

CECS 323



David Brown

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

UML design
Many-to-Many
Many-to-Many 2
Colliding Foreign Keys
Subkey
Repeated Attributes
Multi-Valued Attributes
Domains
Enumerated Domains
Subclasses
Aggregation
Recursive Associations
Normalization
Denormalization
BCNF
4NF
5NF
Transactions
JDBC

Like ugly on an ape



- There are going to be some things that I will caution you **not** to do.
- If I find you doing one of those, I'll be all over you like ugly on an ape.
- I'll highlight such cautions with this picture of my buddy Naruto as a graphic reminder.

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

Quick survey of where we are

- What are some similarities between databases and spreadsheets?
- What are some differences?
- What can you tell me about foreign keys? What makes them foreign?
 - Are foreign keys a conceptual artifact, or strictly physical?
 - Would it make sense to migrate a key **other** than the primary key as a foreign key?
- What are atomic values, and how should you design for them?
- Data dictionary
 - Don't put what the model already says into the definition.
 - I have a template that I'll put on the BeachBoard for you.

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
- ❑ Data quality starts with a specification of what can and **cannot** happen in the data – inventory control example.
 - That specification can be static, specifying states that the data at rest can assume
 - Or dynamic, specifying rules regarding the **transitions** between states.
 - Static constraints are only enforced between transactions.

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

□ Why study relational algebra?

- It provides the mathematical underpinnings for the relational model.
- Be aware that SQL is designed by a committee, many of whom are database vendors, hence it is not a perfect implementation of relational algebra.

Models and Languages

- Discussion question – what **is** a model?
 - Bob Hope as the model husband
- 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.
 - **Don't** include material in your models (designs) just because you think that they **ought** to be there.
 - If the requirement is not called out, do not put it in there.
- There are many models and languages used for this modeling. Some of these are mathematically based. Others are less formal and more intuitive.

Why do we need models?

- ❑ The software development life cycle (SDLC) is a process with specific disciplines that are engaged at various points.
 - Systems architect
 - Data architect/data modeler
 - Business analyst
 - Database Administrator
 - Developer
 - Tester
- ❑ The models that are developed in the course of the SDLC serve as a communication means between various of these disciplines.
- ❑ Through the process of forward engineering, models undergo transformations from one domain of meta data represented to the next, to serve that stage in the life cycle.

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.
- ❑ A good modeling tool does more than just draw.
 - It **should** enforce the semantics of the language as well.
 - And at least partially automate the forward engineering process.

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.
- ❑ Instead of an ER modeling language, we will be using a proprietary language called Relation Scheme that helps emphasize those parts of the database design most important for this course.

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 that we use in the Relational Algebra language are 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.
- ❑ These terms are used to refer to the **physical** view of the database.
- ❑ In this model you'll see familiar terms such as table, referential integrity constraint, column, index, and so forth.

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.
 - The optimizer is the software that takes the SQL and maps it to a procedural approach, based on conditions in the database such as the distribution of the data values and the presence of indexes.
- 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
 - Note that this is selective, we are only going to represent those “things” which are of **interest** to the enterprise.
 - This is the foundation of all good modeling: ignoring what doesn’t interest you.
- 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.

 - What are some other classes that come to mine?
- **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 numbers are not descriptive attributes

The Sample Data Challenge

- ❑ Spreadsheets resemble database tables in that they have columns of data.
- ❑ Each cell in a given column is presumed to have a similar sort of data in it.
- ❑ The meaning of all of the cells in a given column is expected to be the same.
- ❑ When you are designing a class, and you identify descriptive attributes for it, ask yourself how you would populate a spreadsheet with sample data for that entity. If you cannot envision that, you probably need to rethink your design.

Bad definition

-  An employee is a class of things that have an employee ID, department number, annual salary, and some number of jobs that they perform.
 - The notion of class will become very familiar to us before we are done, but your business client is not going to resonate with this. Besides, this information is gratuitous.
 - “Thing” is too generic. It doesn’t say anything.
 - Listing the attributes of the class redundant to the model.
 - Listing the relationships of the class is redundant too.

Better Definition

- An employee is a person who draws a salary from the company at set intervals in exchange for performing stipulated tasks on a given schedule.
 - Start generic. An employee is a person. “Person” may need to be defined in some contexts.
 - Then become more specific. What kind of person are we talking about?
 - Don’t be more specific than the business allows. For instance, if you have some employees getting paid weekly and others getting paid **bi**-weekly, then you have to be generic about the payment cycle.


What the definition achieves

- ❑ Be sure that your class definitions make it clear what criteria a given object must meet in order for it to be a member of the class.
- ❑ For instance, to say that a faculty member is an employee of the university is not specific enough. “Employee” includes the technicians, administrators, cafeteria workers, ...
- ❑ If your definition is not solid enough to allow a reasonable person to reject “things” that do not belong in that class, the definition has failed.

Attribute Definitions

- Describe what the attribute signifies.
 - Values without a definition are meaningless. You might think 90732 is a zip code, but it might be the reading on your odometer.
- Provide enough information that invalid values can be quickly detected.
 - For instance, an odometer reading can never be negative.
 - A person's birthdate must be in the past.
 - A person wishing to buy liquor's birth date must be at least 21 years in the past.

Attribute Definition Example

- Say you are managing information about windows, and one of the attributes of the window-offering class is thickness.
 -  Bad definition: a measure of how thick the window is.
 - It is circular, because “thick” is a related word to “thickness”.
 - It does not specify that this is even a quantity.
 - In this case, it is a number, which means that we need units.
 - Better: The number of centimeters to two decimal places of the amount of depth in the window frame to allow for the installation of the window.
 - It tells us that this is a number, hence it must at least be a decimal quantity.
 - It gives us the units that the quantity is measured in.
 - And it gives us the purpose for the attribute.

What **Is** a Class?

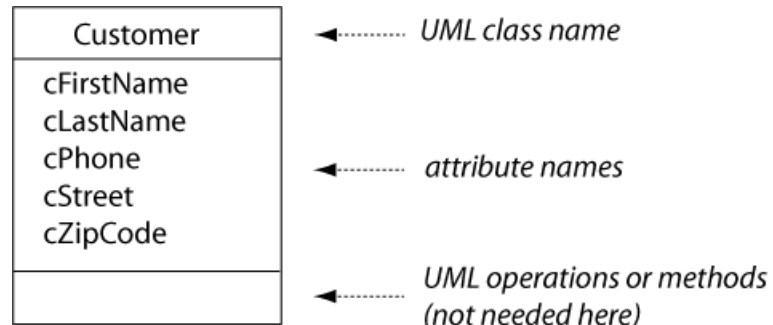
- It represents the information that we want to track about a collection of things.
- Each of those things has certain pieces of information in common.
- They may be tangible or intangible.
 - For instance, an automobile is tangible. It has a model, make, year, VIN, color, current owner ...
 - Automobile **registration** is **intangible**. Yet it's just as real to the DMV. Registration has the VIN, the owner, the date that that particular owner bought the automobile, when they sold it, ...

Class exercise – identifying classes

- Name some possible classes, and discuss the context in which those classes might be important.
- Would a sparkplug be a class?
- How about the act of installing a sparkplug in a specific automobile?
 - What other information would you need for such a record to be meaningful?
- How about a class called Enterprise?
 - How would you define it?
 - What are some example instances of Enterprise?
 - How do we know it's a legitimate class?

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.

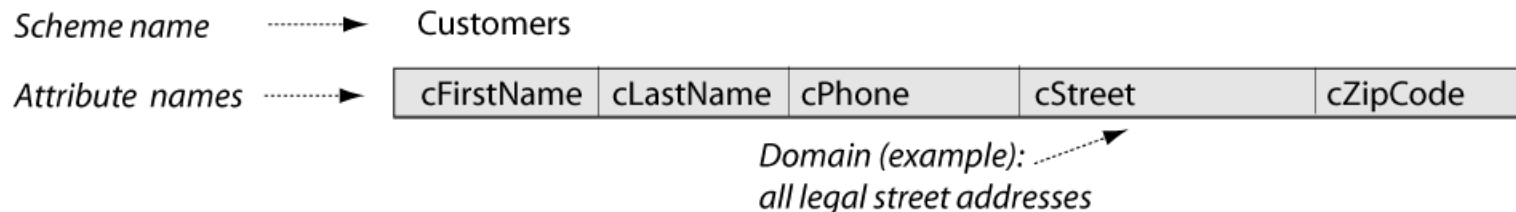


A note on definitions

- The Customer class represents any person who has done business with us or who we think might do business with us in the future.
- Note that a single customer is represented by a single instance of the Customer class.
- Note also that the above definition didn't go into any of the attributes of a customer. That's because the graphics of the model itself convey that.
- What the model **cannot** convey is what it **means** to be a customer in this “universe”.

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 intrinsically **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 (e.g. 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));
```

Life Cycle

- Note that we have intentionally suppressed the datatype of the attributes until the DDL.
- Same observation for optional versus required attributes. We will discuss an approach later that all but removes the need for optional attributes.
- While the attributes have an order, it is not significant. We will soon see that we can display the columns of this data in any order that we wish.

DDL vs. DML

- ❑ DDL, or data **definition** language is used to build the structures that the data of a given relational database have to fit into.
- ❑ DML, or data **manipulation** language is used to declare the operations that are desired on the data stored in the structures defined by the DDL.
- ❑ We'll cover this more in detail later in the course, but be aware that the execution of DDL causes the database management system to create data about the objects created. This is data about data, and hence is referred to as **meta** data.
- ❑ Conceptually, it is not possible to **generate** the DDL from a relation scheme diagram even if we had some tool for composing the relation scheme diagrams that was aware of the semantics of those models.
 - The relation scheme diagram does not carry the datatype
 - Nor does it carry the optionality of the attributes

Basic Structures: Classes and Schemes – Data Models

- Conceptual
- Logical
- Physical

Basic Structures: Rows and Tables

- Definitions

- Each real-world individual of a class is represented by a **row** of information in a database table. (think object in Object Oriented programming)
- The row is defined in the relational model as a **tuple** that is constructed over a given scheme.
 - Remember, the scheme is a template, a cookie cutter if you will, for creating valid tuples.
 - Much as a class in Java is a template for objects.
- 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.

The tuple as a function

- Remember that in mathematics, a function is defined by its set of inputs (called the domain) a set containing the set of outputs, and the set of input-output pairs.
- The set of inputs for a tuple would be any one of the keys for that scheme.
 - “any one of the keys” is shorthand for the set of attribute-value pairs that include all the members of the key to the tuple.
 - For instance, since we need to know the department to know which course a given course number is, CECS, 341 would be a key for a database of courses taught here.
- Each of the dependent attributes would be one output, so technically, a given tuple is likely to be multiple functions, all sharing the same inputs.
- Remember that the domain is merely the set of all possible values for a given variable.
- A term that we will be using more later is *functional dependency*. The **non** key attributes are functionally dependent upon the attributes in the primary key. Other functional dependencies may exist as well within a scheme.

Domain Sneak Peek

- A domain is the set of all possible values for a given attribute.
 - Sometimes, we can express the domain by providing rules about its members.
 - {'A', 'B', 'C', 'D', 'F'} is a static list of values.
 - > 1439 & <= year(now())
 - Other domains are not so easily described.
- We will learn how to use referential integrity to enforce some domains later in this course.

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 that attribute's 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 of 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.

Sneak Peak - Atomicity

- Looking at the previous example, you may be tempted to gloss over the simple fact that each tuple has one and only one value for a given attribute.
- That's quite important. We want to always make our values atomic – that is, they cannot be broken down.
- A value like “Felix Just, S.J., Ph.D” can be broken down.
- A street address may or may not make sense to break down.

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';
```

Our First Relational Algebra

- If you think about your math courses, this should look familiar, just different.
- t_{TJ} is a symbol telling us that this is a particular tuple (symbolized by the lower case t). Just which one is denoted by the TJ subscript.
 - There are no really hard and fast conventions for individual tuple designations, TJ just stands for Tom Jewett.
- Think of a Cartesian space with x, y , and z coordinates. When referring to a particular point, you probably remember referring to the coordinates as x_1, y_1 , and z_1 .

Functional Dependencies

Scheme name **Customers**

Attribute names

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


Row (tuple)

Tom	Jewett	714-555-1212	10200 Slater	92708
-----	--------	--------------	--------------	-------

Data cell
(domain value assigned to attribute name)

- ❑ In the previous example, we mentioned functional dependencies. Simply put, $\{A\} \rightarrow \{B\}$ if knowing the values of the attributes $\{A\}$ is enough information for you to know the values of the attributes $\{B\}$.
- ❑ $\{cFirstName, cLastName, cPhone\} \rightarrow \{cStreet\}$
- ❑ $\{cFirstName, cLastName, cPhone\} \rightarrow \{cZipCode\}$
- ❑ Or, the shorthand notation:
 - $\{cFirstName, cLastName, cPhone\} \rightarrow \{cStreet, cZipCode\}$

Insert options

- ❑ Contrary to set theory, the order of the columns when you create the table is preserved in the database definition.
- ❑ Therefore, the above insert statement **could** be coded: **INSERT INTO** customers VALUES ('Tom', 'Jewett', '714-555-1212', '10200 Slater', '92708');
- ❑  This is categorically evil.
 - It obscures which value is going into which column.
 - If the table columns are in any way changed, your insert statement will produce unexpected results.
 - It's just plain hard to read.

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	----->	PK																				
Column (attribute) names	----->	<table><tr><th>cFirstName</th><th>cLastName</th><th>cPhone</th><th>cStreet</th><th>cZipCode</th></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 in the relation scheme model as the primary key for this scheme, 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. (E.g. it would make no sense to have an Order row in the Customers table.) – (MDC IMS example with cafeteria menus)
- 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 **some** 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.
- A ***minimal super key*** is a super key that will cease to uniquely identify the rows if **any** of its attributes are removed.
- The database designer picks **one** of the minimal super key sets to serve as the ***primary key*** or unique identifier of each row.

Basic Structures: Rows and Tables

– All of the **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

- Note that at the very least, **all** of the attributes of a given scheme must be a superkey, otherwise it is not a valid scheme.

Basic Structures: Rows and Tables

– Insuring Unique Rows - SQL


To add a primary key to existing table:

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

To **create** the table with a primary key

```
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));
```

Putting all the constraints in

- ❑ SQL allows every aspect of the table (including its name) to be updated once the table has been created.
- ❑ Having said that, it's always best to impose all the constraints on the table at once, when its created so that there is no opportunity for rows that violate one of those constraints to get in. If that happens, the constraint cannot be added until the offending row(s) is/are dealt with.
- ❑  In short, I don't want to see you adding a referential integrity or primary key constraints after the table has been defined.

An Example

A student is anyone who attends a four year institution of higher learning for the purpose of obtaining a degree in a particular major.

Student
+firstName
+lastName
+studentID
+GPA
+major
+yearOfStudy

- ❑ Do you see any derived data here?
- ❑ What would you pick for the primary key?
- ❑ How do we determine year of study?

Example Continued

- ❑ What do you think of capturing the gender of a student or faculty member?
 - Who would need to know?
 - Should we protect that information?
- ❑ What about the SSN? How would you propose to secure that information against a data leak?
- ❑ What do you think would be the cost of gathering the information that you're calling for in your design?
- ❑ Do you think that it's easy or difficult to add additional information to a design after it's been implemented?
- ❑ How do you think a business would evaluate whether gathering a given piece of information was “worth it”?

Continued (again)

- If we were to model a pizza class, what were some of the ways that you thought to structure the ingredients of the pizza?
 - Repeating attribute
 - What sort of issues do you see with that?
 - What happens if we add more ingredients?
 - Many to many between pizzas and ingredients
- What are some of the queries that you can think of that would involve the ingredients?

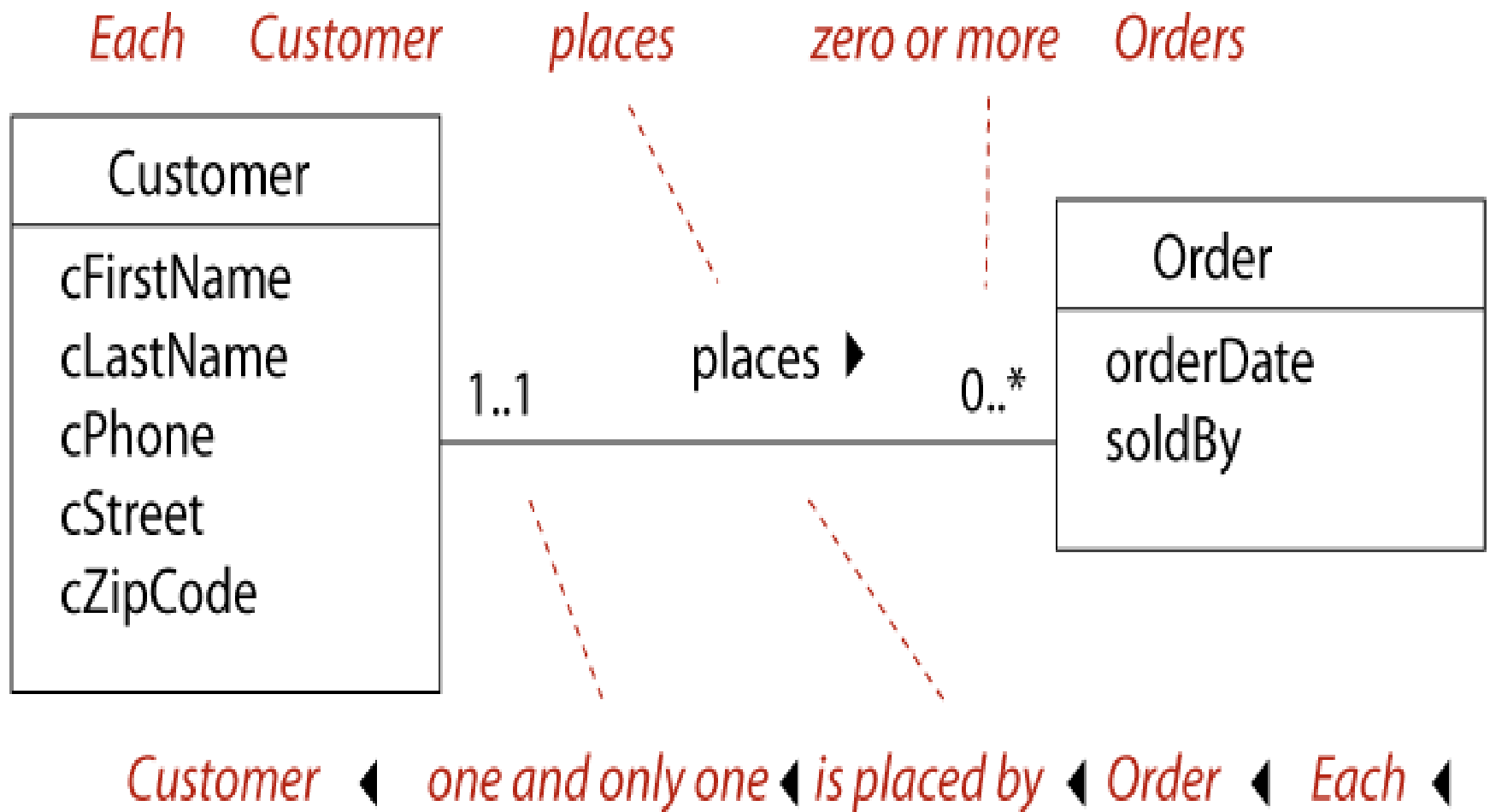
Students & Faculty

- ❑ Did you see any similarities between the students and the faculty?
- ❑ How should we take advantage of those?
- ❑ What about the attributes of student that are distinct from faculty?
- ❑ Always be ready to give examples of your domains. On the work of art, for instance, you might want “art style”. What would the possible values be for that?
- ❑ How would we gauge the **minimum** information needed for a class?
- ❑ Do you think that we might ever need history? For instance, if a faculty member retires, would we keep any records on the courses they taught, the grades that they gave out, and so forth?

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 from our existing customers table to the new orders table.
- ❑ **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



A note about the verb phrase

- It's often been my experience that users will smile and nod at all the right places as you model their requirements, until you read these associations to them aloud.
- Always read both the forward and the reverse of the association to yourself to make sure that it makes sense. If not, figure out what your model is missing.
- Note that this is the first place where we are going to run into a case where the UML model has more semantics than the table model.
 - The cardinality limits on the child cannot be enforced by the RDBMS declaratively.
 - But you **can** put in a trigger to enforce such constraints if those constraints are important to the business.

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
- The Order class represents an event that happens when a customer decides to buy one or more of our products
- 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.
- These relationship sentences form the basis of what are often termed *business rules*. They state things about how the business operates.

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 UML, the details of how that association is physically achieved is left unspecified by design.
- 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 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***.
- The term **foreign** denotes that this key is not native to the orders table; but is rather the **implementation** of the association.

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 – Identifying Relationship

- Note that in the previous diagram, the three migrating foreign keys are all included in the key of the child.
- The way to think about this is that we need to know the customer who placed the order **and** the order date in order to identify the order.
-  It is important to regard the migrating foreign keys as a package. You **cannot** just put some of them in the primary key of the child.
- As you have probably already guessed, if Orders has a child, that association will migrate **four** primary keys.

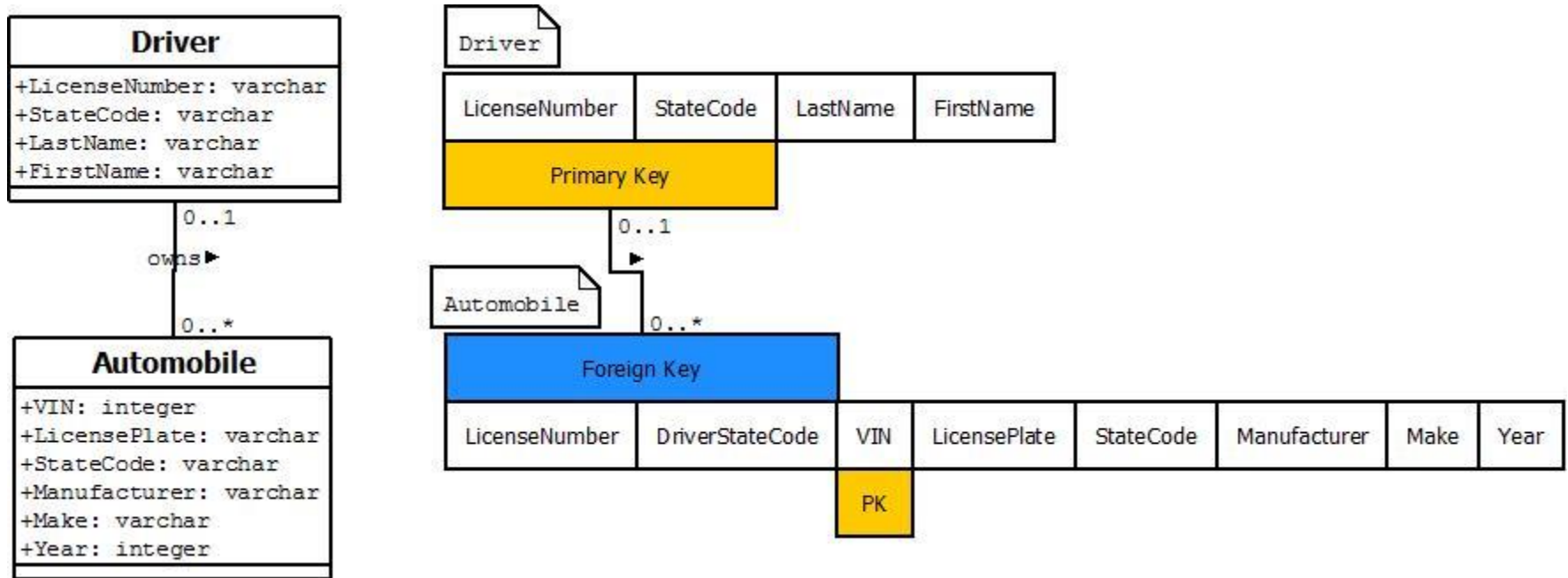
Copying the Key of the Parent

- ❑ There are those who argue that the relational database management system is inherently wasteful.
- ❑ “Cloning” the parent’s PK does not add any information to the database.
- ❑ This leads new designers to want to use a surrogate to reduce the # of migrating keys.
- ❑ If you introduce a surrogate, you **must** provide a candidate key, unless it’s a human.

Identifying vs. **non**-identifying

- In the previous example, to properly identify the order, we need to know who placed the order.
- But consider a different scenario – identifying the owner of a vehicle.
 - The vehicle is identified, not by the owner, but by the Vehicle Identification Number (VIN).
 - The owner of the vehicle is merely another attribute of the vehicle.
 - In fact, the vehicle has a VIN in advance of anyone purchasing the vehicle initially, so there is no existential dependency of the vehicle on the owner.

