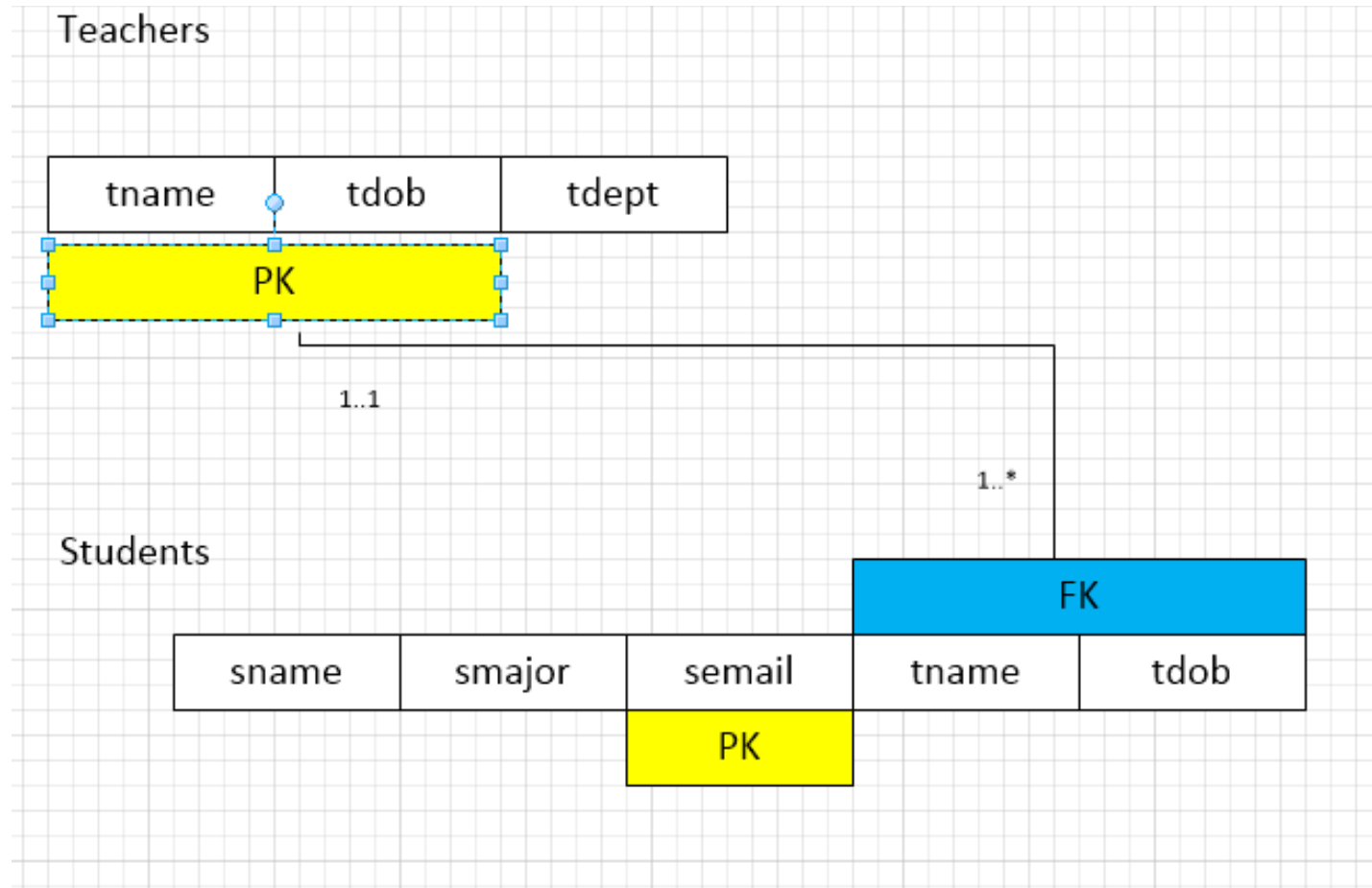
Relationship Definitions

- One teacher teaches one to many students.
- One student is taught by one and only one teacher.

Primary Key

- ❑ For the primary key of Teacher, use ***tname*** and ***tdob***.
- ❑ Copy these PK attributes from the parent (Teacher) to the child (Student) and mark them as Foreign Keys.
- ❑ Question: does a Student need a Teacher to be unique? No, the email itself can be used as the PK of the student.
- ❑ This is an example of when the FK of the parent is NOT part of the FK of a child.

Relation Scheme



SQL

- Even though in this example, the FK of the parent does not become part of the PK of the child, the Students table still needs to have the FK attributes as part of the table and a FK constraint must still be created.

```
CREATE TABLE teachers (  
    tname  VARCHAR(20) NOT NULL,  
    tdept  VARCHAR(20) NOT NULL,  
    tdob   DATE NOT NULL);
```

```
ALTER TABLE teachers  
    ADD CONSTRAINT teachers_pk  
    PRIMARY KEY (tname, tdept, tdob);
```

Students Table

```
CREATE TABLE students (  
    tname  VARCHAR(20) NOT NULL,  
    tdept  VARCHAR(20) NOT NULL,  
    sname  VARCHAR(20) NOT NULL,  
    smajor VARCHAR(20) NOT NULL,  
    semail VARCHAR(90) NOT NULL);
```

```
ALTER TABLE students  
    ADD CONSTRAINT students_pk  
    PRIMARY KEY (semail);
```

```
ALTER TABLE students  
    ADD CONSTRAINT students_teachers_fk  
    FOREIGN KEY (tname,tdob)  
    REFERENCES teachers (tname,tdob);
```

Exercise: Patients and Blood Samples

- ❑ The one-to-many association is perhaps the most common one that you will encounter in database modeling. As an example, we will look at the enterprise of a medical clinic.
- ❑ We wish to track the level of various substances (for example, cholesterol or alcohol) in the blood of patients. For each blood sample that is taken from the patient, one test will be performed and the date of the sample, the substance tested, and the measured level of that substance will be recorded in a database.
 - Describe each class in English.
 - Draw the class diagram.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- ❑ The solution to this exercise will be discussed in class

Exercise: Cities and States

- ❑ Part of a database that you are developing will contain information about cities and states in the United States. Each city is located in only one state. (Texarkana, Texas is a different city than Texarkana, Arkansas.)
 - Describe each class in English.
 - Draw the class diagram.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- ❑ The solution to this exercise will be discussed in class

Exercise: Library Books

- ❑ You are building a very simplified beginning of the database for a library.
- ❑ The library, of course, owns (physical) books that are stored on shelves and checked out by customers. Each of these books is represented by a catalog entry (now in the computer, but think of an old-fashioned card file as a model of this).
- ❑ Assume that there is only one title card for each book in the catalog, but there can be many physical copies of that book on the shelves.
- ❑ Call the title card class a `CatalogEntry` and the physical book class a `BookOnShelf`.

Exercise: Library Books

- ❑ You might think of the book's publisher as a simple attribute of the catalog entry but in fact, the library will probably want to know more than just the publisher's name (for example, the phone number where they can contact a sales representative).
 - Describe each class in English.
 - Draw the class diagram.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- ❑ The solution to this exercise will be discussed in class

Discussion: More About Keys

Customers

cFirstName	cLastName	cPhone	cStreet	cZipCode
Primary Key				

1..1 (*parent*)

Orders

0..* (*child*)

Foreign Key				
cFirstName	cLastName	cPhone	orderDate	soldBy
Primary Key				

Discussion: More About Keys – Super Key

- ❑ Remember that a super key is *any* set of attributes whose values, taken together, uniquely identify each row of a table—and that a primary key is the specific super key set of attributes that we picked to serve as the unique identifier for rows of this table.
- ❑ We are showing these attributes in the scheme diagram for convenience, understanding that keys are a property of the table (relation), not of the scheme.

Discussion: More About Keys – Candidate Keys

- Before picking the PK, we need to identify any ***candidate key*** or keys that we can find for the table. A CK is a minimal super key. Minimal means that if you take away any one attribute from the set, it is no longer a super key.
If you add one attribute to the set, it is no longer minimal but it's still a super key. The word “candidate” simply means that this set of attributes could be used as the primary key.
Whether a set of attributes constitutes a CK or not depends entirely on the data in the table – not just on whatever data happens to be in the table at that moment but on any set of data that could be realistically be in the table over the life of the database.
- Does the set {cFirstName, cLastName, cPhone} meets the test. If not, what other attributes might we need in the table to make a candidate key?

Discussion: More About Keys – PK Size Might Matter

- ❑ Copying three attributes from the parent table to the child table each time a child record is created can be a lot of data even if we have a valid candidate key. It may make sense to “make up” a PK that is small enough. There are two types of “made up” PKs:
- ❑ **surrogate PK** – a single, small attribute (such as a number) that has no descriptive value such as an id number
- ❑ **substitute PK** – a single, small attribute (such as an abbreviation) that has at least some descriptive value.

Discussion: More About Keys – Surrogate/Substitute Keys

- ❑ Do not automatically add surrogate or substitute keys to a table until you are sure that
 1. there is at least one candidate key before the surrogate is added
 2. the table is a parent in at least one association
 3. there is no candidate key small enough for its values to be copied many times into the child table
- ❑ **external key** – a surrogate or substitute key already defined by someone else such as a zip code, UPC, ISBN
- ❑ Only use a Social Security Number as a **NON-KEY** field if required by law

Discussion: More About Keys – Revising the Relation Scheme

Customers

custID	cFirstName	cLastName	cPhone	cStreet	cZipCode
PK	Candidate Key (1 of 2)				

1..1 (*parent*)

Orders

0..* (*child*)

FK		
custID	orderDate	soldBy
Primary Key		

Discussion: More About Keys – Customer Joined to Orders

Customers

custid	cfirstname	clastname	cphone	cstreet	czipcode
1234	Tom	Jewett	714-555-1212	10200 Slater	92708
5678	Alvaro	Monge	562-333-4141	2145 Main	90840
9012	Wayne	Dick	562-777-3030	1250 Bellflower	90840

Orders

custid	orderdate	soldby
5678	2003-07-14	Patrick
9012	2003-07-14	Patrick
5678	2003-07-18	Kathleen
5678	2003-07-20	Kathleen

Customers joined to Orders

custid	cfirstname	clastname	cphone	cstreet	czipcode	orderdate	soldby
5678	Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-14	Patrick
9012	Wayne	Dick	562-777-3030	1250 Bellflower	90840	2003-07-14	Patrick
5678	Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-18	Kathleen
5678	Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-20	Kathleen

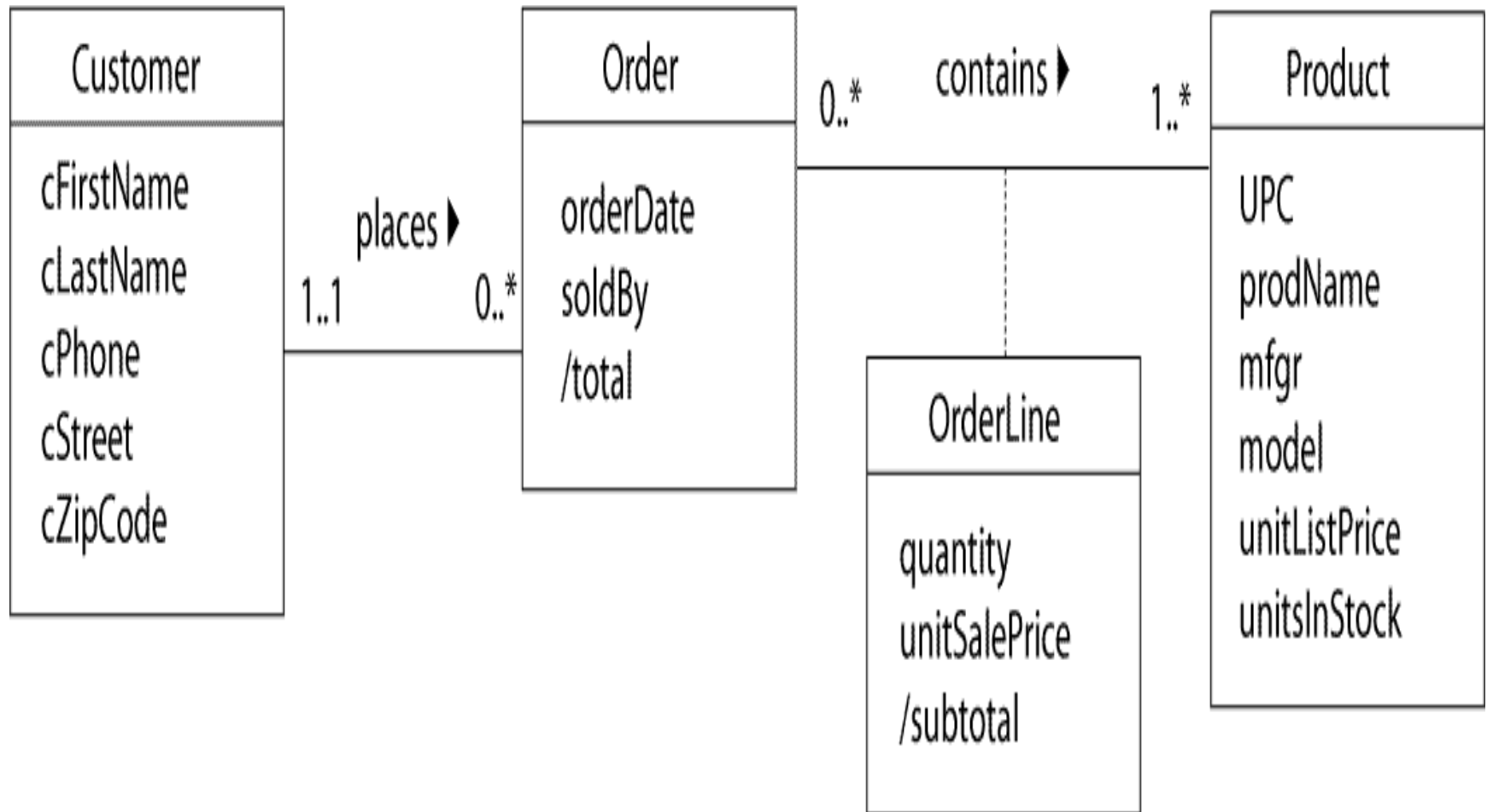
Design Pattern: Many-to-Many (Order Entry)

- ❑ ***Design patterns*** are modeling situations that you will find over and over again as you design real databases. These become tools that you use constantly use to build enterprise models.
- ❑ The sales database represents one of these design patterns called “many-to-many.” In order to complete the pattern we need products to sell.

Design Pattern: Many-to-Many (Order Entry)–Finishing the Pattern

1. Define the product
 2. Define the relationship between orders and products
 3. Define the need for orderlines
- Since the maximum multiplicity in each direction is “many,” this is called a **many-to-many** association between Orders and Products.
 - Each time an order is placed for a product, we need to know how many units of that product are being ordered and what price we are actually selling the product for. These attributes are a result of the association between the Order and the Product. We show them in an **association class** that is connected to the association by a dotted line.

Design Pattern: Many-to-Many (Order Entry) – Class Diagram



Design Pattern: Many-to-Many (Order Entry) – Junction Table

We can't represent a many-to-many association directly in a relation scheme, because two tables can't be children of each other—there's no place to put the foreign keys. So for *every* many-to-many, we will need a ***junction table*** in the database, and we need to show the scheme of this table in our diagram.

Design Pattern: Many-to-Many (Order Entry) – Relation Scheme

Customers

custID	cFirstName	cLastName	cPhone	cStreet	cZipCode
PK	Candidate Key (1 of 2)				

1..1 (*parent*)
Orders 0..* (*child*)

FK					
custID	orderDate	soldBy			
Primary Key					

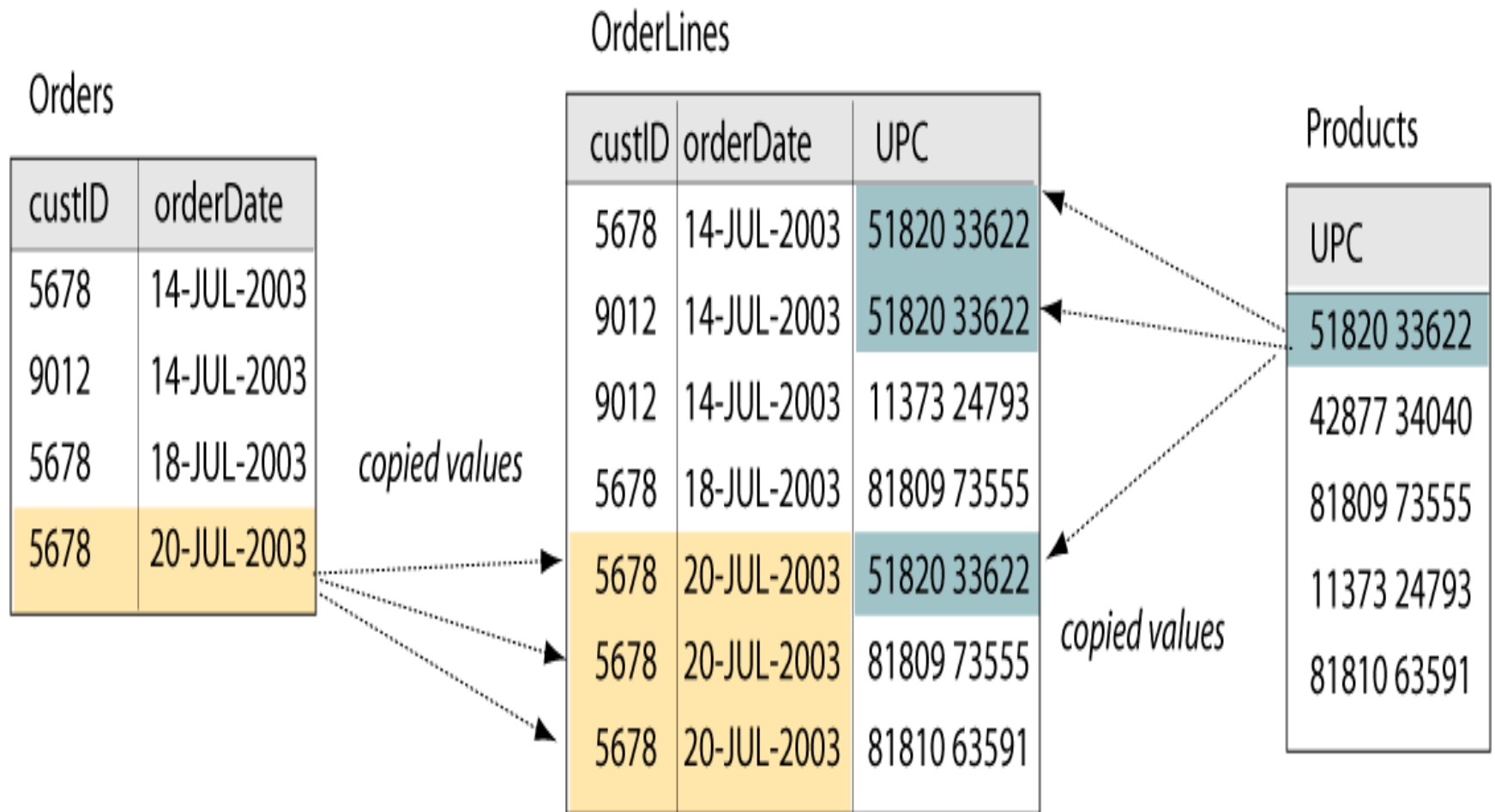
1..1 (*parent*)
OrderLines 1..* (*child*)

Foreign Key					
custID	orderDate	UPC	quantity	unitSalePrice	
Primary Key					
		FK			

Products 1..1 (*parent*)
0..* (*child*)

PK					
UPC	prodName	mfg	model	unitListPrice	unitsInStock
		CK			

Design Pattern: Many-to-Many (Order Entry) – Data Representation



Exercise: Building a Database With Notecards

Exercise

Exercise: Students and Classes

- ❑ If you are using this Web site in a class at a university, you are probably interested in what grade you will get in the class.
- ❑ You will need to model the students and the (university) classes, their attributes, and the relationship between them. Be sure to show where the grade should be recorded.
 - Describe each class in English.
 - Draw the class diagram, including association classes if required.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- ❑ The solution to this exercise will be discussed in class

Exercise: Book Authors

- ❑ Refer back to the [library book exercise](#) in the preceeding section if necessary. Remember that we couldn't include authors in the CatalogEntry class because there might be more than one author for each book.
- ❑ In a real (old-fashioned) library catalog, there was an additional card for each author of a book. (Author and title cards were contained in different sets of card drawers.)

Exercise: Book Authors

- ❑ Design a class diagram that shows the catalog entry plus authors, the attributes of each class, and the association between the classes.
 - Describe each class in English.
 - Draw the class diagram.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- ❑ Your first version of this model might not have included a way to identify which author should be listed first, second, and so on. In real life, authors are very sensitive about this. Revise your model if necessary to accommodate this requirement.
- ❑ The solution to this exercise will be discussed in class

Exercise: Auto Repair

- ❑ You are designing a database for an automobile repair shop. When a customer brings in a vehicle, a service advisor will write up a repair order. This order will identify the customer and the vehicle, along with the date of service and the name of the advisor.
- ❑ A vehicle might need several different types of service in a single visit. These could include oil change, lubrication, rotate tires, and so on.
- ❑ Each type of service is billed at a pre-determined number of hours work, regardless of the actual time spent by the technician.
- ❑ Each type of service also has a flat book rate of dollars-per-hour that is charged.

Exercise: Auto Repair

- Describe each class in English.
 - Draw the class diagram, including association classes if required.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- The solution to this exercise will be discussed in class

Exercise: Redbox

- You are designing a database for a DVD Rental Kiosk. To start, you need the part of the system that will allow a customer to check out a group of dvds and receive an invoice that lists them along with the cost and the date they are due.
 - Describe each class in English.
 - Draw the class diagram, including association classes if required.
 - Describe each association in English (both directions).
 - Draw the relation scheme.

Exercise: Santa's List

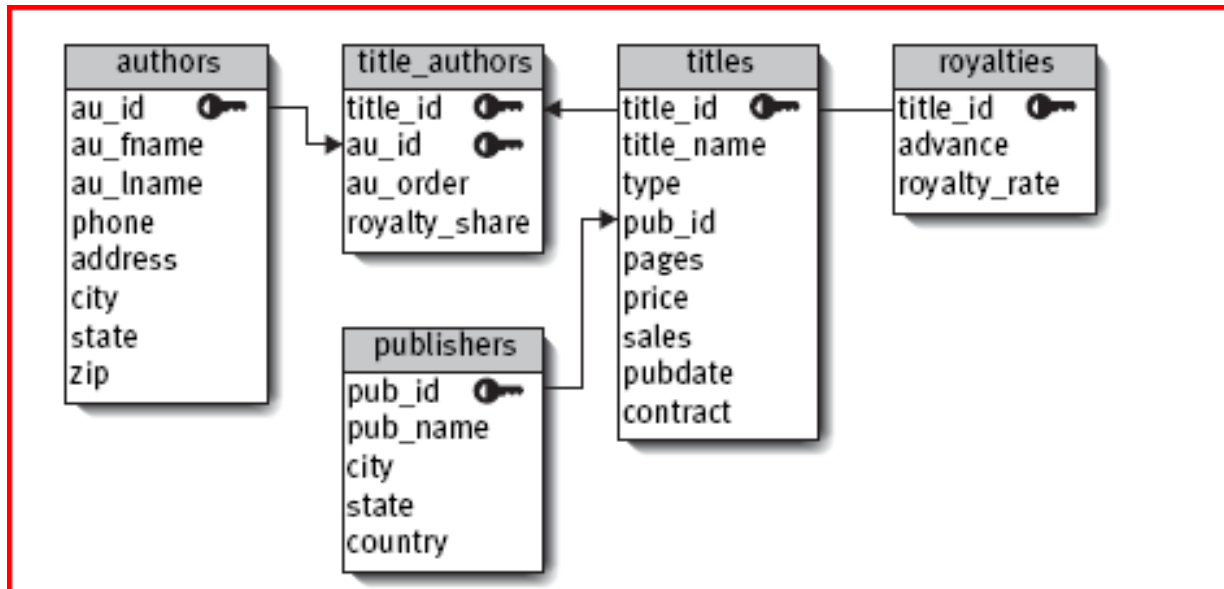
- ❑ You are designing a database to help Santa Claus and his elves to keep track of the toys he gives to children.
- ❑ He obviously needs to know the name and address of each child on his list, and when they were born.
- ❑ Every year, each child will give Santa a list of the toys that he/she wants.
- ❑ Santa will record whether that child has been naughty or nice that year, then pick which toys to actually deliver.
- ❑ A child won't get more than one of each toy, probably won't get everything that he/she asked for, and might get something that wasn't asked for (like a lump of coal if he's been naughty).
- ❑ Of course, Santa doesn't want to give a child any of the same toys this year as he gave them last year.

Exercise: Santa's List

- ❑ Hint: the solution is much easier than you might think the first time you read through this exercise.
 - Describe each class in English.
 - Draw the class diagram, including association classes if required.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- ❑ The solution to this exercise will be discussed in class

Exercise: Books Relation Scheme

- Look at the class diagram for the Books Database in the SQL-QS book (3/e pg 51).
 - Draw the relation scheme.
- The solution to this exercise will be discussed in class



Design Pattern: Many-to-Many With History (the Library Loan)

- ❑ There are times when we need to allow the same two individuals in a many-to-many association to be paired more than once. This frequently happens when we need to keep a history of events over time.
- ❑ Example: In a library, customers can borrow many books and each book can be borrowed by many customers, so this seems to be a simple many-to-many association between customers and books. But any one customer may borrow a book, return it, and then borrow the same book again at a later time. The library records each book loan separately; there is no invoice for each set of borrowed books (that is, there is no equivalent here of the Order in the order entry example).

Design Pattern: Many-to-Many With History (the Library Loan) – Classes

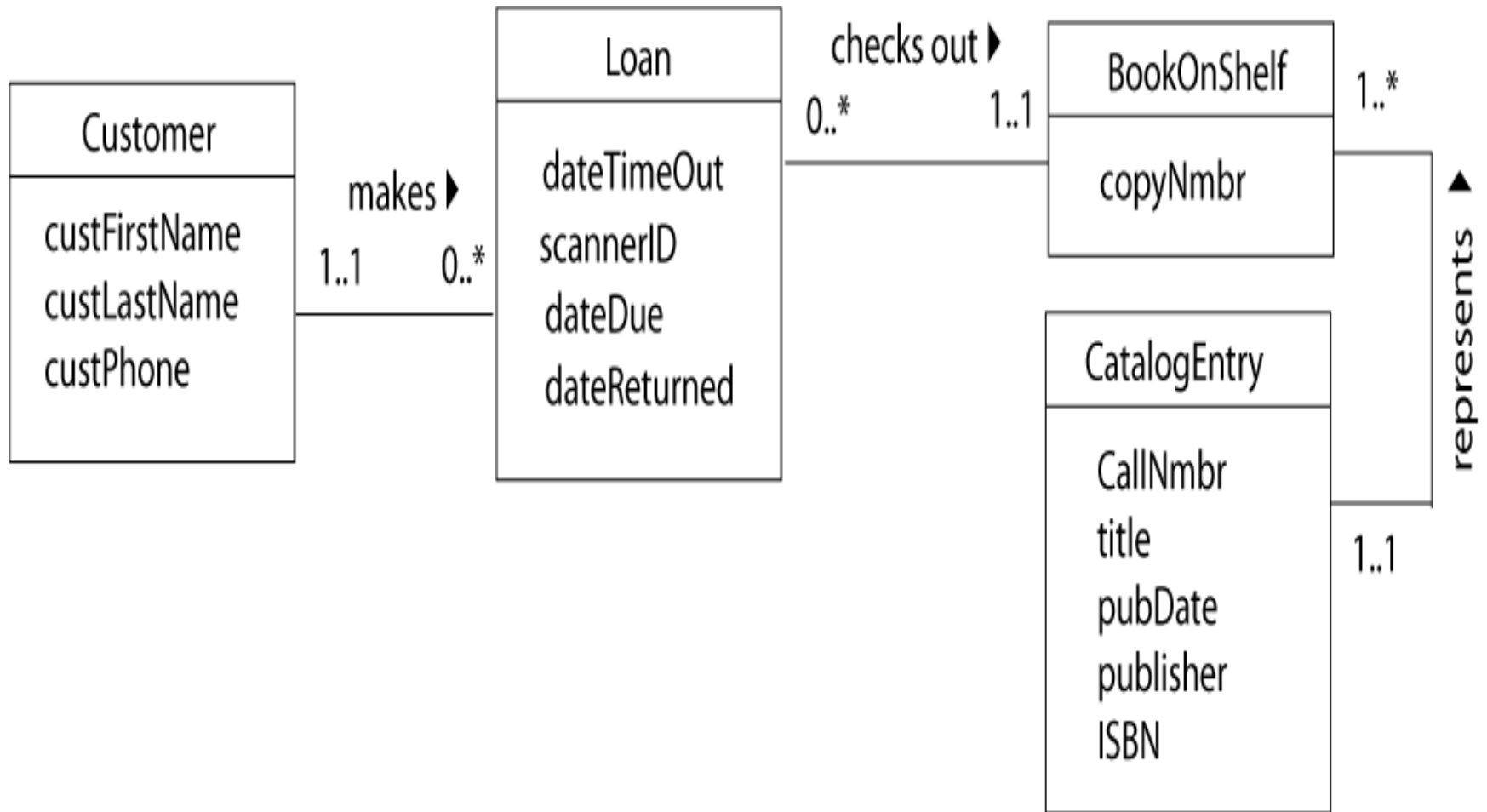
The loan is an event that happens in the real world. We need a regular class to model it correctly. We'll call this the “library loan” design pattern. First, we need to understand what the classes and associations mean:

- “A **customer** is any person who has registered with the library and is eligible to check out books.”
- “A **catalog entry** is essentially the same as an old-fashioned index card that represents the title and other information about books in the library, and allows the customers to quickly find a book on the shelves.”
- “A **book** is the physical volume that is either sitting on the library shelves or is checked out by a customer. There can be many physical books represented by any one catalog entry.”
- “A **loan** event happens when one customer takes one book to the checkout counter, has the book and her library card scanned, and then takes the book home to read.”

Design Pattern: Many-to-Many With History (the Library Loan)-Associations

- “Each Customer makes *zero or more* Loans.”
- “Each Loan is made by *one and only one* Customer.”
- “Each Loan checks out *one and only one* Book.”
- “Each Book is checked out by *zero or more* Loans.”
- “Each Book is represented by *one and only one* CatalogEntry (catalog card).”
- “Each CatalogEntry can represent *one or more* physical copies of the same book.”

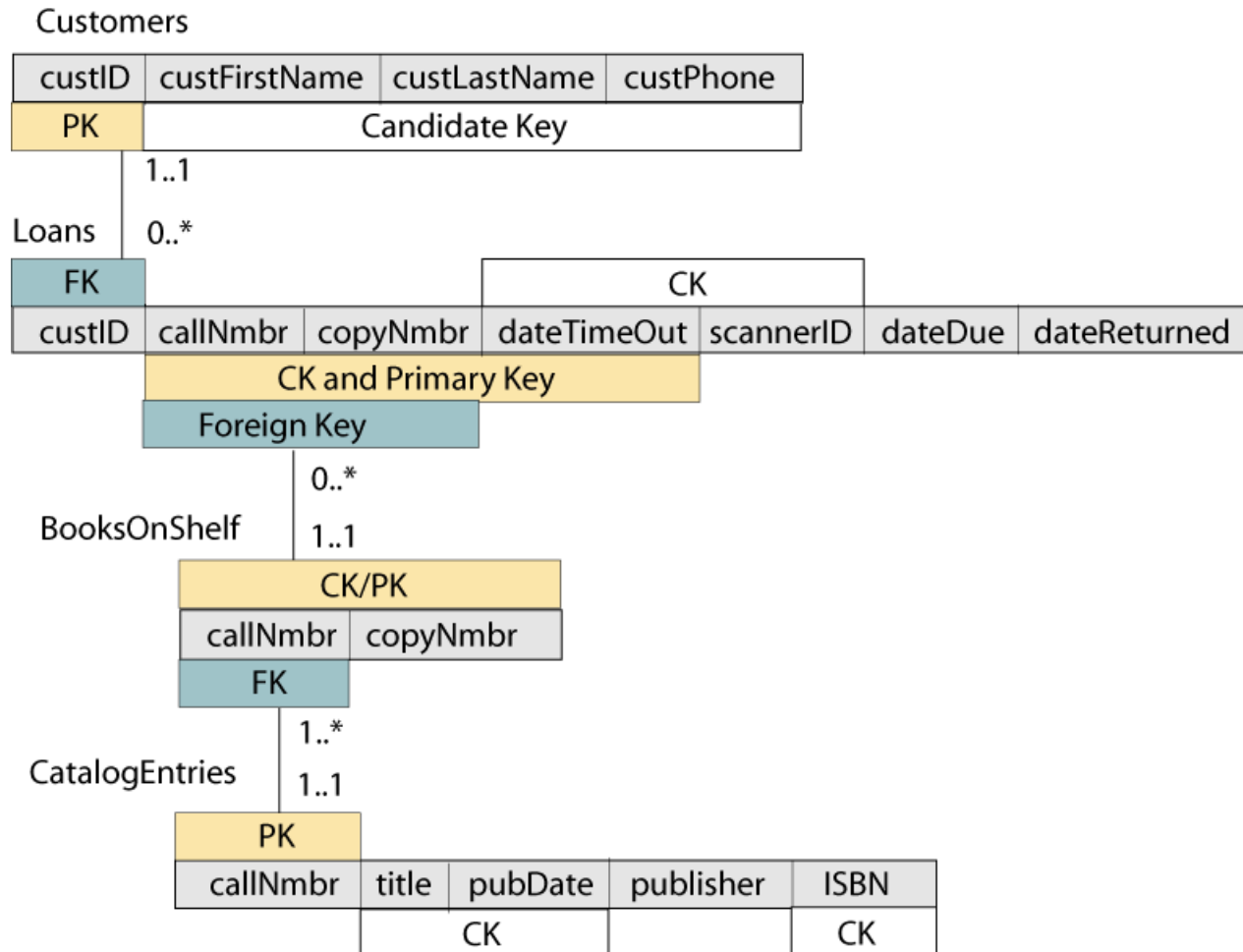
Design Pattern: Many-to-Many With History(Library Loan)–Class Diagram



Design Pattern: Many-to-Many With History (Library Loan)–Keys

- As in the order entry example, the Customers table will need a surrogate key to save space when it is copied in the Loans.
- The CatalogEntry already has two external surrogate keys: the call number and the ISBN.
 - The first of these is defined by the Library of Congress Classification system, and contains codes that represent the subject, author, and year published.
 - The second of these is defined by a standard, number 2108. We'll use the callNmbr as the primary key, since it has more descriptive value than the ISBN and is smaller than the descriptive candidate key:
CK {title, pubDate}
- The dateTimeOut is needed as part of the key in the Loans table in order to pair a customer with the same book more than once.

Design Pattern: Many-to-Many With History (Library Loan)–Diagram



Design Pattern: Many-to-Many With History (Library Loan)

- ❑ As we would do in a junction table scheme, we'll copy the primary key attributes from both the Customers and the BooksOnShelf into the Loans scheme. This tells us which customer borrowed which book, but it doesn't tell when it was borrowed. We have to know the `dateTimeOut` in order to pair a customer with the same book more than once.
- ❑ We can call this a ***discriminator attribute***, since it allows us to discriminate between the multiple pairings of customer and book. If you refer back to the UML class diagram, you'll see that the loan, which would have been a many-to-many association class between customers and books, has become a “real” class because of the discriminator attribute.
- ❑ Notice that there is actually another CK for the loan:
 `{dateTimeOut, scannerID}`
 since it is also physically impossible for the same scanner to read two different books at exactly the same time.
- ❑ We chose `{callNmbr, copyNmbr, dateTimeOut}` because it has just a bit more descriptive value and because we don't care about size here (since the Loan has no children).

Exercise: Employee Timecards

- ❑ In many businesses, employees may work on a number of different projects.
- ❑ Each week, they will submit a time card that lists each project on a separate line, along with the number of hours that they have worked that week on that project.
 - Describe each class in English.
 - Draw the class diagram, including association classes if required.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- ❑ The solution to this exercise will be discussed in class

Exercise: More Many-to-Many

- Model any three many-to-many (without history) relationships that you find. You will need to model the classes, their attributes, and the relationship between them.
 - Describe each class in English.
 - Draw the class diagram, including association classes if required.
 - Describe each association in English (both directions).
 - Draw the relation scheme.
- We'll discuss your examples in class.

Design Pattern: Subkeys (the Zipcode)

- ❑ One of the major goals of relational database design is to prevent unnecessary duplication of data.
- ❑ The Contact class represents any person who is a business associate, friend, or family member. Its attributes are:
 - Contact first name.
 - Contact last name.
 - Contact street.
 - Contact zip code.
 - Contact city.
 - Contact state.

Contact
firstName lastName street zipCode city state

Design Pattern: Subkeys (the Zipcode)

- Problem

It may not be obvious that the model has a problem until you look at the data

Contacts

firstName	lastName	street	zipCode	city	state
George	Barnes	1254 Bellflower	90840	Long Beach	CA
Susan	Noble	1515 Palo Verde	90840	Long Beach	CA
Erwin	Star	17022 Brookhurst	92708	Fountain Valley	CA
Alice	Buck	3884 Atherton	90836	Long Beach	CA
Frank	Borders	10200 Slater	92708	Fountain Valley	CA
Hanna	Diedrich	1699 Studebaker	90840	Long Beach	CA

Design Pattern: Subkeys (the Zipcode)

– Functional Dependency

- ❑ **functional dependency** is simply a more formal term for the super key property. If X and Y are sets of attributes, then the notation $X \rightarrow Y$ is read “ X functionally determines Y ” or “ Y is functionally dependent on X .” This means that if I’m given a table filled with data plus the value of the attributes in X , then I can uniquely determine the value of the attributes in Y .
- ❑ A super key always functionally determines all of the other attributes in a relation (as well as itself). This is a “good” FD. A “bad” FD happens when we have an attribute or set of attributes that are a super key for *some* of the other attributes in the relation, but *not* a super key for the entire relation. We call this set of attributes a **subkey** of the relation.

Design Pattern: Subkeys (the Zipcode)

– Functional Dependency (con't).

- A subkey dependency enables us to detect when a relation scheme has redundancy -- when a table using the scheme will contain unnecessary duplication of information.
- A relation scheme has redundancy whenever there is a subkey dependency.

Design Pattern: Subkeys (the Zipcode)

– Preventing/Removing Redundancy

- There is a very simple way to fix the problem with a relation scheme that has redundancy, as detected by the presence of a subkey.
- In the steps given below, the scheme with redundancy is R and the subkey is represented by the FD $W \rightarrow Z$, where each of W and Z is a set of attributes that is a subset of the attributes in R .

Design Pattern: Subkeys (the Zipcode) – Preventing/Removing Redundancy(con't).

Replace R by two schema, R_1 and R_2 as described in the following steps.

1. Assign R_1 the attributes in the union of the attributes in the subkey FD. That is $R_1 = \{W \cup Z\}$. Since $W \rightarrow Z$, by definition, W is a superkey of R_1 .
2. Assign R_2 the set of attributes $\{R - Z\}$, that is, all the attributes in R except those in Z , the attributes on the right-hand-side of the subkey FD
3. Both R_1 and R_2 share W in common. In R_1 , W is a superkey and in R_2 it becomes a foreign key.

Design Pattern: Subkeys (the Zipcode) – Preventing/Removing Redundancy(con't).

- The “bad” subkey dependency has been removed because we’ve moved the attributes that were functionally dependent on W to another scheme, and we’ve made W the super key of that scheme.

Design Pattern: Subkeys (the Zipcode)

- Solution

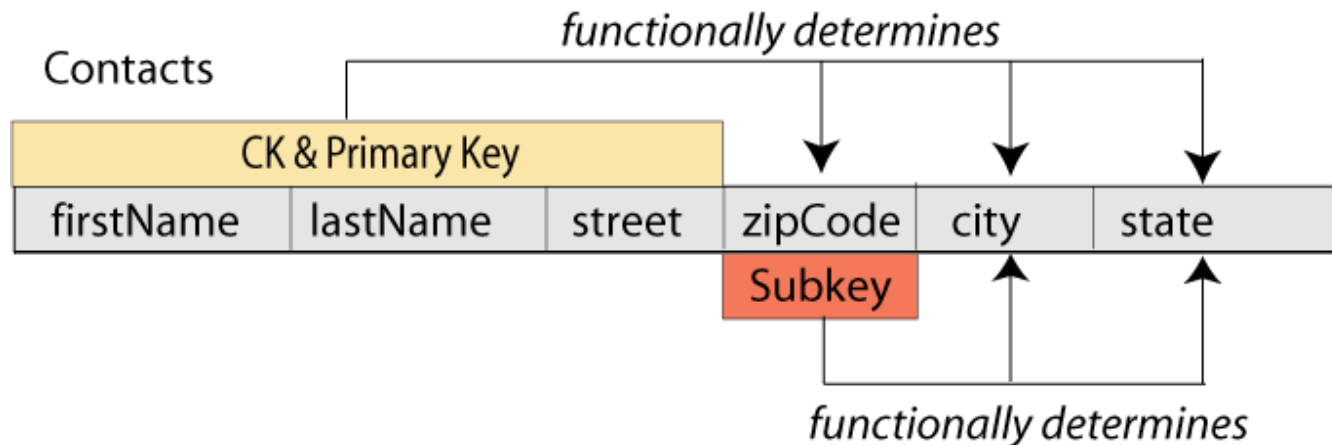
In other words:

1. Remove all of the attributes that are dependent on the subkey. Put them into a new scheme.
2. Duplicate the subkey attribute set in the new scheme, where it becomes the primary key of the new scheme.
3. Leave a copy of the subkey attribute set in the original scheme, where it is now a foreign key. It is no longer a subkey, because you've gotten rid of the attributes that were functionally dependent on it, and you've made it the primary key of its own table. The revised model will have a many-to-one relationship between the original scheme and the new one.

Design Pattern: Subkeys (the Zipcode)

- Subkey

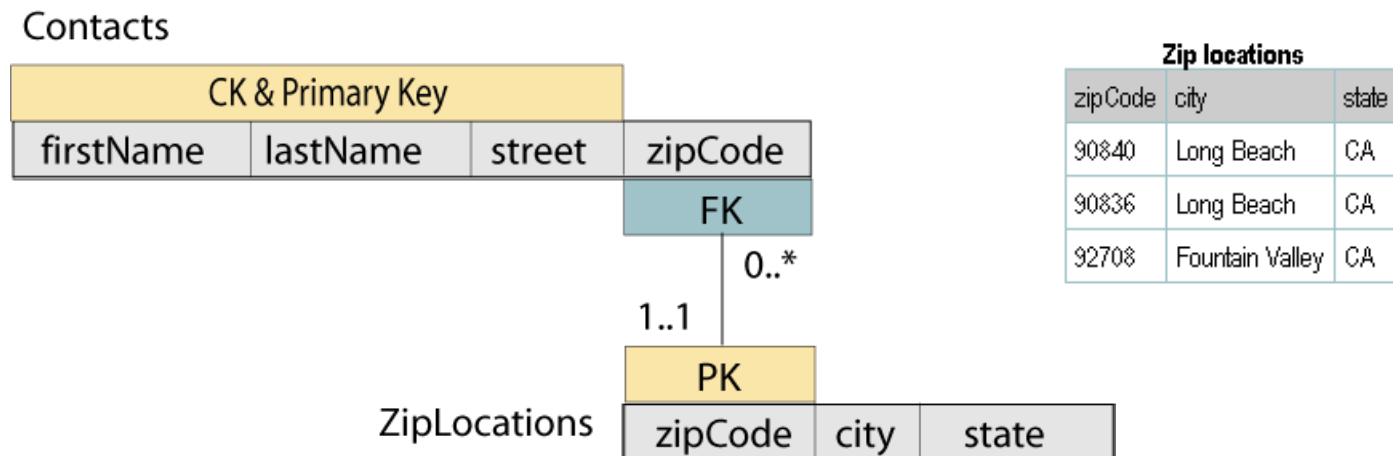
In the Contacts table, the zipCode is a **subkey**. It functionally determines the city and state. The opposite is not true, because many cities have more than one zip code.



Design Pattern: Subkeys (the Zipcode)

– Lossless Join Decomposition

Moving the data to a new table and then joining the two tables is called ***lossless join decomposition*** of the original table



Design Pattern: Subkeys (the Zipcode)

- Normalization

- ❑ **Normalization** means following a procedure or set of rules to insure that a database is well designed. Most normalization rules are meant to eliminate redundant data (that is, unnecessary duplicate data) in the database.
- ❑ Subkeys always result in redundant data, so we need to eliminate them. If there are no subkeys in any of the tables in your database, you have a well-designed model according to what is usually called **third normal form**, or 3NF.
- ❑ Edgar F. Codd was a mathematician and computer scientist who laid the theoretical foundation for relational databases.
- ❑ *“The key, the whole key and nothing but the key so help me Codd”*

Design Pattern: Subkeys (the Zipcode)

– Class Diagram

- When we find a subkey in a relation scheme or table, we also know that the original UML class was badly designed. The problem, always, is that we have actually placed two conceptually different classes in a single class definition.
- In this example, a zipCode is not just an attribute of the Contact class. It is part of a ZipLocation class, which we can describe as “a geographical location whose boundaries have been uniquely identified by the U.S Postal Service for mail delivery.”
- As with all one-to-many associations, the association itself identifies which Contact lives in which ZipLocation. If we had started with this class diagram, we would have produced exactly the same relation scheme that we developed with the normalization process above!



Exercise: Plant Species

- ❑ You are working on a database for a company that grows and sells plants.
- ❑ One important table contains a list of the plant species that they grow, which are identified botanically by their genus and specie name, family, and common name.
- ❑ Even if you have never heard of these terms, you can analyze the table by looking at the data given below:

Exercise: Plant Species

Plant species			
genus	specie	family	commonname
Ardesia	japonica	Myrsinaceae	Marlberry
Beaucarnea	recurvata	Agavaceae	Ponytail
Centaurea	cineraria	Asteraceae	Dusty Miller
Centaurea	gymnocarpa	Asteraceae	Dusty Miller
Centaurea	montana	Asteraceae	
Dracaena	draco	Agavaceae	Dragon Tree
Dracaena	marginata	Agavaceae	
Echeveria	elegans	Crassulaceae	Hen and Chicks
Kalanchoe	beharensis	Crassulaceae	Felt Plant
Kalanchoe	pinnata	Crassulaceae	Air Plant
Pseudosasa	japonica	Poaceae	Arrow Bamboo
Senecio	cineraria	Asteraceae	Dusty Miller

Exercise: Plant Species

- Draw the relation scheme for this table as it is shown above. Identify the primary key.
- Draw the relation scheme for a lossless join decomposition of this table.

Exercise: Student Grades

- You are working on an application to store student data for the CECS department. Based on the data in the following table:

<i>SID</i>	<i>name</i>	<i>email</i>	<i>CID</i>	<i>grade</i>
142	Bart	bart@fox.com	CPS196	B-
142	Bart	bart@fox.com	CPS114	B
123	Milhouse	milhouse@fox.com	CPS196	B+
857	Lisa	lisa@fox.com	CPS196	A+
857	Lisa	lisa@fox.com	CPS130	A+
456	Ralph	ralph@fox.com	CPS114	C
217	Maggie	Maggie@fox.com	CPS196	B-

- Draw the relation scheme for this table as it is shown above. Identify the primary key.
- Draw the relation scheme for a lossless join decomposition of this table.

Design Pattern: Repeated Attributes (the Phone Book)

The contact manager example from our preceding discussion of subkeys is also an excellent illustration of another problem that is found in many database designs.

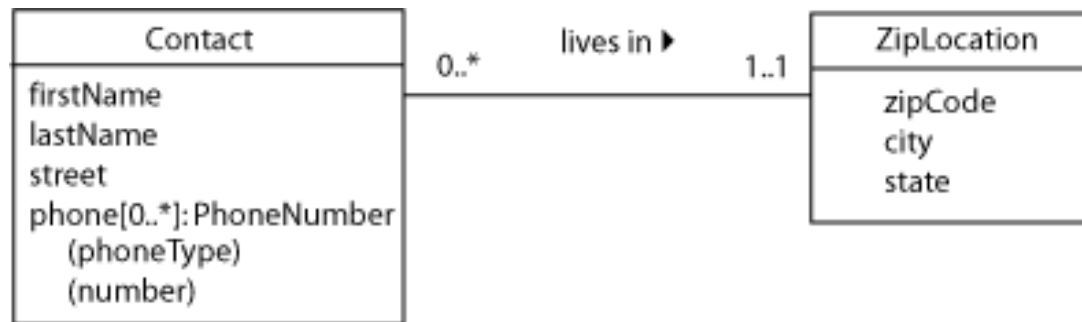


Contact phones

firstName	lastName	homePhone	workPhone	cellPhone	fax	pager
George	Barnes	562-874-1234		310-999-3628		
Susan	Noble	562-975-3388	714-847-3366			
Erwin	Star				714-997-5885	714-997-2428
Alice	Buck		562-577-1200	562-561-1921		
Frank	Borders	714-968-8201				
Hanna	Diedrich			562-786-7727		

Design Pattern: Repeated Attributes (the Phone Book) – Weak Entity

- Note the large amount of null values and the inability of the model to keep up with changing times. Notice that the phone numbers are actually ***repeated attributes***. It is essentially a class within a class.
- This can be called a ***weak entity*** since it can't exist without the parent entity type.

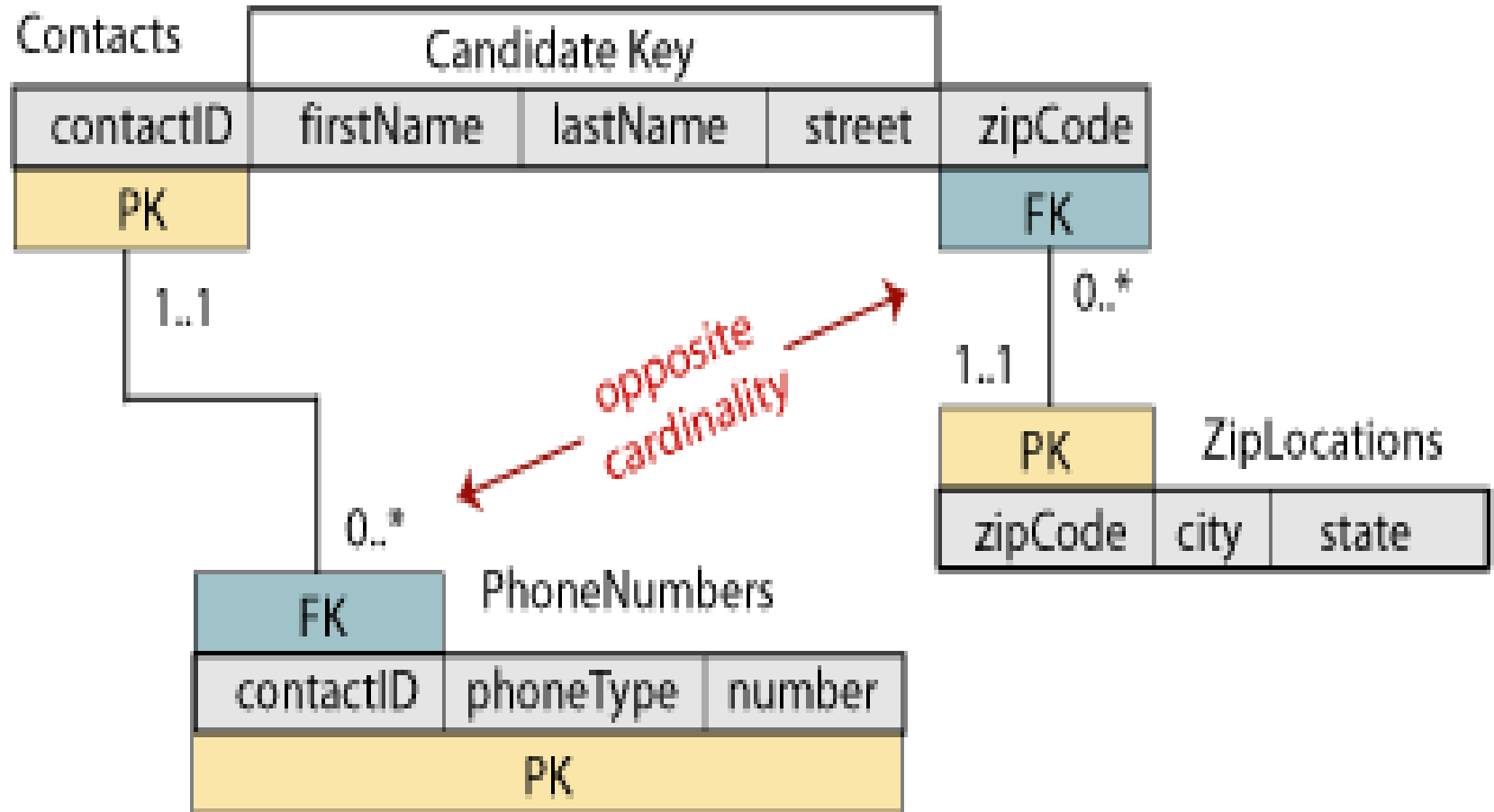


Design Pattern: Repeated Attributes (the Phone Book) - Fix

In order to fix this problem we need to create a new table:

1. Remove all of the phone number fields from the Contacts relation. Create a new scheme that has the attributes of the phone number structure (phone type and number).
2. The Contacts relation has now become a parent so we add a surrogate key and copy it into the new scheme. There is now a one-to-many relationship between Contacts and PhoneNumbers.
3. To identify each phone number, we need to know at least who it belongs to and what type it is. However, a person can have two cell phones so all three attributes are needed for the PK. Since this is not a parent relation, the PK size doesn't matter.