Let's see that in pictures



- Note that the minimum cardinality is zero at **both** ends of this relationship.
 - A driver could exist in the DMV database without any registered vehicles
 - A vehicle can exist without any driver for it at some point in time.

Adding the Foreign Key Constraint

- ❑ To make sure that we never assign an invalid owner to a given vehicle, we impose the constraint that the owner has to be found in the driver table:

```
ALTER TABLE AUTOMOBILE
```

```
ADD CONSTRAINT automobile_driver_fk_01
```

```
FOREIGN KEY (LicenseNumber,  
DriverStateCode) REFERENCES DRIVER  
(LicenseNumber, StateCode);
```

- ❑ Note that the StateCode migrates into an attribute called **Driver**StateCode to distinguish it from the statecode associated with the automobile itself.

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.
- More formally, the orders table has an *existential dependency* on customers: the parent customer must exist in the database before they can place an order.
- 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.
- What happens when/if you delete a customer?

What about deletions in the parent table?

- If the DBMS were to simply remove the parent row without considering any child row(s) that depend on that parent row, you might create “orphans” in the child table.
- There are five alternatives:
 - restrict (the default)
 - cascade
 - set null
 - set default
 - no action

Which type of delete action?

- ❑ Restrict will let the deletion run if there are no child rows impacted.
- ❑ Cascade is thorough, but dangerous
 - There could be a whole chain of relationships starting with the table that you are deleting from, each with a cascade on delete, and you finally bump into the one lonely relationship in that chain with restrict ...
 - The amount of data loss could be huge from a single delete action.
- ❑ Set default might be a workable strategy (delete an instructor, set the courses that they were teaching to reference the ubiquitous “staff” instructor).
- ❑ Set null is almost never a good option, particularly if the relationship is an identifying one.

Basic Structures: Associations – The Child Table

```
CREATE TABLE orders (  
    cfirstname VARCHAR(20) not null,  
    clastname VARCHAR(20) not null,  
    cphone VARCHAR(20) not null,  
    orderdate DATE not null,  
    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
- ❑ Note that the migrated foreign keys must be **not null** because the relationship is **identifying**.

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);
```
- ❑ Strictly speaking, a primary key is only required if a table will be a parent to some other table, but I am going to **require** that every table in your designs have a primary key defined.

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 ensure 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_01
```

```
FOREIGN KEY(cfirstname, clastname,  
cphone)
```

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

Forward Engineering Associations

- ❑ In UML class modeling, these are termed associations, and the ratios of one class to the other in the association is the multiplicity.
- ❑ In the relation scheme diagram, we call it a relationship, and the ratios of one class to the other is termed the cardinality.
- ❑ In the database management system, the association is implemented via a referential integrity constraint, and we don't talk about cardinality at all because the database is agnostic in that regard.

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

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, that is, it makes sense to the business.

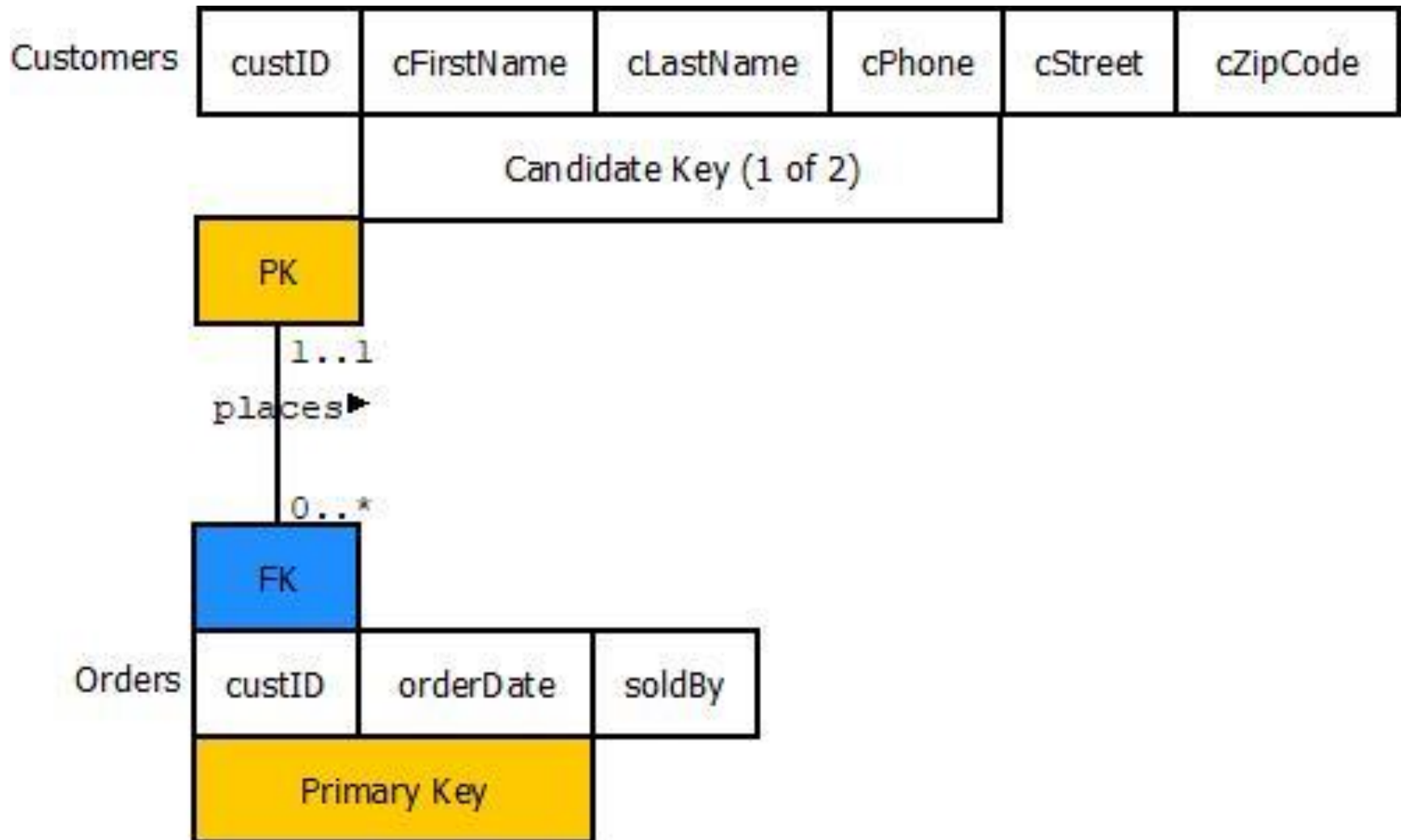
Discussion: More About Keys – Surrogate/Substitute Keys

-  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
 - a. Remember that we require that a **Primary Key** be identified for a table, regardless whether it is a parent or not.
 - b. We will only introduce a surrogate or substitute key as a convenience if the table is a parent **and** these other conditions apply.
 3. there is no candidate key small enough for its values to be copied many times into the child table(s)
- **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 PKs

- Even if an attribute is a member of a candidate key, if that attribute is going to change regularly, it's a poor candidate for going into a PK, which makes any candidate key that uses that attribute a poor candidate for a PK.
 - This is because those changes in the PK attributes have to ripple through the migrating foreign keys.
 - If there is a string of identifying relationships, that ripple effect will keep going.
- If the attribute is optional, that disqualifies it as a member of a Primary Key as well.

Discussion: More About Keys – Revising the Relation Scheme



Candidate Key Enforcement

- ❑ Remember from lab that when you create a primary key, the database automatically creates what is called a *backing index* on the columns in the primary key to enforce uniqueness.
- ❑ The candidate key is more than just a concept, it is a constraint on the data.
- ❑ Therefore, each candidate key that you model must be declared to the database in the form of a uniqueness constraint which will, in turn, cause the database to create a unique index for the candidate 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

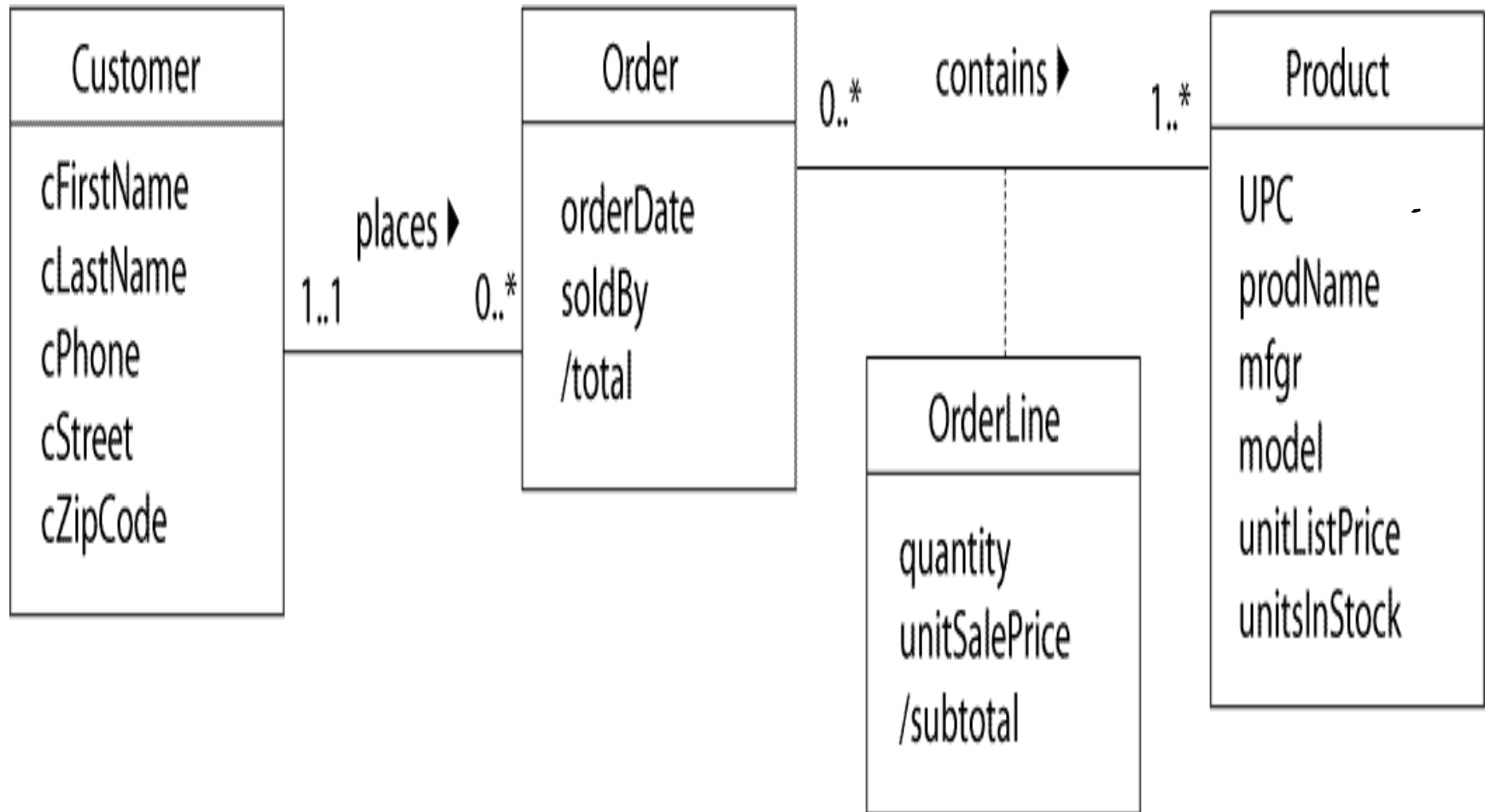
Design Pattern: Many-to-Many (Order Entry)

- ❑ ***Design patterns*** are modeling situations that you will find repeatedly 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



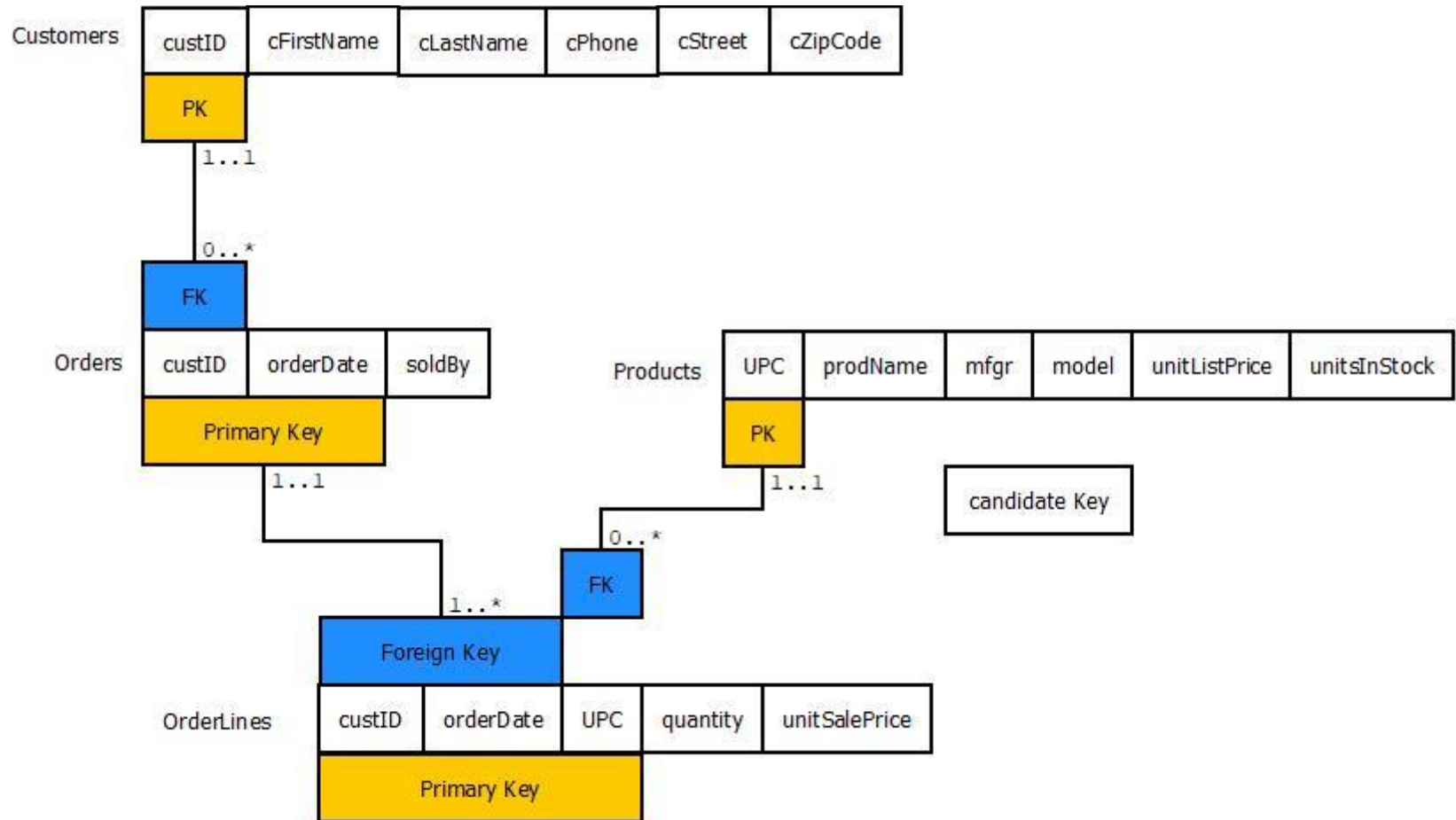
Modeling Style Note

- There are those modelers who would argue that if you don't have anything to put into the association class, then leave the class out.
- I will **insist** that you show the association class every time you have a many to many relationship without fail:
 - It is there as a reminder in case association class attributes and/or associations to/from the association class emerge later
 - It makes mapping the conceptual model into the physical model easier because the association class materializes as the junction table.

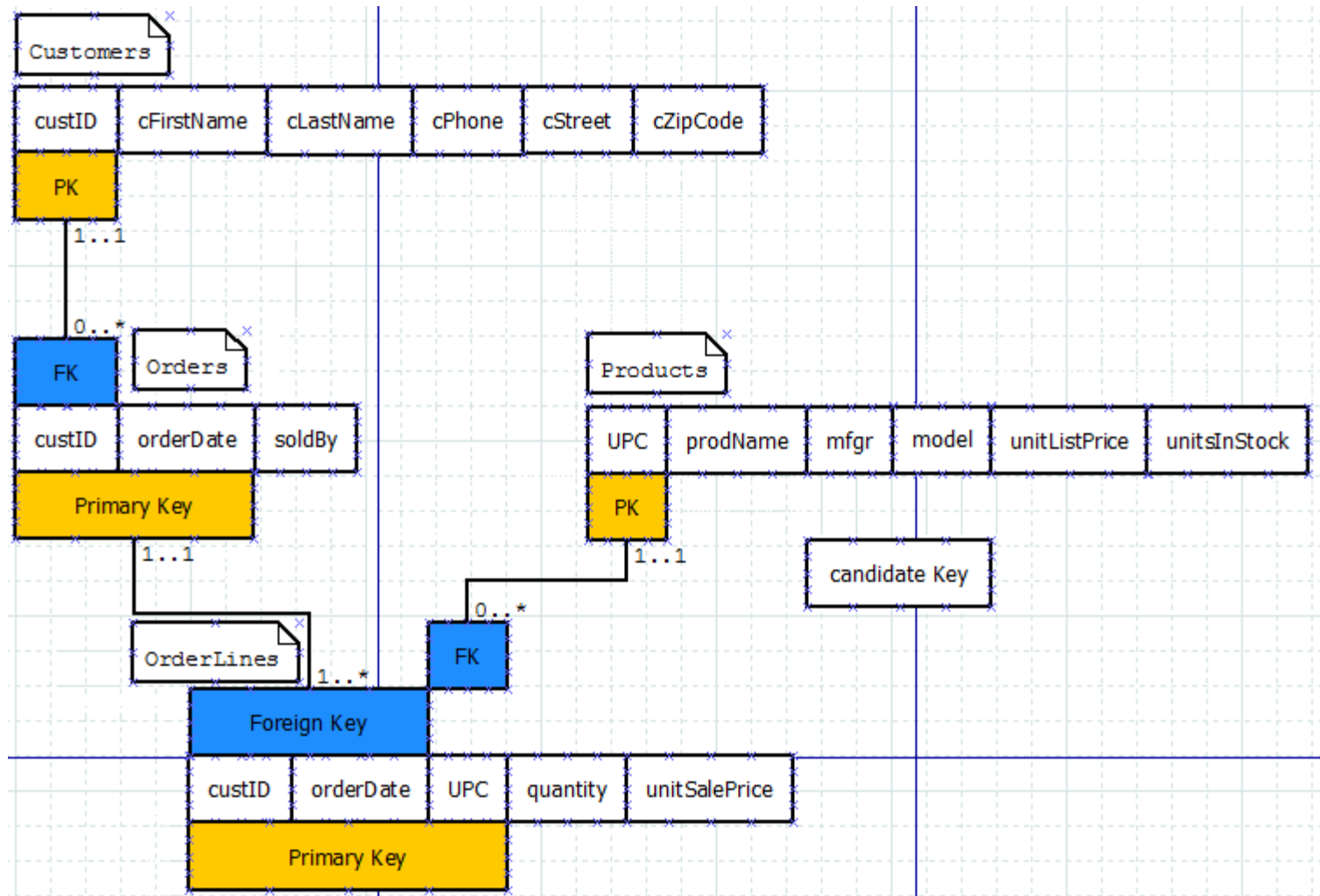
Design Pattern: Many-to-Many (Order Entry) – Junction Table

- 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.
- Hence the need to have the association class in UML every time, it means that we have a UML class for every relation scheme.

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



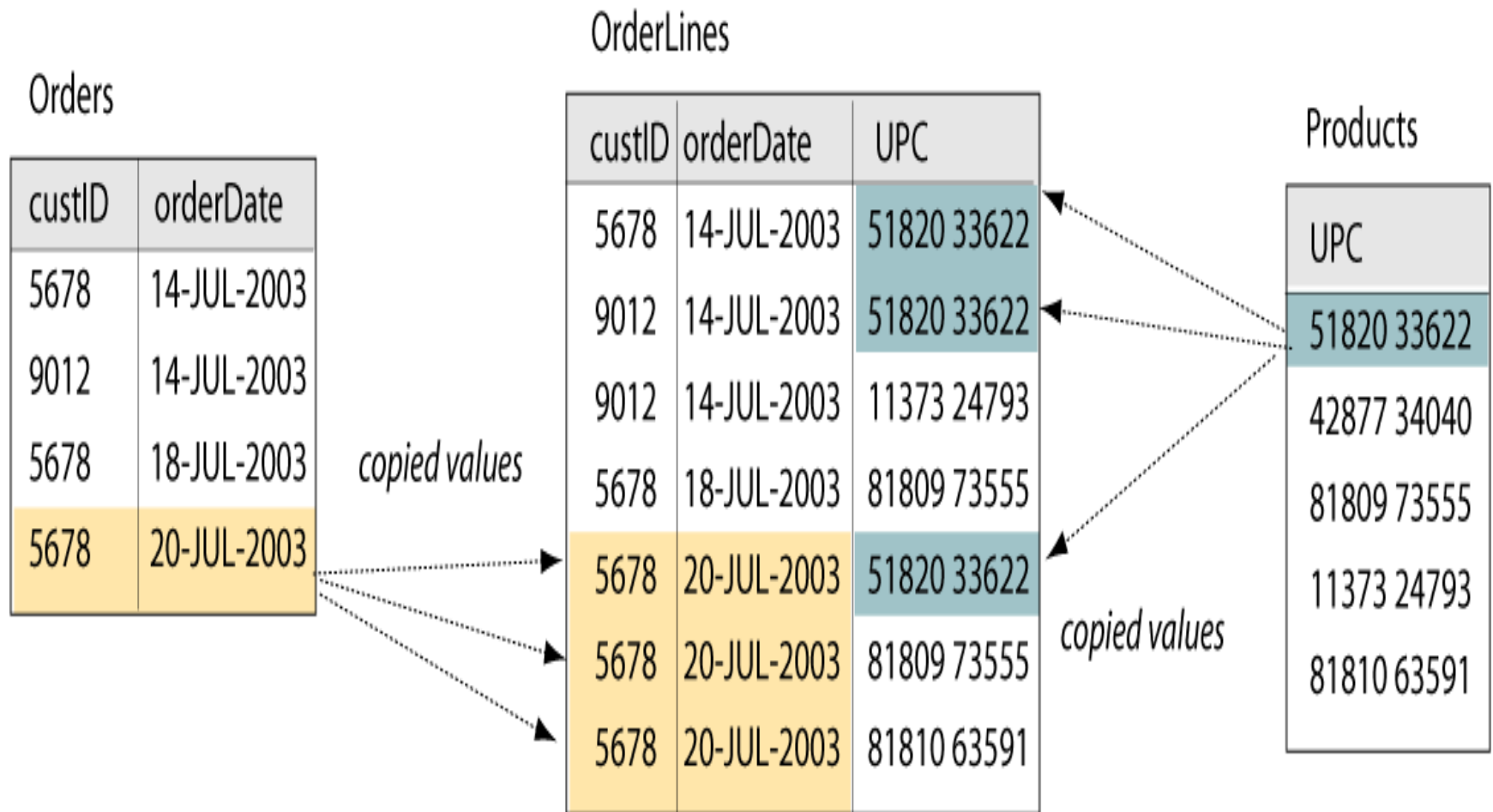
Yet More Style (rows of keys)




Foreign Key Rows in Relation Scheme Diagram

- Each key, no matter what sort, needs to have its own row in the diagram.
- Notice how the two foreign keys for OrderLines (one for the custID and OrderDate versus the other one for UPC) have a separate row of keys in the diagram.
- This helps make it clearer which foreign keys are coming from which relationship.
- This is particularly important when the key attributes from a given relationship cannot all be adjacent to each other in the diagram.

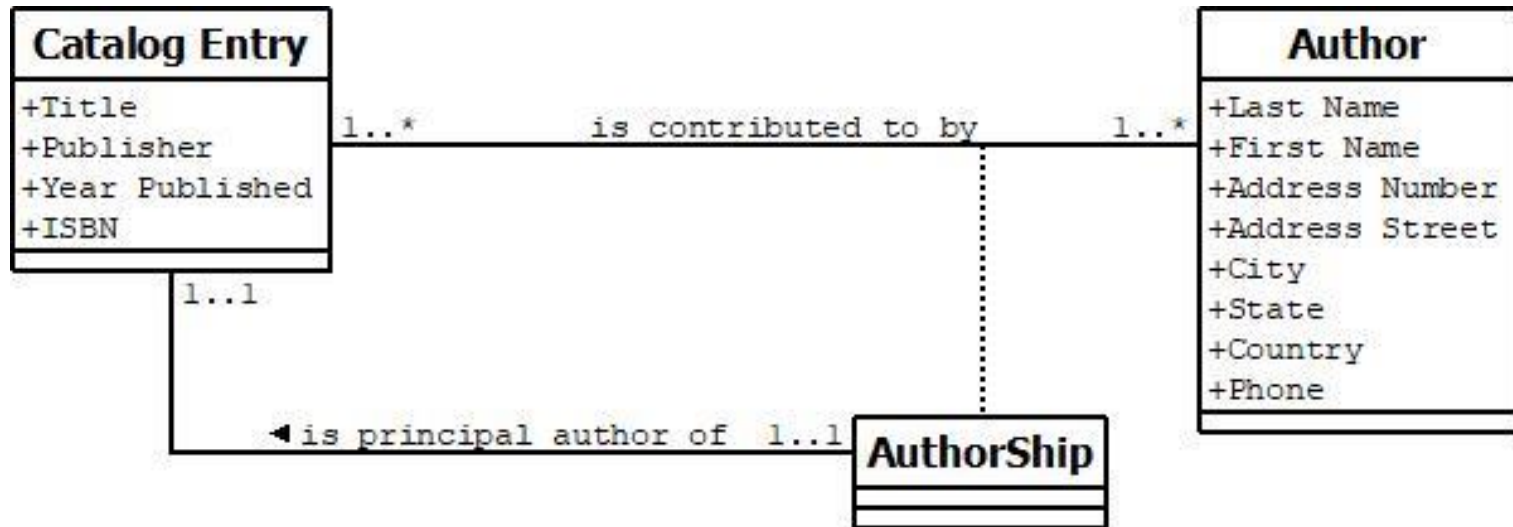
Design Pattern: Many-to-Many (Order Entry) – Data Representation



Surrogate Key in the Junction Table

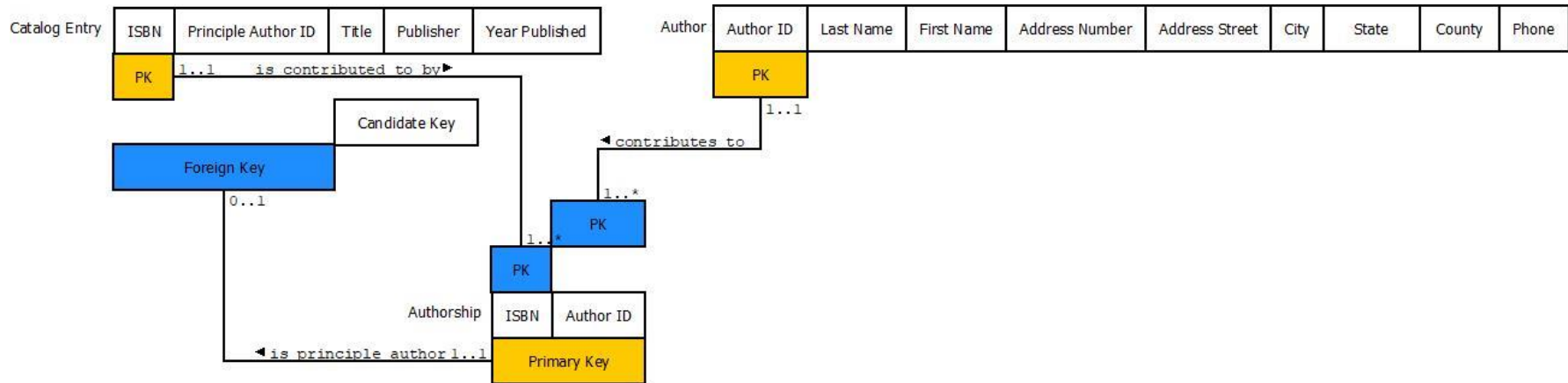
- Normally speaking, the primary key of a many to many without history is strictly the union of the primary keys coming in from the two parents.
- **But**, if that junction table is a parent to other tables, and you decide to give it a surrogate key, that is acceptable **only if** the set of keys migrating in from the two parents is a candidate key.
-  Failure to capture the candidate key when you add a surrogate key will cost you points every time.

Many to Many “loopback”



- We want just **one** of the authors designated as the principal author.
- We want to treat the principal author the same way that we do any other.

The Relation Scheme Diagram



- ❑ The ISBN in Authorship comes back into Catalog Entry and must have the same value.
- ❑ This is what I call Foreign Key collision, and we will see this pattern come up over and over.

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 later. 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

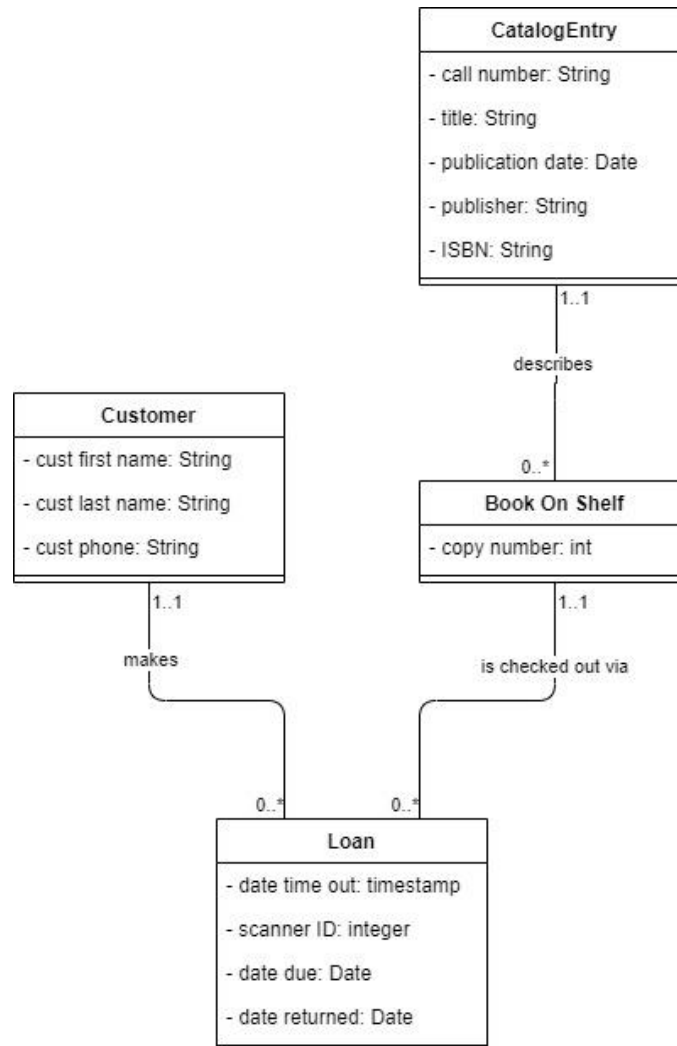
The loan is an event that happens in the real world. We need a regular class to model it correctly. We'll call this the "library loan" design pattern. First, we need to understand what the classes and associations mean:

- ❑ "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." Think "book on shelf".
- ❑ "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.”

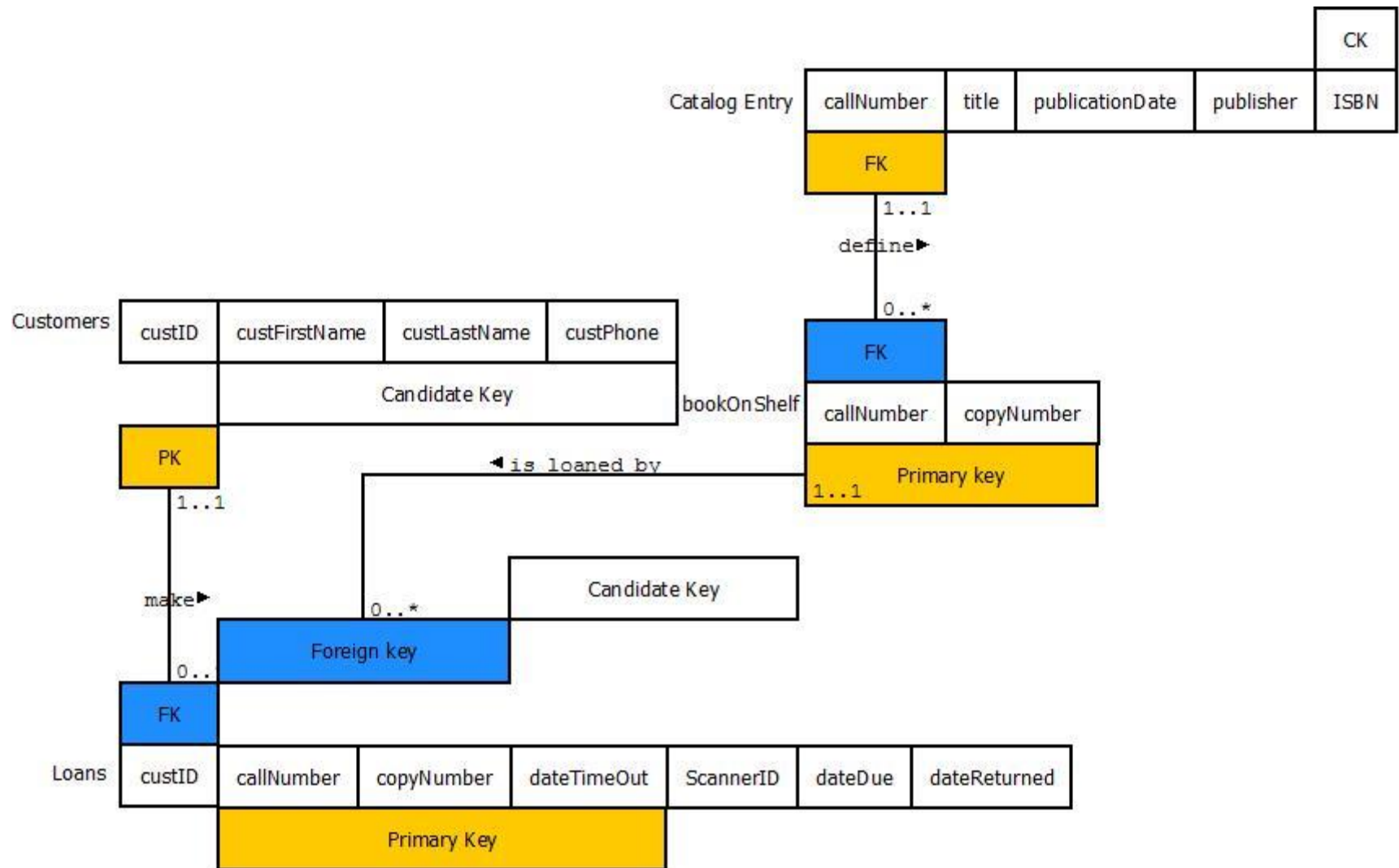
Design Pattern: Many-to-Many With History(Library Loan)–Class Diagram



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



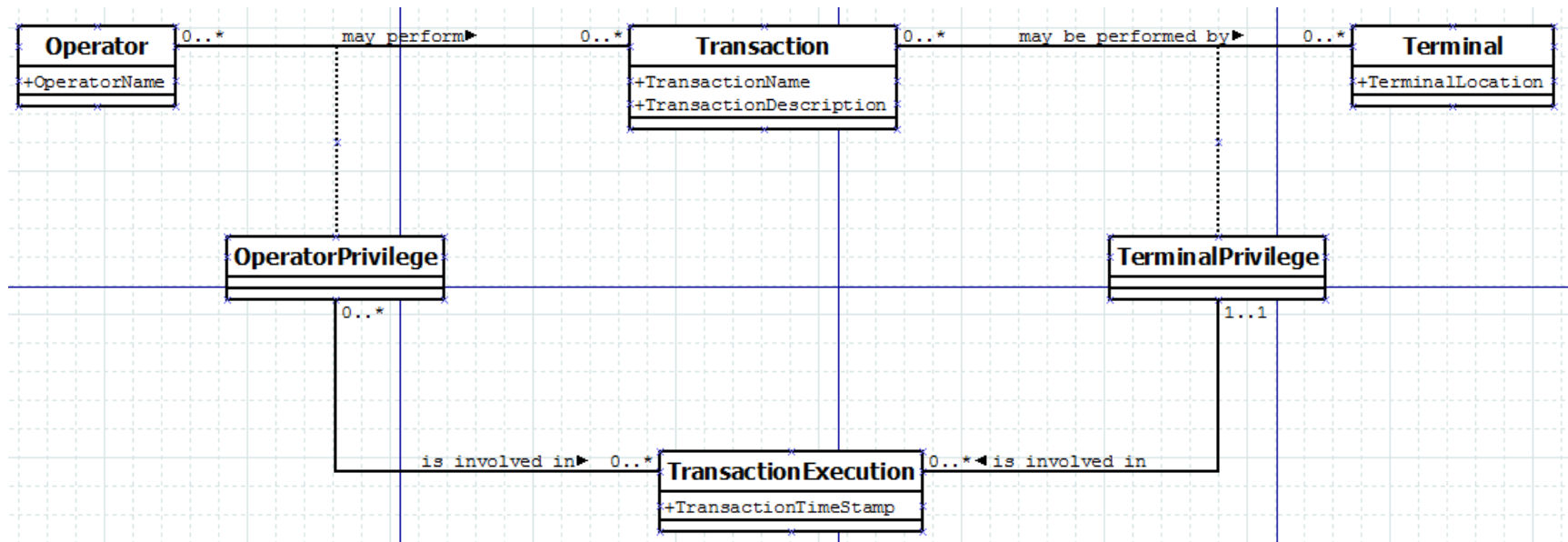
Design Pattern: Many-to-Many With History (Library Loan)

- ❑ 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 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).

Colliding Foreign Keys

- ❑ Occasionally a connection between two classes can occur along more than one relationship path.
- ❑ When that happens, as a designer, you need to decide whether the instance of that common ancestor must be the same for two or more of those paths, or whether it should be different.
- ❑ Unfortunately, UML, because it does not deal with migrating foreign keys, has no way to capture this design decision.
- ❑ However, the relation scheme diagram **will** allow you to capture this aspect of your design.

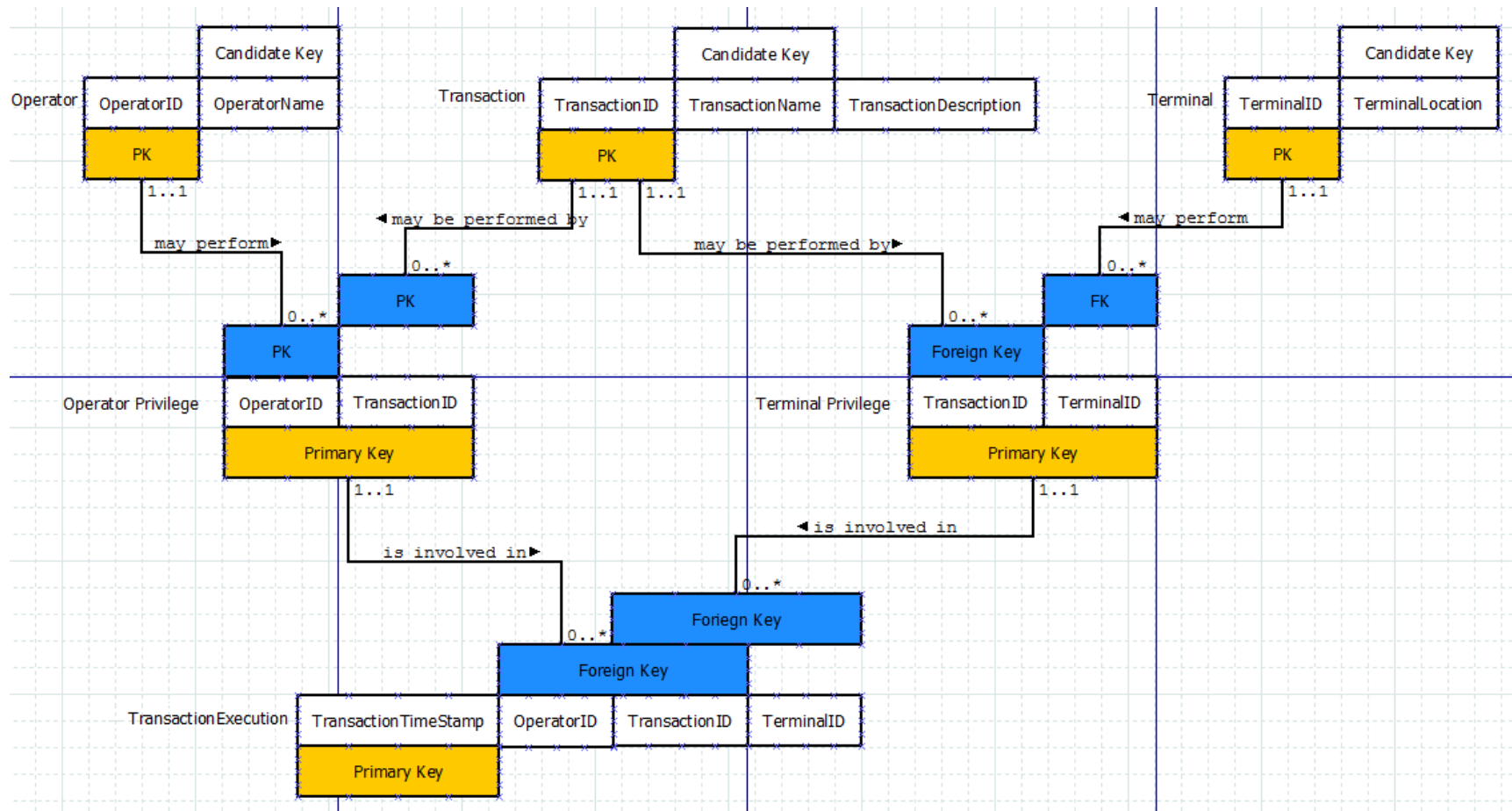
Colliding Foreign Keys Example



- ❑ Back in the day, we used to authorize certain transactions to certain hard-wired terminals.
- ❑ In addition, only certain operators could perform certain transactions.
- ❑ A given transaction execution was then an intersection between the authorization of a terminal and an operator.

Thomas Bruce *Designing Quality Databases with IDEF1X Information Models*
Dorset House Publishing 1992

Colliding Foreign Keys in the Relation Scheme Diagram



- ❑ The TransactionID migrates through OperatorPrivilege as well as TerminalPrivilege.
- ❑ The assertion is that it has to be the same transaction from both sources in order for the transaction to be valid.
- ❑ Note that you cannot capture that constraint without the relation scheme model.

And Finally, the DDL