Design Pattern: Repeated Attributes (the Phone Book) - Relation Scheme



Design Pattern: Repeated Attributes (the Phone Book) – Tables

Contacts

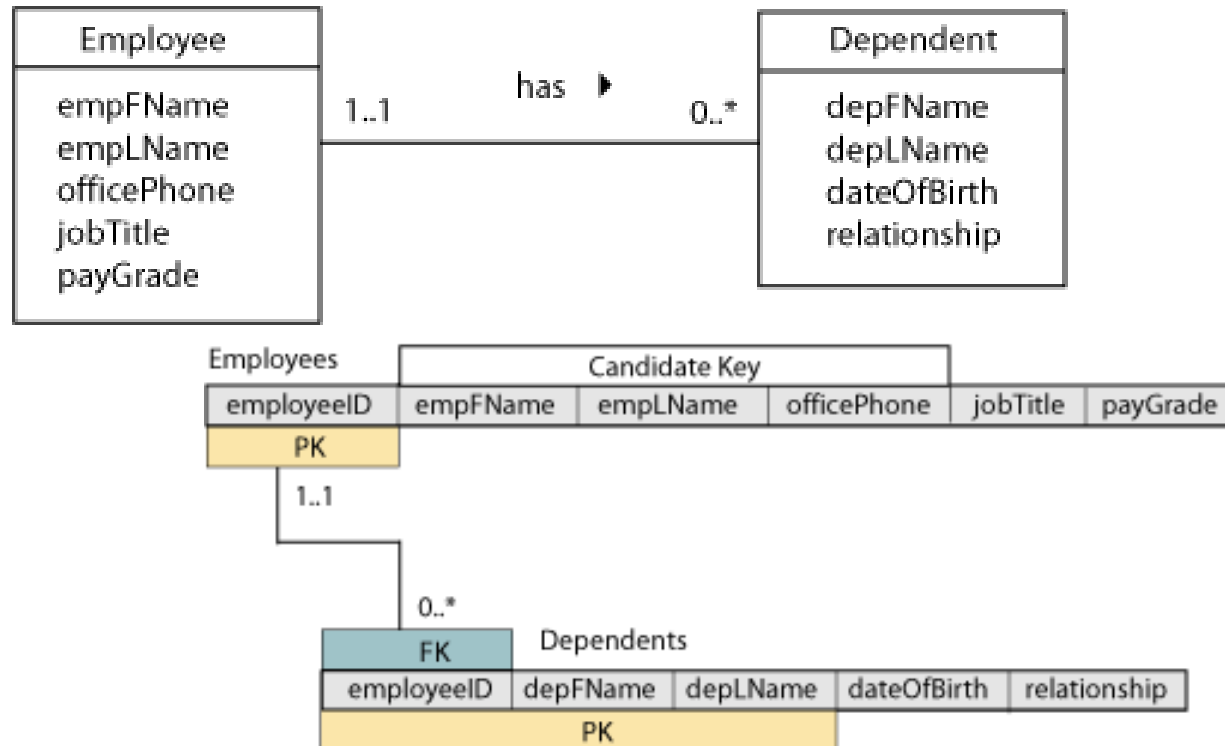
contactid	firstname	lastname	street	zipcode
1639	George	Barnes	1254 Bellflower	90840
5629	Susan	Noble	1515 Palo Verde	90840
3388	Erwin	Star	17022 Brookhurst	92708
5772	Alice	Buck	3884 Atherton	90836
1911	Frank	Borders	10200 Slater	92708
4848	Hanna	Diedrich	1699 Studebaker	90840

Phone numbers

contactid	phonetype	number
1639	Home	562-874-1234
1639	Cell	310-999-3628
5629	Home	562-975-3388
5629	Work	714-847-3366
3388	Fax	714-997-5885
3388	Pager	714-997-2428
5772	Work	562-577-1200
5772	Cell	562-561-1921
1911	Home	714-968-8201
4848	Cell	562-786-7727

Design Pattern: Repeated Attributes (the Phone Book) - Employees

The modeling technique shown above is useful where the parent class has relatively few attributes and the repeated attribute has only one or a very few attributes of its own. However, you can also model the repeated attribute as a separate class in the UML diagram. One classic textbook example is an employee database.



Design Pattern: Repeated Attributes (the Phone Book) – Employees (cont)

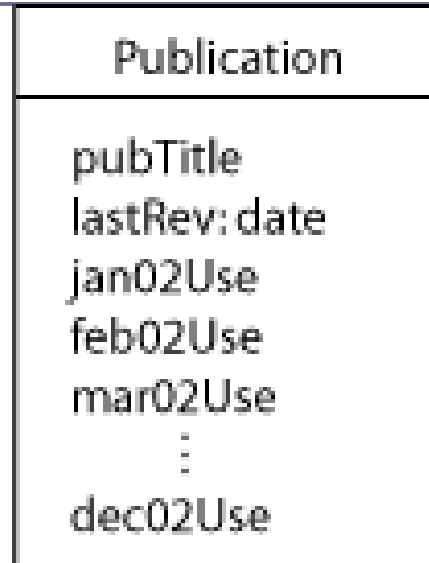
- This is the same example using the “shorthand” notation.

Employee
-empFname -empLname -officePhone -jobTitle -payGrade -dependent[0..*]:Dependents -(depFname) -(depLname) -(dateOfBirth) -(relationship)

Exercise: Monthly Publication Usage

- ❑ A large task for computer manufacturers is to keep track of their publications: product specification sheets, marketing brochures, user manuals, and so on.
- ❑ One major manufacturer had developed a spreadsheet to record the monthly usage of each publication (that is, how many copies of the publication were distributed each month).
- ❑ Then they decided to export the spreadsheet into a database.
- ❑ The export program naturally converted each column of the spreadsheet into a database table attribute, so the result looked something like this:

Exercise: Monthly Publication Usage



- ❑ Revise the class diagram to correct any problems that you find in this design.
- ❑ Then draw the relation scheme for your corrected model.

Exercise: Faculty Degrees

- A number of years ago, I taught evening courses at a small college extension center. The center staff kept a list of faculty members in one of the very early PC-based database programs. There was only one faculty table, part of which looked like this:

Faculty
facFirstName
facLastName
degree1
degree2
degree3

- No, I'm not making this up. There really were three fields to hold information about degrees that instructors had earned. The data in each degree field looked something like: MS in Computer Science, 1980, UC Santa Barbara.

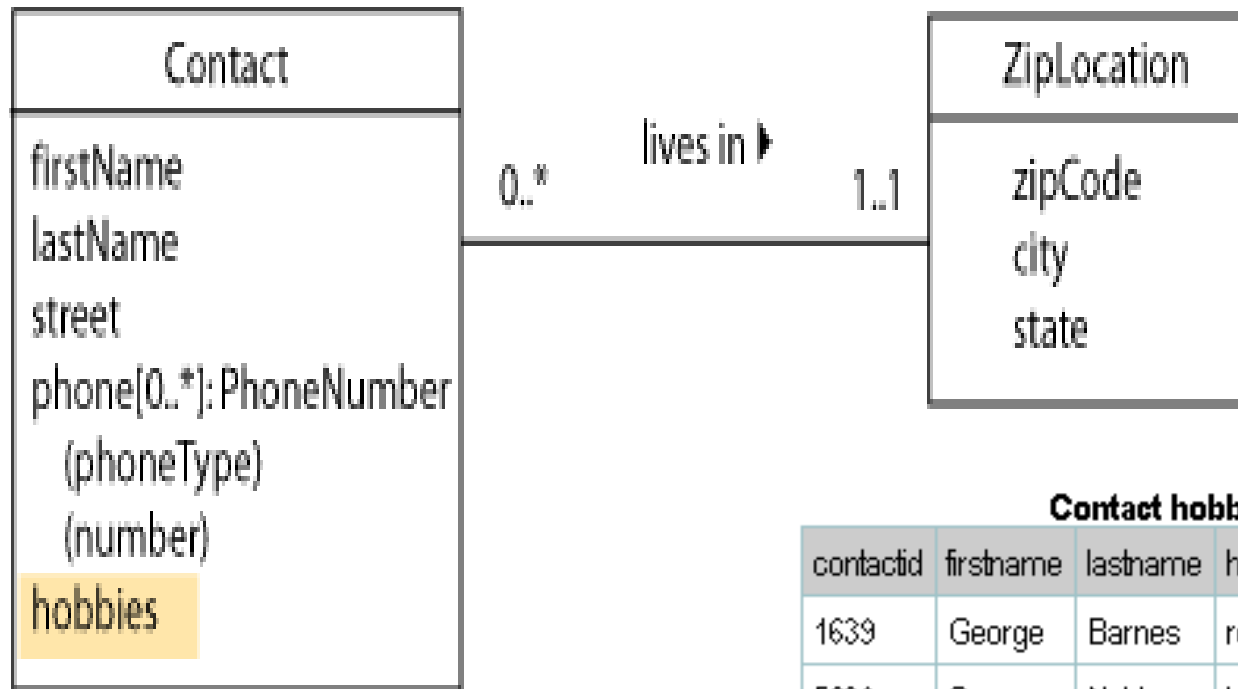
Exercise: Faculty Degrees

- ❑ Revise the class diagram to correct any problems that you find in this design.
- ❑ Then draw the relation scheme for your corrected model.

Design Pattern: Multi-Valued Attributes (Hobbies)

- ❑ When there are many distinct values entered in the same column of the table we have another design problem called ***multi-valued attributes*** .
- ❑ This makes it difficult to search the table for any one particular value and it is impossible to create a query that will individual list the values in that column. For example: hobbies.

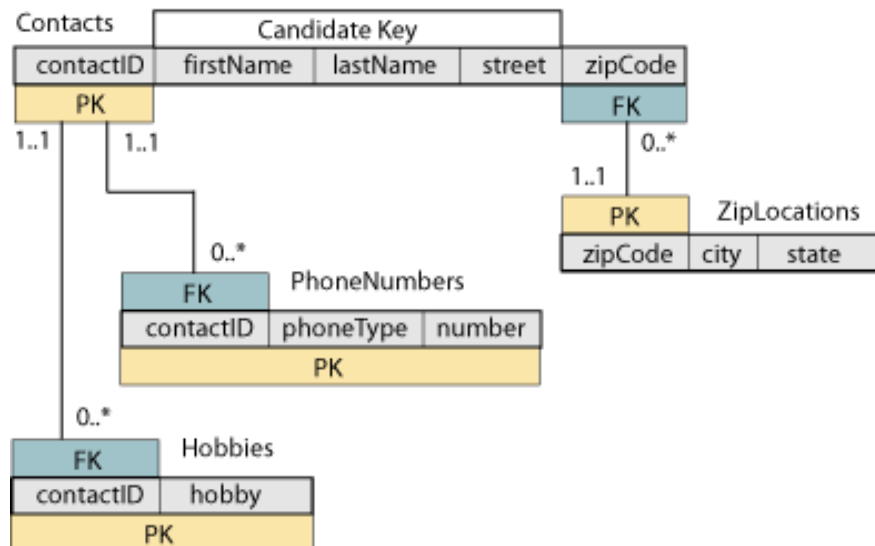
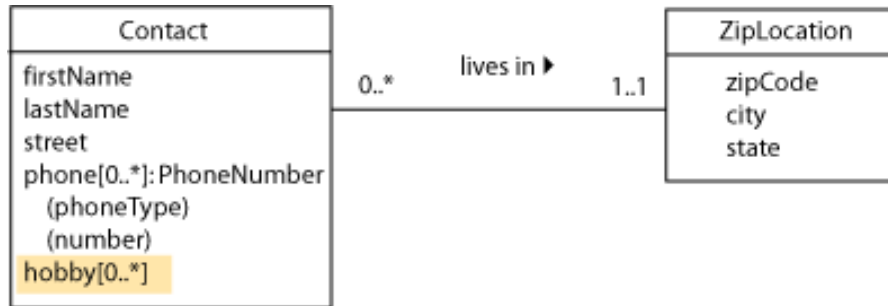
Design Pattern: Multi-Valued Attributes (Hobbies) – Class Diagram



Contact hobbies

contactid	firstname	lastname	hobbies
1639	George	Barnes	reading
5629	Susan	Noble	hiking, movies
3388	Erwin	Star	hockey, skiing
5772	Alice	Buck	
1911	Frank	Borders	photography, travel, art
4848	Hanna	Diedrich	gourmet cooking

Design Pattern: Multi-Valued Attributes (Hobbies) - Corrected

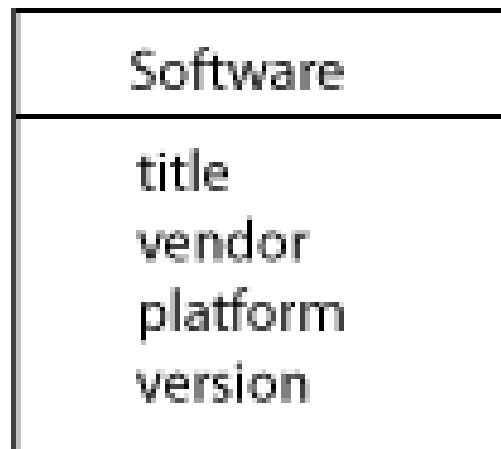


Hobbies

contactid	hobby
1639	reading
5629	hiking
5629	movies
3388	hockey
3388	skiing
1911	photography
1911	travel
1911	art
4848	gourmet cooking

Exercise: Software List

- Sometimes it takes more than just a glance at the class diagram to spot problems with a design.
- Consider the following class type that might be used by a software vendor to list software titles that are available.



Exercise: Software List

- There is nothing obviously wrong with this design. However, the users of this database might enter data that would cause problems, as shown in this table:

Software			
TITLE	VENDOR	PLATFORM	VERSION
Wordy	Macrosoft	Win, Mac	9.4, 6.7
Visual B--	Macrosoft	Win	6.0
Cherokee	Open Source	Linux, Solaris	10.4.5.2, 10.3.1.7
Inlook	Hinkysoft	Win, Linux, Palm OS	0.5
Corral Draw	Corral	Mac	22.1

- Revise the class diagram to correct any problems that you find in this design. Then draw the relation scheme for your corrected model.

Discussion: More About Domains

- ❑ Remember that a domain is the set of legal values that can be assigned to an attribute. Each attribute in a database must have a well-defined domain.
- ❑ One goal of database developers is provide ***data integrity***, part of which means insuring that the value entered in each field of a table is consistent with its attribute domain.
- ❑ Sometimes it's possible to devise a ***validation rule*** to help with this. Before you can design the data type and input format for an attribute, you have to understand the characteristic of its domain.

Discussion: More About Domains - Validation

- ❑ Some domains can only be described with a general statement of what they contain. Examples: name, addresses. For this, use a VARCHAR2 string that is long enough to hold the expected value.
- ❑ Some domains have at least some pattern. Examples, email addresses, URLs, North American phone numbers. These must be validated programmatically such as with a regular expression.
- ❑ Some domains have precise patterns. For example, SSNs and zip code.

Discussion: More About Domains – More Validations

- ❑ Easy domains to handle are those which can be specified by a well-defined, built-in system data type. These include integers, real numbers, and dates/times.
- ❑ You might need a range check.
- ❑ In most systems, a Boolean data type is also available although Oracle does not.
- ❑ Finally, there are many domains that may be specified by a well-defined, reasonably-sized set of constant values. These are enumerated domains and we'll cover these in the next section. For example, departments in the school, states.

Design Pattern: Enumerated Domains

- Attribute domains that may be specified by a well-defined, reasonable-size set of constant values are called ***enumerated domains***. These values should be kept in a separate table. Tables that are created for this purpose are commonly called ***lookup tables***.
- Although it might appear that this technique will create too many tables and query joins, the advantages outweigh the disadvantages. You can always break the rule if you have to for performance reasons.

Design Pattern: Enumerated Domains

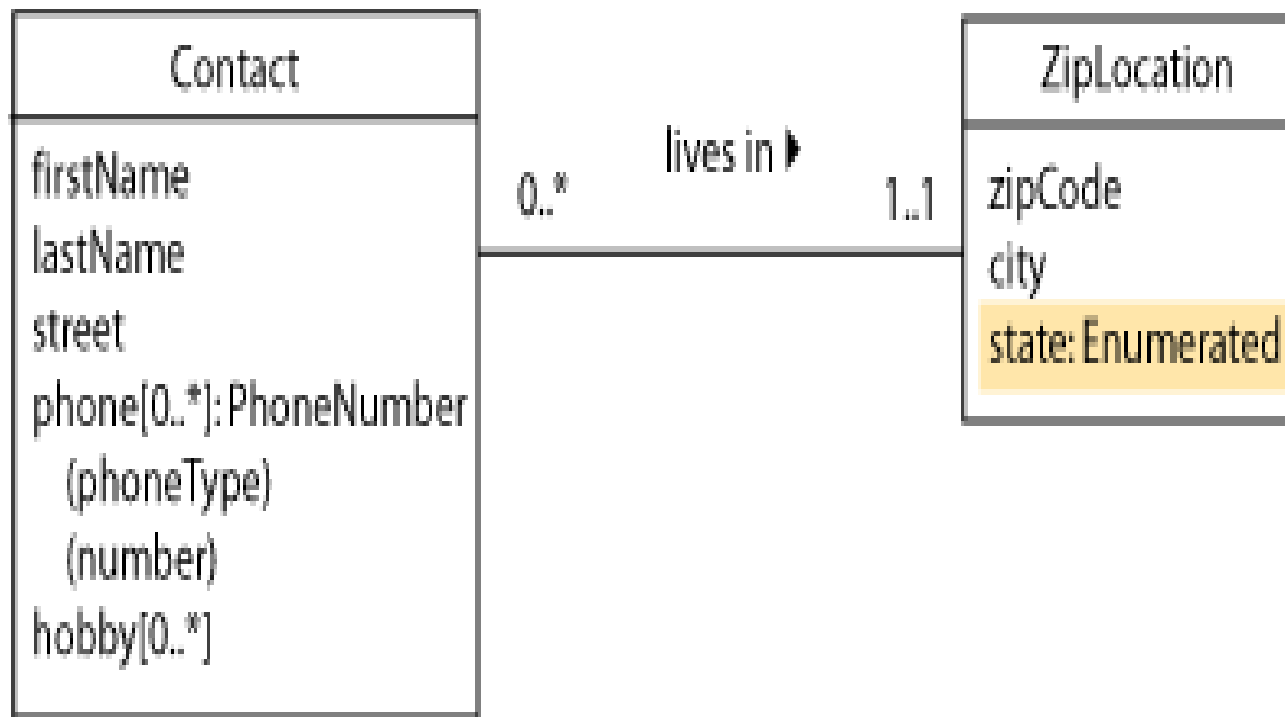
- Advantages

1. Data Integrity – changing the data in one place, avoiding deletion anomalies
2. You can read the values from the table into a combo box, list box or similar input control on either a web page or a GUI form. This allows the user to easily select only values that are valid in this domain at this time.
3. You can always update the table if new values are added to the domain, or if existing values are changed. This is much easier than modifying your user-interface code or your table structure.

Design Pattern: Enumerated Domains

– Class Diagram

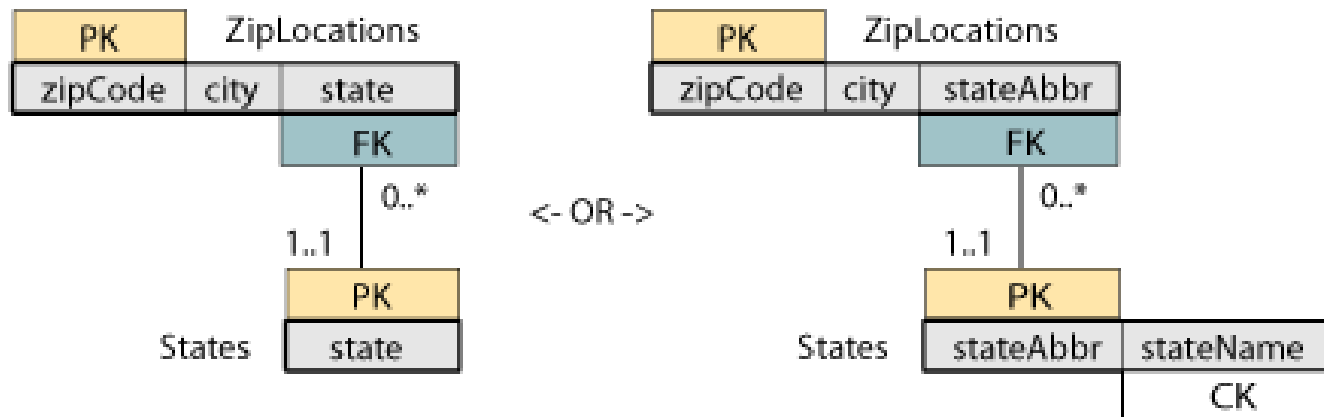
In our earlier ZipLocations example, the *state* attribute clearly fits the definition of an enumerated domain. In UML, we can simply use a data type specification to show this, without adding a new class type.



Design Pattern: Enumerated Domains

– Relation Scheme

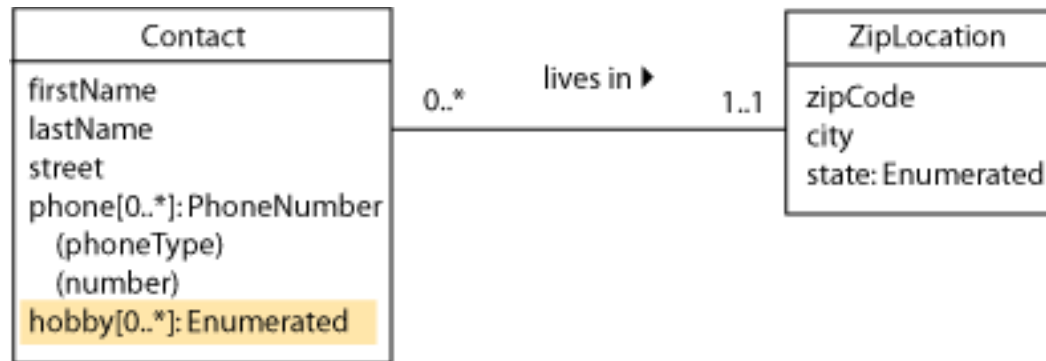
The relation scheme will show the table that contains the enumerated domain values. This table might have a single attribute, or it might have two attributes: one for the true values and one for a substitute key. The true values always form a candidate key of the table.



Design Pattern: Enumerated Domains

– Multi-Valued Attributes

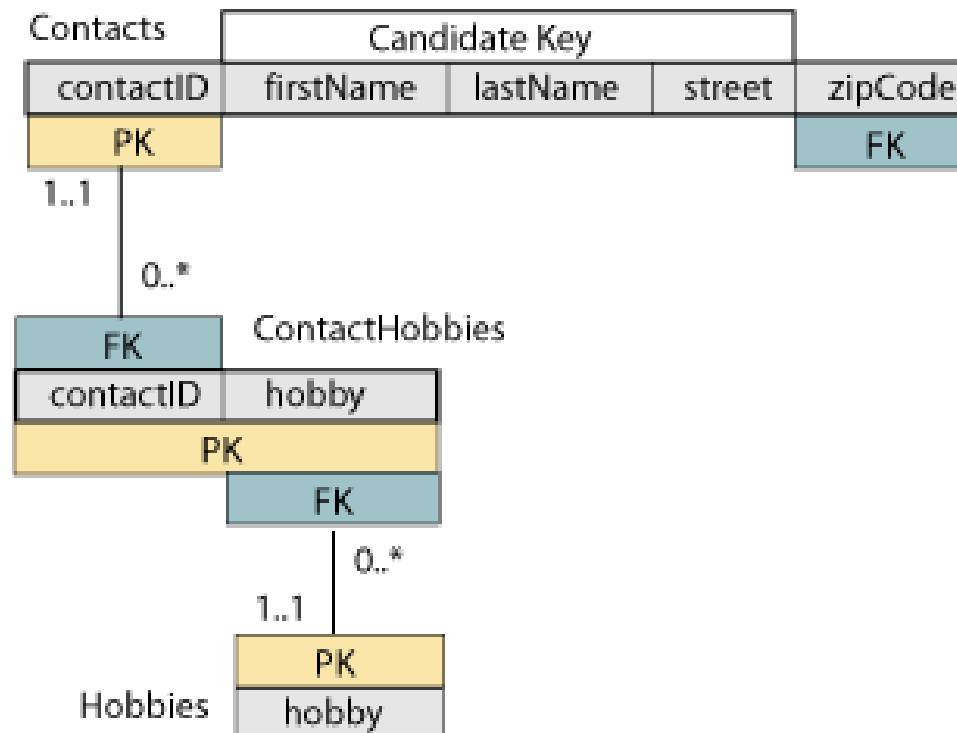
Multi-valued attributes might also have enumerated domains.



Design Pattern: Enumerated Domains

– Many-to-Many Relation Scheme

In the scheme, the relationship between Contacts and Hobbies has become many-to-many, instead of one-to-many. This is shown in the scheme by linking an enumeration table to the previous Hobbies tables (which now functions like an association class).



Exercise: a Pizza Shop

- You are designing a database for a pizza shop that wants to get into Web-based sales. Your client has given you the transcript of a typical phone order conversation:

Pizza shop associate (Lori): Thank you for calling the Pizza Shop; this is Lori. How may I help you?

Caller (Rick): What toppings do you put on your all-meat special?

Lori: Italian sausage, pepperoni, ground beef, salami, and bacon.

Rick: OK, I'd like a large one, but without the bacon.

Lori: Do you want regular crust, extra-thin, or whole wheat?

Rick: Regular is fine. And a medium wheat crust with just cheese.

Lori: We have mozzarella, parmesan, romano, smoked cheddar, and jalapeno jack.

Rick: Uhhh just mozzarella and romano. What kind of sauce is on that?

Lori: Marinara, spicy southwestern, tandoori masala, or pesto your choice.

Rick: Pesto sounds good.

Exercise: a Pizza Shop

Lori: You can add a large order of breadsticks for just 99 cents.

Rick: Sure, why not? And I'd like three small salads (muffled) make two of'em with Italian dressing and one with ranch.

Lori: The dressing comes on the side; we'll give you an extra one of each flavor. What would you like to drink?

Rick: Keg'a beer, maybe?

Lori: Sorry, we just have soft drinks.

Rick: (laughs) Just kidding how about two medium diet colas and a large iced tea.

Lori: That's one large regular crust all-meat special, no bacon, one medium wheat crust with pesto sauce, mozzarella and romano, one large order of breadsticks, three small salads, three Italian dressing, two ranch, two medium diet colas and one large iced tea. Just a minute, please (cash register clicks several times) your total with tax is 27 dollars and 39 cents. Is this for pickup or delivery?

Exercise: a Pizza Shop

Rick: I'll pick it up.

Lori: And your name?

Rick: Rick.

Lori: Thank you for your order, Rick. It'll be ready in about 20 minutes.

Rick: See'ya then.

- Develop a list of the enumerated attributes that are discussed in this conversation.

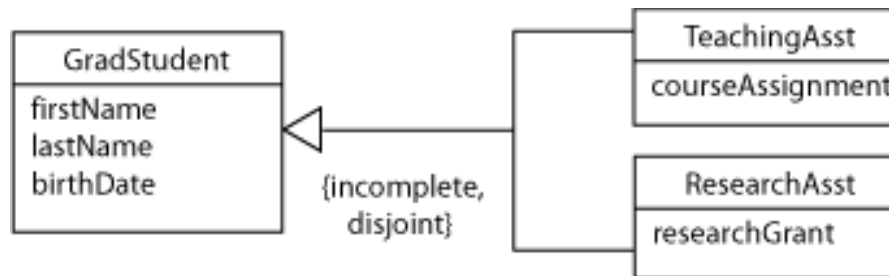
Design Pattern: Subclasses –Top Down Design

- As you are developing a class diagram, you might discover that one or more attributes of a class are characteristics of only *some* individuals of that class, but not of others. This probably indicates that you need to develop a **subclass** of the basic class type. We call the process of designing subclasses from “top down” **specialization**. A class that represents a subset of another class type can also be called a specialization of its parent class.
- Example: we will model the graduate students at a university. Some are employed by the university as teaching associates (TAs). Some are employed as research associates (RAs). Some are not employed by the university at all. For the TAs, we need to know which course they are assigned to teach. For the RAs, we need to know the grant number of the research project to which they are assigned.

Design Pattern: Subclasses –Top Down Design – Class Diagrams



- Note that one of the attributes `courseAssignment` or `researchGrant` will always be null



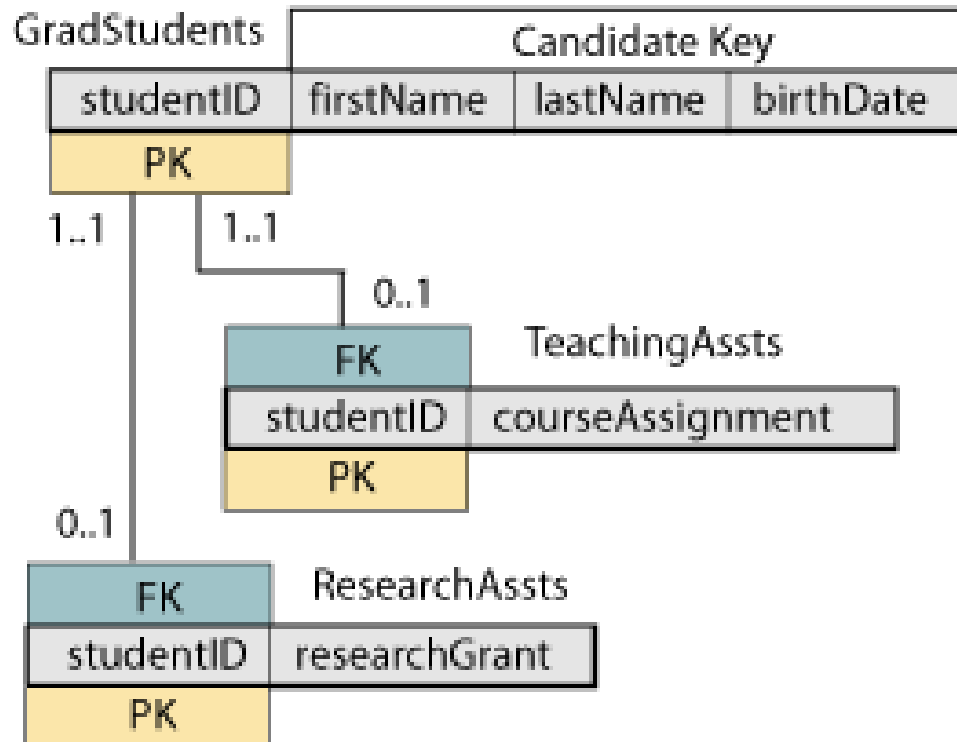
- Common attributes are in the parent class, unique attributes are in the subclass

Design Pattern: Subclasses –Top Down Design–Specialization Constraints

- Rather than with the usual cardinality symbols, the subclass association line is label with ***specialization constraints***. Constraints are described along two dimensions: ***incomplete*** vs. ***complete***, and ***disjoint*** vs. ***overlapping***.
 - ***incomplete*** or ***partial specialization***, only some individuals of the parent class are specialized.
 - ***complete specialization***, all individuals of the parent class have one or more unique attributes that are not common to the generalized (parent) class.
 - ***disjoint*** or ***exclusive*** specialization, an individual of the parent class may be a member of only one specialized subclass.
 - ***overlapping*** specialization, an individual of the parent class may be a member of more than one specialized subclass.

Design Pattern: Subclasses – Top Down Design – Relation Scheme

Note that the PK of the parent is the PK/FK of the child as this is a one-to-one relationship.

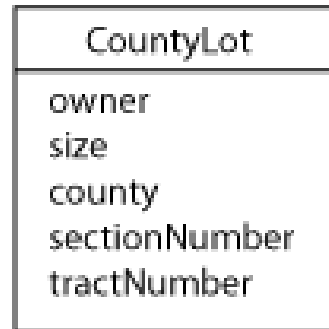


Design Pattern: Subclasses - Bottom Up Design

- Sometimes, instead of finding unique attributes in a single class type, you might find two or more classes that have many of the *same* attributes. This probably indicates that you need to develop a ***superclass*** of the classes with common attributes.
- We call the process of designing subclasses from “bottom up” ***generalization***. A class or entity that represents a superset of other class types can also be called a ***generalization*** of the child types.
- Example: Consider a brush-clearing service. This is a fairly specialized business, where dried plant growth (brush) can present a severe fire hazard if it is not cleared from around houses and other structures.

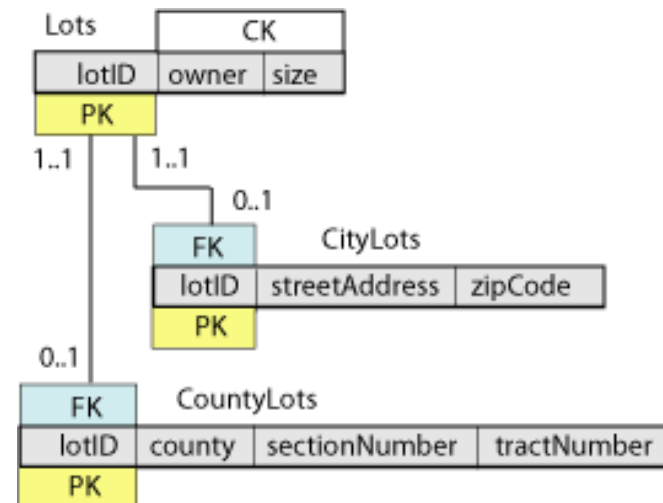
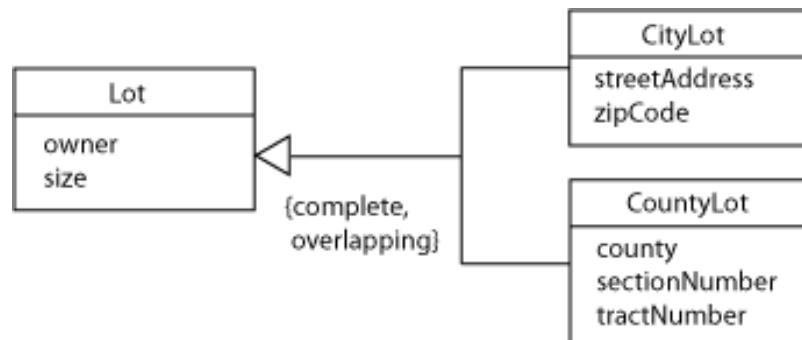
Design Pattern: Subclasses – Bottom Up Design – Class Diagram

- ❑ One important class type was the lot (or property) to be cleared. Some lots were in the city, with a standard street-and-number address.
- ❑ Other lots were not on a city street, but were described by the county surveyor's section and tract number.
- ❑ It seemed as if there were two class types:



Design Pattern: Subclasses – Bottom Up Design – Relation Scheme

Actually, a few of the lots were identified by both address schemes. A closer look at the city and county lot classes also shows two common descriptive attributes (the owner and the lot size). The common attributes should go in a ***generalization*** or ***superclass*** that is simply called a “lot.”

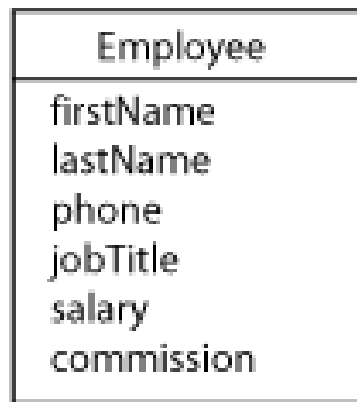


Exercise: Sales Commission

- ❑ The training material for a well-known database system includes an example employee class with standard attributes such as employee ID, name, and so on.
- ❑ The same entity also includes both a salary attribute and a commission attribute, even though only sales representatives earn a commission. (Everyone earns a salary.)
- ❑ In their lesson plans, this vendor emphasizes various ways that their software can handle the inevitable null values of the commission attribute.

Exercise: Sales Commission

- ❑ I believe that their model is simply wrong.



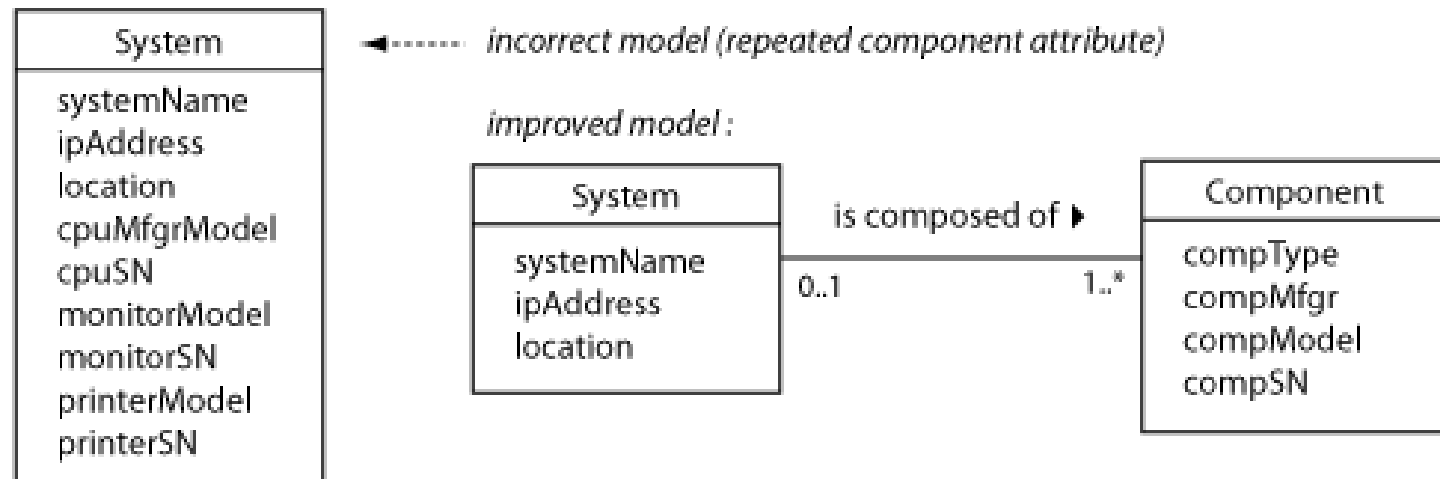
- ❑ Design a better way to preserve the salary and commission information but preclude null values in any of the attributes.
- ❑ Draw the class diagram and the relation scheme.

Design Pattern: Aggregation

- An ***aggregation*** is when a class type really represents a collection of individual components. Although this pattern can be modeled by an ordinary association, its meaning becomes much clearer if we use the UML notation for an ***aggregation***.
- Example: a small business needs to keep track of its computer systems. They want to record information such as model and serial number for each system and its components.

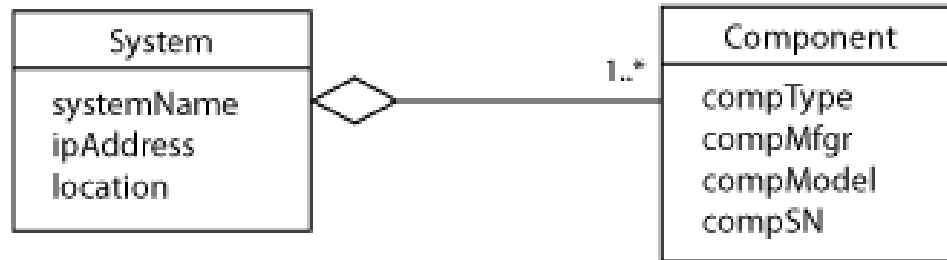
Design Pattern: Aggregation – Incorrect Model vs. Improved Model

Note that the incorrect model uses repeated attributes for the components. This is fixed in the improved model although there are still some problems – what if there are components on the shelf that don't belong to a system



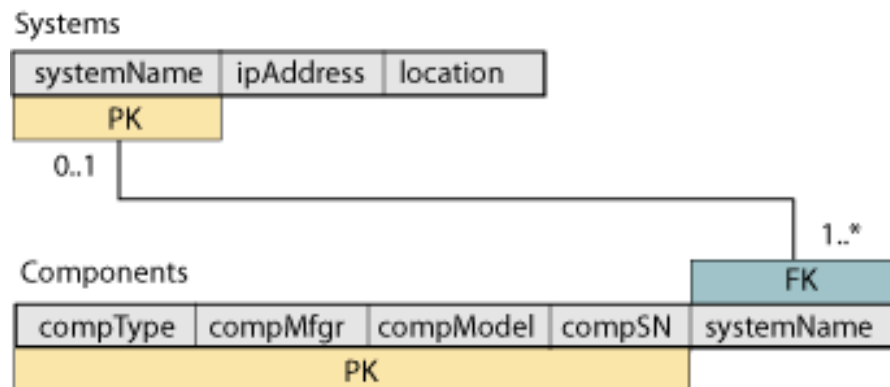
Design Pattern: Aggregation – Model with UML Aggregation

- ❑ The system is an aggregation of components.
- ❑ There is an implied multiplicity on the diamond end of 0..1 with multiplicity of the other end shown in the diagram.



Design Pattern: Aggregation – Relation Scheme

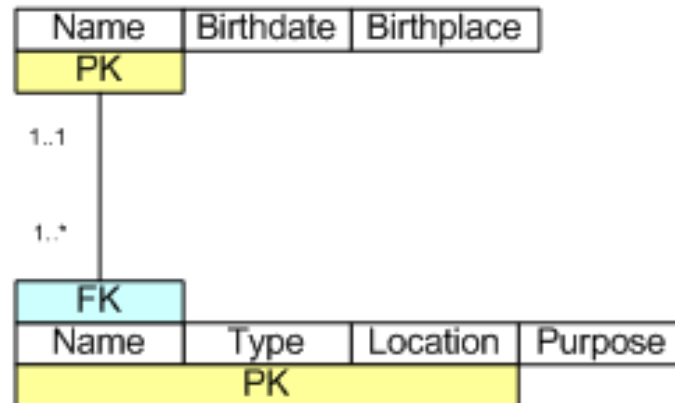
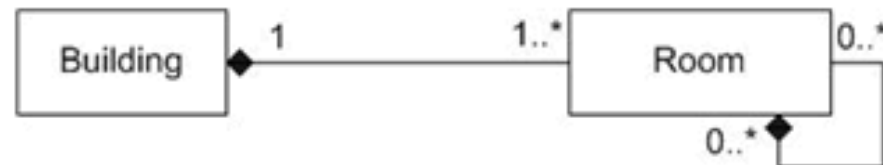
- Since the component can exist by itself in this model the system name can't be part of its PK.
- We use the only candidate key {type, mgr, model, SN} as the PK since this class is not a parent. The system name, will be filled in if the component is installed as part of a system, otherwise it will be null.



Design Pattern: Aggregation – Composition

- ❑ A **composition** is a stronger form of aggregation. The notation is similar using a filled-in diamond instead of an open one. In a composition, a component instance cannot exist on its own without a parent.
- ❑ The implied multiplicity on the diamond end is 1..1.

Design Pattern: Aggregation – Composition Examples



Design Pattern: Aggregation – Definitions

- *Aggregation* – (“has a” relationship)

In an aggregation relationship, the part may be independent of the whole but the whole requires the part.

- *Composition* – (“uses a” relationship)

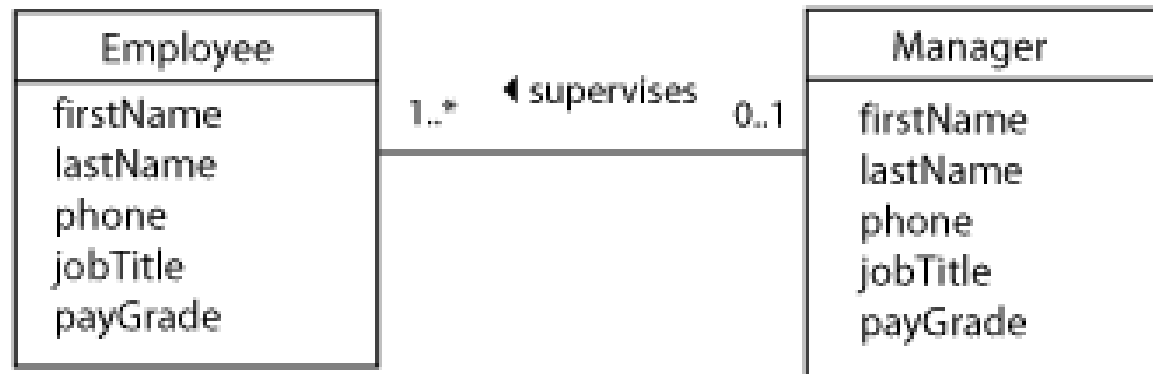
A composition relationship, also known as a composite aggregation, is a stronger form of aggregation where the part is created and destroyed with the whole.

Design Pattern: Recursive Associations

- A ***recursive association*** connects a single class type (serving in one role) to itself (serving in another role).
- Example: employees and their managers

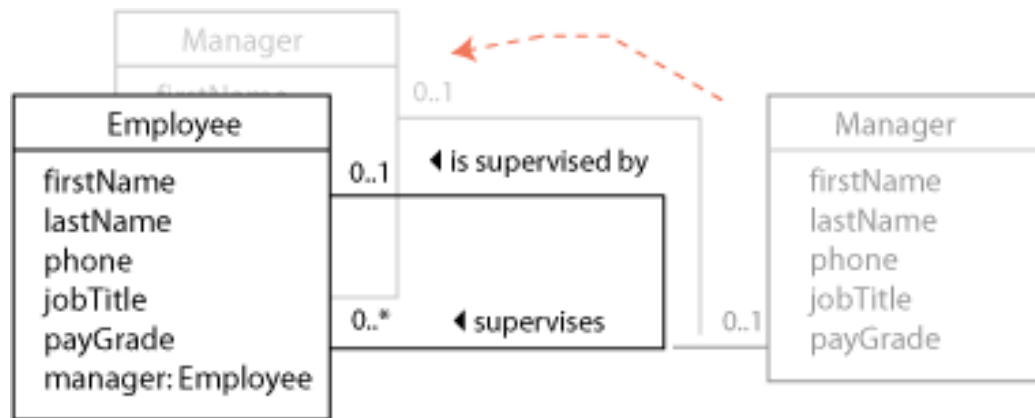
Design Pattern: Recursive Associations – Incorrect Relationship

The problem with this model is that each manager is also an employee so the manager table duplicates information in the employee table.



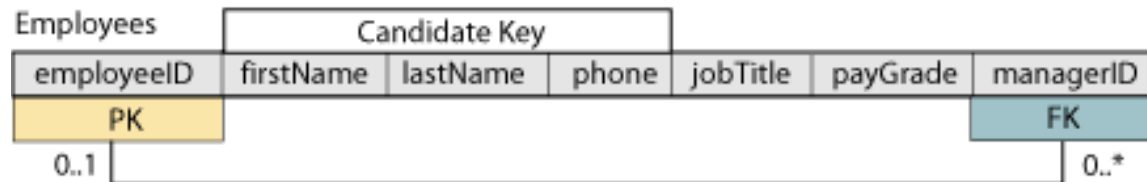
Design Pattern: Recursive Associations – Correct Relationship

Normally, we wouldn't show an FK in the class diagram; however, including the manager as an attribute of the employee here (in addition to the association line) can help in understanding the model.



Design Pattern: Recursive Associations – Relation Scheme

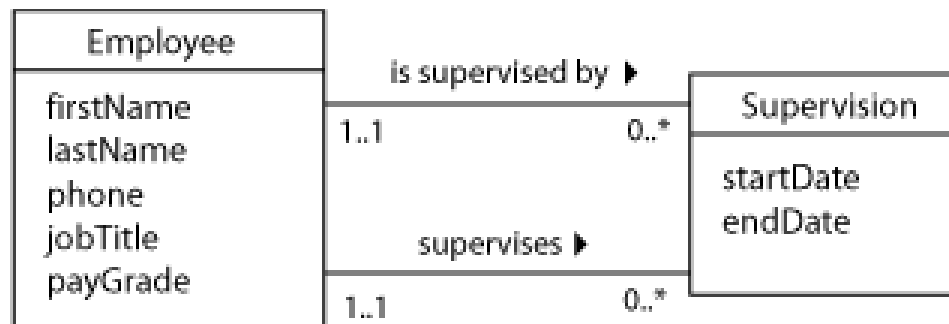
In the relation scheme, we can explicitly show the connection between the surrogate PK (employeeID) and the managerID (which is an FK, even though it is in the same scheme).



Design Pattern: Recursive Associations

– Many-to-Many Class Diagram

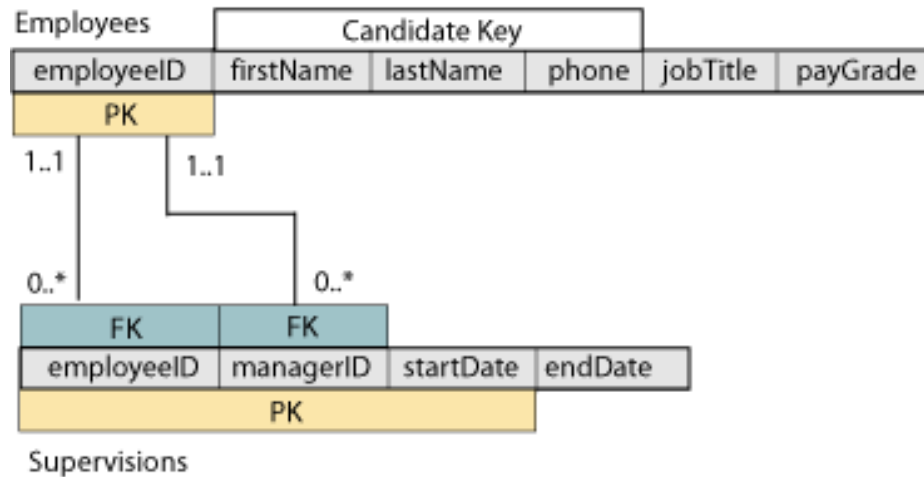
- ❑ In some project-oriented companies, an employee might work for more than one manager at a time.
- ❑ We also might want to keep a history of the employees' supervision assignments over time. We can model either case by revising the class diagram to a many-to-many pattern.



Design Pattern: Recursive Associations

– Many-to-Many Relation Scheme

The relation scheme for this model looks exactly like other many-to-many applications, with the exception that both foreign keys come from the same PK table.



Design Pattern: Recursive Associations – Retrieving Data

```
SELECT E.lastName AS "Employee",  
       M.lastName AS "Manager"  
FROM Employees E LEFT OUTER JOIN  
       Employees M  
ON E.managerID = M.employeeID  
ORDER BY E.lastName
```

Manager (supervised by someone else)

53	Steven	Buchanan	5-1599	Manager	exempt 5	22
PK						

Employee (supervised by this manager)

						FK
82	Robert	King	5-1221	Clerk	admin 3	53

Design Pattern: Recursive Associations

– Retrieving Data Many-to-Many

```
SELECT E.lastName AS "Employee",  
       M.lastName AS "Manager"  
FROM Employees E LEFT OUTER JOIN  
    (Supervisions S INNER JOIN  
     Employees M ON S.managerID =  
     M.employeeID)  
ON E.employeeID = S.employeeID  
ORDER BY E.lastName
```

Exercise: Team Games

- ❑ You are modeling a sports league. It could be any team sport.
- ❑ Certainly one important class is the team, which will have a name (unique within the league), a coach, a home field or venue, and probably more attributes.
- ❑ Obviously, teams play games. In each game, there is a designated home team and away team (even if the game is played at a neutral venue).
- ❑ Teams may play each other more than once during a season, possibly even with the same home and away roles (for example, during playoffs).
- ❑ Design a class diagram that shows the teams, the games that they play, and the score of each game.
- ❑ Then draw the relation scheme for your model.

Appendix: Traditional Normalization

- Normalization is usually thought of as a process of applying a set of rules to your database design, mostly to achieve minimum redundancy in the data. Most textbooks present this as a three-step process, with correspondingly label “normal forms.
- In theory, you could start with a single relation scheme (sometimes called the universal scheme, or U) that contains all of the attributes in the database—then apply these rules recursively to develop a set of increasingly-normalized sub-relation schemes. When all of the schemes are in third normal form, then the whole database is properly normalized.
- In practice, you will more likely apply the rules gradually, refining each relation scheme as you develop it from the UML class diagram or ER model diagram. The final table structures should be the same no matter which method (or combination of methods) you’ve used.