```
create table operator (  
    OperatorID          int not null generated always as identity  
                        (start with 1, increment by 1) constraint  
operator_PK primary key,  
    OperatorName        varchar(100) not null constraint operator_UK01  
unique);  
  
create table "transaction" (  
    TransactionID       int not null generated always as identity  
                        (start with 1, increment by 1),  
    TransactionName     varchar(100) not null constraint transaction_UK01  
unique,  
    TransactionDescription  varchar(500),  
    constraint transaction_PK primary key(TransactionID));  
  
create table terminal (  
    TerminalID          int not null generated always as identity  
                        (start with 1, increment by 1) primary key,  
    TerminalLocation     varchar(100) not null constraint terminal_UK01  
unique);
```

DDL Continued

```
create table operatorPrivilege (  
    OperatorID      int not null constraint Operator_OperatorPrivilege  
                    references Operator (OperatorID),  
    TransactionID   int not null constraint Transaction_OperatorPrivilege  
                    references "transaction" (TransactionID),  
    constraint      operatorPrivilege_PK primary key (OperatorID, TransactionID)  
);  
  
create table terminalPrivilege (  
    TransactionID   int not null constraint Transcation_TerminalPrivilege  
                    references "transaction" (TransactionID),  
    TerminalID      int not null constraint Terminal_TerminalPrivilege  
                    references terminal (TerminalID),  
    constraint      terminalPrivilege_PK primary key (TransactionID, TerminalID))
```

DDL Continued

```
create table transactionExecution(  
    transactionTimeStamp    timestamp not null constraint  
                           transactionExecution_PK primary key,  
    OperatorID              int not null,  
    TransactionID           int not null,  
    TerminalID             int not null,  
    constraint              operatorPrivilege_transaction_FK01 foreign key  
                           (OperatorID, TransactionID)  
                           references OperatorPrivilege  
                           (OperatorID, TransactionID),  
    constraint              terminalPrivilege_transaction_FK01 foreign key  
                           (TransactionID, TerminalID)  
                           references terminalPrivilege  
                           (TransactionID, TerminalID));
```

- ❑ Notice that TransactionID comes in to transactionExecution twice, once from OperatorPrivilege, and once from TerminalPrivilege.
- ❑ Since we do **not** change the name of the attribute in the child table, the value of TransactionID coming from those two sources must agree.

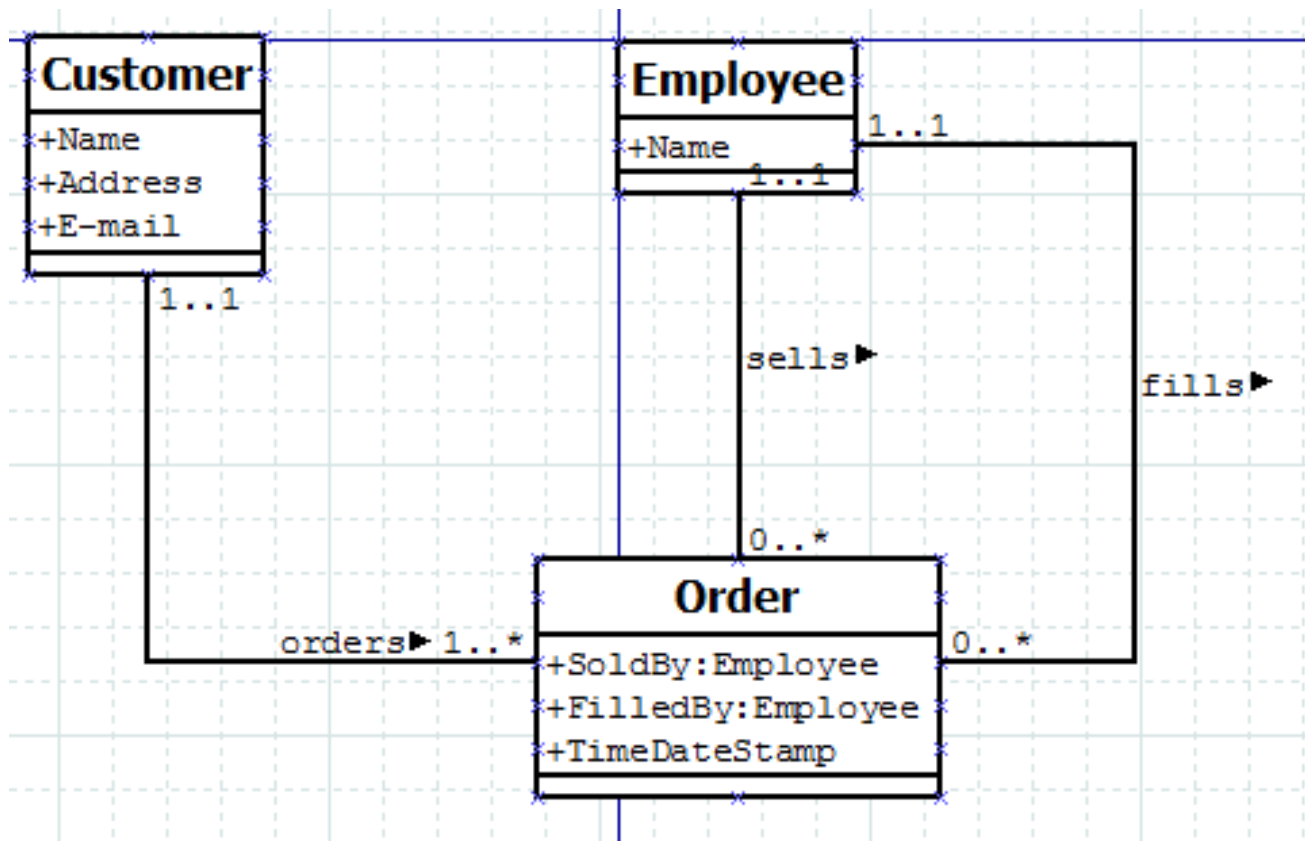
Additional Points

- ❑ We could have role named the TransactionID coming from each relationship and used the same name in both cases, and that would have achieved the same constraint.
- ❑ In this case, there was a surrogate in the common ancestor, but you could have a case with multiple members of the key.
- ❑ In such a case, you **could** role name just some of the common ancestor's keys for clarity.

What Happens With Surrogates?

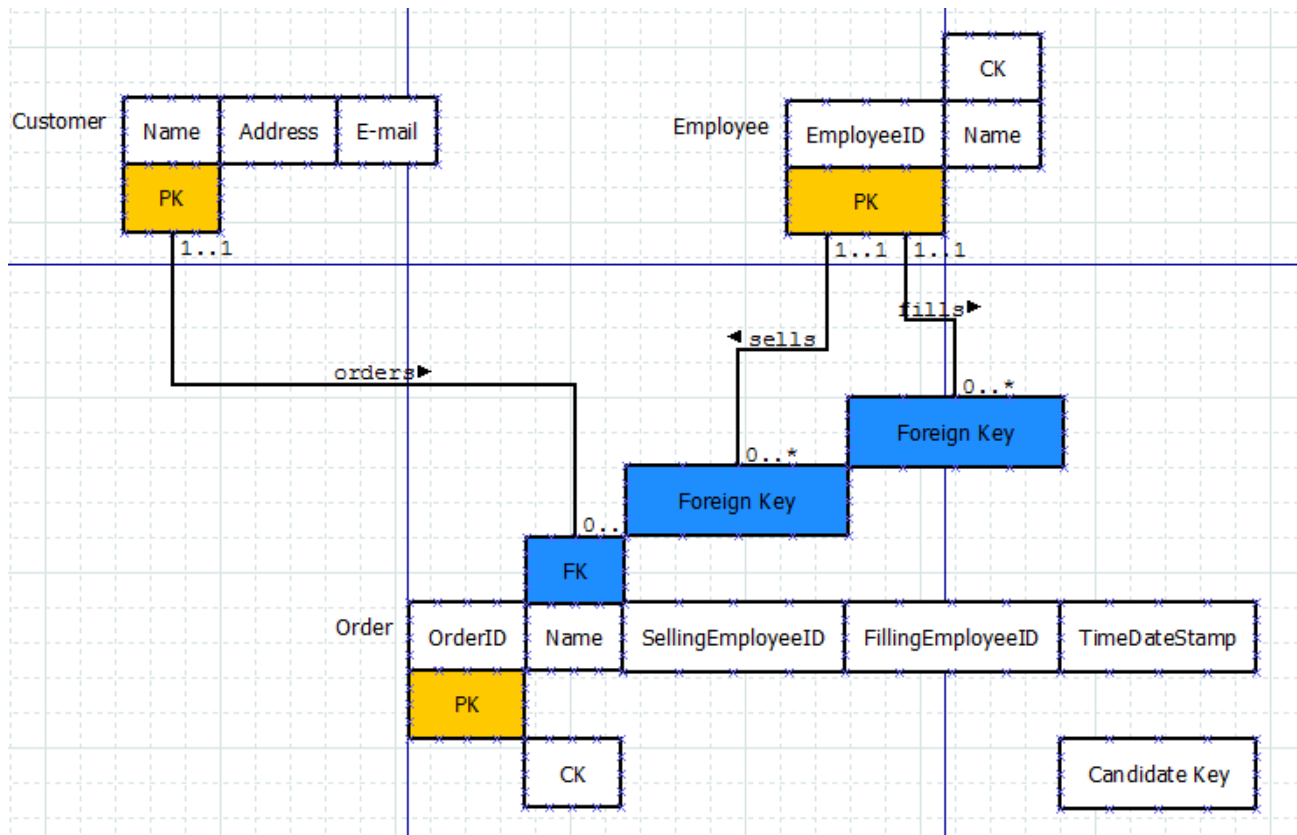
- ❑ Surrogates encapsulate.
- ❑ The relationship from the parent is to the child is **non**-identifying if the child has a surrogate.
- ❑ You **could** still get the same constraint using a 3rd generation language trigger.
- ❑ The database structure is more declarative, easier to understand.
- ❑ One more thing to consider when deciding whether to go with surrogates or natural keys.

Role naming “out of the way”



- The employee plays two separate roles for the order.
- UML allows us to specify the type of our attributes, and in this case, it's worthwhile to indicate that each of these roles is of the Employee type.

The Relation Scheme Diagram



- There is a separate attribute in Order for each of the employees related to the order.
- We only **had** to role name one of them, but I chose to role name both in order to be more explicit.

And Finally the DDL

```
create table customer (  
    name            varchar(100) not null constraint customer_PK primary key,  
    Address         varchar(100) not null,  
    E_mail          varchar(100) not null);  
  
create table employee (  
    EmployeeID      int not null generated always as identity  
                    (start with 1, increment by 1)  
                    constraint employee_pk primary key,  
    Name            varchar(100) constraint employee_UK01 unique);
```

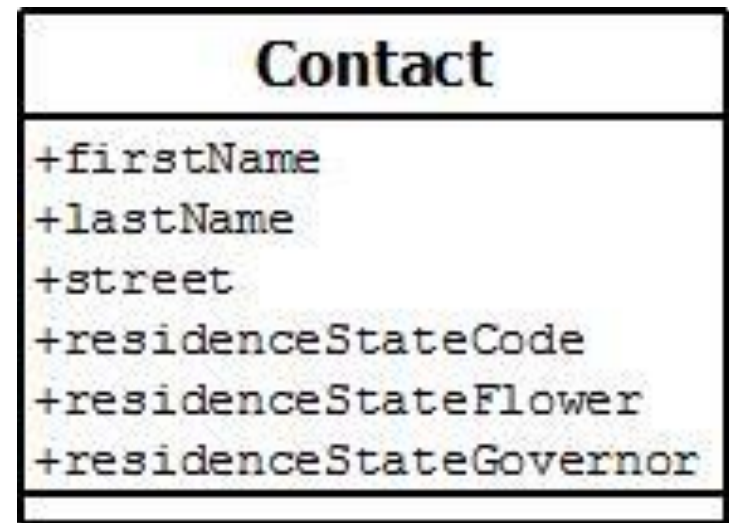
DDL for the Order Table