Appendix: Traditional Normalization

– Normal Forms

Normal forms

Normal form	Traditional definition	As presented here
First normal form (1NF)	<ul style="list-style-type: none">▪ All attributes must be atomic, and▪ No repeating groups	<ul style="list-style-type: none">▪ Eliminate <u>multi-valued attributes</u>, and▪ Eliminate <u>repeated attributes</u>
Second normal form (2NF)	<ul style="list-style-type: none">▪ First normal form, and▪ No partial functional dependencies	<ul style="list-style-type: none">▪ Eliminate <u>subkeys</u> (where the subkey is part of a composite primary key)
Third normal form (3NF)	<ul style="list-style-type: none">▪ Second normal form, and▪ No transitive functional dependencies	<ul style="list-style-type: none">▪ Eliminate <u>subkeys</u> (where the subkey is not part of the primary key)

Functional Dependency-FD

- A **functional dependency** (FD) is a type of relationship between attributes
- If A and B are sets of attributes of relation R, we say B is functionally dependent on A if each A value in R has associated with it exactly one value of B in R.
- Alternatively, if two tuples have the same A values, they must also have the same B values
- Write **$A \rightarrow B$** , read **A functionally determines B**, or B functionally dependent on A
- FD is actually a many-to-one relationship between A and B

Example of FDs

stuid	lastName	major	credits	status	socSecNo
S1001	Smith	History	90	Senior	100429500
S1003	Jones	Math	95	Senior	010124567
S1006	Lee	CSC	15	Freshman	088520876
S1010	Burns	Art	63	Junior	099320985
S1060	Jones	CSC	25	Freshman	064624738

□ Let R be

NewStudent(stuid, lastName, major, credits, status, socSecNo)

□ FDs in R include

$\{\text{stuid}\} \rightarrow \{\text{lastName}\}$, but not the reverse

$\{\text{stuid}\} \rightarrow \{\text{lastName}, \text{major}, \text{credits}, \text{status}, \text{socSecNo}, \text{stuid}\}$

$\{\text{socSecNo}\} \rightarrow \{\text{stuid}, \text{lastName}, \text{major}, \text{credits}, \text{status}, \text{socSecNo}\}$

$\{\text{credits}\} \rightarrow \{\text{status}\}$, but not $\{\text{status}\} \rightarrow \{\text{credits}\}$

Trivial Functional Dependency

- The FD $X \rightarrow Y$ is **trivial** if set $\{Y\}$ is a subset of set $\{X\}$

Examples: If A and B are attributes of R,

$\{A\} \rightarrow \{A\}$

$\{A, B\} \rightarrow \{A\}$

$\{A, B\} \rightarrow \{B\}$

$\{A, B\} \rightarrow \{A, B\}$

are all trivial FDs

First Normal Form-1NF

- A relation is in **1NF** if and only if every attribute is single-valued for each tuple
- Each cell of the table has only one value in it
- Domains of attributes are **atomic**: no sets, lists, repeating fields or groups allowed in domains

Counter-Example for 1NF

stuid	lastName	major	credits	status	socSecNo
S1001	Smith	History	90	Senior	100429500
S1003	Jones	Math	95	Senior	010124567
S1006	Lee	Math	15	Freshman	088520876
		CSC			
S1010	Burns	English	63	Junior	099320985
		Art			
S1060	Jones	CSC	25	Freshman	064624738

NewStu(stuid, lastName, major, credits, status, socSecNo) – Assume students can have more than one major

The *major* attribute is not single-valued for each tuple

Ensuring 1NF

- Best solution: For each multi-valued attribute, create a new table, in which you place the **key of the original table** and the **multi-valued attribute**. Keep the original table, with its key

Ex. NewStu2(stuId, lastName, credits, status, socSecNo)

Majors(stuId, major)

NewStu2

StuId	lastName	credits	status	socSecNo
S1001	Smith	90	Senior	100429500
S1003	Jones	95	Senior	010124567
S1006	Lee	15	Freshman	088520876
S1010	Burns	63	Junior	099320985
S1060	Jones	25	Freshman	064624738

Majors

<u>stuId</u>	<u>major</u>
S1001	History
S1003	Math
S1006	CSC
S1006	Math
S1010	Art
S1010	English
S1060	CSC

Full Functional Dependency

- In relation R, set of attributes B is **fully functionally dependent** on set of attributes A of R if B is functionally dependent on A but not functionally dependent on any proper subset of A
- This means every attribute in A is needed to functionally determine B
- Must have multivalued key for relation to have FD problem

Partial Functional Dependency Example

NewClass(courseNo, stuld, stuLastName, facld, schedule, room, grade)

FDs:

$\{\text{courseNo}, \text{stuld}\} \rightarrow \{\text{lastName}\}$

$\{\text{courseNo}, \text{stuld}\} \rightarrow \{\text{facld}\}$

$\{\text{courseNo}, \text{stuld}\} \rightarrow \{\text{schedule}\}$

$\{\text{courseNo}, \text{stuld}\} \rightarrow \{\text{room}\}$

$\{\text{courseNo}, \text{stuld}\} \rightarrow \{\text{grade}\}$

$\text{courseNo} \rightarrow \text{facld}$ **partial FD

$\text{courseNo} \rightarrow \text{schedule}$ **partial FD

$\text{courseNo} \rightarrow \text{room}$ ** partial FD

$\text{stuld} \rightarrow \text{lastName}$ ** partial FD

...plus trivial FDs that are partial...

Second Normal Form-2NF

- A relation is in **second normal form** (2NF) if it is in first normal form and all the non-key attributes are **fully** functionally dependent on the key.
- No non-key attribute is FD on just part of the key
- If key has only one attribute, and R is 1NF, R is automatically 2NF

Converting to 2NF

- ❑ Identify each partial FD
- ❑ Remove the attributes that depend on each of the determinants so identified
- ❑ Place these determinants in separate relations along with their dependent attributes
- ❑ In original relation keep the composite key and any attributes that are fully functionally dependent on all of it
- ❑ Even if the composite key has no dependent attributes, keep that relation to connect logically the others

2NF Example

NewClass(courseNo, stuld, stuLastName, facId, schedule, room, grade)

FDs grouped by determinant:

{courseNo} → {courseNo, facId, schedule, room}

{stuld} → {stuld, lastName}

{courseNo, stuld} → {courseNo, stuld, facId, schedule, room, lastName, grade}

Create tables grouped by determinants:

Class2(courseNo, facId, schedule, room)

Stu(stuld, lastName)

Keep relation with original composite key, with attributes FD on it, if any

Register(courseNo, stuld, grade)

2NF Example

<u>courseNo</u>	<u>stuld</u>	stuLastName	facld	schedule	room	grade
ART103A	S1001	Smith	F101	MWF9	H221	A
ART103A	S1010	Burns	F101	MWF9	H221	
ART103A	S1006	Lee	F101	MWF9	H221	B
CSC201A	S1003	Jones	F105	TUTHF10	M110	A
CSC201A	S1006	Lee	F105	TUTHF10	M110	C
HST205A	S1001	Smith	F202	MWF11	H221	

First Normal Form Relation

Register		
<u>courseNo</u>	<u>stuld</u>	grade
ART103A	S1001	A
ART103A	S1010	
ART103A	S1006	B
CSC201A	S1003	A
CSC201A	S1006	C
HST205A	S1001	

Stu	
<u>stuld</u>	stuLastName
S1001	Smith
S1010	Burns
S1006	Lee
S1003	Jones

Class2			
<u>courseNo</u>	facld	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TUTHF10	M110
HST205A	F202	MWF11	H221

Second Normal Form Relations

Transitive Dependency

- If A, B, and C are attributes of relation R, such that $A \rightarrow B$, and $B \rightarrow C$, then C is **transitively dependent** on A

Example:

NewStudent (stuld, lastName, major, credits, status)

FD:

credits \rightarrow status (and several others)

By transitivity:

stuld \rightarrow credits \wedge credits \rightarrow status implies stuld \rightarrow status

Transitive dependencies cause update, insertion, deletion anomalies.

Third Normal Form-3NF

- A relation is in **third normal form (3NF)** if whenever a non-trivial functional dependency $X \rightarrow A$ exists, then either X is a superkey or A is a member of some candidate key
- To be 3NF, relation must be 2NF and have no transitive dependencies
- No non-key attribute determines another non-key attribute. Here key includes “candidate key”

Example Transitive Dependency

NewStudent

Stuid	lastName	Major	Credits	Status
S1001	Smith	History	90	Senior
S1003	Jones	Math	95	Senior
S1006	Lee	CSC	15	Freshman
S1010	Burns	Art	63	Junior
S1060	Jones	CSC	25	Freshman

Transitive Dependency: Stuid \rightarrow Credits and Credits \rightarrow Status

NewStu2

Stuid	lastName	Major	Credits
S1001	Smith	History	90
S1003	Jones	Math	95
S1006	Lee	CSC	15
S1010	Burns	Art	63
S1060	Jones	CSC	25

Stats

Credits	Status
15	Freshman
25	Freshman
63	Junior
90	Senior
95	Senior

Removed Transitive Dependency

Making a relation 3NF

- For example,
NewStudent (stuld, lastName, major, credits, status)
with FD $\text{credits} \rightarrow \text{status}$
- Remove the dependent attribute, *status*, from the relation
- Create a new table with the dependent attribute and its determinant, *credits*
- Keep the determinant in the original table

NewStu2 (stuld, lastName, major, credits)
Stats (credits, status)

Appendix: Traditional Normalization

– Denormalization

- ❑ You database will always be part of a larger system, which will include at least a user interface and reporting structure or the back-end of a Web site, with both middle-tier business logic and front-end presentation code dependent on it.
- ❑ It is not uncommon for developers to “break the rules” of database design in order to accommodate other parts of a system. This is called ***denormalization***.

Appendix: Traditional Normalization

– Denormalization for Efficiency

- ❑ An example of **denormalization**, using our “phone book” problem, would be to store the city and state attributes in the basic contacts table, rather than making a separate zip codes table.
- ❑ At the cost of extra storage, this would save one join in a SELECT statement. Although this would certainly not be needed in such a simple system, imagine a Web site that supports thousands of “hits” per second, with much more complicated queries needed to produce the output.
- ❑ With today’s terabyte disk systems, it might be worth using extra storage space to keep Web viewers from waiting excessively while a page is being generated.

Boyce-Codd Normal Form-BCNF

- A relation is in Boyce/Codd Normal Form (BCNF) if whenever a non-trivial functional dependency $X \rightarrow A$ exists, then X is a superkey (trivial means that A is a subset of X)
- Stricter than 3NF, which allows A to be part of a candidate key
- If there is just one single candidate key, the forms are equivalent

BCNF Example

NewFac (facName, dept, office, rank, dateHired)

FDs:

office \rightarrow dept

facName,dept \rightarrow office, rank, dateHired

facName,office \rightarrow dept, rank, dateHired

- ❑ NewFac is 3NF but not BCNF because office is not a superkey
- ❑ To make it BCNF, remove the dependent attributes to a new relation, with the determinant as the key
- ❑ Decompose into

Fac1 (office, dept)

Fac2 (facName, office, rank, dateHired)

Note we have lost a functional dependency in Fac2 – no longer able to see that {facName, dept} is a determinant, since they are in different relations

Example Boyce-Codd Normal Form

Faculty

facName	dept	office	rank	dateHired
Adams	Art	A101	Professor	1975
Byrne	Math	M201	Assistant	2000
Davis	Art	A101	Associate	1992
Gordon	Math	M201	Professor	1982
Hughes	Mth	M203	Associate	1990
Smith	CSC	C101	Professor	1980
Smith	History	H102	Associate	1990
Tanaka	CSC	C101	Instructor	2001
Vaughn	CSC	C101	Associate	1995

Fac1

office	dept
A101	Art
C101	CSC
C105	CSC
H102	History
M201	Math
M203	Math

Fac2

facName	office	rank	dateHired
Adams	A101	Professor	1975
Byrne	M201	Assistant	2000
Davis	A101	Associate	1992
Gordon	M201	Professor	1982
Hughes	M203	Associate	1990
Smith	C101	Professor	1980
Smith	H102	Associate	1990
Tanaka	C101	Instructor	2001
Vaughn	C101	Associate	1995

Converting to BCNF

- Identify all determinants and verify that they are superkeys in the relation
- If not, break up the relation by decomposition
 - for each non-superkey determinant, create a separate relation with all the attributes it determines, also keeping it in original relation
 - Preserve the ability to recreate the original relation by joins.
- Repeat on each relation until you have a set of relations all in BCNF

Normalization Example

- Relation that stores information about projects in large business
 - Work (projName, projMgr, empld, hours, empName, budget, startDate, salary, empMgr, empDept, rating)

prijName	projMgr	empld	hours	Emp Name	budget	startDate	salary	Emp Mgr	Emp Dept	rating
Jupiter	Smith	E101	25	Jones	100000	01/15/04	60000	Levine	10	9
Jupiter	Smith	E105	40	Adams	100000	01/15/04	55000	Jones	12	
Jupiter	Smith	E110	10	Rivera	100000	01/15/04	43000	Levine	10	8
Maxima	Lee	E101	15	Jones	200000	03/01/04	60000	Levine	10	
Maxima	Lee	E110	30	Rivera	200000	03/01/04	43000	Levine	10	
Maxima	Lee	E120	15	Tanaka	200000	03/01/04	45000	Jones	15	

Normalization Example (cont)

1. Each project has a unique name.
2. Although project names are unique, names of employees and managers are not.
3. Each project has one manager, whose name is stored in `projMgr`.
4. Many employees can be assigned to work on each project, and an employee can be assigned to more than one project. The attribute `hours` tells the number of hours per week a particular employee is assigned to work on a particular project.
5. `budget` stores the amount budgeted for a project, and `startDate` gives the starting date for a project.
6. `salary` gives the annual compensation of an employee.

Normalization Example (cont)

7. `empMgr` gives the name of the employee's manager, who might not be the same as the project manager.
8. `empDept` gives the employee's department. Department names are unique. The employee's manager is the manager of the employee's department.
9. `rating` gives the employee's performance for a particular project. The project manager assigns the rating at the end of the employee's work on the project.

Normalization Example (cont)

□ Functional dependencies

- $\text{projName} \rightarrow \text{projMgr}, \text{budget}, \text{startDate}$
- $\text{empId} \rightarrow \text{empName}, \text{salary}, \text{empMgr}, \text{empDept}$
- $\text{projName}, \text{empId} \rightarrow \text{hours}, \text{rating}$
- $\text{empDept} \rightarrow \text{empMgr}$
- empMgr does not functionally determine empDept since people's names were not unique (different managers may have same name and manage different departments or a manager may manage more than one department)
- projMgr does not determine projName

□ Primary Key

- $\text{projName}, \text{empId}$ since every member depends on that combination

Normalization Example (cont)

□ First Normal Form

- With the primary key each cell is single valued,
Work in 1NF

□ Second Normal Form

■ Partial dependencies

- projName → projMgr, budget, startDate
- empld → empName, salary, empMgr, empDept

■ Transform to

- Proj (projName, projMgr, budget, startDate)
- Emp (empld, empName, salary, empMgr, empDept)
- Work1 (projName, empld, hours, rating)

Normalization Example (cont)

Second Normal Form

Proj

prijName	projMgr	budget	startDate
Jupiter	Smith	100000	01/15/04
Maxima	Lee	200000	03/01/04

Work1

prijName	empld	hours	rating
Jupiter	E101	25	9
Jupiter	E105	40	
Jupiter	E110	10	8
Maxima	E101	15	
Maxima	E110	30	
Maxima	E120	15	

Emp

empld	empName	salary	empMgr	empDept
E101	Jones	60000	Levine	10
E105	Adams	55000	Jones	12
E110	Rivera	43000	Levine	10
E101	Jones	60000	Levine	10
E110	Rivera	43000	Levine	10
E120	Tanaka	45000	Jones	15

Normalization Example (cont)

□ Third Normal Form

- Proj in 3NF – no non-key attribute functionally determines another non-key attribute
- Work1 in 3NF – no transitive dependency involving hours or rating
- Emp not in 3NF – transitive dependency
 - $\text{empDept} \rightarrow \text{empMgr}$ and empDept is not a superkey, nor is empMgr part of a candidate key
 - Need two relations
 - Emp1 (empId, empName, salary, empDept)
 - Dep (empDept, empMgr)

Normalization Example (cont)

Third Normal Form

Emp1

empId	empName	salary	empDept
E101	Jones	60000	10
E105	Adams	55000	12
E110	Rivera	43000	10
E120	Tanaka	45000	15

Dept

empDept	empMgr
10	Levine
12	Jones
15	Jones

Proj

prijName	projMgr	budget	startDate
Jupiter	Smith	100000	01/15/04
Maxima	Lee	200000	03/01/04

Work1

prijName	empId	hours	rating
Jupiter	E101	25	9
Jupiter	E105	40	
Jupiter	E110	10	8
Maxima	E101	15	
Maxima	E110	30	
Maxima	E120	15	

This is also BCNF since the only determinant in each relation is the primary key

Decomposition

- **Definition:** A **decomposition** of a relation R is a set of relations $\{R_1, R_2, \dots, R_n\}$ such that each R_i is a subset of R and the union of all of the R_i is R .
- Starting with a universal relation that contains all the attributes of a schema, we can decompose into relations

Desirable Properties of Decompositions

- **Attribute preservation** - every attribute is in some relation
- **Dependency preservation** – all FDs are preserved
- **Lossless decomposition** – can get back the original relation by joins

Dependency Preservation

- If R is decomposed into $\{R_1, R_2, \dots, R_n\}$ so that for each functional dependency $X \rightarrow Y$ all the attributes in $X \cup Y$ appear in the same relation, R_i , then all FDs are preserved
- Allows DBMS to check each FD constraint by checking just one table for each

Example

Sometimes more important to maintain functional dependencies than it is to get the relation in BCNF

NewFac (facName, dept, office, rank, dateHired)

FDs:

office \rightarrow dept

facName, dept \rightarrow office, rank, dateHired

facName, office \rightarrow dept, rank, dateHired

- ❑ NewFac is not BCNF because office is not a superkey
- ❑ To make it BCNF, remove the dependent attributes to a new relation, with the determinant as the key
- ❑ Project into

Fac1 (office, dept)

Fac2 (facName, office, rank, dateHired)

Note we have lost a functional dependency in Fac2 – no longer able to see that {facName, dept} is a determinant, since they are in different relations

Multi-valued Dependency

- In $R(A,B,C)$ if each A value has associated with it a set of B values and a set of C values such that the B and C values are independent of each other, then **A multi-determines B** and **A multi-determines C**
- Multi-valued dependencies occur in pairs
- Example: JointAppoint(facId, dept, committee) assuming a faculty member can belong to more than one department and belong to more than one committee
- Table must list all combinations of values of department and committee for each facId

Lossless Decomposition

- A decomposition of R into $\{R_1, R_2, \dots, R_n\}$ is **lossless** if the natural join of R_1, R_2, \dots, R_n produces exactly the relation R
- No **spurious tuples** are created when the projections are joined.
- always possible to find a BCNF decomposition that is lossless

Example of Lossy Decomposition

Original EmpRoleProj table: tells what role(s) each employee plays in which project(s)

<u>EmpName</u>	<u>role</u>	<u>projName</u>
Smith	designer	Nile
Smith	programmer	Amazon
Smith	designer	Amazon
Jones	designer	Amazon

Project into two tables **Table a**(empName, role), **Table b**(role, projname)

Table a

<u>EmpName</u>	<u>role</u>
Smith	designer
Smith	programmer
Jones	designer

Table b

<u>role</u>	<u>projName</u>
designer	Nile
programmer	Amazon
designer	Amazon

Joining Table a and Table b produces

<u>EmpName</u>	<u>role</u>	<u>projName</u>
Smith	designer	Nile
Smith	designer	Amazon
Smith	programmer	Amazon
Jones	designer	Nile
Jones	designer	Amazon

← spurious tuple

Lossless Decomposition

- ❑ Lossless property guaranteed if for each pair of relations that will be joined, the set of common attributes is a superkey of one of the relations
- ❑ Binary decomposition of R into $\{R_1, R_2\}$ lossless iff one of these holds

$$R_1 \cap R_2 \rightarrow R_1 - R_2$$

or

$$R_1 \cap R_2 \rightarrow R_2 - R_1$$

- ❑ If projection is done by successive binary projections, can apply binary decomposition test repeatedly

Algorithm to Test for Lossless Join

- Given a relation $R(A_1, A_2, \dots, A_n)$, a set of functional dependencies, F , and a decomposition of R into Relations R_1, R_2, \dots, R_m , to determine whether the decomposition has a lossless join
 - Construct an m by n table, S , with a column for each of the n attributes in R and a row for each of the m relations in the decomposition
 - For each cell $S(i, j)$ of S ,
 - If the attribute for the column, A_j , is in the relation for the row, R_i , then place the symbol $a(j)$ in the cell else place the symbol $b(i, j)$ there
 - Repeat the following process until no more changes can be made to S for each FD $X \rightarrow Y$ in F
 - For all rows in S that have the same symbols in the columns corresponding to the attributes of X , make the symbols for the columns that represent attributes of Y equal by the following rule:
 - If any row has an a value, $a(j)$, then set the value of that column in all the other rows equal to $a(j)$
 - If no row has an a value, then pick any one of the b values, say $b(i, j)$, and set all the other rows equal to $b(i, j)$
 - If, after all possible changes have been made to S , a row is made up entirely of a symbols, $a(1), a(2), \dots, a(n)$, then the join is lossless. If there is no such row, the join is lossy.

Normalization Methods

□ Analysis

- Decomposition method shown previously

□ Synthesis

- Begin with attributes, combine them into groups having the same determinant
- Use functional dependencies to develop a set of normalized relations

□ Mapping from ER diagram provides almost-normalized schema

De-normalization

- When to stop the normalization process
 - When applications require too many joins
 - When you cannot get a non-loss decomposition that preserves dependencies

Inference Rules for FDs

□ Armstrong's Axioms

- **Reflexivity** If B is a subset of A , then $A \rightarrow B$.
- **Augmentation** If $A \rightarrow B$, then $AC \rightarrow BC$.
- **Transitivity** If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$

Additional rules that follow:

- **Additivity** If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$
- **Projectivity** If $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$
- **Pseudotransitivity** If $A \rightarrow B$ and $CB \rightarrow D$, then $AC \rightarrow D$

Closure of Set of FDs

- If F is a set of functional dependencies for a relation R , then the set of all functional dependencies that can be derived from F , F^+ , is called the **closure of F**
- Could compute closure by applying Armstrong's Axioms repeatedly

Closure of an Attribute

- If A is an attribute or set of attributes of relation R , all the attributes in R that are functionally dependent on A in R form the **closure of A** , A^+
- Computed by Closure Algorithm for A
- $result \leftarrow A$;
 while (result changes) do
 for each functional dependency $B \rightarrow C$ in F
 if B is contained in result then $result \leftarrow result \cup C$;
 end;
 $A^+ \leftarrow result$;

Uses of Attribute Closure

- Can determine if A is a superkey-if every attribute in R functionally dependent on A
- Can determine whether a given FD $X \rightarrow Y$ is in the closure of the set of FDs. (Find X^+ , see if it includes Y)

Redundant FDs and Covers

- Given a set of FDs, can determine if any of them is **redundant**, i.e. can be derived from the remaining FDs, by a simple
- If a relation R has two sets of FDs, F and G
 - then F is a **cover** for G if every FD in G is also in F^+
 - F and G are equivalent if F is a cover for G and G is a cover for F (i.e. $F^+ = G^+$)

Minimal Set of FDs

- Set of FDs, F is **minimal** if
 - The right side of every FD in F has a single attribute (called standard or canonical form)
 - No attribute in the left side of any FD is extraneous
 - F has no redundant FDs

Minimal Cover for Set of FDs

- A minimal cover for a set of FDs is a cover such that no proper subset of itself is also a cover
- A set of FDs may have several minimal covers

Synthesis Algorithm for 3NF

- Can always find 3NF decomposition that is lossless **and that preserves all FDs**
- 3NF Algorithm uses synthesis
 - Begin with universal relation and set of FDs, G
 - Find a minimal cover for G
 - Combine FDs that have the same determinant
 - Include a relation with a key of R

Basic Queries: SQL and RA

- ❑ To look at the data in tables, we use the **SELECT** statement. The result of this statement is always a new table that we can view with our database client software or use with programming languages to build dynamic web pages or desktop applications.
- ❑ Although the result table is not stored in the database we can also use it as part of other **SELECT** statements.

Basic Queries: SQL and RA – Required Clauses

Only the **SELECT** and **FROM** clauses are required.

SELECT <attribute names>

FROM <table names>

WHERE <condition to pick rows>

ORDER BY <attribute names>;

Basic Queries: SQL and RA – Retrieving Data Step 1

1. Look at *all* of the relevant data—this is called the **result set** of the query, and it is specified in the **FROM** clause. We have only one table, so the result set should consist of all the columns (* means all attributes) and rows of this table.

```
SELECT * FROM customers;
```

Customers				
cfirstname	clastname	cphone	cstreet	czipcode
Tom	Jewett	714-555-1212	10200 Slater	92708
Alvaro	Monge	562-333-4141	2145 Main	90840
Wayne	Dick	562-777-3030	1250 Bellflower	90840

Basic Queries: SQL and RA – Retrieving Data Step 2

2. Pick the specific rows you want from the result set (for example here, all customers who live in zip code 90840). Notice the *single* quotes around the string you're looking for—search strings *are* case sensitive!

```
SELECT * FROM customers  
WHERE cZipCode = '90840';
```

Customers in zip code 90840

cfirstname	clastname	cphone	cstreet	czipcode
Alvaro	Monge	562-333-4141	2145 Main	90840
Wayne	Dick	562-777-3030	1250 Bellflower	90840

Basic Queries: SQL and RA – Retrieving Data Step 3

3. Pick the attributes (columns) you want. Notice that changing the order of the columns (like showing the last name first) does not change the meaning of the data.

```
SELECT cLastName, cFirstName, cPhone  
FROM customers  
WHERE cZipCode = '90840';
```

Columns from SELECT

cLastName	cFirstName	cPhone
Monge	Alvaro	562-333-4141
Dick	Wayne	562-777-3030

Basic Queries: SQL and RA – Retrieving Data Step 4

4. In SQL, you can also specify the order in which to list the results. Once again, the order in which rows are listed does not change the meaning of the data in them.

```
SELECT cLastName, cFirstName, cPhone  
FROM customers  
WHERE cZipCode = '90840'  
ORDER BY cLastName, cFirstName;
```

Rows in order

cLastName	cFirstName	cPhone
Dick	Wayne	562-777-3030
Monge	Alvaro	562-333-4141

Basic queries: SQL and RA – Why SQL Works

- Like all algebras, ***Relational Algebra (RA)*** applies operators to operands to produce results. ***RA*** operands are relations. Results are new relations that can be used as operands in building more complex expressions.
- *select(RA)* and *project(RA)* are two ***RA*** operators that were used in the prior example

Basic queries: SQL and RA – Select and Project Step 1

1. To represent a single relation in RA, we only need to use its name. We can also represent relations and schemes symbolically with small and capital letters, for example relation r over scheme R . In this case, $r = \text{customers}$ and $R = \text{the Customers scheme}$.

Basic queries: SQL and RA – Select and Project Step 2

2. The ***select* (RA)** operator (written σ) picks tuples, like the SQL WHERE clause picks rows. It is a unary operator that takes a single relation or expression as its operand. It also takes a **predicate**, θ , to specify which tuples are required. Its syntax is $\sigma\theta r$, or in our example: $\sigma_{cZipCode='90840'} customers$.

The scheme of the result of $\sigma\theta r$ is R —the same scheme we started with—since we haven't done anything to change the attribute list. The result of this operation includes all tuples of r for which the predicate θ evaluates to *true*.

Basic queries: SQL and RA – Select and Project Step 3

3. The **project (RA)** operator (written π) picks attributes, confusingly like the SQL SELECT clause. It is also a unary operator that takes a single relation or expression as its operand. Instead of a predicate, it takes a **subscheme**, X (of R), to specify which attributes are required. Its syntax is $\pi X r$, or in our example:

$\pi \text{cLastName, cFirstName, cPhone} \text{customers}.$

The scheme of the result of $\pi X r$ is X . The tuples resulting from this operation are tuples of the original relation, r , cut down to the attributes contained in X .

- For X to be a subscheme of R , it must be a subset of the attributes in R , and preserve the assignment rule from R (that is, each attribute of X must have the same domain as its corresponding attribute in R).
- If X is a super key of r , then there will be the same number of tuples in the result as there were to begin with in r . If X is *not* a super key of r , then any duplicate (non-distinct) tuples are eliminated from the result.