```
create table "order" (  
    OrderID                int not null generated always as identity  
                           (start with 1, increment by 1)  
                           constraint order_pk primary key,  
    name varchar(100)      not null constraint customer_order_fk01  
                           references customer(name),  
    sellingEmployeeID      int not null constraint employee_order_fk01  
                           references employee (employeeID),  
    fillingEmployeeID      int not null constraint employee_order_fk02  
                           references employee (employeeID),  
    TimeDateStamp          timestamp not null,  
    constraint order_uk01 unique (name, timeDateStamp));
```

- ❑ The employee ID comes in through two separate relationships: one sells the order, the other employee fills it (pulls stock, prepares for shipping, ...)
- ❑ Both of those employees **could** be the same person
- ❑ But they **don't** have to be.

Design Pattern: Subkeys

- ❑ 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 residence state code.
 - Contact residence state flower.
 - Contact residence state Governor.



Design Pattern: Subkeys - Problem

- It does not take a great deal of imagination to see where this can cause problems:
 - The state information is redundant, so it is taking up more room than is strictly necessary.
 - Because the state information is repeated, inconsistencies can come up.
 - In order to prevent those inconsistencies, any time that you change the governor, for instance, you have to be sure to change that value for all of the rows in which that state is referenced.
- By this time, it should be intuitive what to do to fix this.

Design Pattern: Subkeys – 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 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 – Functional Dependency (continued).

- 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 – 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 – 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 – 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.

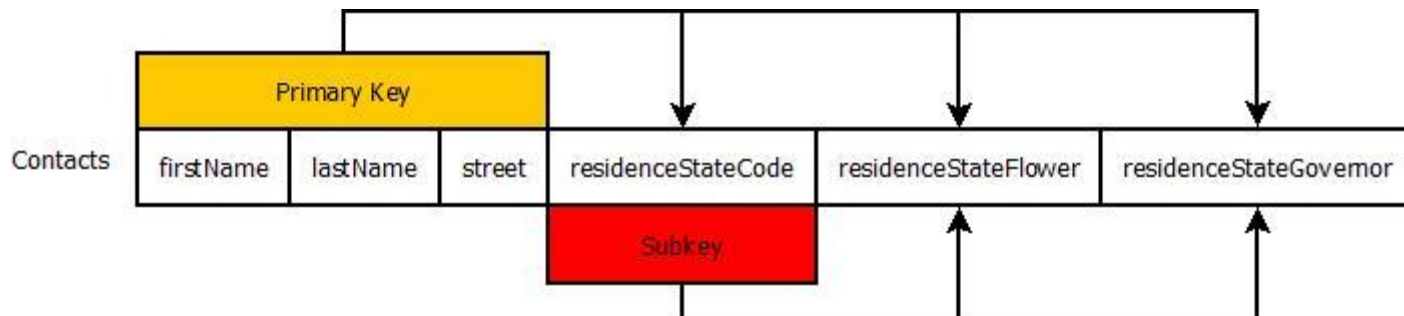
Design Pattern: Subkeys - Solution

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.

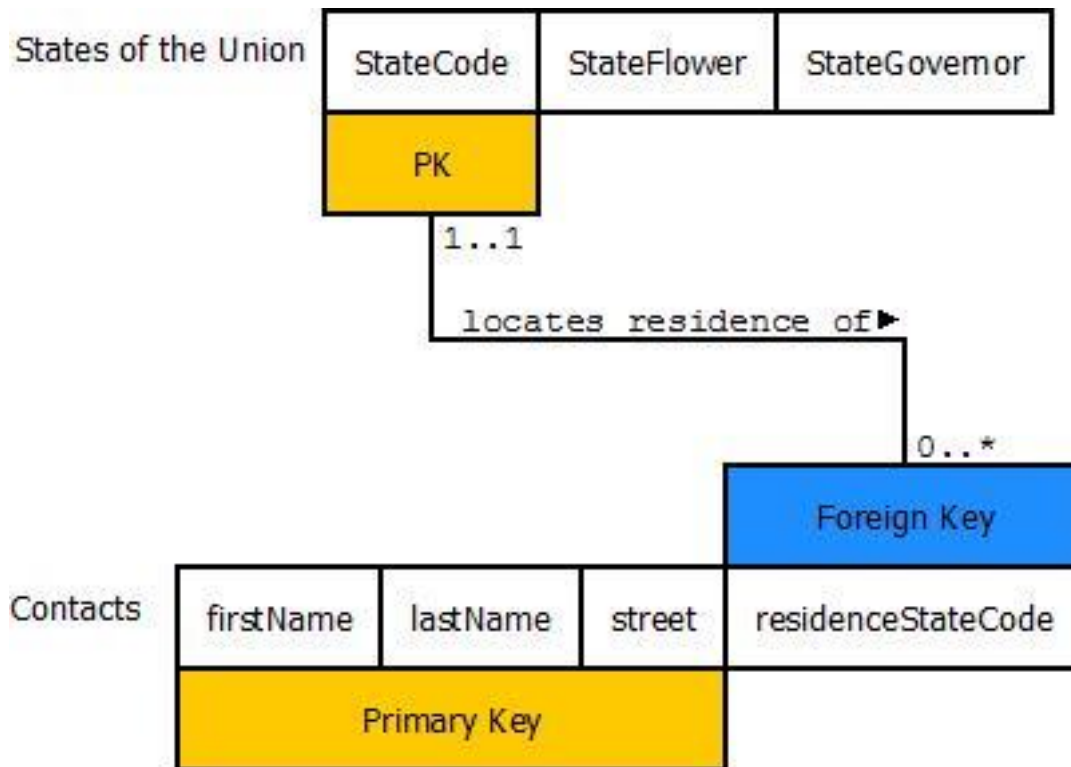
Design Pattern: Subkeys - Subkey

In the Contacts table, the residenceStateCode is a subkey. It functionally determines the state flower and the state governor. The opposite is not true because there is nothing preventing two states from having the same flower, or two state governors having the same name.



Design Pattern: Subkeys – 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



Another Benefit of Subkey resolution

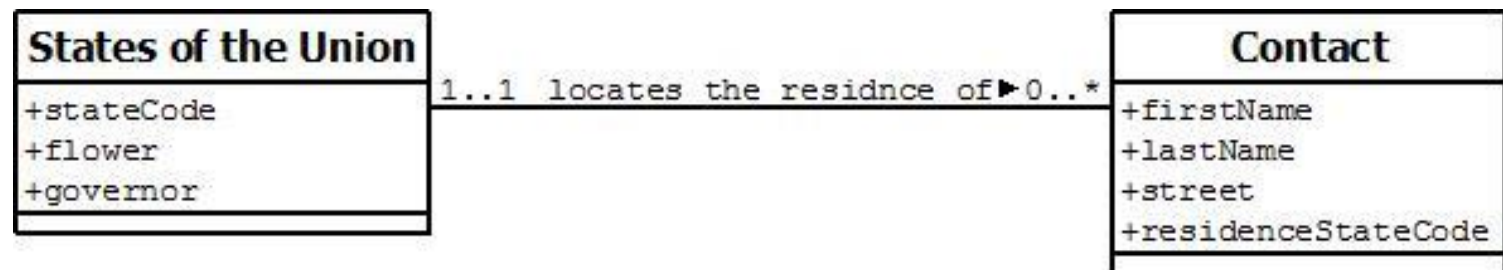
- ❑ If you then chose to have a second reference to the state, the birth state of the contact for instance, that would merely require adding another migrating foreign key (with appropriate role name).
- ❑ Further, the States of the Union table could be used in other relationships to yet other tables.
- ❑ This sort of information, which can serve many purposes, is often called “Reference Data” because it is referred to from multiple places.

Design Pattern: Subkeys - 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– 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 residenceStateCode is not just an attribute of Contact, it's the role named key of States of The Union.
- As with all one-to-many associations, the association itself identifies which Contact lives in which State of the Union. 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!

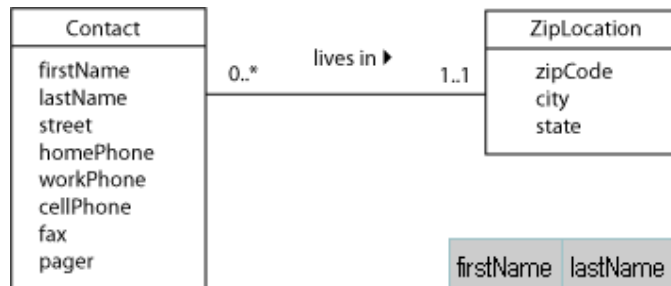


Detecting Functional Dependencies

- ❑ So far, the functional dependencies were easily detected based on business rules.
- ❑ If you are presented with a data table of values, and you are looking for subkeys:
 - If I know (the value of some collection of attributes) then I know (the value of some other collection of attributes) then a functional dependency exists.
 - Sort the data by the suspected subkey values
 - Then, if every time a row pops up with a specific set of values for the suspected subkey, one or more other attributes have the same value, then you might have a functional dependency.

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.

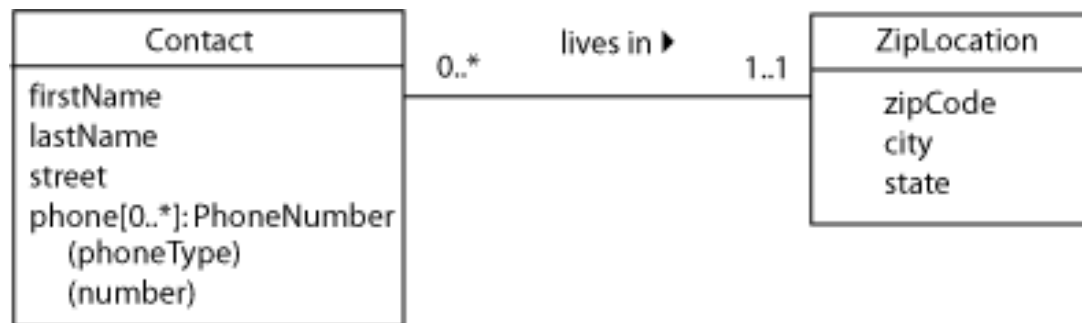


Contact phones

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 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.



Another option

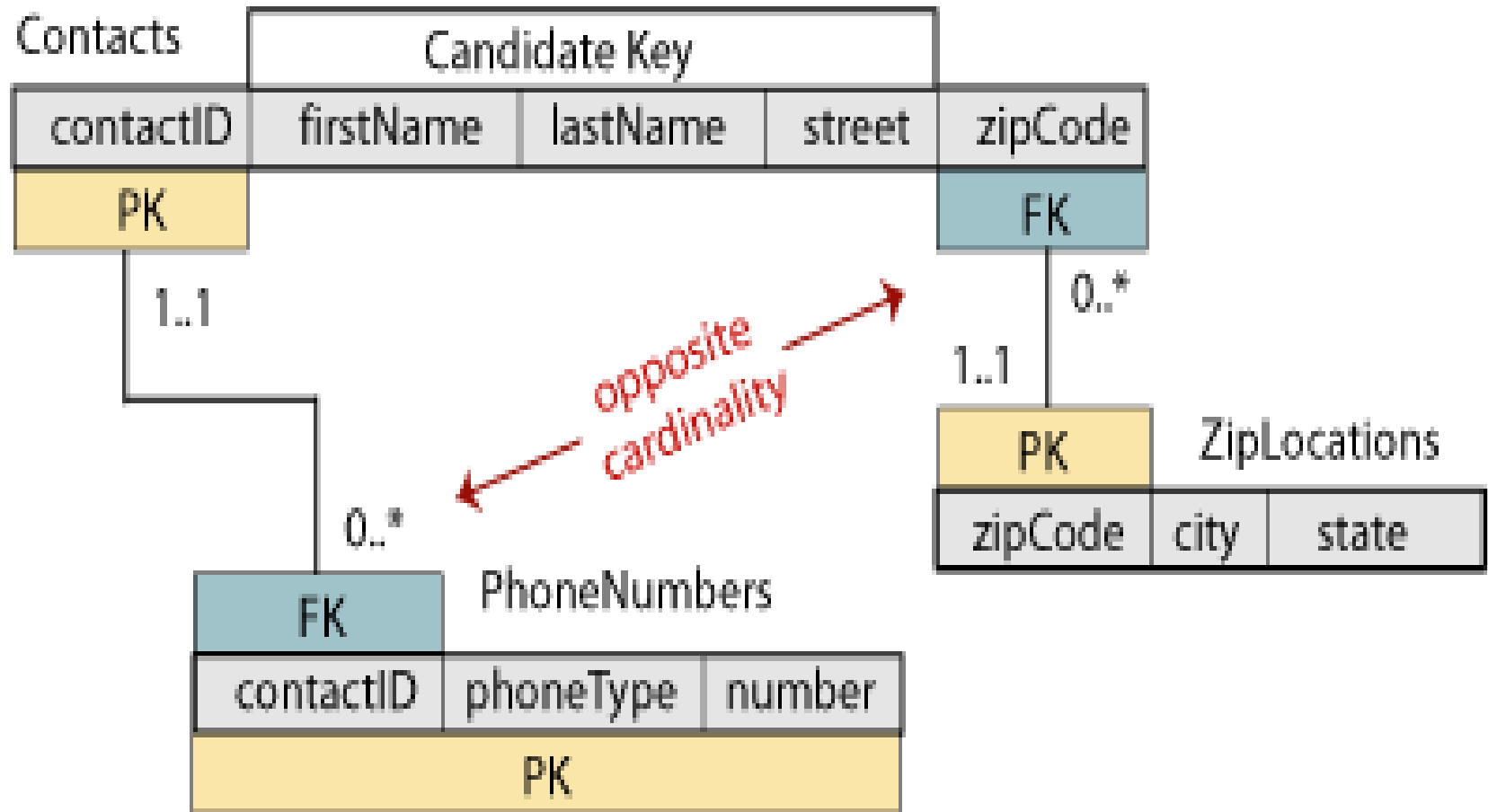
- When you find that a given class has attributes or associations which are optional, and at least several of them are optional together, or they are mutually exclusive, it may be time for a subtype.
- For instance, student could be a graduate student, and a given graduate student could be a teaching assistant (which needs the course number that they are assisting on) or a research assistant (which needs the grant number).

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

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

1. Remove all 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

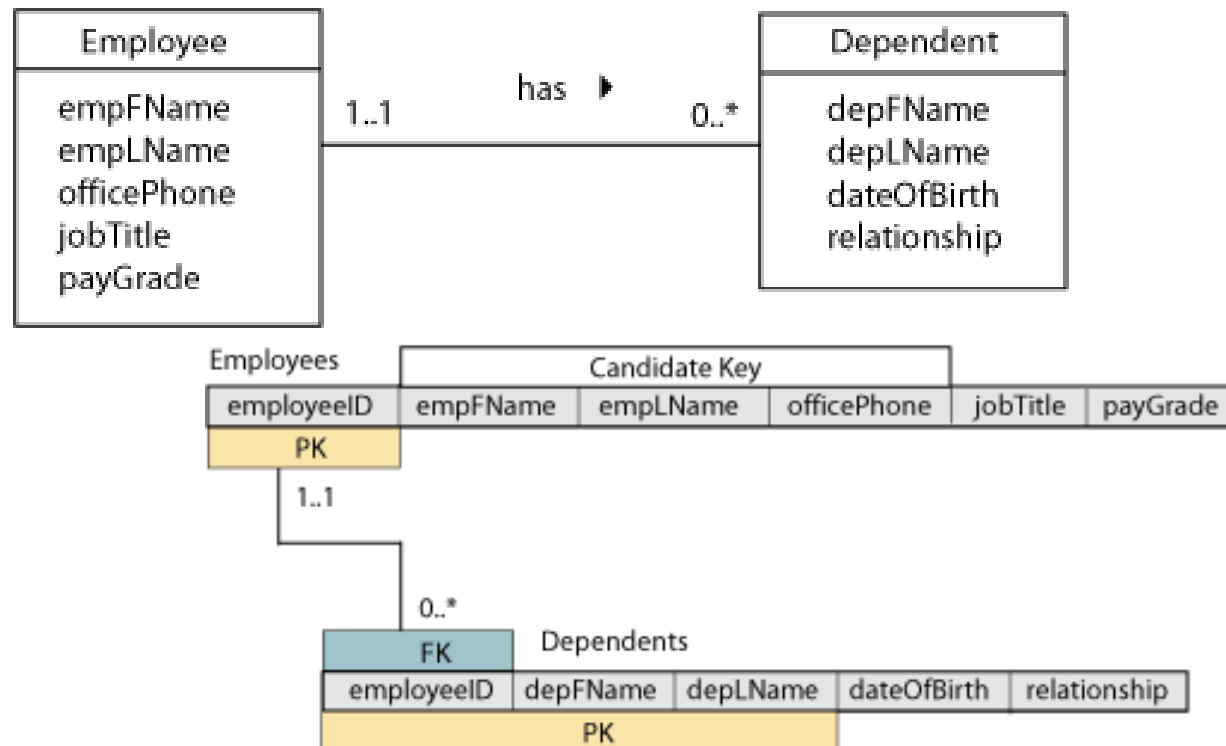
contactid	firstname	lastname	street	zipcode
1639	George	Barnes	1254 Bellflower	90840
5629	Susan	Noble	1515 Palo Verde	90840
3388	Erwin	Star	17022 Brookhurst	92708
5772	Alice	Buck	3884 Atherton	90836
1911	Frank	Borders	10200 Slater	92708
4848	Hanna	Diedrich	1699 Studebaker	90840

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

Another Example

Instructor Name	Degree 1	Degree 2	Degree 3	Degree 4
William Shatner	MS in Computer Science, 2016, UC Santa Barbara	PhD in Astronomy, 2018, Cal Berkeley	PhD in Xenobiology, 2020, California Institute of Technology	
Leonard Nimoy	PhD in Rocket Propulsion, 2012, MIT	PhD in Physics, 2012, Carnegie Mellon	PhD in Stellar Navigation, 2014, USC	MS in Vulcan poetry, 2016, UC Santa Barbara

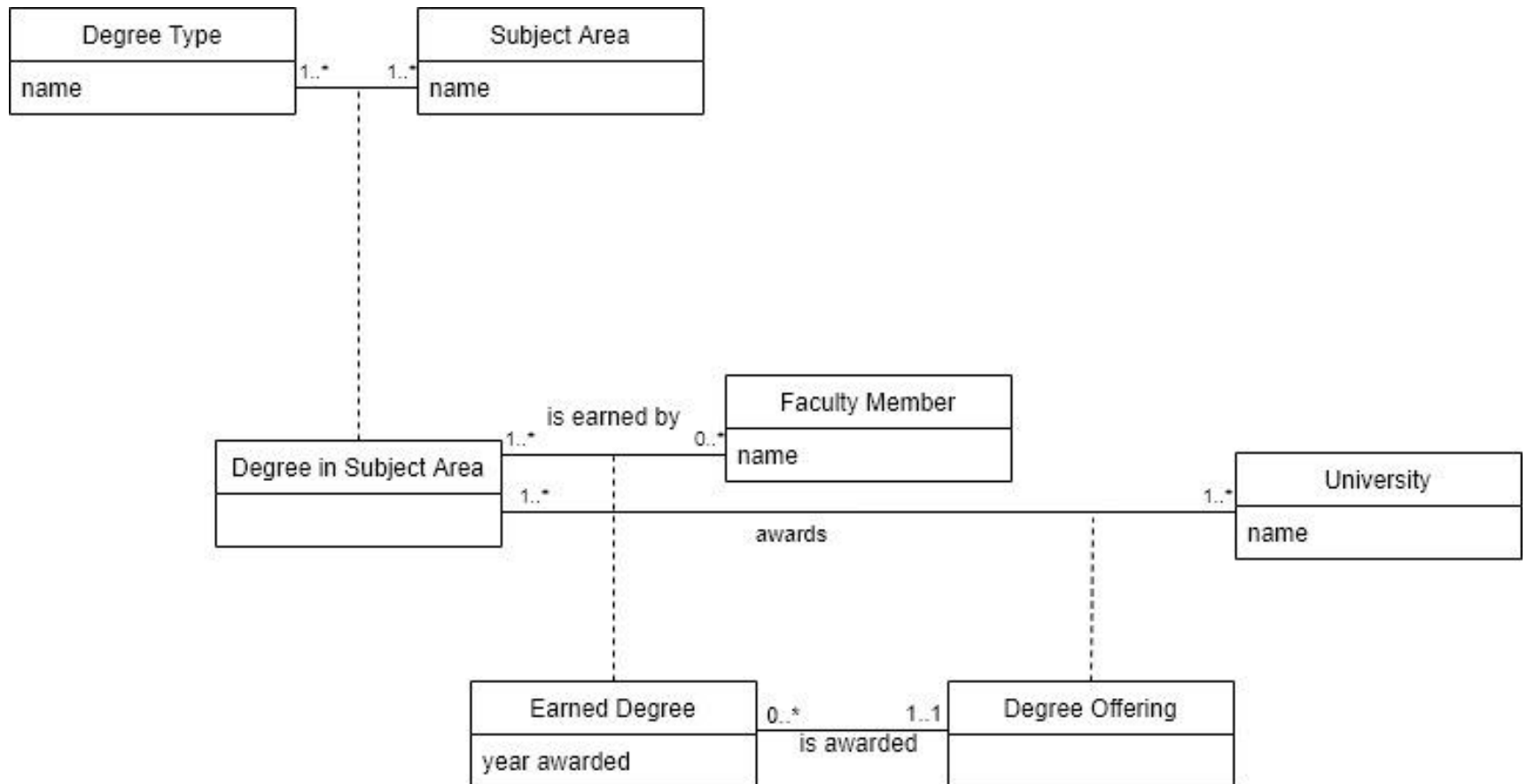
Recognizing Repeating Attributes

- ❑ Sometimes the name gives it away, as in the above case.
- ❑ Sometimes the values themselves suggest that there is a repeating attribute afoot.
- ❑ If in doubt, ask for sample data.
- ❑ And/or ask whether there is any specific format for a set of field values.

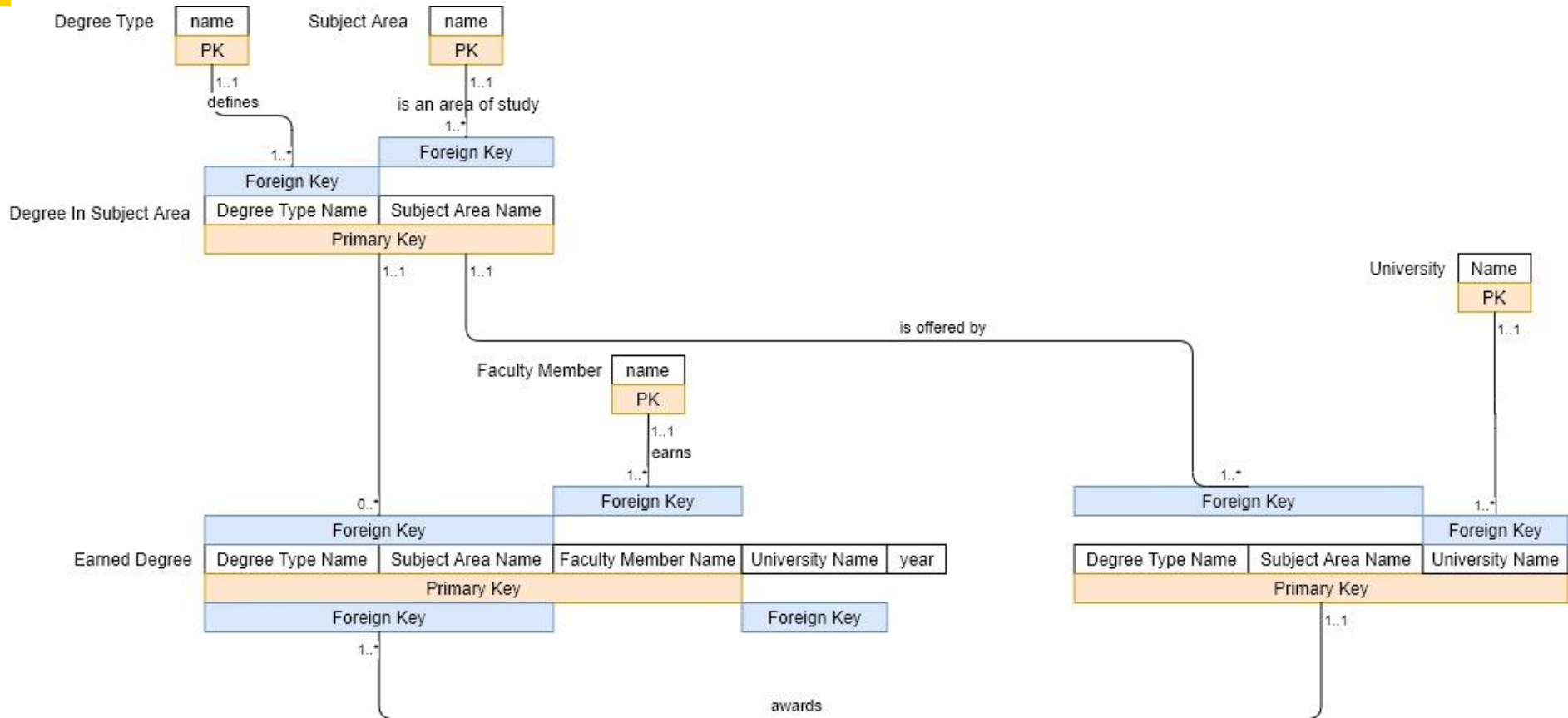
Let's put some business rules in

- ❑ A given faculty member (person, really) can only earn one degree of a given type in a given subject. For instance, you can only get a Physics BS once. Believe me, that's enough.
- ❑ Then let's also say that only certain universities offer a given degree type (BS, MS, PhD, ...) in certain subjects.
- ❑ What would that look like in the UML?

UML



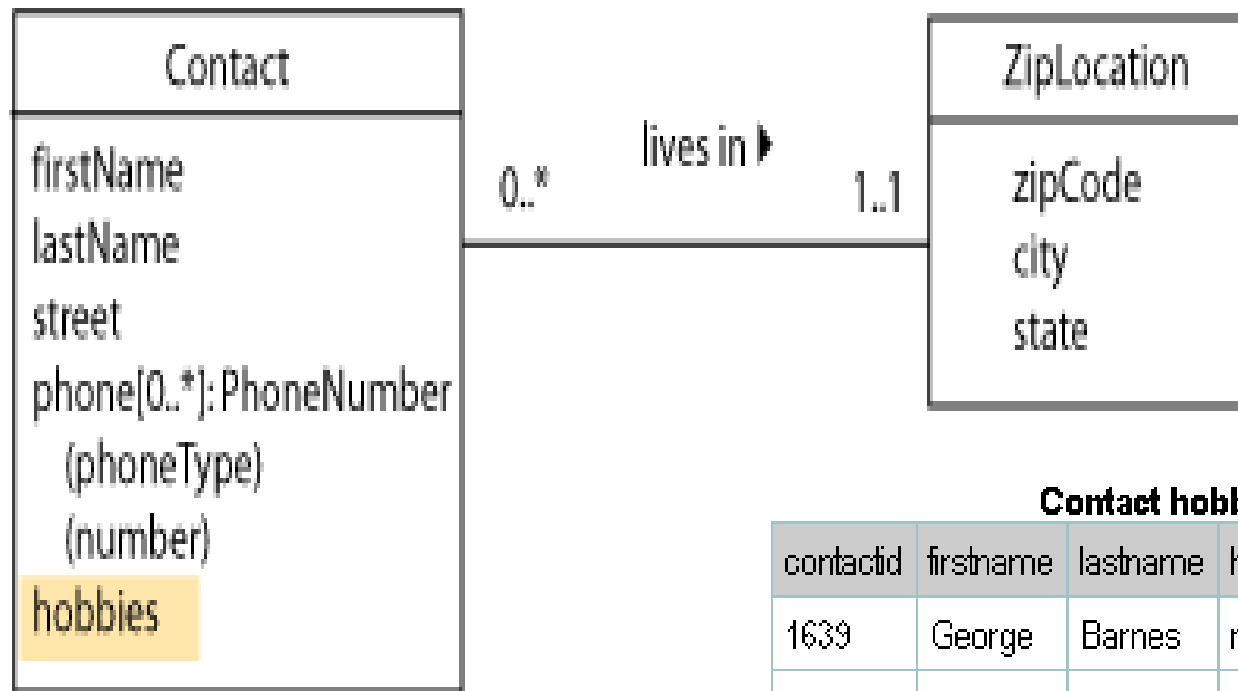
Relation Scheme



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 specific value and it is impossible to create a query that will individually list the values in that column. For example: hobbies.

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



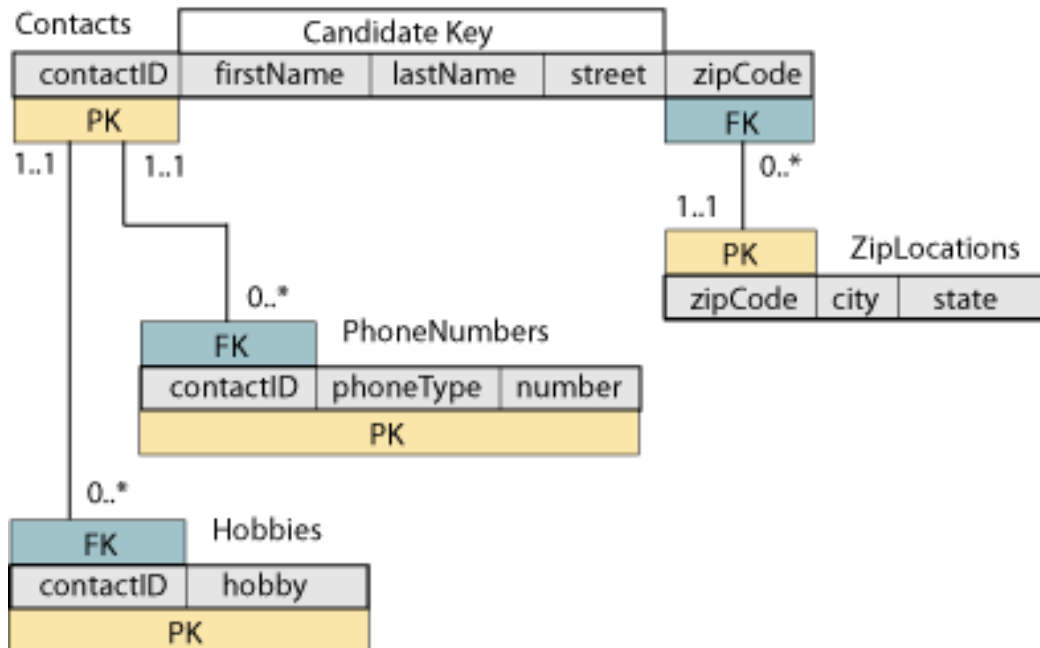
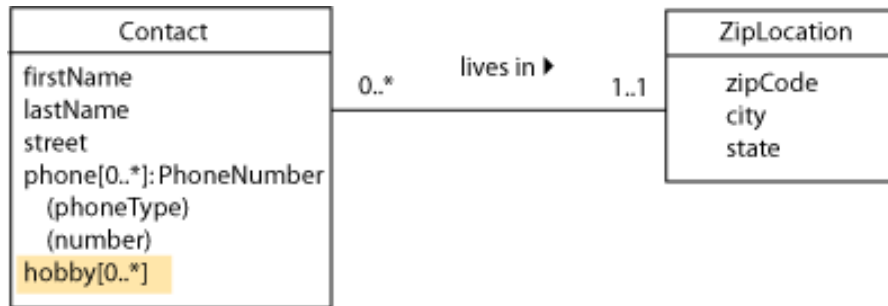
Contact hobbies

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

Multi-valued attribute issues

- ❑ It becomes very difficult to search efficiently for anyone with a given hobby.
- ❑ It is impossible to use referential integrity to make sure that only a valid hobby is entered.
- ❑ There is no library of valid hobbies to use as a drop down to aid in data entry.
- ❑ It is pointless to index the hobbies list.
- ❑ When this course started, I saw several of you making designs that seemed to imply a multi-valued assumption. Now we'll see how to do better than that.

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



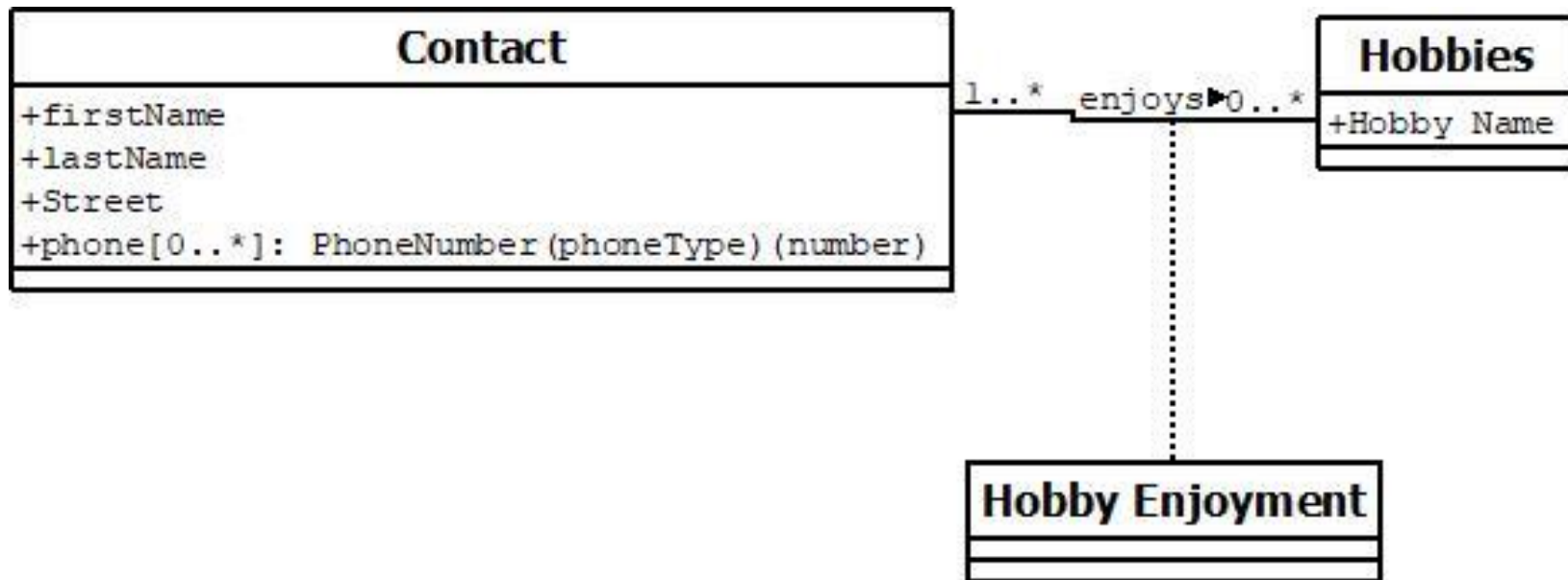
Hobbies

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

Signs that you have a multi-valued attribute

- ❑ The value in the legacy system is variable length and can be very long.
- ❑ The attribute name is plural.
- ❑ The values have some sort of delimiter in them like , | or something else to mark the end of one value and the start of the next.
- ❑ Be sure to pay attention to the order. Something like author order might be implicit in the data and will need to be made **explicit** in the design.

Another Look at Hobbies



- ❑ With this design, we have a list of valid hobbies that we can use to make sure that they don't put in an invalid hobby name.
- ❑ The Hobbies class can be used as a drop-down menu in your GUI application.
- ❑ This is a classic many to many without history since we have only one association of a given contact to a given hobby.

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 to provide ***data integrity***, which includes 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 must 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 have such a datatype, but Derby does.
- ❑ 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, gender.

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.
- Another example of an enumerated domain is the ProductLine column in Products in our lab database.

Design Pattern: Enumerated Domains

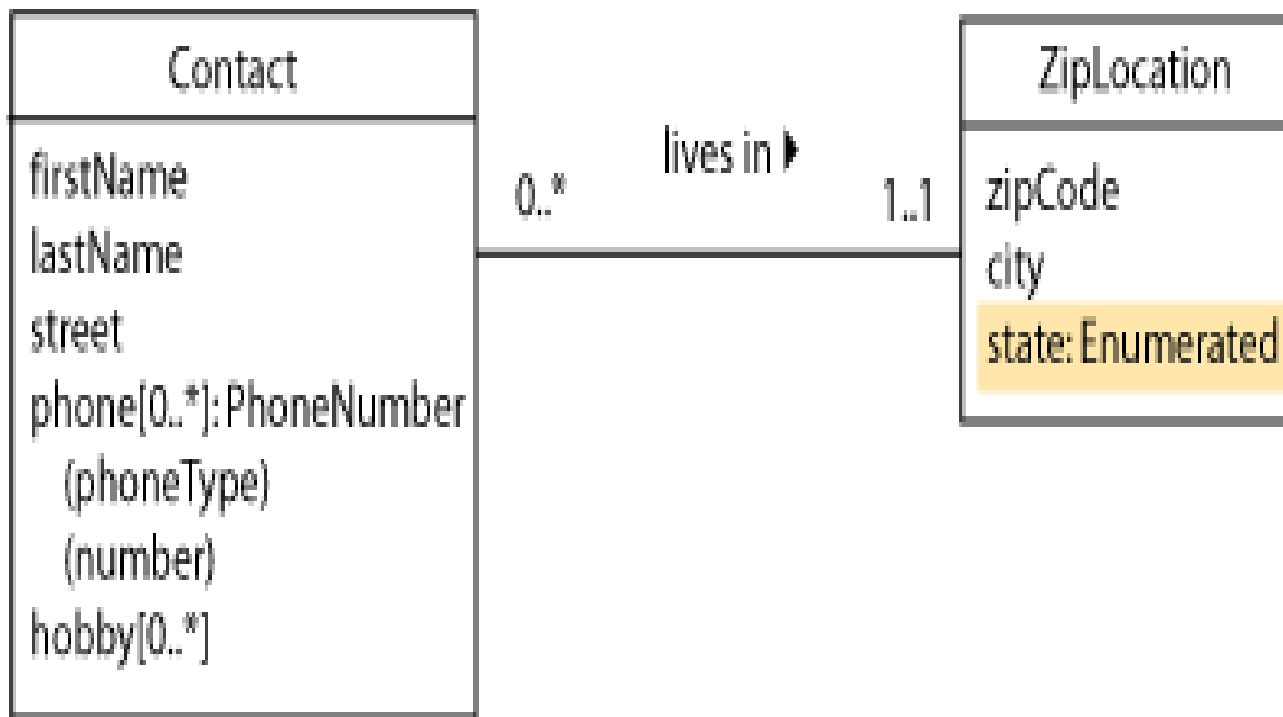
- Advantages

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.

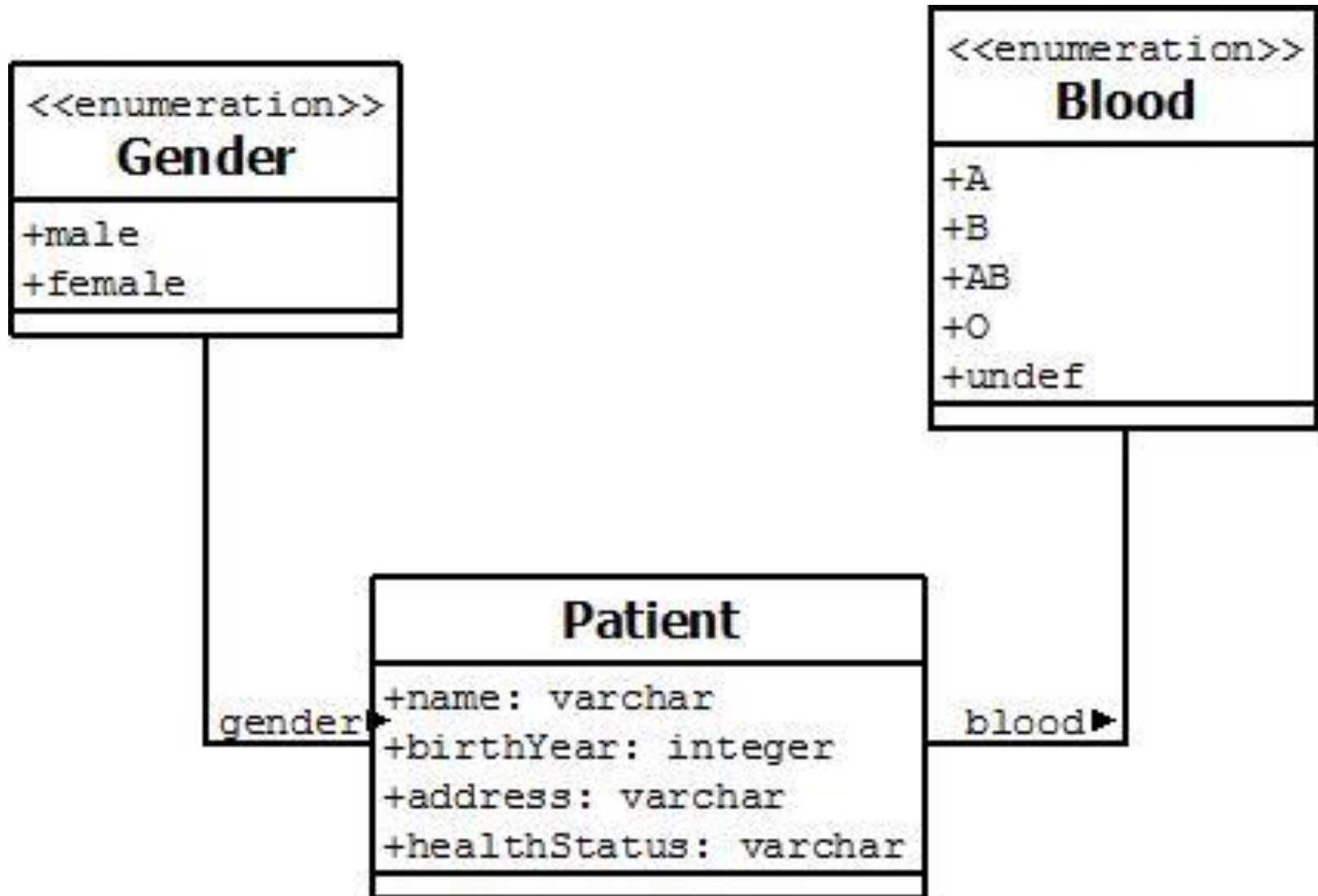
Design Pattern: Enumerated Domains

– 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 – Class Diagram (Cont)



Design Pattern: Enumerated Domains – Class Diagram (Cont)

- ❑ The <<enumeration>> at the top of the class indicates the stereotype for the class. Think of it as a class template or class type.
- ❑ The instances of the enumeration class are listed in the body of the class, in no order.
- ❑ The association from the enumeration to the Patient class (in this case) indicates that the Patient has one enumeration attribute for the gender, and another for the blood type.
- ❑ The same enumeration could be used by multiple classes.
- ❑ Note that there is no explicit attribute in the Patient class for Gender or Blood type as that is implied by the association, just like an aggregation.

One more alternative

- ❑ As if that weren't enough, you can think of the enumeration type as a type, like varchar or int.
- ❑ With that perspective in mind, you could have modeled Patient with attributes like sex: Gender and bloodType: Blood, and leave the associations out of the picture.
- ❑ The disadvantage of this method is that you then leave it to the reader to hunt down the enumeration to see what the values are.

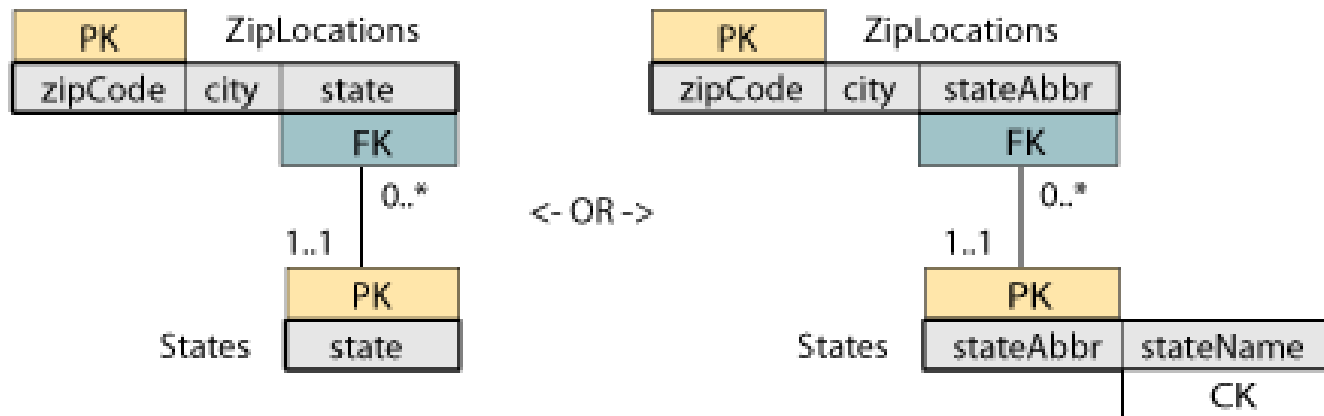
Guidance on how to diagram

- ❑ If you have an enumeration which is used in multiple classes or serves more than one role (for instance size might be used for soft drinks as well as pizzas), then use the separate class for the enumeration.
- ❑ Otherwise, it's more compact to simply indicate that the given attribute is an enumeration and leave it to the relation scheme diagram to indicate that there is a specific reference data table to support the enumeration.

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.



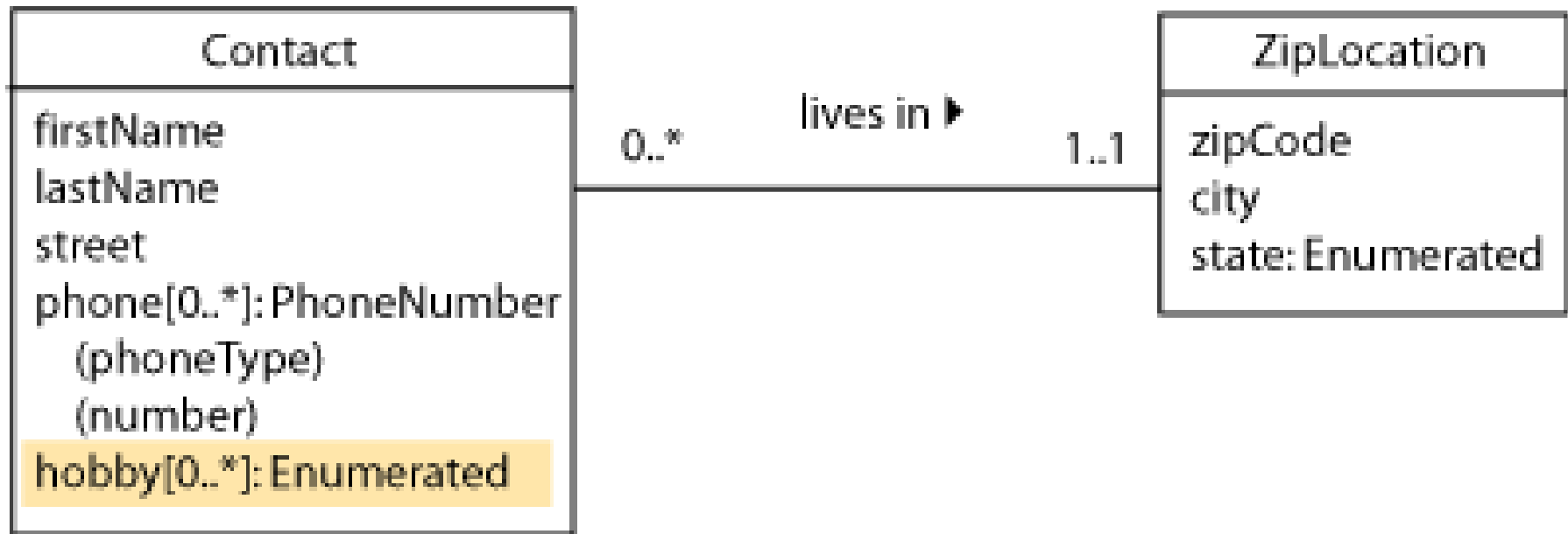
Design Pattern – Enumerated Domains, when **not** to

- ❑ Consider the state table: {stateName, stateCode}. Should we enumerate the stateCode, the stateName?
- ❑ The answer is categorically **no**, because the state table **is** the enumeration.
- ❑ The rule of thumb is, if there is only one row in the table for each value of a given column, and that column is not needed anywhere else, no enumeration is needed.
- ❑ On the other hand, if there is only one row/value for the given column, and that column is the key of the table, then you can use that table as the enumeration.
- ❑ For instance, the state table would serve as a lookup table for state code for any number of other tables.

Design Pattern: Enumerated Domains

– Multi-Valued Attributes

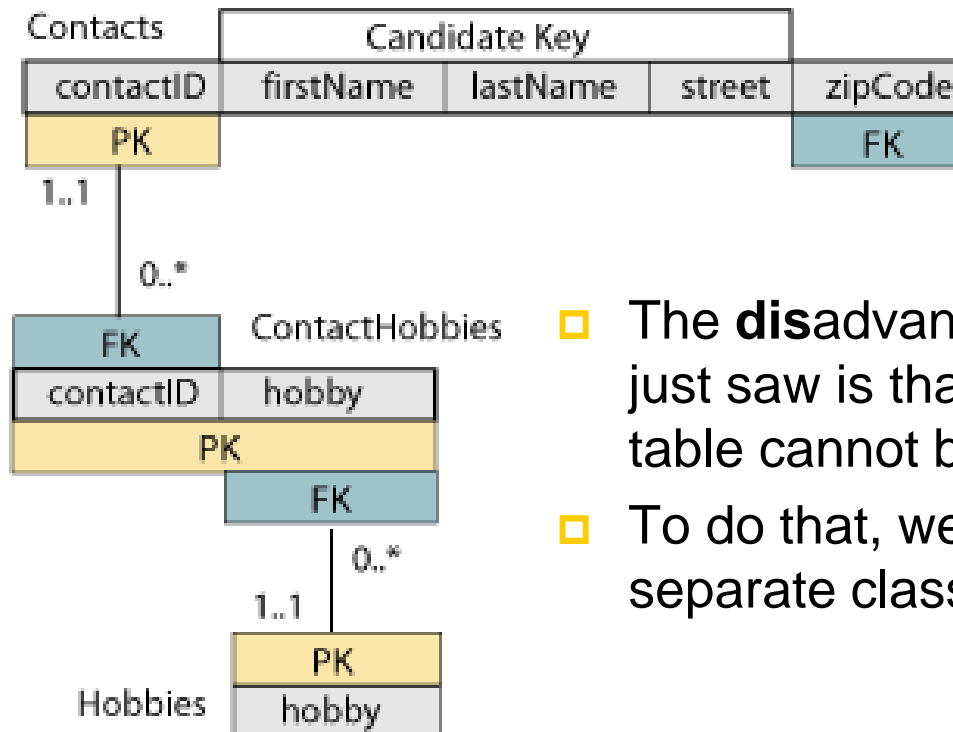
Multi-valued attributes might also have enumerated domains.



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).



- The **dis**advantage of the UML syntax we just saw is that the Hobbies enumeration table cannot be used anywhere else.
- To do that, we would need to model it as a separate class of its own.

A Word of Warning

- ❑  Do **not** show an enumeration in your model, and then forget to put in the appropriate relation scheme in the relation scheme diagram.
- ❑ I see this all of the time, and I think that it comes of “out of sight, out of mind” because there is no corresponding class in the UML diagram.
- ❑ But remember, the enumeration must be implemented physically, and to do that, you must show it in the relation scheme diagram.

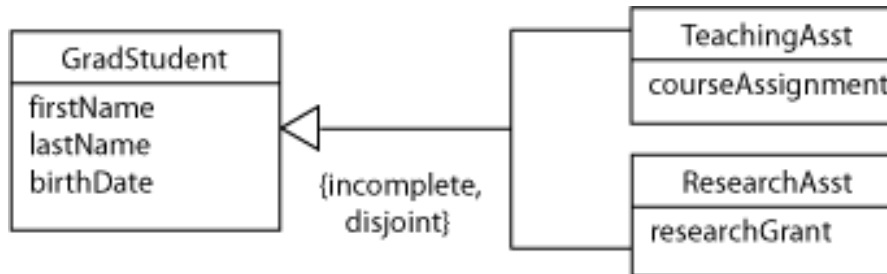
Design Pattern: Subclasses –Top Down Design

- 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).
 - For the TAs, we need to know which course they are assigned to teach.
 - Some are employed as research associates (RAs).
 - For the RAs, we need to know the grant number of the research project to which they are assigned.
 - Some are not employed by the university at all.

Design Pattern: Subclasses –Top Down Design – Class Diagrams



- Note that one of the attributes courseAssignment or researchGrant will **always** be null. If the given student is **neither** a TA nor a grad student, both attributes will 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. E.g. an instance of the generic parent may not fall into **any** of the categories.
 - ***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.

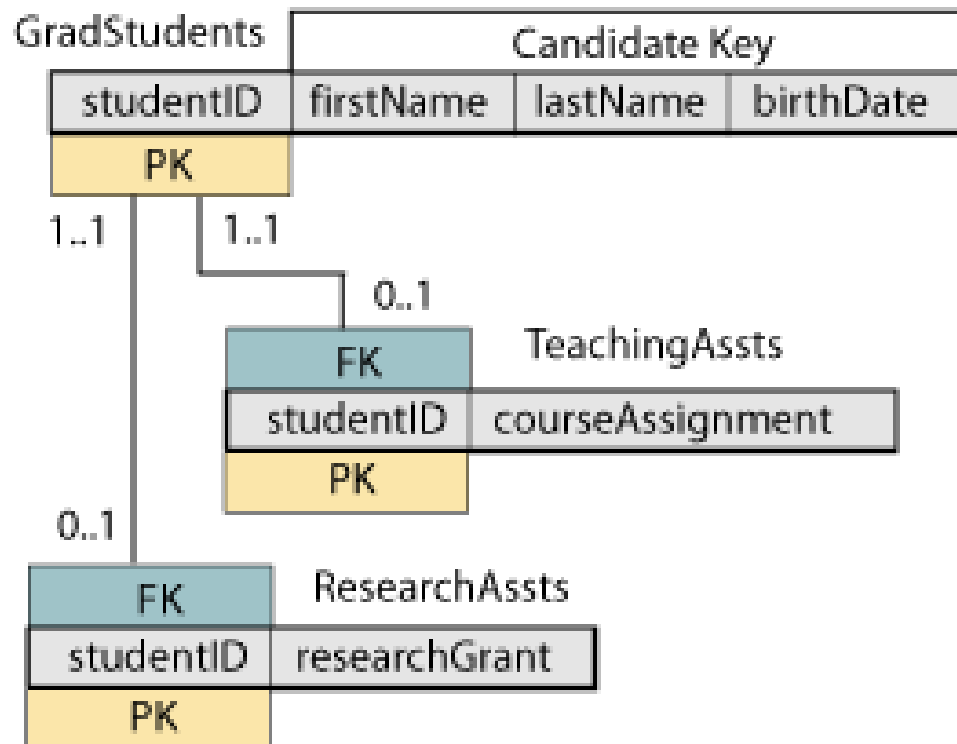
Cardinality Implications of the categorization constraints

- The number of categories that a given instance of the generic parent can belong to for a given combination of categorization constraints is summarized below.
- As a special case, remember that a categorization with only one category can never be complete, otherwise there would be no reason for the categorization at all.
- Note that this is not the cardinality on the physical relationships from the generic parent to the category table.


	Complete	Incomplete
Disjoint	One and only one	Zero or one
Overlapping	One or more	Zero or more

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-zero-or-one relationship.



Primary Key of the subtype=the Primary Key of the supertype

- ❑  **Don't** try to change the composition of the primary key of the category class/table from what is inherited from the supertype.
- ❑ Non-identifying relationships to (that is the subtype is the child) the subtype are fine. But no **identifying** relationships aside to the one from the generic parent are allowed.
- ❑ **Other** candidate keys beyond the Primary Key can and should be identified where they exist.
 - Just remember that those candidate keys cannot **include** the entire primary key.
- ❑ The cardinality to the subtype tables is always going to be 0..1. The database is not going to be able to implement the {...} constraints on the categorization. Use triggers for that.

A class can have only **one** supertype

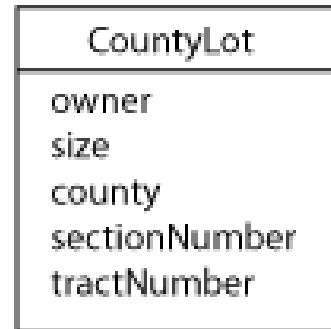
- ❑ This was a huge debate when object-oriented modeling first came out, whether to allow multiple inheritance or not.
- ❑ The verdict finally came in for most languages that multiple inheritance was not going to be allowed. C++ is a notable exception.
- ❑ From the database perspective, that's a huge benefit, because we can enforce the cardinality from the supertype class to the subtype with the foreign keys.

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.

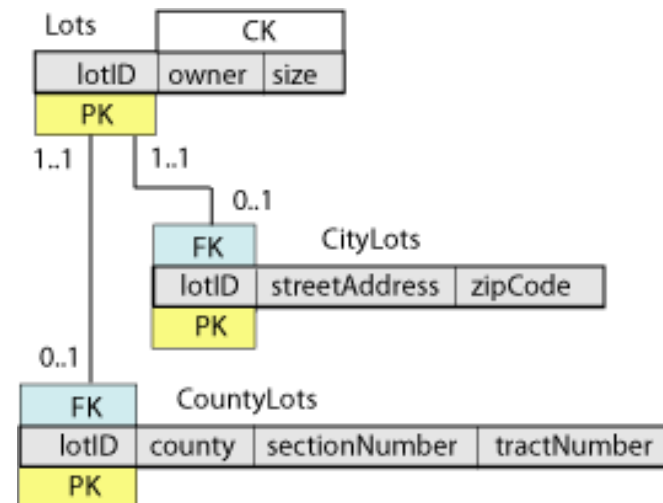
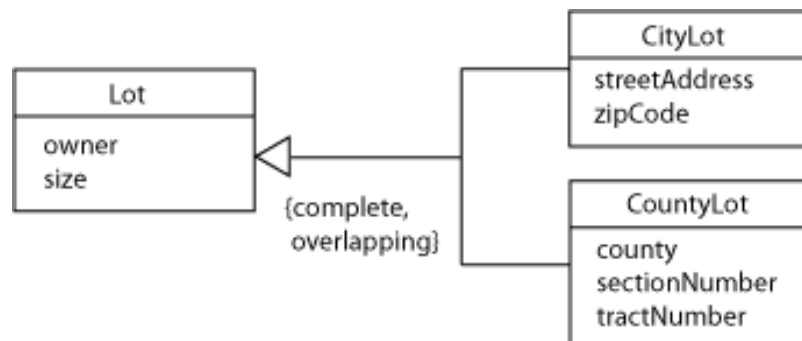
Design Pattern: Subclasses – Bottom Up Design – Class Diagram

- ❑ 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.”



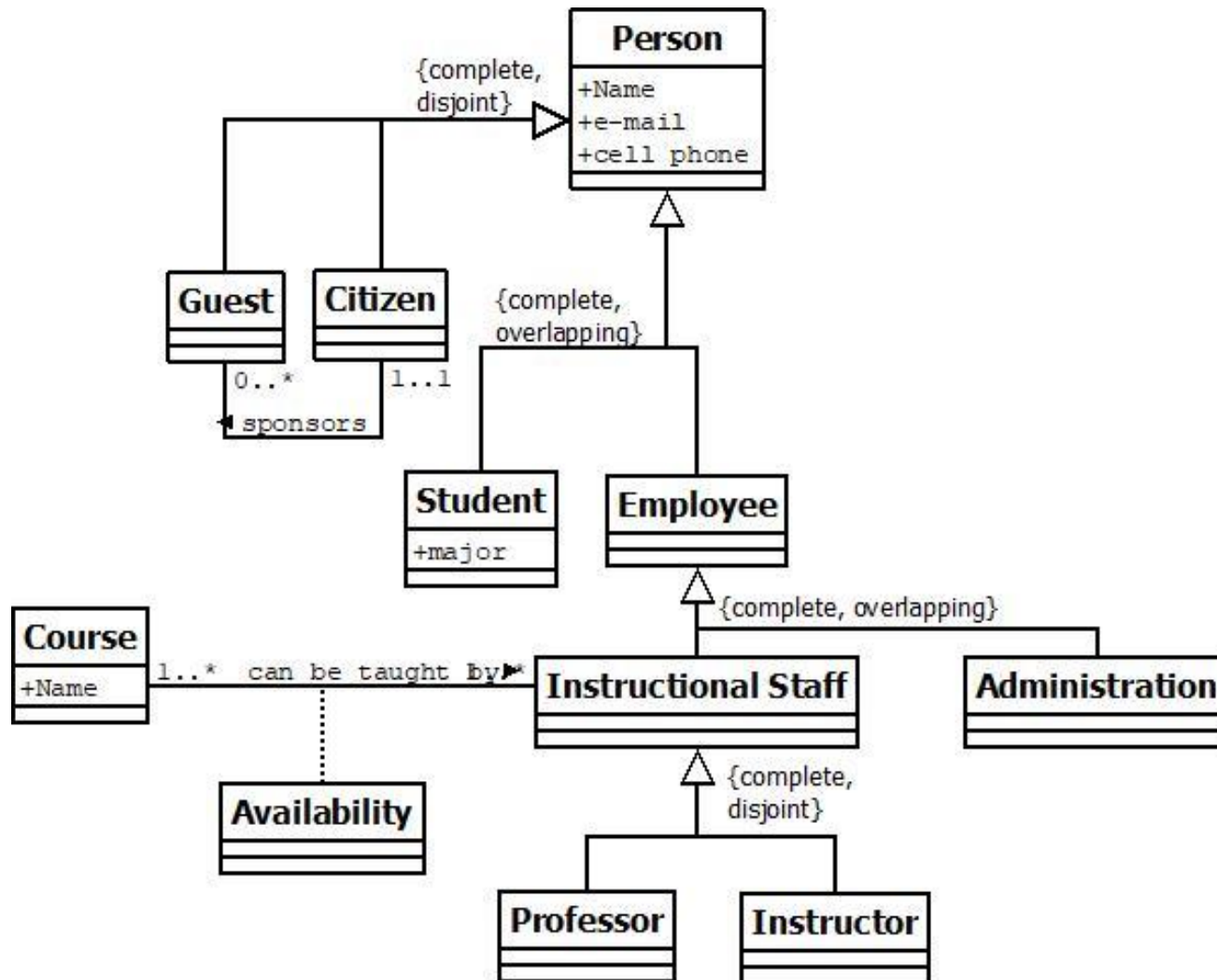
What about a categorization with only one category?

- Occasionally it becomes necessary to model a categorization in which only one category is at all significant.
- An example would be employees in a firm in which the executives receive incentive compensation in the form of a % of the overall profits.
- The rest of the employees don't have that attribute, so you can argue that the executives form a specialization of employees.
- In such a case, the categorization constraints should always be {incomplete, disjoint}
 - If it **were** complete, there would be no need to separate the category out from the generic parent.
 - If there were other categories, it would be possible for the categorization to be overlapping. But since there are not, we'll call it disjoint.

Does each category **have** to have distinct attributes?

- ❑ In a word, “no”.
- ❑ It's common to feel uncomfortable with categorizations that have one or more categories with no peculiar attributes.
- ❑ The mere existence of the category is significant.
- ❑ Also, any **associations** that are only proper to that category are significant.

A More Complex Example



Comments on the Above

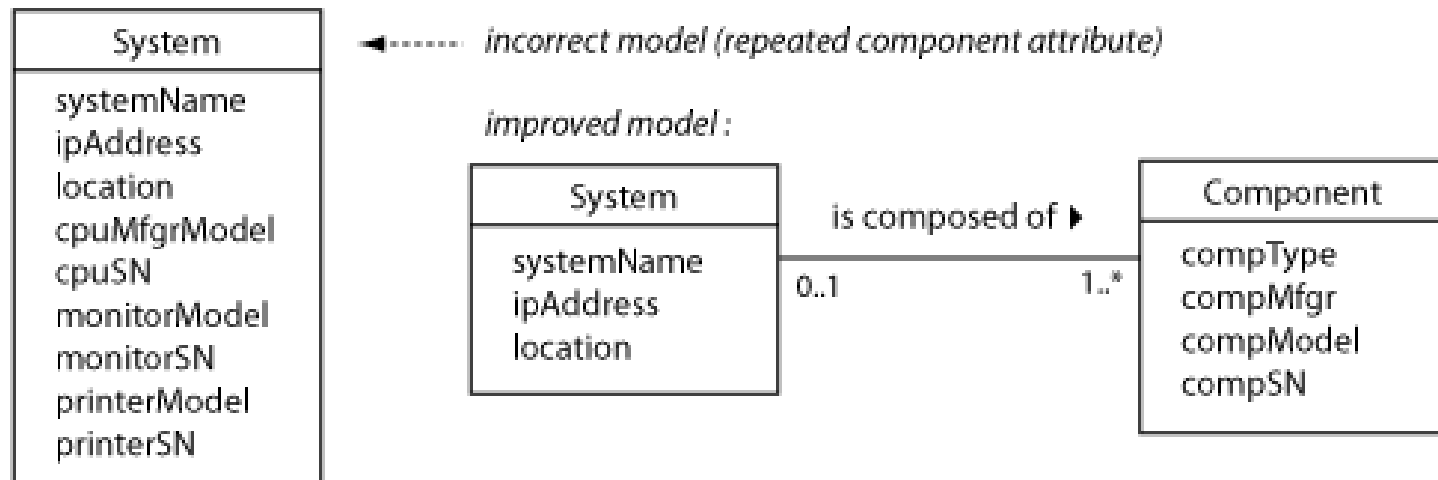
- ❑ Note that *Guest* and *Citizen* do not have any attributes that are peculiar to them, but *Guest* has a unique association: sponsors.
- ❑ An employee could also be a student at some time.
- ❑ A member of administration (I know one of these) can also teach here.
- ❑ On the other hand, you are either a Professor or an Instructor, you cannot be both. Which is **not** to say that an instructor could not have a PhD.