Just as in the SQL statement, we can apply the project operator to the output of the select operation to produce the results that we want: $\pi X \sigma \theta r$ or

$\pi \text{cLastName, cFirstName, cPhone} \sigma \text{ZipCode}='90840' \text{customers}.$

Basic queries: SQL and RA – Select and Project Step 4

4. Since **RA** considers relations strictly as *sets* of tuples, there is no way to specify the order of tuples in a result relation.

Basic SQL Statements: DDL and DML

- SQL statements are divided into two major categories:
 - ***data definition language (DDL)*** - used to build and modify the structure of your tables and other objects in the database
 - ***data manipulation language (DML)*** - used to work with the data in tables
- All of the information about objects in your schema is contained in a set of tables called the ***data dictionary***.

Basic SQL Statements: DDL and DML

– DDL- CREATE Statement

- ❑ The CREATE TABLE statement does exactly that:
`CREATE TABLE <table name> (
 <attribute name 1> <data type 1>,
 ...
 <attribute name n> <data type n>);`
- ❑ The **data types** that you will use most frequently are character strings, which might be called VARCHAR or CHAR for variable or fixed length strings; numeric types such as NUMBER or INTEGER, which will usually specify a precision; and DATE or related types.
- ❑ Data type syntax is variable from system to system; the only way to be sure is to consult the documentation for your own software.

Basic SQL Statements: DDL and DML

– DDL- ALTER TABLE Statement

- ❑ The ALTER TABLE statement may be used as you have seen to specify primary and foreign key constraints, as well as to make other modifications to the table structure. Key constraints may also be specified in the CREATE TABLE statement.

```
ALTER TABLE <table name>  
ADD CONSTRAINT <constraint name>  
PRIMARY KEY (<attribute list>);
```

- ❑ You get to specify the constraint name. Get used to following a convention of tablename_pk (for example, Customers_pk), so you can remember what you did later.
- ❑ The attribute list contains the one or more attributes that form this PK; if more than one, the names are separated by commas.

Basic SQL Statements: DDL and DML

– DDL- Foreign Key Constraint

- The FOREIGN KEY CONSTRAINT is a bit more complicated, since we have to specify both the FK attributes in this (child) table, and the PK attributes that they link to in the parent table.

```
ALTER TABLE <table name>
```

```
ADD CONSTRAINT <constraint name>
```

```
FOREIGN KEY (<attribute list>)
```

```
REFERENCES <parent table name> (<attribute list>);
```

- Name the constraint in the form childtable_parenttable_fk (for example, Orders_Customers_fk). If there is more than one attribute in the FK, all of them must be included (with commas between) in both the FK attribute list and the REFERENCES (parent table) attribute list.
- You need a separate foreign key definition for each relationship in which this table is the child.

Basic SQL Statements: DDL and DML

– DDL- DROP statement

- ❑ If you totally mess things up and want to start over, you can always get rid of any object you've created with a drop statement. The syntax is different for tables and constraints.

`DROP TABLE <table name>;`

`ALTER TABLE <table name>`

`DROP CONSTRAINT <constraint name>;`

- ❑ This is where consistent constraint naming comes in handy, so you can just remember the PK or FK name rather than remembering the syntax for looking up the names in another table.
- ❑ The DROP TABLE statement gets rid of its own PK constraint, but won't work until you separately drop any FK constraints (or child tables) that refer to this one. It also gets rid of all data that was contained in the table—and it doesn't even ask you if you really want to do this!

Basic SQL statements: DDL and DML

– DML

- ❑ When you are connected to most multi-user databases (whether in a client program or by a connection from a Web page script), you are in effect working with a private copy of your tables that can't be seen by anyone else until you are finished .
- ❑ The SELECT statement is considered to be part of DML even though it just retrieves data rather than modifying it.

Basic SQL Statements: DDL and DML

– DML – INSERT Statement

- ❑ The insert statement is used to add new rows to a table.
`INSERT INTO <table name>`
`VALUES (<value 1>, ... <value n>);`
- ❑ The comma-delimited list of values must match the table structure exactly in the number of attributes and the data type of each attribute.
 - Character type values are always enclosed in single quotes
 - Number values are never in quotes
 - Date values are often (but not always) in the format 'yyyy-mm-dd' (for example, '2006-11-30')
- ❑ You will need a separate INSERT statement for every row.

Basic SQL Statements: DDL and DML

– DML – UPDATE Statement

- The update statement is used to change values that are already in a table.

UPDATE <table name>

SET <attribute> = <expression>

WHERE <condition>;

- The update expression can be a constant, any computed value, or even the result of a SELECT statement that returns a single row and a single column.
- If the WHERE clause is omitted, then the specified attribute is set to the same value in every row of the table (which is usually not what you want to do).
- You can also set multiple attribute values at the same time with a comma-delimited list of *attribute=expression* pairs.

Basic SQL Statements: DDL and DML

– DML – DELETE Statement

- ❑ The DELETE statement deletes rows in a table.
DELETE FROM <table name>
WHERE <condition>;
- ❑ If the WHERE clause is omitted, then every row of the table is deleted!!!

Basic SQL Statements: DDL and DML

– DML – Transactions

- ❑ If you are using a large multi-user system, you may need to make your DML changes visible to the rest of the users of the database. Although this might be done automatically when you log out, you could also just type:
`COMMIT;`
- ❑ If you need to back out your changes and want to restore your private copy of the database to the way it was before you started or since the last `COMMIT` just type:
`ROLLBACK;`
- ❑ Although single-user systems don't support **COMMIT** and **ROLLBACK** statements, they are used in large systems to control *transactions*, which are sequences of changes to the database.

Basic SQL Statements: DDL and DML

– Privileges

- ❑ If you want anyone else to be able to view or manipulate the data in your tables, and if your system permits this, you will have to explicitly GRANT the appropriate privilege or privileges (SELECT, INSERT, UPDATE, or DELETE) to them. This has to be done for each table.
- ❑ The most common case where you would use grants is for tables that you want to make available to scripts running on a Web server, for example:
`GRANT select, insert ON customers TO webuser;`

Basic Query Operation: the Join

- In order to see data from two or more tables, the tables must be ***joined***.
- A ***join*** operation matches up the right information from each table.

```
SELECT * FROM customers  
NATURAL JOIN orders;
```

Customers joined to Orders

cfirstname	clastname	cphone	cstreet	czipcode	orderdate	soldby
Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-14	Patrick
Wayne	Dick	562-777-3030	1250 Bellflower	90840	2003-07-14	Patrick
Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-18	Kathleen
Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-20	Kathleen

Basic Query Operation: the Join – Natural Join

- ❑ The NATURAL JOIN keyword specifies that the attributes are to be matched between the two tables is those with matching names and matching data types. These should be the PK/FK attributes. The join attributes are shown only once in the result along with the remaining attributes of both tables.
- ❑ Notice that all of the customer info is repeated for each order that the customer has placed. This is expected because of the one-to-many relationship between Customers and Orders.
- ❑ Notice also that any customer who has not placed an order is missing from the results. This is also expected because there is no FK in the Orders table to match that Customer's PK in the Customers table.

Basic Query Operation: the Join – How it Works

- The easiest way to understand the join is to think of the database software looking one-by-one at each pair of rows from the two tables.
- Go through each row of the Customers table and see if it matches each of the rows in the Orders table.

Customers				
cfirstname	clastname	cphone	cstreet	czipcode
Tom	Jewett	714-555-1212	10200 Slater	92708
Alvaro	Monge	562-333-4141	2145 Main	90840
Wayne	Dick	562-777-3030	1250 Bellflower	90840

Orders				
cfirstname	clastname	cphone	orderdate	soldby
Alvaro	Monge	562-333-4141	2003-07-14	Patrick
Wayne	Dick	562-777-3030	2003-07-14	Patrick
Alvaro	Monge	562-333-4141	2003-07-18	Kathleen
Alvaro	Monge	562-333-4141	2003-07-20	Kathleen

Basic Query Operation: the Join – RA Syntax

- The **RA join** of two relations, r over scheme R and s over scheme S , is written $r \bowtie s$, or in our example, $customers \bowtie orders$. The scheme of the result, exactly as you have seen in the SQL syntax, is the union of the two relation schemes, $R \cup S$. The join attributes are found in the intersection of the two schemes, $R \cap S$. Clearly, the intersection attributes must inherit the same assignment rule from R and S . This makes the two schemes **compatible**.
- The result of the RA join consists of the pairwise **paste** of all tuples from the two relations, written $\text{paste}(t, u)$ for any tuple t from relation r over scheme R and u from relation s over scheme S . The result of the paste operation is exactly as explained in the preceding section.

Basic Query Operation: the Join – Cross Join

- ❑ In the very, very rare case where there is no intersection between schemes R and S (that is, $R \cap S = \{\text{null}\}$), the schemes are still compatible and every tuple from relation r is pasted to every tuple from relation s , with all of the attributes from both schemes contained in the resulting tuples.
- ❑ In set theory, this is a ***Cartesian product*** of the two relations; in practice, it is almost always nonsense and not what you want. The Cartesian product can also be written in RA as $r \times s$, or intentionally specified in SQL with the CROSS JOIN keyword.

Basic Query operation: the Join – Paste Operation

1.

$$R \cap S = \{\text{attrC}, \text{attrD}\}$$

Scheme R:

attrA	attrB	attrC	attrD
val1	val2	val3	val4

tuple t:

Scheme S:

attrC	attrD	attrE	attrF
val3	val4	val5	val6

tuple u:

=

Result:

Scheme $R \cup S$:

tuple paste(t,u):

attrA	attrB	attrC	attrD	attrE	attrF
val1	val2	val3	val4	val5	val6

2.

$$R \cap S = \{\text{attrC}, \text{attrD}\}$$

Scheme R:

attrA	attrB	attrC	attrD
val1	val2	val3	val4

tuple t:

Scheme S:

attrC	attrD	attrE	attrF
val3	val5	val6	val7

tuple u:

≠

Result:

Scheme $R \cup S$:

tuple paste(t,u): {null}

attrA	attrB	attrC	attrD	attrE	attrF
-------	-------	-------	-------	-------	-------

3.

$$R \cap S = \{\text{null}\}$$

Scheme R:

attrA	attrB	attrC	attrD
val1	val2	val3	val4

tuple t:

Scheme S:

attrE	attrF	attrG	attrH
val5	val6	val7	val8

tuple u:

Result:

Scheme $R \cup S$:

tuple paste(t,u):

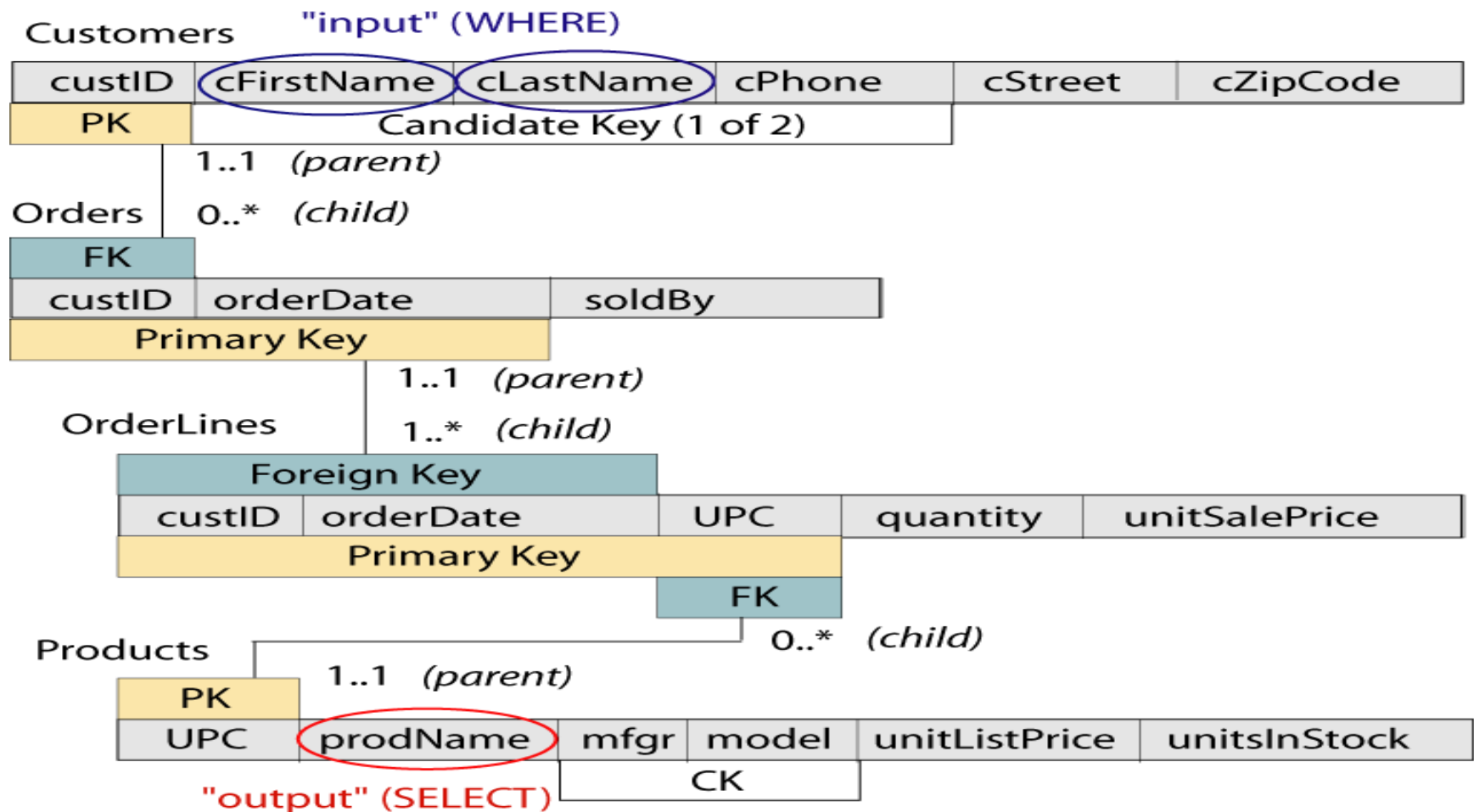
attrA	attrB	attrC	attrD	attrE	attrF	attrG	attrH
val1	val2	val3	val4	val5	val6	val7	val8

SQL Technique: Multiple Joins and the *Distinct* Keyword

- ❑ It is important to realize that if you have a properly designed and linked database, you can retrieve information from as many tables as you want, specify retrieval conditions based on any data in the tables, and show the results in any order that you like.
- ❑ *Example:* We'll use the order entry model. We'd like a list of all the products that have been purchased by a specific customer.

π cLastName, cFirstName, prodName
 σ cFirstName='Alvaro' and cLastName='Monge'
(customers \bowtie orders \bowtie orderlines \bowtie
products)

SQL Technique: Multiple Joins and the *Distinct* Keyword – Relation Scheme



SQL Technique: Multiple Joins and the *Distinct* Keyword – Explained

- ❑ The needed information is found in the Products table (“output” from the query). The retrieval condition or “input” to the query is based on the Customers tables. These attributes will be needed for the WHERE clause. They are only included in the SELECT list to be sure that the query is showing the data that you want.
- ❑ You also need to include the intervening tables in the FROM clause of the query since this is the only way to correctly associate the Customers data with the Products data. They can’t be joined directly because they don’t have any common attributes.
- ❑ Another way to think about this is to simply follow the PK-FK pairs from table to table until you have completely linked all of the info you need.

SQL Technique: Multiple Joins and the *Distinct* Keyword – Step 1

1. Look at the result set (all of the linked data).

```
SELECT * FROM customers  
NATURAL JOIN orders  
NATURAL JOIN orderlines  
NATURAL JOIN products;
```

- Your database system might not support the NATURAL JOIN syntax that we show here.
- The multiple natural joins in our example work correctly because there are no non-pk/fk attributes in any of our tables that have the same name. In larger, more complicated databases, this might not be true.

SQL Technique: Multiple Joins and the *Distinct* Keyword – Step 2

2. Pick the rows you want, and be sure that all of the information makes sense and is really what you are looking for.

```
SELECT * FROM customers
NATURAL JOIN orders
NATURAL JOIN orderlines
NATURAL JOIN products
WHERE cFirstName = 'Alvaro' AND
      cLastName = 'Monge';
```

SQL Technique: Multiple Joins and the *Distinct* Keyword – Step 3

3. Now pick the columns that you want, and again check the results. Notice that we are including the retrieval condition attributes in the SELECT clause, to be sure that this really is the right answer.

```
SELECT cFirstName, cLastName, prodName
FROM customers
NATURAL JOIN orders
NATURAL JOIN orderlines
NATURAL JOIN products
WHERE cFirstName = 'Alvaro' AND
      cLastName = 'Monge';
```

SQL Technique: Multiple Joins and the *Distinct* Keyword – Results

Products purchased

cFirstName	cLastName	prodName
Alvaro	Monge	Hammer, framing, 20 oz.
Alvaro	Monge	Hammer, framing, 20 oz.
Alvaro	Monge	Screwdriver, Phillips #2, 6 inch
Alvaro	Monge	Screwdriver, Phillips #2, 6 inch
Alvaro	Monge	Pliers, needle-nose, 4 inch

SQL Technique: Multiple Joins and the *Distinct* Keyword – Distinct keyword

- ❑ Notice that there are duplicate rows in the result set. *Why?*
- ❑ The DISTINCT keyword will remove the duplicate rows

```
SELECT DISTINCT cFirstName, cLastName, prodName
FROM customers
NATURAL JOIN orders
NATURAL JOIN orderlines
NATURAL JOIN products
WHERE cFirstName = 'Alvaro' AND cLastName = 'Monge'
ORDER BY prodName;
```

Distinct products		
cFirstName	cLastName	prodName
Alvaro	Monge	Hammer, framing, 20 oz.
Alvaro	Monge	Pliers, needle-nose, 4 inch
Alvaro	Monge	Screwdriver, Phillips #2, 6 inch

SQL Technique: Join Types – Natural Join

- ❑ Remember that the NATURAL JOIN is the intersection of the tables, however, if there are any non-PK/FK attributes that have the same name, they will also be joined. This is probably not what you want.
- ❑ It also produces only one copy of those attributes in the result table.

```
SELECT cFirstName, cLastName, orderDate  
FROM customers
```

```
NATURAL JOIN orders;
```

```
r ⋈ s
```

SQL Technique: Join Types – Inner Join Using

- ❑ The INNER JOIN .. USING specifies the columns to join.
- ❑ It also produces only one copy of the join attributes in the result set.

```
SELECT cFirstName, cLastName, orderDate  
FROM customers  
INNER JOIN orders USING (custID);
```

$r \bowtie_{a,b} s$

SQL Technique: Join Types – Inner Join On

- ❑ The INNER JOIN .. ON specifies the columns and the join condition (normally equality).
- ❑ It also requires the join attributes to be prefaced with the table name since both columns will be included in the result set.

```
SELECT cFirstName, cLastName, orderDate  
FROM customers
```

```
INNER JOIN orders
```

```
ON customers.custID = orders.custID;
```

$r \bowtie_{a=b} S$

SQL Technique: Join Types – Inner Join – Alias

- ❑ You can specify an ***alias*** for each table name (such as c and o in this example), then using the alias instead of the full name when you refer to the attributes.
- ❑ This is the only syntax that will let you join a table to itself.

```
SELECT cFirstName, cLastName, orderDate  
FROM customers c  
INNER JOIN orders o  
ON c.custID = o.custID;
```


SQL Technique: Join Types – Outer Join

- ❑ One important effect of all natural and inner joins is that any unmatched PK value simply drops out of the result. In our example, this means that any customer who didn't place an order isn't shown. Suppose that we want a list of *all* customers, along with order date(s) for those who did place orders. To include the customers who did *not* place orders, we will use an OUTER JOIN.
- ❑ An OUTER JOIN allows unmatched rows to be part of the result set. Undefined values are NULL.

SQL Technique: Join Types – Left Outer Join

```
SELECT cFirstName, cLastName, orderDate
FROM customers c
LEFT OUTER JOIN orders o
ON c.custID = o.custID;
```

$r = \bowtie s$

All customers and order dates

cfirstname	clastname	orderdate
Tom	Jewett	
Alvaro	Monge	2003-07-14
Alvaro	Monge	2003-07-18
Alvaro	Monge	2003-07-20
Wayne	Dick	2003-07-14

SQL Technique: Join Types – Right Outer Join

- The word “left” refers to the order of the tables in the FROM clause (customers on the left, orders on the right). The left table here is the one that might have unmatched join attributes—the one from which we want *all* rows. We could have gotten exactly the same results if the table names and outer join direction were reversed:

```
SELECT cFirstName, cLastName, orderDate
FROM orders o
RIGHT OUTER JOIN customers c
ON o.custID = c.custID;      r ⋈= s
```

- An outer join makes sense only if one side of the relationship has a minimum cardinality of zero (as Orders does in this example). Otherwise, the outer join will produce exactly the same result as an inner join (for example, between Orders and OrderLines).
- The SQL standard also allows a FULL OUTER JOIN, in which unmatched join attributes from either side are paired with null values on the other side. You will probably not have to use this with most well-designed databases.
 $r = \bowtie s$

SQL Technique: Join Types – Evaluation Order

- Multiple joins in a query are evaluated left-to-right in the order that you write them, unless you use parentheses to force a different evaluation order. The schemes of the joins are also cumulative in the order that they are evaluated; in RA, this means that
$$r1 \bowtie r2 \bowtie r3 = (r1 \bowtie r2) \bowtie r3$$
- It is especially important to remember this rule when outer joins are mixed with other joins in a query. For example, if you write:

```
SELECT cFirstName, cLastName, orderDate, UPC, quantity FROM
customers
LEFT OUTER JOIN orders USING (custID)
NATURAL JOIN orderlines;
```

you will lose the customers who haven't placed orders. They will be retained if you force the second join to be executed first:

```
SELECT cFirstName, cLastName, orderDate, UPC, quantity
FROM customers
LEFT OUTER JOIN (orders NATURAL JOIN orderlines)
USING (custID);
```

SQL Technique: Join Types – Other Join Types

- ❑ If you try to join two tables with no join condition, the result will be that every row from one side is paired with every row from the other side. It is easy to do this accidentally, by forgetting to put the join condition in the WHERE clause.
- ❑ If you ever have an occasion to really need a Cartesian product of two tables, use a CROSS JOIN:

```
SELECT cFirstName, cLastName, orderDate  
FROM customers  
CROSS JOIN orders;
```
- ❑ It is possible, but confusing, to specify a join condition other than equality of two attributes; this is called a ***non-equi-join***. If you see such a thing in older code, it probably represents a WHERE clause or subquery in disguise.
- ❑ You may also hear the term ***self join***, which is nothing but an inner or outer join between two attributes in the same table.

SQL Technique: Functions

- Sometimes, the information that we need is not actually stored in the database, but has to be computed in some way from the stored data.
- In our order entry example, there are two derived attributes (/subtotal in OrderLines and /total in Orders) that are part of the class diagram but not part of the relation scheme. We can compute these by using SQL functions in the SELECT statement.

SQL Technique: Functions – Computed Columns – How To

- We can compute values from information that is in a table simply by showing the computation in the SELECT clause. Each computation creates a new column in the output table, just as if it were a named attribute.
- *Example:* We want to find the subtotal for each line of the OrderLines table.

```
SELECT custID, orderDate, UPC,  
       unitSalePrice * quantity  
FROM orderlines;
```

SQL Technique: Functions – Computed Columns - Result

Notice that the computation itself is shown as the heading for the computed column. This is awkward to read, and doesn't really tell us what the column means.

Order line subtotals			
custid	orderdate	upc	unitsaleprice * quantity
5678	2003-07-14	51820 33622	11.95
9012	2003-07-14	51820 33622	23.90
9012	2003-07-14	11373 24793	21.25
5678	2003-07-18	81809 73555	18.00
5678	2003-07-20	51820 33622	23.90
5678	2003-07-20	81809 73555	9.00
5678	2003-07-20	81810 63591	24.75

SQL Technique: Functions – As Keyword

We can create our own column heading or alias using the **AS** keyword. If you want your column alias to have spaces in it, you will have to enclose it in *double* quote marks.

```
SELECT custID,orderDate,UPC,  
       unitSalePrice * quantity AS subtotal  
FROM orderlines;
```

Order line subtotals			
custid	orderdate	upc	subtotal
5678	2003-07-14	51820 33622	11.95
9012	2003-07-14	51820 33622	23.90
9012	2003-07-14	11373 24793	21.25
5678	2003-07-18	81809 73555	18.00
5678	2003-07-20	51820 33622	23.90
5678	2003-07-20	81809 73555	9.00
5678	2003-07-20	81810 63591	24.75

SQL Technique: Functions – Aggregate Functions

- ❑ SQL **aggregate functions** let us compute values based on multiple rows in our tables. They are also used as part of the SELECT clause, and also create new columns in the output.
- ❑ *Example:* First, let's just find the total amount of all our sales. To compute this, all we need is to do is to add up all of the price-times-quantity computations from every line of the OrderLines. We will use the **SUM** function to do the calculation.

```
SELECT SUM(unitSalePrice * quantity) AS  
totalsales  
FROM orderlines;
```

Sales
totalsales
132.75

SQL Technique: Functions – Aggregate Functions – Group By

- Next, we'll compute the total for each order. We still need to add up order lines, but we need to group the totals for each order. We can do this with the **GROUP BY** clause.
- This time, the output will contain one row for every order, since the `customerID` and `orderDate` form the PK for *Orders*, not *OrderLines*.
- Notice that the `SELECT` clause and the `GROUP BY` clause contain exactly the same list of attributes, except for the calculation. This is a must!!!!

```
SELECT custID, orderDate,  
SUM(unitSalePrice * quantity) AS total  
FROM orderlines  
GROUP BY custID, orderDate;
```

Order totals		
custid	orderdate	total
5678	2003-07-14	11.95
5678	2003-07-18	18.00
5678	2003-07-20	57.65
9012	2003-07-14	45.15

SQL Technique: Functions – Aggregate Functions – Common Functions

- ❑ Other frequently-used functions that work the same way as SUM include MIN, MAX, and AVG.
- ❑ The COUNT function is slightly different, since it returns the *number* of rows in a grouping. To count all rows, we can use the *.

```
SELECT COUNT(*) FROM orders;
```

Orders	
COUNT(*)	
4	

SQL Technique: Functions – Aggregate Functions – Count

- We can also count groups of rows with identical values in a column. In this case, COUNT will ignore NULL values in the column.
- Here, we'll find out how many times each product has been ordered.

```
SELECT prodname AS "product name", COUNT(prodname)  
AS "times ordered"
```

```
FROM products
```

```
NATURAL JOIN orderlines
```

```
GROUP BY prodname;
```

Product orders

product name	times ordered
Hammer, framing, 20 oz.	3
Pliers, needle-nose, 4 inch	1
Saw, crosscut, 10 tpi	1
Screwdriver, Phillips #2, 6 inch	2

SQL Technique: Functions – Aggregate Functions – Having

- If we want to select output rows based on the results of the group function, the HAVING clause is used instead.
- For example, we could ask for only those products that have been sold more than once:

```
SELECT prodname AS "product name",  
COUNT(prodname) AS "times ordered"  
FROM products  
NATURAL JOIN orderlines  
GROUP BY prodname  
HAVING COUNT(prodname) > 1;
```

SQL Technique: Functions – Aggregate Functions – Other Functions

- Most database systems offer a wide variety of functions that deal with formatting and other miscellaneous tasks. These functions tend to be proprietary, differing widely from system to system in both availability and syntax.
- Most are used in the SELECT clause, although some might appear in a WHERE clause expression or an INSERT or UPDATE statement. Typical functions include:
 - Rounding, truncating, converting, and formatting numeric data types.
 - Concatenating, altering case, and manipulating character data types.
 - Formatting dates and times, or retrieving the date and time from the operating system.
 - Converting data types such as date or numeric to character string, and vice-versa.
 - Supplying visible values to null attributes, allowing conditional output, and other miscellaneous tasks.

SQL Technique: Subqueries

- Sometimes you don't have enough information available when you design a query to determine which rows you want. In this case you'll have to find the required information with a ***subquery***.
- Example: Find the name of customers who live in the same zipcode as Wayne Dick.
- Problem: Putting the zipcode in the query without knowing what it is

SQL Technique: Subqueries – Finding the Unknown

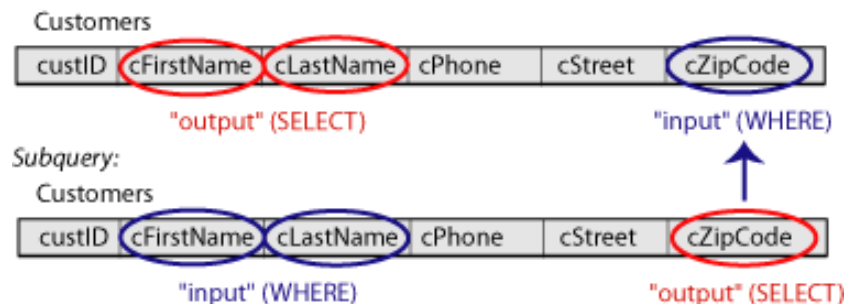
- First, we find the right zip code by writing another query:

```
SELECT cZipCode  
FROM Customers
```

Zip code
czipcode
90840

```
WHERE cFirstName = 'Wayne' AND cLastName = 'Dick';
```

- Since this query returns a single column and a single row. We can use the result as the condition value for cZipCode in our original query. In effect, the output of the second query becomes input to the first one.



SQL Technique: Subqueries – Finding the Unknown

- Syntactically, all we have to do is to enclose the subquery in parentheses, in the same place where we would normally use a constant in the WHERE clause.
- We'll include the zip code in the SELECT line to verify that the answer is what we want:

```
SELECT cFirstName, cLastName, cZipCode
FROM customers
WHERE cZipCode =
    (SELECT cZipCode FROM customers
     WHERE cFirstName = 'Wayne' AND
          cLastName = 'Dick');
```

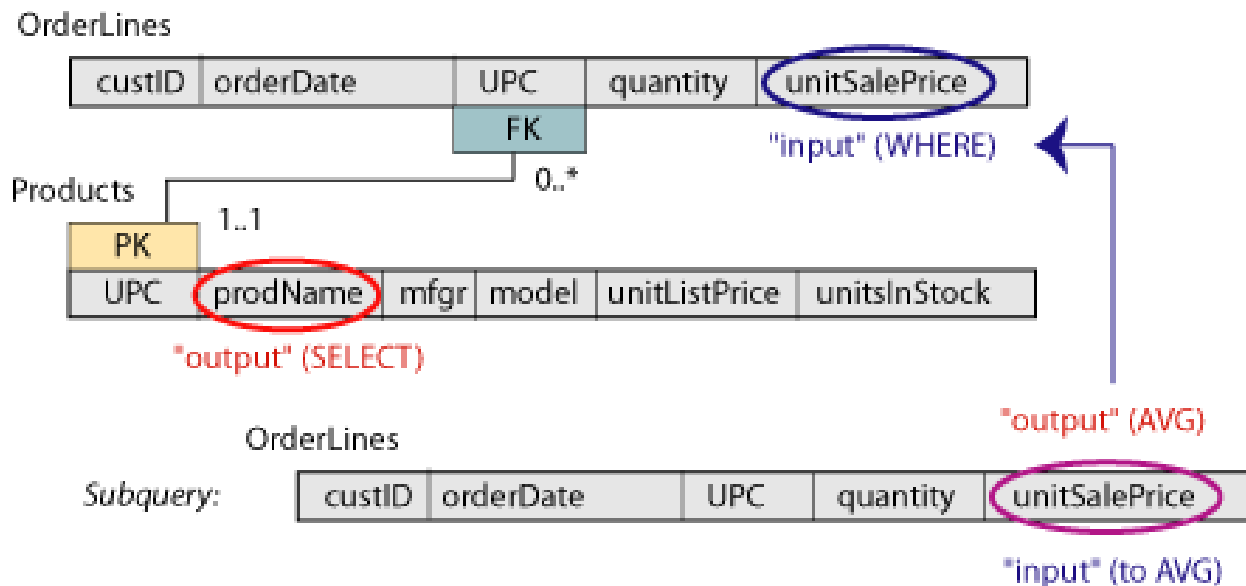
Customers		
cfirstname	clastname	czipcode
Alvaro	Monge	90840
Wayne	Dick	90840

SQL Technique: Subqueries – Another Example

- ❑ A subquery that returns only one column and one row can be used any time that we need a single value.
- ❑ Subqueries can also be used when we need more than a single value as part of a larger query.
- ❑ Another example would be to find the product name and sale price of all products whose unit sale price is greater than the average of all products. The DISTINCT keyword is needed, since the SELECT attributes are not a super key of the result set:

```
SELECT DISTINCT prodName, unitSalePrice
FROM Products NATURAL JOIN OrderLines
WHERE unitSalePrice >
      (SELECT AVG(unitSalePrice) FROM
        OrderLines);
```

SQL Technique: Subqueries – Relation Scheme

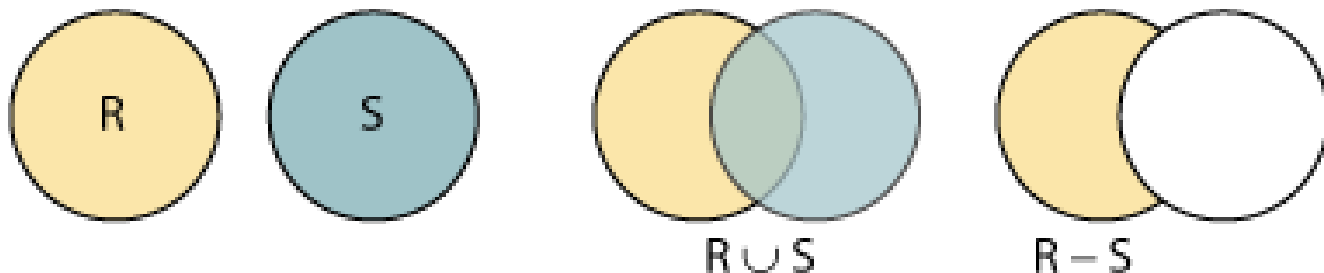


Above average

prodname	unitsaleprice
Hammer, framing, 20 oz.	11.95
Saw, crosscut, 10 tpi	21.25

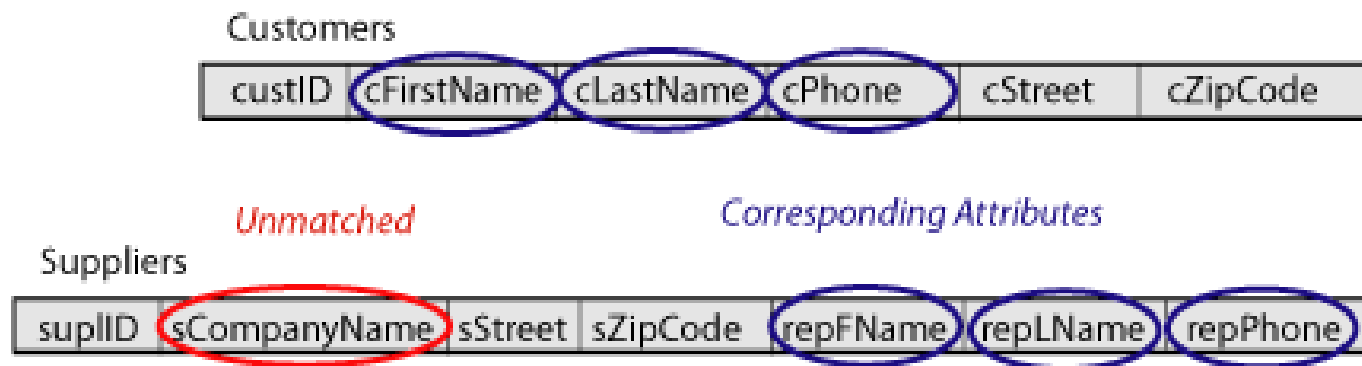
SQL Technique: Union and Minus - Set Operations on Tables

- **Union** includes members of both sets with no duplicates, **minus** includes only those members of the set on the left side of the expression that are not contained in the set on the right side of the expression.
- Both sets have to contain objects of the same type. SQL and RA set operations treat tables as sets of rows. Therefore both tables must have the same number of attributes of the same data type.



SQL Technique: Union and Minus - Union

- Example: add a suppliers table to the order entry model. Produce a listing that shows the names and phone numbers of all people we deal with, whether they are customers or suppliers.
- We need rows from both tables, but they have to have the same attribute list.



SQL Technique: Union and Minus – Union – Building the Query

- We can create an extra column in the query output for the Customers table by simply giving it a name and filling it with a constant value. Here, we'll use the value 'Customer' to distinguish these rows from supplier representatives. SQL uses the column names of the *first* part of the union query as the column names for the output, so we will give each of them aliases that are appropriate for the entire set of data.
- Build and test each component of the union query individually, then put them together. The ORDER BY clause has to come at the end.

```
SELECT cLastName AS "Last Name", cFirstName AS  
"First Name", cPhone as "Phone", 'Customer' AS  
"Company"  
FROM customers  
UNION  
SELECT repLName, repFName, repPhone, sCompanyName  
FROM suppliers  
ORDER BY "Last Name";
```

SQL Technique: Union and Minus – Union – Result

Phone list

Last Name	First Name	Phone	Company
Bradley	Jerry	888-736-8000	Industrial Tool Supply
Dick	Wayne	562-777-3030	Customer
Jewett	Tom	714-555-1212	Customer
Monge	Alvaro	562-333-4141	Customer
O'Brien	Tom	949-567-2312	Bosch Machine Tools

SQL Technique: Union and Minus – Union – Minus

- ❑ Sometimes you have to think about both what you do want and what you don't want in the results of a query. If there is a WHERE clause predicate that completely partitions all rows of interest into those you want and those you don't want, then you have a simple query with a test for inequality.
- ❑ The multiplicity of an association can help you determine how to build the query. Since each product has one and only one supplier, we can partition the Products table into those that are supplied by a given company and those that are not.

SQL Technique: Union and Minus – Union – Simple Inequality

- Sometimes you have to think about both what you do want and what you don't want in the results of a query. If there is a WHERE clause predicate that completely partitions all rows of interest into those you want and those you don't want, then you have a simple query with a test for inequality.
- The multiplicity of an association can help you determine how to build the query. Since each product has one and only one supplier, we can partition the Products table into those that are supplied by a given company and those that are not.

```
SELECT prodName, sCompanyName  
FROM Products NATURAL JOIN Suppliers  
WHERE sCompanyName <> 'Industrial Tool Supply';
```

SQL Technique: Union and Minus – Union – Complications

Contrast this to finding customers who did not make purchases in 2002. Because of the optional one-to-many association between Customers and Orders, there are actually four possibilities:

1. A customer made purchases in 2002 (only).
2. A customer made purchases in other years, but not in 2002.
3. A customer made purchases both in other years and in 2002.
4. A customer made no purchases in any year.

SQL Technique: Union and Minus – Union – Incorrect Solution

- ❑ If you try to write this as a simple test for inequality,
`SELECT DISTINCT cLastName, cFirstName,
cStreet, cZipCode
FROM Customers NATURAL JOIN Orders
WHERE TO_CHAR(orderDate, 'YYYY') <>
'2002';`
- ❑ You will correctly exclude group 1 and include group 2,
but falsely include group 3 and falsely exclude group 4.
- ❑ We can show in set notation what we need to do:
 $\{\text{customers who did not make purchases in 2002}\}$
 $= \{\text{all customers}\} - \{\text{those who did}\}$

SQL Technique: Union and Minus – Union – Correct Solution 1

The easiest syntax in this case is to compare only the customer IDs. We'll use the NOT IN set operator in the WHERE clause, along with a subquery to find the customer ID of those who did make purchases in 2002.

```
SELECT cLastName, cFirstName, cStreet,  
cZipCode
```

```
FROM Customers WHERE custID NOT IN  
  (SELECT custID FROM Orders  
   WHERE TO_CHAR(orderDate, 'YYYY') =  
     '2002');
```

SQL Technique: Union and Minus – Union – Correct Solution 2

We can also use the MINUS operator to subtract rows we don't want from all rows in Customers. (Some versions of SQL use the keyword EXCEPT instead of MINUS.) Like the UNION, this requires the schemes of the two tables to match exactly in number and type of attributes.

```
SELECT cLastName, cFirstName, cStreet,  
       cZipCode
```

```
FROM Customers
```

```
MINUS
```

```
SELECT cLastName, cFirstName, cStreet,  
       cZipCode
```

```
FROM Customers NATURAL JOIN Orders
```

```
WHERE TO_CHAR(orderDate, 'YYYY') = '2002';
```

SQL Technique: Union and Minus – Union – Other Set Operations

SQL has two additional set operators:

- UNION ALL works like UNION, except it keeps duplicate rows in the result.
- INTERSECT operates just like you would expect from set theory.

SQL Technique: Views and Indexes

- ❑ A **view** is simply any SELECT query that has been given a name and saved in the database.
- ❑ A view is also called a **named query** or a **stored query**.
CREATE OR REPLACE VIEW <view_name> AS
SELECT <any valid select query>;
- ❑ The view query itself is saved in the database, but it is not actually run until it is called with another SELECT statement. For this reason, the view does not take up any disk space for data storage, and it does not create any redundant copies of data that is already stored in the tables that it references (which are sometimes called the **base tables** of the view).

SQL Technique: Views and Indexes

- Views

- Although it is not required, many database developers identify views with names such as `v_Customers` or `Customers_view`. This not only avoids name conflicts with base tables, it helps in reading any query that uses a view.
- The keywords `OR REPLACE` in the syntax shown above are optional. Although you don't need to use them the first time that you create a view, including them will overwrite an older version of the view with your latest one, without giving you an error message.
- The syntax to remove a view from your schema is exactly what you would expect:

```
DROP VIEW <view_name>;
```

SQL Technique: Views and Indexes

– Using Views

- ❑ A view name may be used in exactly the same way as a table name in any SELECT query. Once stored, the view can be used again and again, rather than re-writing the same query many times.
- ❑ The most basic use of a view would be to simply SELECT * from it, but it also might represent a pre-written subquery or a simplified way to write part of a FROM clause.
- ❑ In many systems, views are stored in a pre-compiled form. This might save some execution time for the query, but usually not enough for a human user to notice.
- ❑ One of the most important uses of views is in large multi-user systems, where they make it easy to control access to data for different types of users. As a very simple example, suppose that you have a table of employee information on the scheme
Employees = {employeeID, empFName, empLName, empPhone, jobTitle, payRate, managerID}
- ❑ Obviously, you can't let everyone in the company look at all of this information, let alone make changes to it.

SQL Technique: Views and Indexes

– Roles

- ❑ Your database administrator (DBA) can define **roles** to represent different groups of users, and then grant membership in one or more roles to any specific user account (schema). In turn, you can grant table-level or view-level permissions to a role as well as to a specific user. Suppose that the DBA has created the roles *managers* and *payroll* for people who occupy those positions.
- ❑ In Oracle®, there is also a pre-defined role named *public*, which means every user of the database.
- ❑ You could create separate views even on just the Employees table, and control access to them.

SQL Technique: Views and Indexes

– SQL

```
CREATE VIEW phone_view AS
    SELECT empFName, empLName, empPhone
    FROM Employees;
GRANT SELECT ON phone_view TO public;
CREATE VIEW job_view AS
    SELECT employeeID, empFName, empLName,
           jobTitle, managerID
    FROM Employees;
GRANT SELECT, UPDATE ON job_view TO managers;
CREATE VIEW pay_view AS
    SELECT employeeID, empFName, empLName, payRate
    FROM Employees;
GRANT SELECT, UPDATE ON pay_view TO payroll;
```

SQL Technique: Views and Indexes

– Privileges

- ❑ Only a very few trusted people would have SELECT, UPDATE, INSERT, and DELETE privileges on the entire Employees base table; everyone else would now have exactly the access that they need, but no more.
- ❑ When a view is the target of an UPDATE statement, the base table value is changed. You can't change a computed value in a view, or any value in a view that is based on a UNION query.
- ❑ You may also use a view as the target of an INSERT or DELETE statement, subject to any integrity constraints that have been placed on the base tables.

SQL Technique: Views and Indexes

– Materialized Views

- ❑ Sometimes, the execution speed of a query is so important that a developer is willing to trade increased disk space use for faster response, by creating a ***materialized view***. A materialized view *does* create and store the result table in advance, filled with data. The scheme of this table is given by the SELECT clause of the view definition.
- ❑ This technique is most useful when the query involves many joins of large tables, or any other SQL feature that could contribute to long execution times.
- ❑ Since the view would be useless if it is out of date, it must be re-run, at the minimum, when there is a change to any of the tables that it is based on.

SQL Technique: Views and Indexes

– Indexes

- ❑ An **index** is a data structure that the database uses to find records within a table more quickly.
- ❑ Indexes are built on one or more columns of a table. Each index maintains a list of values within that field that are sorted in ascending or descending order.
- ❑ Rather than sorting records on the field or fields during query execution, the system can simply access the rows in order of the index.

SQL Technique: Views and Indexes – Indexes - Unique and Non-Unique

- ❑ When you create an index, you may allow the indexed columns to contain duplicate values. The index will still list all of the rows with duplicates.
- ❑ You may also specify that values in the indexed columns must be unique, just as they must be with a primary key. In fact, when you create a primary key constraint on a table, Oracle and most other systems will automatically create a unique index on the primary key columns, as well as not allowing null values in those columns.
- ❑ One good reason for you to create a unique index on non-primary key fields is to enforce the integrity of a candidate key, which otherwise might end up having duplicate values in different rows.

SQL Technique: Views and Indexes – Queries/Insert/Update

- ❑ You should not create an index on every column or group of columns that will ever be used in an ORDER BY clause.
- ❑ Each index will have to be updated every time that a row is inserted or a value in that column is updated. Although index structures allow this to happen very quickly, there still might be circumstances where too many indexes would detract from overall system performance.

SQL Technique: Views and Indexes – SQL

```
CREATE INDEX <indexname> ON <tablename>
    (<column>, <column>...);
```

- ❑ To enforce unique values, add the UNIQUE keyword:

```
CREATE UNIQUE INDEX <indexname> ON
    <tablename> (<column>, <column>...);
```

- ❑ To specify sort order, add the keyword ASC or DESC after each column name, just as you would do in an ORDER BY clause.
- ❑ To remove an index, simply enter:

```
DROP INDEX <indexname>;
```

Transactions –

Why Do We Need Transactions?

- Many enterprises use databases to store information about their state
 - e.g., Balances of all depositors at a bank
- When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way
 - e.g., Bank balance must be updated when deposit is made

Transactions - Concurrency

- ❑ The goal in a 'concurrent' DBMS is to allow multiple users to access the database simultaneously without interfering with each other.
- ❑ A problem with multiple users using the DBMS is that it may be possible for two users to try and change data in the database simultaneously. If this type of action is not carefully controlled, inconsistencies are possible.
- ❑ To control data access, we first need a concept to allow us to encapsulate database accesses. Such encapsulation is called a '**Transaction**'.

Transactions - What is a Transaction?

- A sequence of many actions which are considered to be one atomic unit of work.
- Basic operations a transaction can include are:
 - Reads, Writes
 - Commits, Rollbacks

Transactions –

What Does a Transaction Do?

- Return information from the database
 - RequestBalance transaction: Read customer's balance in database and output it
- Update the database to reflect the occurrence of a real world event
 - Deposit transaction: Update customer's balance in database
- Cause the occurrence of a real world event
 - Withdraw transaction: Dispense cash (and update customer's balance in database)

Transactions - Outcomes

- After work is performed in a transaction, two outcomes are possible:
 - Commit - Any changes made during the transaction by this transaction are committed to the database.
 - Abort - All the changes made during the transaction by this transaction are not made to the database. The result of this is as if the transaction was never started.

Transactions - Commit and Abort

- If the transaction successfully completes it is said to commit
 - The system is responsible for ensuring that all changes to the database have been saved
- If the transaction does not successfully complete, it is said to abort
 - The system is responsible for undoing, or rolling back, all changes the transaction has made

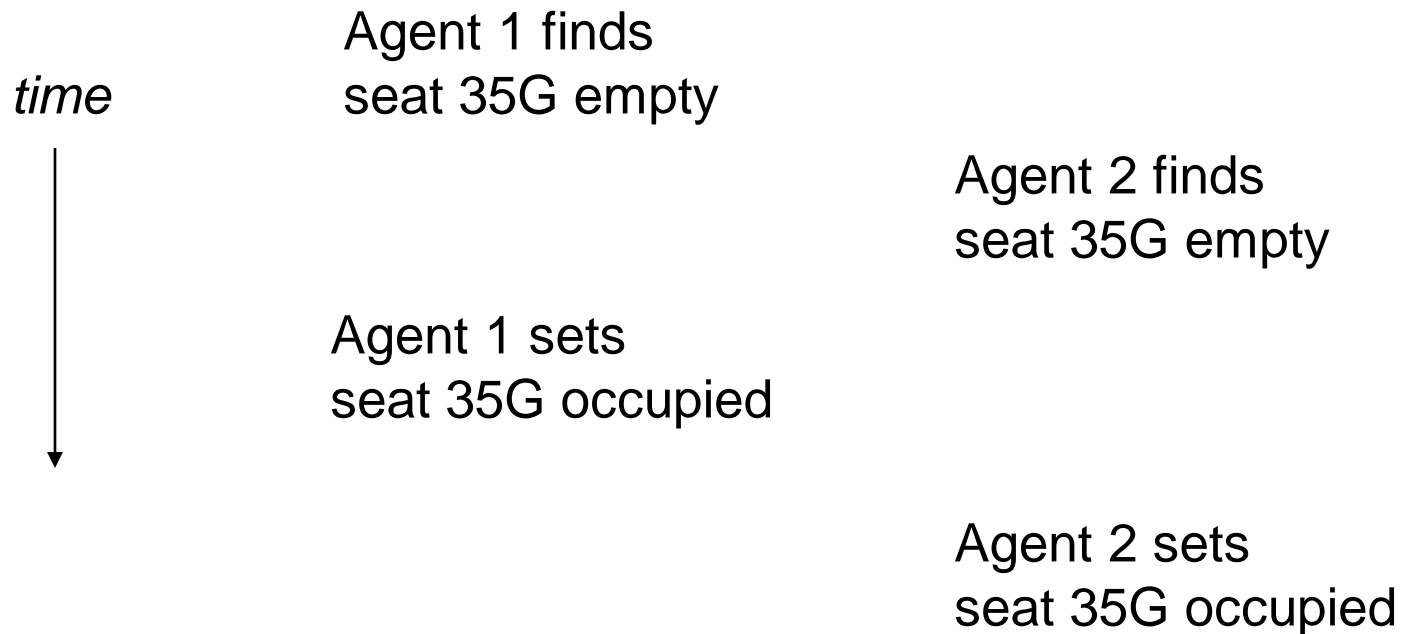
Transactions - Reasons for Abort

- System crash
- Transaction aborted by system
 - Execution cannot be made atomic (a site is down)
 - Execution did not maintain database consistency (integrity constraint is violated)
 - Execution was not isolated
 - Resources not available (deadlock)
- Transaction requests to roll back

Transactions – Problem 1

Ex. Reserving a seat for a flight --

If concurrent access is allowed to data in a DBMS, two users may try to book the same seat simultaneously



Transactions - Schedules

- A transaction schedule is a tabular representation of how a set of transactions were executed over time. This is useful when examining problem scenarios. Within the diagrams various nomenclatures are used:
 - READ(a) - This is a read action on an attribute or data item called 'a'.
 - WRITE(a) - This is a write action on an attribute or data item called 'a'.
 - WRITE(a[x]) - This is a write action on an attribute or data item called 'a', where the value 'x' is written into 'a'.
 - t_n (e.g. t_1, t_2, t_{10}) - This indicates the time at which something occurred. The units are not important, but t_n always occurs before t_{n+1} .

Transactions – Problem 2

- Problems can occur when concurrent transactions execute in an uncontrolled manner.
- Examples of one problem.
 - A original equals 100, after executing T1 and T2, A is supposed to be $100+10-8=102$ but A is 92

Add 10 To A	Minus 8 from A	Value of A on the disk
T1	T2	
Read(A) A=A+10		100
	Read(A) A=A-8	100
Write(A)	Write(A)	100
		100
		110
		92

Transactions – Problem 3

- Consider transaction A, which loads in a bank account balance X (initially \$20) and adds \$10 to it. Such a schedule would look like this:

Time	Transaction
t1	TOTAL:=READ(X)
t2	TOTAL:=TOTAL+10
t3	WRITE(X[30])

Transactions – Problem 3

- Now consider that, at the same time as trans A runs, trans B runs.
- Transaction B gives all accounts a 10% increase. Will X be 32 or 33?

Time	Transaction A	Transaction B
t1		BALANCE:=READ(X)
t2	TOTAL:=READ(X)	
t3	TOTAL:=TOTAL+10	
t4	WRITE(X[30])	
t5		BONUS:=BALANCE*110%
t6		WRITE(X[22])

- Woops... X is 22! Depending on the interleaving, X can also be 32, 33, or 30.

Transactions – Lost Update Scenario

Time	Transaction A	Transaction B
t1	READ(R)	
t2		READ(R)
t3	WRITE(R)	
t4		WRITE(R)

Transaction A's update is lost at t4, because Transaction B overwrites it. B missed A's update at t4 as it got the value of R at t2.

Transactions – Uncommitted Dependency

Time	Transaction A	Transaction B
t1		WRITE(R)
t2	READ(R)	
t3		ABORT

Transaction A is allowed to READ (or WRITE) item R which has been updated by another transaction but not committed (and in this case ABORTed).