- ❑ From a layout perspective, please try to have your categorizations point upward if possible.
- ❑ Person participates in two **independent** categorizations: *Guest/Citizen* and *Student/Employee* at the same time. Those two categorizations are independent of each other. Which category the Person falls into in one categorization says nothing about which category the fall into in the other.

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.

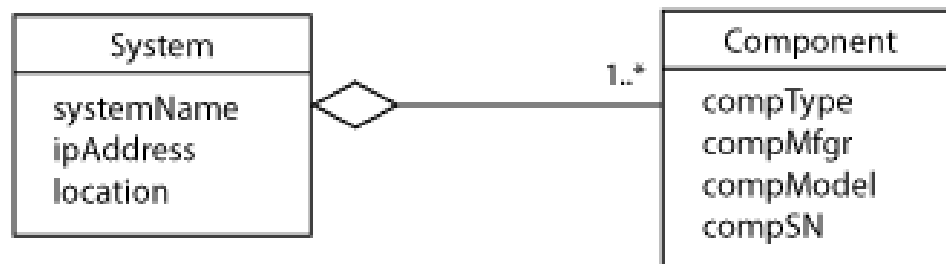
Design Pattern: Aggregation – Incorrect Model vs. Improved Model

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



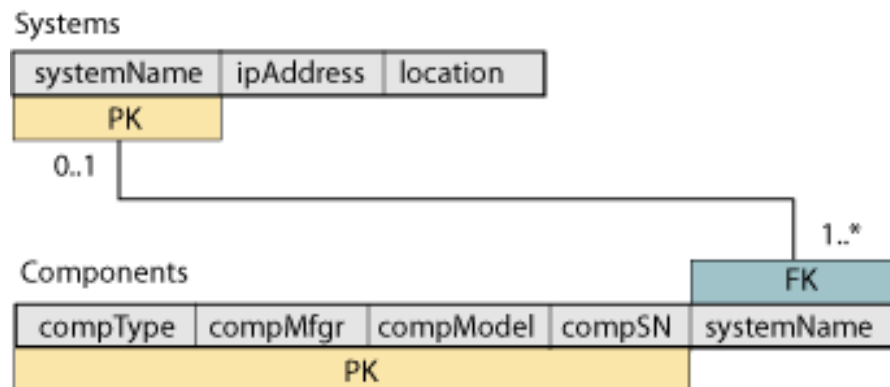
Design Pattern: Aggregation – Model with UML Aggregation

- ❑ Each System object 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.
- ❑ If the multiplicity were 1..1, that would imply that the child **must** have a parent, which is not the case in aggregation.
- ❑ The components **can** exist apart from any system using them.



Design Pattern: Aggregation – Relation Scheme

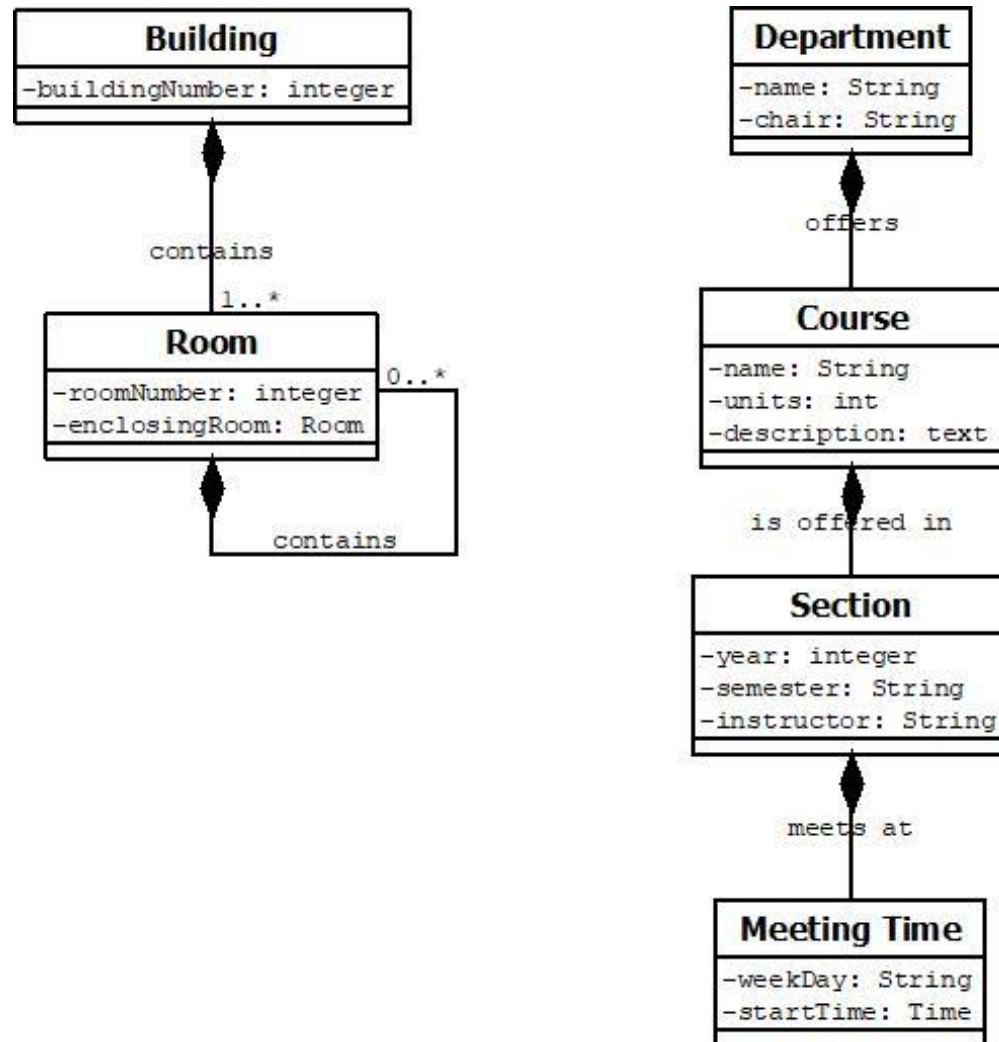
- Since the component **can** exist by itself in this model the system name **can not** be part of its PK. That is, this relationship is **non-identifying**.
- We use the only candidate key {type, mgr, model, SN} as the PK since this class is not a parent. Otherwise, we would have created a surrogate key with this many keys. 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, so we don't spell the multiplicity out as it would be redundant.

Design Pattern: Aggregation – Composition Examples



Comments on these examples

- ❑ The first model on the previous foil shows one room composed of several other rooms. Think of a conference center in which a room can be broken up into smaller rooms by deploying solid curtains as dividers.
- ❑ That's an example of a **recursive** relationship which we'll be covering in more detail later. Basically it's just another flavor of one to many.
- ❑ The second example shows a cascade of compositions. Courses only exist because of their offering departments, sections only exist because of their courses, and so forth.
- ❑ A section could have an association from instructor (which would have been better) and still be a composition.

Aggregation vs. Composition

- There is no existential dependency implied by an aggregation, there is in a composition. Or, when the parent dies, it takes all of the children with it in composition.
- The relationship in the relation scheme is **non** identifying for an aggregation.
 - That comes from the association being optional.
- From the OO programming perspective, aggregation can be likened to collection by pointers (reference) where composition is collection by value.

Some examples

- ❑ You run an engine rebuilding shop, and you have folks bring in their cars and boats to get the engine serviced. Would you model the relationship between engine and boat/car as an aggregation or a composition?
- ❑ You are managing a set of stadiums. Each stadium has many seats, some of which are the cheap seats, while others are veritable salons. Would you use aggregation or composition to model the relationship from stadium to seat?
- ❑ You are managing a chess tournament. Each board has players on it during the scheduled match times. Each board has 64 squares on it. At any point in time, you are recording which players are on a board and where all the pieces are, square by square. What is the relationship between board and player vs board and square?

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.

Aggregation – A Summary

Relationship Type	Symbol	Identifying	Required	Existential Dependency?
Aggregation		No	No	No
Composition		Maybe	Yes	Yes
Non Descript	<u>m..n</u>	Maybe	Maybe	Maybe

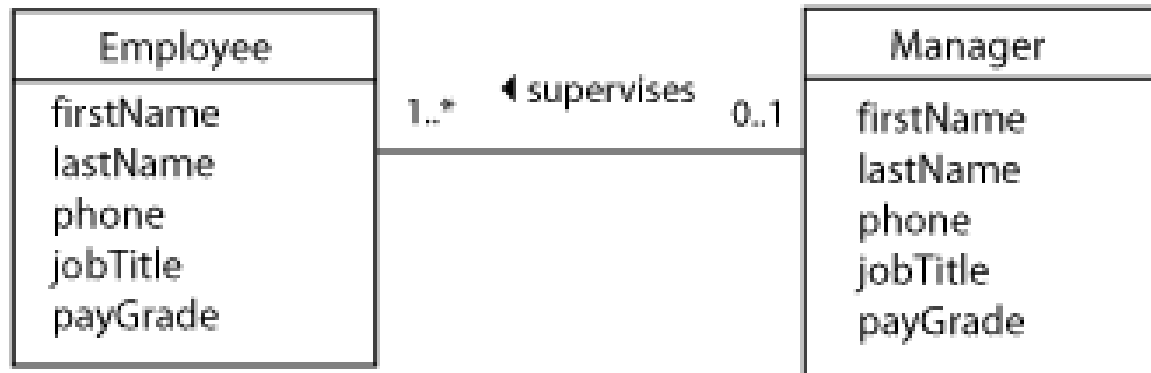
- ❑ Identifying – the primary key of the parent becomes part of the primary key of the child.
- ❑ Required – The multiplicity at the parent end is 1..1. Optional → multiplicity of 0..1.
- ❑ Existential Dependency – the child cannot exist apart from the parent – implied by Required

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

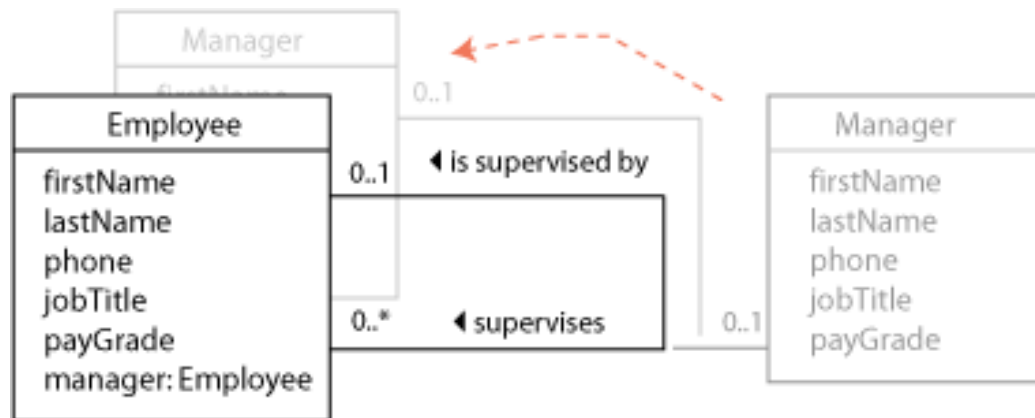
Design Pattern: Recursive Associations – **Incorrect Relationship**

- 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

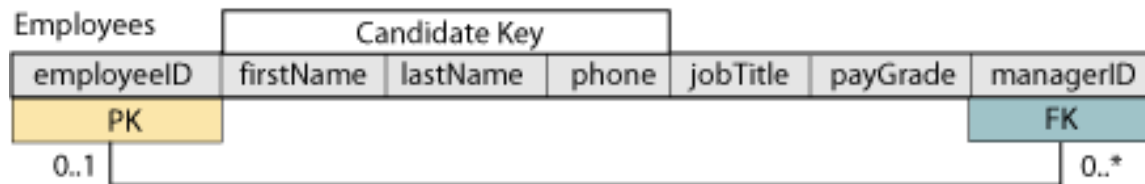
- Normally, we wouldn't show an FK in the class diagram; however, including the manager as an attribute of the employee here (in addition to the association line) can help in understanding the model.



- Note that the manager attribute is of type Employee.

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).

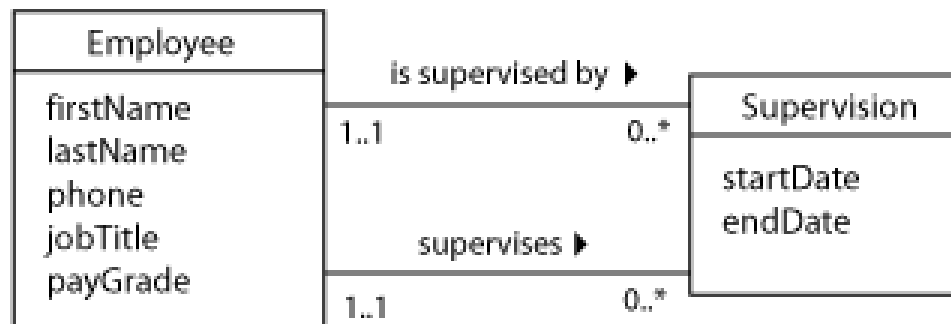


- Note that the employeeID has a new name when it comes in as a foreign key: managerID. Without the role name, the manager and employee would have to be the same.

Design Pattern: Recursive Associations

– Many-to-Many Class Diagram

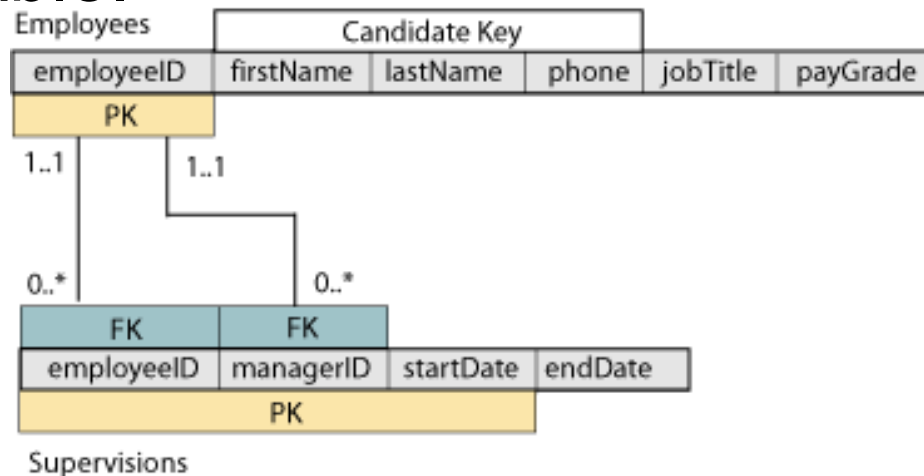
- ❑ In some project-oriented companies, an employee might work for more than one manager at a time.
- ❑ We also might want to keep a history of the employees' supervision assignments over time. We can model either case by revising the class diagram to a many-to-many pattern.



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.

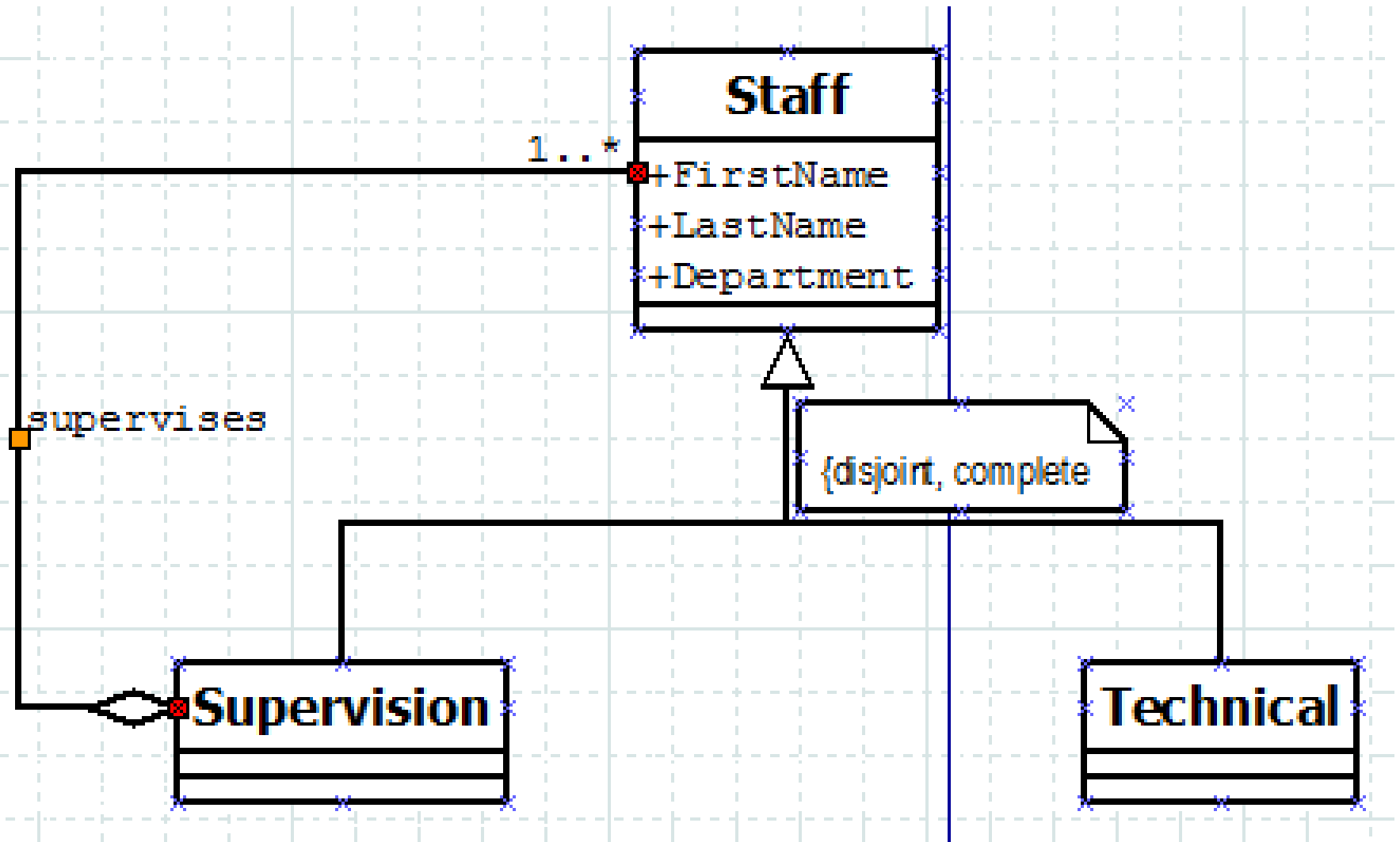


- Note that this is a many to many with history since the same manager could supervise the same employee multiple times.

Role names, again

- Note that we had to give the migrating foreign key `employeeID` a different role name for the manager.
- The physical **implementation** of the role name comes about when the referential integrity constraint is created:
 - `alter table supervisions add constraint employees_supervisions_fk01 (managerID) references employees (employeeID)`
- You must give the migrating foreign key a different name, otherwise employees are managing themselves.

Categorization by role



The Recursion Can be Transitive

- ❑ Notice that the aggregation relationship runs from the supervision to the Staff class.
- ❑ TRW calls their technical folks Member Technical Staff (MTS) and those are the non managers.
- ❑ The aggregation tells us that the one who is supervised could be another Supervisor, or a member of Staff.
- ❑ Why was this an aggregation rather than a composition?
- ❑ Why have the aggregation run to Staff?

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						

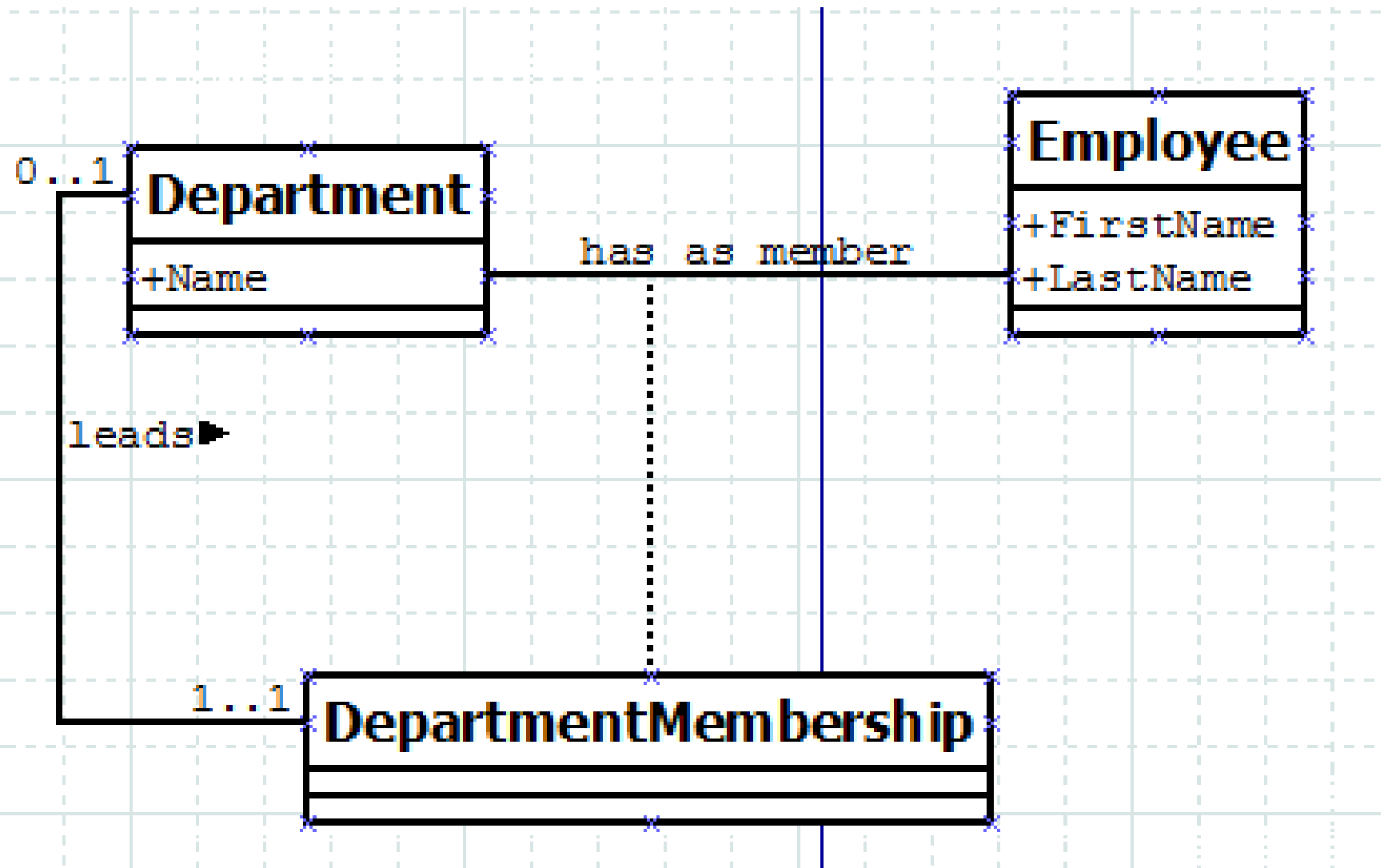
Employee (supervised by this manager)

						FK
82	Robert	King	5-1221	Clerk	admin 3	53

Recursion & Many to Many

- ❑ Say that you have your company broken into departments and each department can have many employees, and a given employee could belong to many departments.
- ❑ Say further that a given department will be run by one member of that department. You want to be sure that the person running the department comes from that department.
- ❑ The department name uniquely identifies the department within the company.

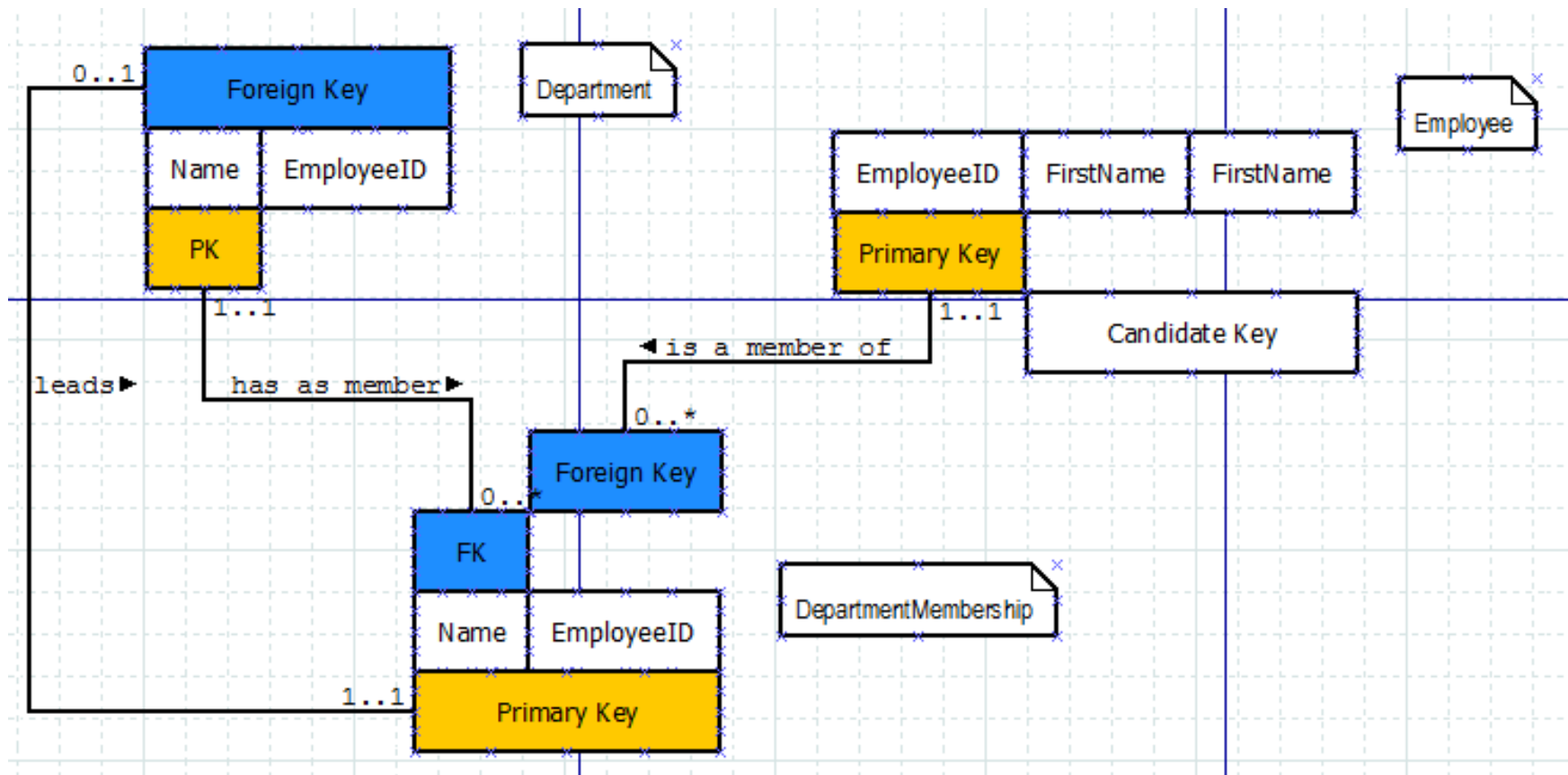
The UML



Points to Ponder

- Notice that the association from DepartmentMembership to Department is not aggregation nor a composition.
- The relationship is mandatory, every department has to be managed by **someone**.
- But the relationship is not identifying. The department name identifies the department.
 - So we cannot use an aggregation, because the relationship is mandatory
 - We cannot use a composition because it is non-identifying.

The Relation Scheme Diagram



Points to Ponder (again)

- We have a “chicken and egg” situation between DepartmentMembership and Department.
 - Department needs to be there before we can create any instances of DepartmentMembership.
 - At least one instance of DepartmentMembership needs to be in the database to be the leader of the Department.
- We can use “set foreign_key_checks=0;” to disable foreign key constraints long enough to get the department and its first member stored and related to each other, then “set foreign_key_checks=1;” to put the foreign key checks back online.
- This only applies to the **current** session, leaving the foreign key constraints intact for everyone else.

SQL for recursive relationships

- ❑ You want to find the orders that have products in common with each other.
- ❑ So you do a self join on orderDetails to get the orders that have a common product.
- ❑ Then, count the number of common products between a given pair of orders.
- ❑ Then, join, based on those order numbers, to the orders and customers to find the information about those orders.

Recursive relationship and subqueries