Transactions – Inconsistency Scenario

TIME	X	Y	Z	TRANSACTION A		TRANSACTION B
				ACTION	SUM	
t1	40	50	30	SUM:=READ(X)	40	
t2	40	50	30	SUM+=READ(Y)	90	
t3	40	50	30			READ(Z)
t4	40	50	20			WRITE(Z[20])
t5	40	50	20			READ(X)
t6	50	50	20			WRITE(X[50])
t7	50	50	20			COMMIT
t8	50	50	20	SUM+=READ(Z)	110	
				SUM should have been 120		

Transactions - Requirements

- The execution of each transaction must maintain the relationship between the database state and the enterprise state
- Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs:

- Atomicity
- Consistency
- Isolation
- Durability

ACID properties

Transactions-ACID Properties of Transactions

- ❑ **Atomicity:** Transaction is either performed in its entirety or not performed at all.
- ❑ **Consistency:** Transaction must take the database from one consistent state to another.
- ❑ **Isolation:** Transaction should appear as though it is being executed in isolation from other transactions.
- ❑ **Durability:** Changes applied to the database by a committed transaction must persist, even if the system fails before all changes reflected on disk.

Transactions - Atomicity

- A real-world event either happens or does not happen
 - Student either registers or does not register
- Similarly, the system must ensure that either the corresponding transaction runs to completion or, if not, it has no effect at all
 - Not true of ordinary programs. A crash could leave files partially updated on recovery

Transactions – Atomicity (cont.)

- Partial effects of a transaction must be undone when
 - User explicitly aborts the transaction using ROLLBACK
 - Application asks for user confirmation in the last step and issues COMMIT or ROLLBACK depending on the response
 - An error, exception, or constraint violation occurs during a transaction
 - The DBMS crashes before a transaction commits
- How is atomicity achieved?
 - Logging

Transactions - Isolation

- Serial Execution: transactions execute in sequence
 - Each one starts after the previous one completes.
 - Execution of one transaction is not affected by the operations of another since they do not overlap in time
 - The execution of each transaction is isolated from all others.
- If the initial database state and all transactions are consistent, then the final database state will be consistent and will accurately reflect the real-world state, *but*
- Serial execution is inadequate from a performance perspective

Transactions – Isolation (cont.)

- Concurrent execution offers performance benefits:
 - A computer system has multiple resources capable of executing independently (e.g., cpu's, I/O devices), *but*
 - A transaction typically uses only one resource at a time
 - Hence, only concurrently executing transactions can make effective use of the system
 - Concurrently executing transactions yield interleaved schedules

Transactions – Isolation (cont.)

- ❑ Transactions must ***appear*** to be executed in a serial schedule (with no interleaving operations)
- ❑ For performance, DBMS executes transactions using a serializable schedule
 - In this schedule, operations from different transactions can interleave and execute concurrently
 - But the schedule is guaranteed to produce the same effects as a serial schedule
- ❑ How is isolation achieved?
 - Locking, multi-version concurrency control (method commonly used to provide concurrent access to the database)

Transactions – Database Consistency

- Enterprise (Business) Rules limit the occurrence of certain real-world events
 - Student cannot register for a course if the current number of registrants equals the maximum allowed
- Correspondingly, allowable database states are restricted
$$cur_reg \leq max_reg$$
- These limitations are called (static) integrity constraints: assertions that must be satisfied by all database states

Transactions – Transaction Consistency

- A consistent database state does not necessarily model the actual state of the enterprise
 - A deposit transaction that increments the balance by the wrong amount maintains the integrity constraint $balance \geq 0$, but does not maintain the relation between the enterprise and database states
- A consistent transaction maintains database consistency and the correspondence between the database state and the enterprise state (implements its specification)
 - Specification of deposit transaction includes
$$balance' = balance + amt_deposit ,$$
$$(balance' \text{ is the next value of } balance)$$

Transactions - Consistency

- Consistency of the database is guaranteed by constraints and triggers declared in the database and/or transactions themselves
 - When inconsistency arises, abort the transaction or fix the inconsistency within the transaction

Transactions - Durability

- The system must ensure that once a transaction commits, its effect on the database state is not lost in spite of subsequent failures
 - Not true of ordinary programs. A media failure after a program successfully terminates could cause the file system to be restored to a state that preceded the program's execution

Transactions – Durability (cont.)

- Effects of committed transactions must survive DBMS crashes
- How is durability achieved?
 - DBMS manipulates data in memory; forcing all changes to disk at the end of every transaction is very expensive
 - Logging

Transactions – Implementing Durability

- ❑ Database stored redundantly on mass storage devices to protect against media failure
- ❑ Architecture of mass storage devices affects type of media failures that can be tolerated
- ❑ Related to Availability: extent to which a (possibly distributed) system can provide service despite failure
 - ❑ Non-stop DBMS (mirrored disks)
 - ❑ Recovery based DBMS (log)

Transactions – Isolation Levels

- Strongest isolation level: **SERIALIZABLE**
 - Complete isolation
 - SQL default
- Weaker isolation levels: **REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED**
 - Increase performance by eliminating overhead and allowing higher degrees of concurrency
 - Trade-off: sometimes you get the wrong answer

Transactions – Example Schema

```
CREATE TABLE Account
(accno INTEGER NOT NULL PRIMARY KEY,
name CHAR(30) NOT NULL,
balance FLOAT NOT NULL
CHECK(balance >= 0));
```


Transactions – Read Uncommitted

- ❑ Can read dirty data
- ❑ A data item is dirty if it is written by an uncommitted transaction
- ❑ Problem: What if the transaction that wrote the dirty data eventually aborts?
- ❑ Example: wrong average

-- T1:

```
UPDATE Account  
SET balance = balance - 200  
WHERE accno = 142857;
```

ROLLBACK;

-- T2:

```
SELECT AVG(balance)  
FROM Account;
```

COMMIT;

Transactions – Read Committed

- ❑ No dirty reads, but non-repeatable reads possible
 - Reading the same data item twice can produce different results
- ❑ Example: different averages

-- T1:

```
UPDATE Account
SET balance = balance - 200
WHERE accno = 142857;
COMMIT;
```

-- T2:

```
SELECT AVG(balance)
FROM Account;
```

```
SELECT AVG(balance)
FROM Account;
COMMIT;
```

Transactions – Repeatable Read

- Reads are repeatable, but may see phantoms
 - A phantom read occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first.
- Example: different average (still!)

-- T1:

```
INSERT INTO Account  
VALUES(428571, 1000);  
COMMIT;
```

-- T2:

```
SELECT AVG(balance)  
FROM Account;
```

```
SELECT AVG(balance)  
FROM Account;  
COMMIT;
```

Transactions - Serializable

- None of these problems can happen

Transactions – Summary of SQL Isolation Levels

Isolation Level/Anomaly	Dirty Reads	Non-Repeatable Reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

Transactions - Serializability

- A 'schedule' is the actual execution sequence of two or more concurrent transactions.
- A schedule of two transactions T1 and T2 is 'serializable' if and only if executing this schedule has the same effect as either
T1;T2 or T2;T1.

Transactions - Precedence Graph

- In order to know that a particular transaction schedule can be serialized, we can draw a precedence graph. This is a graph of nodes and vertices, where the nodes are the transaction names and the vertices are attribute collisions.
- The schedule is said to be serialized if and only if there are no cycles in the resulting diagram.

Transactions – Precedence Graph - Method

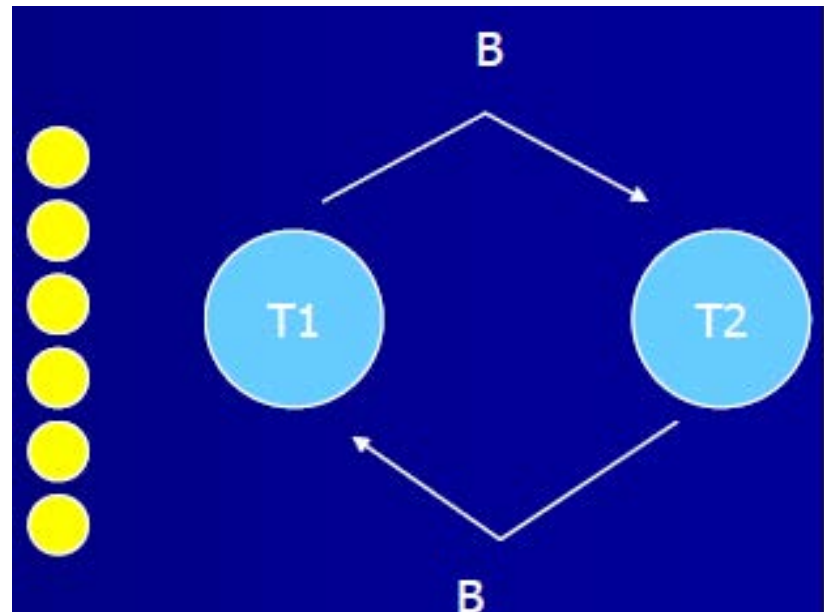
To draw one:

1. Draw a node for each transaction in the schedule
2. Where transaction A writes to an attribute which transaction B has read from, draw a line pointing from B to A.
3. Where transaction A writes to an attribute which transaction B has written to, draw a line pointing from B to A.
4. Where transaction A reads from an attribute which transaction B has written to, draw a line pointing from B to A.

Transactions - Schedule – Ex. 1

□ Consider the following Schedule:

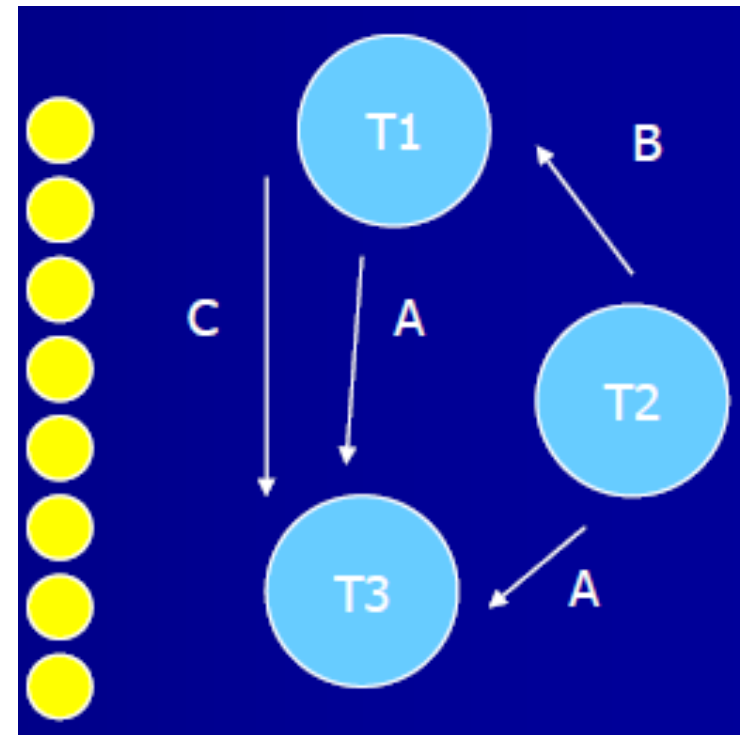
Time	T1	T2
t1	READ(A)	
t2	READ(B)	
t3		READ(A)
t4		READ(B)
t5	WRITE(B)	
t6		WRITE(B)



Transactions - Schedule – Ex. 2

□ Consider the following Schedule:

Time	T1	T2	T3
t1	READ(A)		
t2	READ(B)		
t3		READ(A)	
t4		READ(B)	
t5			WRITE(A)
t6	WRITE(C)		
t7	WRITE(B)		
t8		WRITE(C)	



Methods to Ensure Serializability

- ❑ **Locking**
- ❑ **Timestamping**
- ❑ Concurrency control subsystem is "part of the package" and not directly controllable by either the users or the DBA
- ❑ A **scheduler** is used to allow operations to be executed immediately, delayed, or rejected
- ❑ If an operation is delayed, it can be done later by the same transaction
- ❑ If an operation is rejected, the transaction is aborted but it may be restarted later

Locks

- ❑ Transaction can ask DBMS to place locks on data items
- ❑ Lock prevents another transaction from modifying the object
- ❑ Transactions may be made to wait until locks are released before their lock requests can be granted
- ❑ Objects of various sizes (DB, table, page, record, data item) can be locked.
- ❑ Size determines the fineness, or **granularity**, of the lock
- ❑ Lock implemented by inserting a flag in the object or by keeping a list of locked parts of the database
- ❑ Locks can be **exclusive** or **shared** by transactions
 - Shared locks are sufficient for read-only access
 - Exclusive locks are necessary for write access

Lock Compatibility Matrix

	Transaction 2 Requests Shared Lock	Transaction 2 Requests Exclusive Lock
Transaction 1 Holds No Lock	Yes	Yes
Transaction 1 Holds Shared Lock	Yes	No
Transaction 1 Holds Exclusive Lock	No	No

Deadlock

- Often, transaction cannot specify in advance exactly what records it will need to access in either its **read set** or its **write set**
- **Deadlock**- two or more transactions wait for locks being held by each another
- Deadlock detection uses a **wait-for** graph to identify deadlock
 - Draw a node for each transaction
 - If transaction S is waiting for a lock held by T, draw an edge from S to T
- **Cycle** in the graph shows deadlock

Deadlock with Two Transactions

Time	Transaction S	Transaction T
t1	request Xlock a	
t2	grant Xlock a	...
t3		request Xlock b
t4	...	grant Xlock b
t5	request Xlock b	...
t6	wait	request Slock a
t7	wait	wait
t8	wait	wait
t9	wait	wait
...

Deadlock

- Deadlock is resolved by choosing a **victim**-newest transaction or one with least resources
- Should avoid always choosing the same transaction as the victim, because that transaction will never complete - called **starvation**

Two-phase locking protocol

- ❑ Guarantees serializability
- ❑ Every transaction acquires all its locks before releasing any, but not necessarily all at once
- ❑ Transaction has two phases:
 - In **growing** phase, transaction obtains locks
 - In **shrinking** phase, it releases locks
- ❑ Once it enters its shrinking phase, a transaction can never obtain a new lock
- ❑ For **standard two-phase locking**, the rules are
 - Transaction must acquire a lock on an item before operating on the item.
 - For read-only access, a shared lock is sufficient. For write access, an exclusive lock is required.
 - Once the transaction releases a single lock, it can never acquire any new locks
- ❑ Deadlock can still occur

What is JDBC ?

- ❑ JDBC stands for “Java DataBase Connectivity”
- ❑ The standard interface for communication between a Java application and a SQL database
- ❑ Allows a Java program to issue SQL statements and process the results.

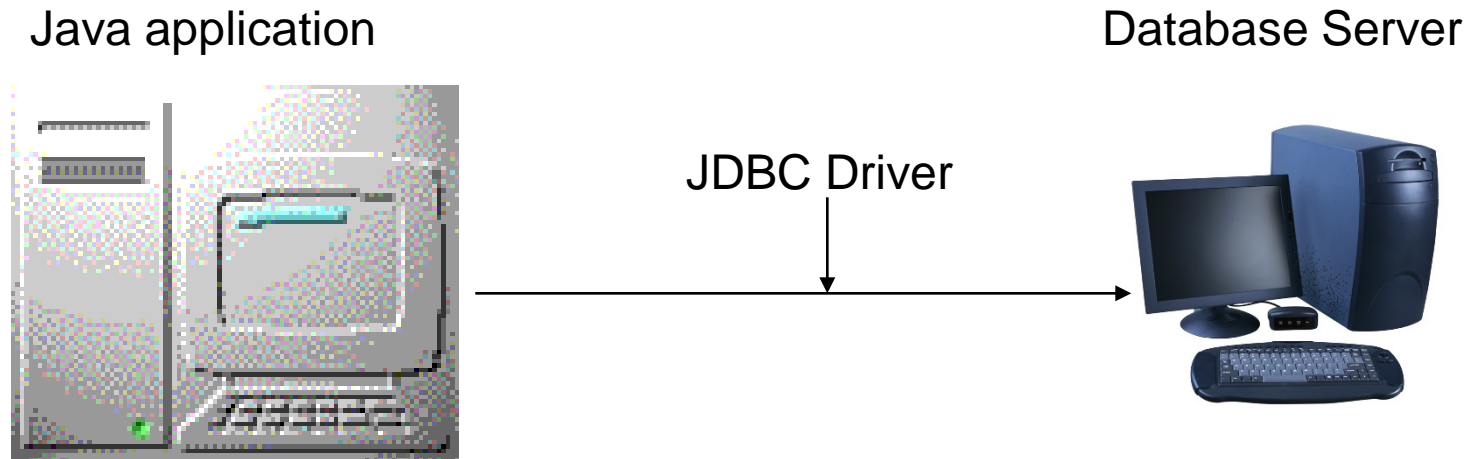
JDBC Classes and Interfaces

Steps to using a database query:

- Load a JDBC “driver”
- Connect to the data source
- Send/execute SQL statements
- Process the results

JDBC Driver

- ❑ Acts as the gateway to a database
- ❑ Not actually a “driver”, just a .jar file



JDBC Driver Installation

- ❑ Must download the driver, then add the .jar file to your \$CLASSPATH or NetBeans project
- ❑ To set up your classpath on Windows
 - Control panel
 - Search for Environment Variables
 - Add the jar file to your CLASSPATH variable

JDBC Driver Management

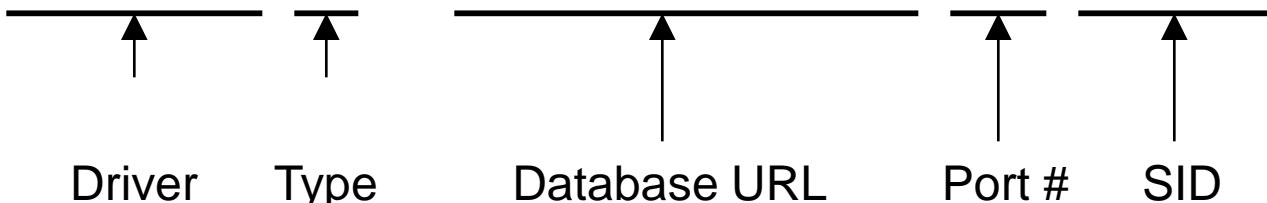
- All drivers are managed by the DriverManager class
- Example - loading an Oracle JDBC driver:
 - In the Java code:
`Class.forName("oracle.jdbc.driver.OracleDriver")`
- Driver class names:
 - Oracle: `oracle.jdbc.driver.OracleDriver`
 - MySQL: `com.mysql.jdbc.Driver`
 - MS SQL Server:
`com.microsoft.jdbc.sqlserver.SQLServerDriver`

Establishing a Connection

- Create a Connection object
- Use the DriverManager to grab a connection with the getConnection method
- Necessary to follow exact connection syntax
- Problem 1: the parameter syntax for getConnection varies between JDBC drivers
- Problem 2: one driver can have several different legal syntaxes

Establishing a Connection (cont.)

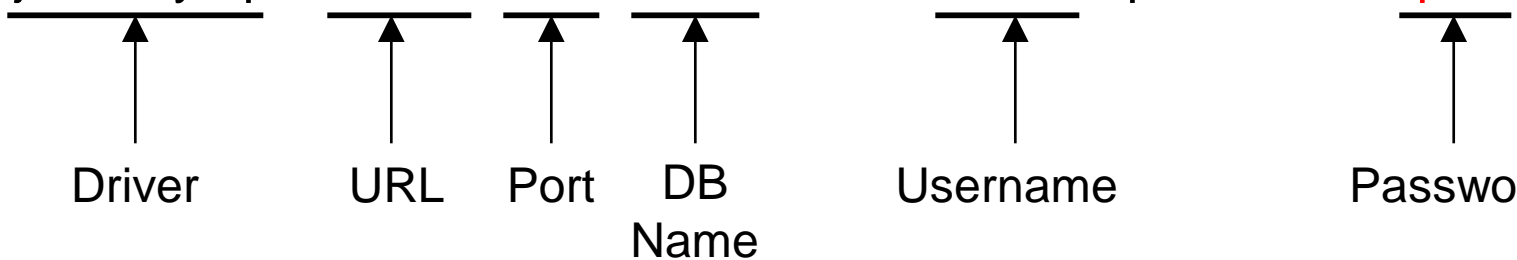
Oracle Example

- Connection con =
DriverManager.getConnection(string,
"username", "password");
- what to supply for string ?
- “jdbc:oracle:thin:@augur.seas.gwu.edu:1521:orcl10g2”


Driver	Type	Database URL	Port #	SID
--------	------	--------------	--------	-----

Establishing a Connection (cont.)

MySQL Example

- Connection con =
DriverManager.getConnection(string);
- what to supply for string ?
- “jdbc:mysql://<URL>:3306/<DB>?user=<user>&password=<pw>”


The diagram shows the string "jdbc:mysql://<URL>:3306/<DB>?user=<user>&password=<pw>" with horizontal lines under the following segments: "jdbc:mysql://", "<URL>", "3306/", "<DB>", "<user>", and "<pw>". Arrows point from these underlined segments to labels below: "Driver" points to "jdbc:mysql://", "URL" points to "<URL>", "Port" points to "3306/", "DB Name" points to "<DB>", "Username" points to "<user>", and "Password" points to "<pw>".

Executing Statements

- Obtain a statement object from the connection:
 - `Statement stmt = con.createStatement ();`
- Execute the SQL statements:
 - `stmt.executeUpdate("update table set field='value'");`
 - `stmt.executeUpdate("INSERT INTO mytable VALUES (1, 'name')");`
 - `stmt.executeQuery("SELECT * FROM mytable");`

Retrieving Data

- `ResultSet rs = stmt.executeQuery("SELECT id,name FROM employees where id = 1000")`
- Some methods used in `ResultSet`:
 - `next()`
 - `getString()`
 - `getInt()`

Using the Results

```
while (rs.next())  
{  
    int s = rs.getInt("id");  
    String n = rs.getString("name");  
    System.out.println(s + " " + n);  
}
```

Executing SQL Statements

- Three different ways of executing SQL statements:
 - Statement (both static and dynamic SQL statements)
 - PreparedStatement (semi-static SQL statements)
 - CallableStatement (stored procedures)

PreparedStatement class: Precompiled, parametrized SQL statements:

- Structure is fixed
- Values of parameters are determined at run-time

The Challenge

- ❑ The createStatement objects that you saw in the example code are OK for simple statements, but they become cumbersome if you want to prompt the user for values at runtime.
- ❑ Also, the database optimizer (at least in Oracle) will take the literal string of a statement, and create a map from that string to the compiled version.
 - If you run the same exact statement twice, with the only difference being the value of a literal, the optimizer will **not** recognize that, and parse the statement from scratch each time you submit it with a new literal.
 - Also, it's cumbersome to have to translate your variables to String before including them into the statement.

Bind ambition

- The PreparedStatement is an alternative that is **somewhat** better.
- For example, to prompt the user for the name of the album that they wish to query on, you would:
 - String stmt = "select * from albums where albumtitle = ?";
 - PreparedStatement pstmt = conn.prepareStatement(stmt);
- What's going on is that stmt string has a bind variable, that "?" in it, that stands for a value. In this case, it's a String, but it could be any supported primitive datatype.
- The beauty of the prepared statement, even in this case, is that the value that is substituted in at run time for that "?" could have embedded quotes.
- The next line creates the pstmt object. It's not ready to execute yet, we still need to supply the value that we want to substitute in for the "?". That is called **binding** the variable.

Mapping from the bind variables to a value

- ❑ `pstmt.setString(1, "Please Please Me");` tells JDBC to map the value "Please Please Me" to the first (this is where this gets messy) bind variable in the pstmt statement.
- ❑ That's not too bad if you only have one or two bind variables, you just have to keep track of which is in which order, and call the proper set method with the proper index.
- ❑ However, be **very careful** if you should ever insert a new bind variable into the middle of your statement.

A set of sets

- ❑ If you go to URL: <http://docs.oracle.com/javase/8/docs/api/>, and then click on the link labeled “java.sql”, you’ll find that there is an API just for the various JDBC methods.
- ❑ Once in the java.sql API, if you select PreparedStatement, you’ll see a collection of set methods, like setDate, setFloat (I’ll have root beer in mine thank you) setInt to name just a few.
- ❑ The setDate will take a java Date object (which you can build once you import the java.util.date package).

The execution

- Once you have prepared the statement, and bound all of its variables, you're ready to execute.
- There is an `executeQuery`, `executeUpdate` or just `execute`.
 - The `executeQuery` returns a `ResultSet` object that you're already familiar with.
 - The `executeUpdate` method will not just do updates, it performs inserts and deletes as well, and returns the integer number of records impacted by your DDL statement.

Executing SQL Statements (cont.)

```
String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatement pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4,age);
// we know that no rows are returned, thus we use
    executeUpdate()
int numRows = pstmt.executeUpdate();
```

ResultSet

- ❑ `PreparedStatement.executeUpdate` only returns the number of affected records
- ❑ `PreparedStatement.executeQuery` returns data, encapsulated in a `ResultSet` object (a cursor)

```
ResultSet rs=pstmt.executeQuery();  
// rs is now a cursor  
While (rs.next()) {  
    // process the data  
}
```

ResultSets (cont.)

A ResultSet is a very powerful cursor:

- ❑ `previous()`: moves one row back
- ❑ `absolute(int num)`: moves to the row with the specified number
- ❑ `relative (int num)`: moves forward or backward
- ❑ `first()` and `last()`

Retrieving the results

- The ResultSet interface sports getDate, getInt, getString, and so on to convert from the database representation of the data to a Java representation of the data.

Matching Java-SQL Data Types

<u>SQL Type</u>	<u>Java class</u>	<u>ResultSet get method</u>
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Data About the Data

- ❑ `ResultSetMetaData rsmd = rs.getMetaData();`
- ❑ Use the import: `java.sql.*` for the `ResultSetMetaData` interface.
- ❑ For more information, see the documentation at: <https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html>.
- ❑ What I'm going to cover is just a fraction of what this class can do.
- ❑ Bear in mind that these methods are all operating on the entire result set, not any given row in the result set.

Useful methods

- ❑ `int – getColumnCount()` returns the # of columns in the result set. Good first call to make when looping through the meta data for a given result set.
- ❑ These utilities all start at 1 for the first column. Do not let that confuse you with starting at 0 for arrays.
- ❑ `int – getColumnDisplaySize (int)` – returns the designated column's maximum width.
- ❑ `String – getColumnName (int)`
- ❑ `int getColumnType (int)` – returns an integer corresponding to the column type. See `java.sql.Types` for the list of valid types that can be returned.
- ❑ `String getColumnTypeName (int)` – returns name of the column type.

You Also Might be Interested In:

- ❑ `int isNullable (int)` – you receive an integer {columnNoNulls, columnNullable, columnNullableUnknown} in return
- ❑ `String getTableName (int)` – returns the table name from the specified column index
- ❑ `boolean isReadOnly (int)`
- ❑ `boolean isWritable (int)`

JDBC: Exceptions and Warnings

- ❑ Most of java.sql can throw and SQLException if an error occurs (use try/catch blocks to find connection problems)
- ❑ SQLWarning is a subclass of SQLException; not as severe (they are not thrown and their existence has to be explicitly tested)

JDBC MySQL example:

- Excellent instructions can be found at:
- <http://www.tutorialspoint.com/jdbc/jdbc-environment-setup.htm>
- Sample Code can be found at:
- <http://www.csulb.edu/~mopkins/cecs323/FirstExample.java>