```
select one.orderNumber, one.productCode,  
       two.ORDERNUMBER  
from   orderDetails one inner join orderDetails two using  
       (productCode)  
where  one.ORDERNUMBER < two.ORDERNUMBER  
order by one.ORDERNUMBER, two.ORDERNUMBER;
```

- Here again, we use aliases to separate one use of the orderDetails table from the other.
- There is no real distinction between them, so I'm just going with one and two so that I can distinguish them from each other.
- Note the < to make sure I only get each pair once.

But that's not very useful

```
select firstOrderNum, secondOrderNum, count(*) as countCommonProducts
from      (select one.orderNumber as firstOrderNum,
                  one.productCode, two.ORDERNUMBER as
                  secondOrderNum
            from    orderDetails one inner join
                  orderDetails two using (productCode)
            where   one.ORDERNUMBER < two.ORDERNUMBER)
            commonOrders
group by firstOrderNum, secondOrderNum;
```

- Now, we count the number of common product items shared between two orders by grouping by the two order numbers.
- That count gets called “countCommonProducts”.

Now for the coup de grace

```
select  c1.customerName, o1.ORDERNUMBER, o1.orderDate, c2.customerName,
        o2.ORDERNUMBER, o2.orderDate, countCommonProducts
from    (select firstOrderNum, secondOrderNum, count(*) as
          countCommonProducts
        from  (select one.orderNumber as firstOrderNum,
                      one.productCode, two.ORDERNUMBER as secondOrderNum
        from    orderDetails one inner join orderDetails two using
                (productCode)
        where   one.ORDERNUMBER < two.ORDERNUMBER)
          commonOrders
        group by firstOrderNum, secondOrderNum)
          countedCommonOrders
inner join orders o1 on (firstOrderNum = o1.ORDERNUMBER)
inner join customers c1 on
        (o1.CUSTOMERNUMBER = c1.CUSTOMERNUMBER)
inner join orders o2 on (secondOrderNum = o2.ORDERNUMBER)
inner join customers c2 on (o2.CUSTOMERNUMBER =
        o2.CUSTOMERNUMBER)
where   c1.CUSTOMERNAME = 'Mini Caravy' and
        c2.CUSTOMERNAME = 'Baane Mini Imports';
```

o1, o2, c1, c2?

- We are joining from that collection of orders with common products along two different paths.
- We need to distinguish which order number gets joined to orders and thence to customers so that we keep them straight.
- The names, as you would guess, are immaterial. I chose these for brevity.

Relational Algebra Review

- ❑ Remember that any expression that yields a relation in relational algebra can be renamed, given an alias: $\rho_X(R)$ where R is again the original relation, and X is the new name that you wish to give the relation.
- ❑ Much like an alias for a subquery in SQL, you can use the new name elsewhere in the statement to refer to individual columns of that query.

RA Aliasing Continued

- ❑ Aliasing columns in SQL is more than just cosmetic.
- ❑ To give attributes in a relation new names:
 $\rho_{\substack{a,b \\ y,z}}(R)$ where R is the original relation (either a table or an expression), a, b, \dots are the new name(s), and y, z, \dots are the old names.
- ❑ To remember: ρ is the Greek letter rho, which we use to rename something in Relational Algebra.

RA Recursive Query Example

- Return the employee last name, first name, their manager's last name and first name:

```
select  e.lastName, e.firstName, m.lastName, m.firstName
from    employees e inner join employees m
        on e.reportsTo = m.employeeNumber
order by e.lastName, m.lastname;
```

- That equates to this in Relational Algebra:

$$\pi_{E.LASTNAME, E.FIRSTNAME, M.LASTNAME, M.FIRSTNAME} \rho_E EMPLOYEES \bowtie_{E.REPORTSTO=M.EMPLOYEEENUMBER} \rho_m EMPLOYEES$$

- We had to assign a role name/alias name to Employees to do the recursive join, as well as refer to the manager's first & last name.

Role naming the attributes

- The relational algebra query:
 - $\pi_{E.LASTNAME, E.FIRSTNAME, M.LASTNAME, M.FIRSTNAME}$
 $\rho_E EMPLOYEES \bowtie_{E.REPORTSTO=M.EMPLOYEEENUMBER}$
 $\rho_m EMPLOYEES$
 - Doesn't give us distinctive column headers for the manager names.
 - $\pi_{E.LASTNAME, E.FIRSTNAME, M.LASTNAME, M.FIRSTNAME}$
 $\rho \frac{firstname, lastname}{employee\ first\ name, employee\ last\ name} \rho_E EMPLOYEES$
 $\bowtie_{E.REPORTSTO=M.EMPLOYEEENUMBER}$
 $\rho \frac{firstname, lastname}{manager\ first\ name, manager\ last\ name} \rho_m EMPLOYEES$

This is more than cosmetics

- When we use an expression as a sub query, the column names output by the sub query are exactly what we need to reference in the outer query.
- If we had the above query as a subquery, without the role names it would look as though it had two first names and two last names.
- Technically, that's a syntax error in Relational Algebra since the relation scheme is a set of attributes, and there can be no duplicates in a set.

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)

An Example

- Consider the relation: Work {projName, projMgr, empId, hours, empName, budget, salary, startDate, empMgr, empDept, rating, skills}
 - Each project has a unique name
 - Names of employees and managers are **not** unique
 - Each project has one manager: projMgr
 - Each employee can work many projects and vice versa. Hours tells the # of hours/week the employee is supposed to work on that project.
 - budget stores the budget for the project
 - startDate gives the starting date for the project
 - salary is the annual salary of the employee

More assumptions

- ❑ empMgr gives the name of the employee's manager, who might **not** = the project manager.
- ❑ empDept gives the employee's department. Department names **are** unique.
- ❑ The employee's manager is the manager of the employee's department.
- ❑ Rating gives the employee's rating for the particular **project**.
- ❑ Each employee is associated with a **set** of skills that they are master of. Many employees can share the same skill.

Functional Dependencies

- Note that these are all directly derived from the text of the business rules.
- $\{\text{projName}\} \rightarrow \{\text{projMgr}, \text{budget}, \text{startDate}\}$
- $\{\text{empId}\} \rightarrow \{\text{empName}, \text{salary}, \text{empMgr}, \text{empDept}, \text{skills}\}$
- $\{\text{projName}, \text{empId}\} \rightarrow \{\text{hours}, \text{rating}\}$
- $\{\text{empDept}\} \rightarrow \{\text{empMgr}\}$
- Leading us to a primary key of $\{\text{projName}, \text{empId}\}$

1st Normal Form

- ❑ We don't have a sample table for this relation, but “skills” sounds very much like “hobbies”. We'll treat it the same way.
- ❑ Note that the UML diagram can call this out simply by the attribute `skills[0..*]`
- ❑ If we want to have those skills enumerated, we simply say: `skills[0..*]:Enumerated`

2nd Normal Form

- Are there any partial dependencies (i.e. subkeys) in which the subkey is **part** of the primary key?
 - {projName} → {projMgr, budget, startDate}
 - {empId} → {empName, salary, empMgr, empDept}

Brief Aside

- As we move through the three normal forms, the resulting relations have less and less redundancy.
- One way to remember the difference between 2nd and third normal form is that a subkey that's part of the primary key is more severe than a subkey that is not part of the primary key.
- 3rd normal form will tackle the subkeys that are not part of the primary key.

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 (e.g. multiple attributes) key for relation to have FD problem

3rd Normal Form

- Do any of the tables that we've created have any transitive dependencies?
 - The Proj relation is now in 3rd normal form
 - $\{\text{empDept}\} \rightarrow \{\text{empMgr}\}$ so we need to factor that out of the Emp relation.
 - This is a transitive dependency because $\{\text{empID}\} \rightarrow \{\text{empDept}\}$
 - Since $\{\text{empDept}\}$ is not part of the primary key, this is only taken up when achieving 3rd normal form.

Lossless Decomposition Defined

- 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 (that is, each original attribute of R appears in at least one R_i).
- A decomposition of R $\{R_1, R_2, \dots, R_n\}$ is **lossless** for R if the natural join of $\{R_1, R_2, \dots, R_n\}$ produces exactly the relation R .
- Not all decompositions are lossless. Achieving 3rd normal form can always be done with a lossless decomposition. Achieving Boyce Codd Normal Form (BCNF) will sometimes lose dependencies.

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

- Credits is the number of units that the student has earned to date, so if the standing is purely a matter of the units ...
- Transitive dependencies cause update, insertion, deletion anomalies.

Partial Functional Dependency Example

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

FDs:

{courseNo,stuld} \rightarrow {lastName}

{courseNo,stuld} \rightarrow {facld}

{courseNo,stuld} \rightarrow {schedule}

{courseNo,stuld} \rightarrow {room}

{courseNo,stuld} \rightarrow {grade}

courseNo \rightarrow facld **partial FD

courseNo \rightarrow schedule **partial FD

courseNo \rightarrow room ** partial FD

stuld \rightarrow lastName ** partial FD

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

Appendix: Traditional Normalization

– Denormalization

- Your 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***.

Denormalization Life Cycle

- ❑ Always take the time and effort to fully design the database in **normalized** form. Hang onto that model as a reference for the business rules.
- ❑ Separately document the denormalization decisions made.
- ❑ Reflect the denormalization decisions in the **physical** model (e.g. the relation scheme diagram).
- ❑ Generally, the mapping between the logical and physical models is not rigorously maintained, you'll have to find ways to reconcile the differences between the logical and physical.

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.

More Denormalization Examples

- ❑ Denormalizations generally bring in redundancy.
- ❑ Putting the data from a parent into the child, for instance bringing in the state name and storing along with the state abbreviation is redundant because state abbreviation → state name.
- ❑ The protection would be to not allow updates to the state name except by changing the state abbreviation. In that case, the trigger would have to go to the state table and pull up the proper state name and put that into the row being modified.

Denormalization – Migrate through non-identifying associations

- ❑ Say that you have owner as a non-identifying parent to vehicle, and vehicle as an identifying parent to service_visit.
- ❑ You decide that you do the double join from service_vist to owner so many times, that you want to keep the owner information in service_visit, redundantly.
- ❑ We'll show you how to build a trigger that will “migrate” that information down automatically if any of it changes in the source table.
- ❑ By “migrating” that information, you save yourself two joins. But
 - You also incur the risk of those two copies of the same information getting out of sync.
 - And you have redundant data taking up disk space.

Denormalization – multi-valued attributes

- If we back away from the 1st normal form and allow a column to have more than one value, how would we regulate that?
 - Have the hobbies table and contact_hobbies tables still there, redundantly.
 - If a contact gains/loses a hobby, make that change in the contact_hobbies table.
 - When a row is inserted/deleted from contact_hobbies, re-derive the list of hobbies for that contact (might as well sort them) and store the multi-valued attribute again. This can be done automatically in the database by means of a trigger.

Denormalization – Category roll up/roll down

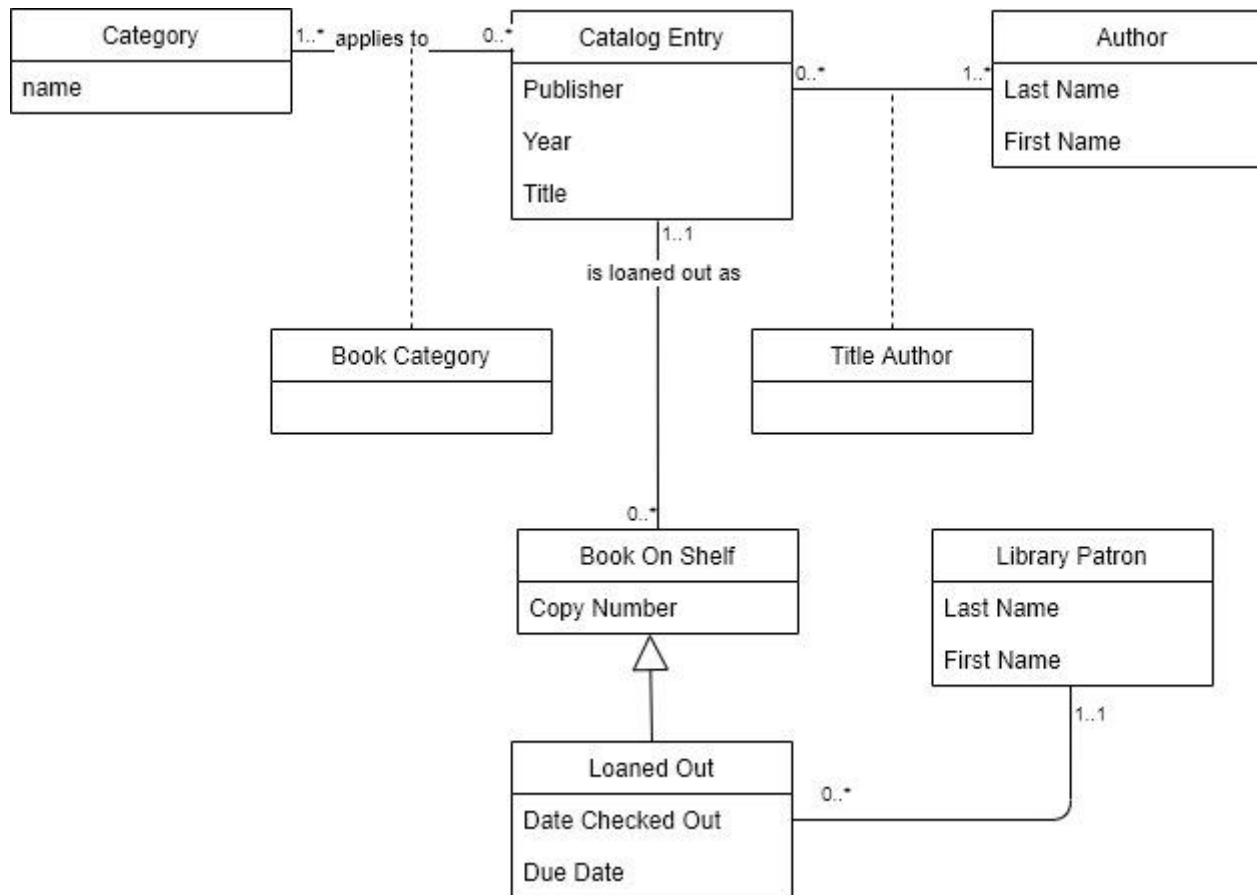
- Strictly speaking, optional attributes are not a violation of any normal form, but it can still be regarded as a denormalization to merge the category table into the generic parent (roll up) or the parent into the child (roll down).
 - For instance, the grad student example could be rolled down if the common attributes in Student were put into ResearchAssistant as well as TeachingAssistant.
 - That is redundant from a structural standpoint, but not redundant from a **data** standpoint.

Denormalization: Category Roll up (Continued)

- By contrast, a roll up would be to go back to the grad student table as it originally existed, in which there were two optional attributes: the grant and the class.
 - You could add another attribute, call it GradStudentType that was either “RA” or “TA”.
 - You could use a check constraint (Oracle, but not MySQL) to do that much.
 - Then, depending on the value of your GradStudentType, you would check in an on insert or on update trigger to make sure that RA records only had a value for the grant, and the TA records only had a value for the course.

An Example

Publisher	Year	Categories	Authors	Title	Copy #	Date Checked out	Date Due	Client
Random House	1851	Adventure, Historical Fiction	Herman Melville	Moby Dick	1			
Random House	1851	Adventure, Historical Fiction	Herman Melville	Moby Dick	2	4/25/2020	5/11/2020	Kermit the Frog
Random House	1851	Romance, Comedy	Herman Melville	Moby Dick	3	3/22/2020	4/5/2020	Miss Piggy



Necessary Triggers

- ❑ Create a many to many with Categories so that you have the benefit of a list of legitimate categories, but then need to keep that in sync with the categories column.
- ❑ Same with Authors.
- ❑ Make sure that Date Checked Out and Due date are either both there or neither of them.
- ❑ Make sure that Library Patron is only associated with a book if that book is loaned out.

Collapse Parents into Many to Many

- ❑ Particularly if one or both parents in a many to many do not have any association to any other classes, a viable denormalization is to collapse one or both into the association class.
- ❑ Then, you need a trigger on the association class to make sure that all of the redundant copies of each parent are consistent.

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
- Another way to say it is that a relation is in BCNF if every determinant is a candidate key.
 - Remember that a determinant is any combination of one or more attributes which determine the value of some other attribute.
 - The **determinand** is the combination of attributes on the right-hand side of the functional determination expression.

BCNF Example

Relation Faculty: (facultyName, dept, office, rank, dateHired)

Functional Dependencies:

{office} → {dept} – a given office belongs to a department

{facultyName, dept} → {office, rank, dateHired} – name is only unique within a given department.

{facultyName, office} → {dept, rank, dateHired} – since {office} → {dept}

- ❑ Faculty 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, the same way that we do with any subkey.
- ❑ Two candidate keys: {facultyName, dept} and {facultyName, office}
- ❑ Decompose into
 - Fac1 (office, dept) – not the other way around since each department can have many offices.
 - 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

How was that 3rd Normal Form?

- The easy way to express 3rd Normal Form is to say that every **non** key attribute depends upon the key, the whole key, and nothing but the key.
 - This applies to **all** the candidate keys
- {office} → {dept} does **not** violate that 3rd Normal form mnemonic because {dept} is part of the primary key, which is, of course, also a candidate key.

Example Boyce-Codd Normal Form

Faculty

facultyName	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	Math	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

facultyName	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

Trigger needed to enforce the lost functional dependency

```
/*
Boyce-Codd normal form example from class. Faculty table refactored into Fac1 and
Fac2 to resolve subkey of {office}->{dept}.
```

```
*/
```

```
create table          Fac1(
                        office      varchar(4) not null primary key,
                        dept        varchar(12) not null);
```

```
/*
```

This table has office as a migrating foreign key from Fac1, which means that dept is no longer here since we can deduce that by virtue of a join with Fac1. But that separation has made it impossible for us to enforce the functional dependency: {facultyName, dept} --> {office, rank, dateHired}.

```
*/
```

```
create table          Fac2(
                        facultyName  varchar(10) not null,
                        office        varchar(4) not null,
                        constraint Fac1_Fac2_FK01 foreign key (office)
                                references Fac1(office),
                        rank          varchar(12) not null,
                        dateHired     int not null,
                        constraint Fac2_hire_date check (dateHired > 1940 and
                                dateHired <= year(now())),
                        constraint Fac2_PK primary key (facultyName, office));
```

Inserts for the sample data

```
insert into Fac1 (office, dept) values  
( 'A101', 'Art'),  
( 'C101', 'CSC'),  
( 'C105', 'CSC'),  
( 'H102', 'History'),  
( 'M201', 'Math'),  
( 'M203', 'Math');
```

```
insert into Fac2 (facultyName, office, rank, dateHired) values  
( '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);
```

Stored procedure for the constraint enforcement

```
CREATE DEFINER=`root` @`localhost` PROCEDURE `fac2_dup_count`(in
faculty_name varchar(40),
    in in_office varchar(4))
BEGIN
    declare dup_count int;
    select    count(*) into dup_count
    from      Fac1 inner join Fac2 using (office)
    where     facultyName = faculty_name and
            dept = (select    dept
                    from      Fac1
                    where     office=in_office);
    if (dup_count > 0) then
        signal sqlstate '45000'
        set message_text=
            '{facultyName,dept} → {office, rank, dateHired} FD violated';
    end if;
END
```

Triggers to use the stored proc

```
CREATE DEFINER=`root` @`localhost` TRIGGER
`programming`.`fac2_BEFORE_INSERT` BEFORE INSERT ON `fac2` FOR
EACH ROW
BEGIN
    call fac2_dup_count(new.facultyName, new.office);
END
```

```
CREATE DEFINER=`root` @`localhost` TRIGGER
`programming`.`fac2_BEFORE_UPDATE` BEFORE UPDATE ON `fac2` FOR
EACH ROW
BEGIN
    call fac2_dup_count(new.facultyname, new.office);
END
```

Demonstration inserts

/*

We already have a record with Tanaka in the CSC department and in office C101. This insert would create a Tanaka in C105, which doesn't violate the uniqueness constraint in Fac2, but it would be a second Tanaka in that department, since office C105 is the CSC department, which violates the {facultyName, office} --> {dept, rank, dateHired} functional dependency.

*/

```
insert into Fac2 (facultyName, office, rank, dateHired) values  
('Tanaka', 'C105', 'Professor', 2016);
```

-- Just to show that a legitimate insert is possible.

```
insert into Fac2 (facultyName, office, rank, dateHired) values  
('Brown', 'C105', 'Instructor', 2015);
```

-- And then to demonstrate that the update checks as well, now that we have an on update trigger.

```
update fac2 set facultyName='Tanaka' where facultyName='Brown';
```

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
- ❑ Note that you don't have to go through 1NF, 2NF, 3NF to get to BCNF.

Distinction between 3NF and BCNF

- A relation, R , is in **3NF** iff for every nontrivial Functional Dependency ($X \rightarrow A$) satisfied by R at least ONE of the following conditions is true:
 - a) X is a superkey for R , or
 - b) A is a key attribute for R
- BCNF requires (a) but doesn't treat (b) as a special case of its own. In other words BCNF requires that every nontrivial determinant is a superkey even its dependent attributes happen to be part of a key.
- A relation, R , is in **BCNF** iff for every nontrivial FD ($X \rightarrow A$) satisfied by R the following condition is true:
 - X is a superkey for R
- BCNF is therefore more strict.
- The 3NF rule of thumb is “all **non key** attributes depend upon the key, the whole key, and nothing but the key.”

3NF but not BCNF – an example

- ❑ Supplier_part (supplier_no, supplier_name, part_no, quantity)
- ❑ {supplier_**no**, part_no} → {quantity}
- ❑ {supplier_**name**, part_no} → {quantity}
- ❑ {supplier_no} → {supplier_name}
- ❑ {supplier_name} → {supplier_no}
- ❑ We have two overlapping candidate keys.
- ❑ Since supplier_name and supplier_no are each part of a candidate key, this is in 3NF already.
- ❑ But it's not in BCNF yet.
- ❑ Note that breaking it into two tables loses one of the dependencies.

Normalization Example

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

projName	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, therefore the Work relation is in 1NF

□ Second Normal Form

■ Partial dependencies

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

■ Transform to

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

Normalization Example (cont)

Second Normal Form

Proj

projName	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

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

Multi-valued Dependency Example

- Say you have a single table that captures the books required for a course and the instructors who can teach that course.
- The text books are **not** optional. No matter which instructor teaches a given course, they must use the same textbook(s) for that course.
- Going in the other direction, you cannot list a given textbook for a course without also listing all of the instructors for that course.

Sample Data

University courses		
<u>Course</u>	<u>Book</u>	<u>Lecturer</u>
AHA	Silberschatz	John D
AHA	Nederpelt	John D
AHA	Silberschatz	William M
AHA	Nederpelt	William M
AHA	Silberschatz	Christian G
AHA	Nederpelt	Christian G
OSO	Silberschatz	John D
OSO	Silberschatz	William M

How to fix it

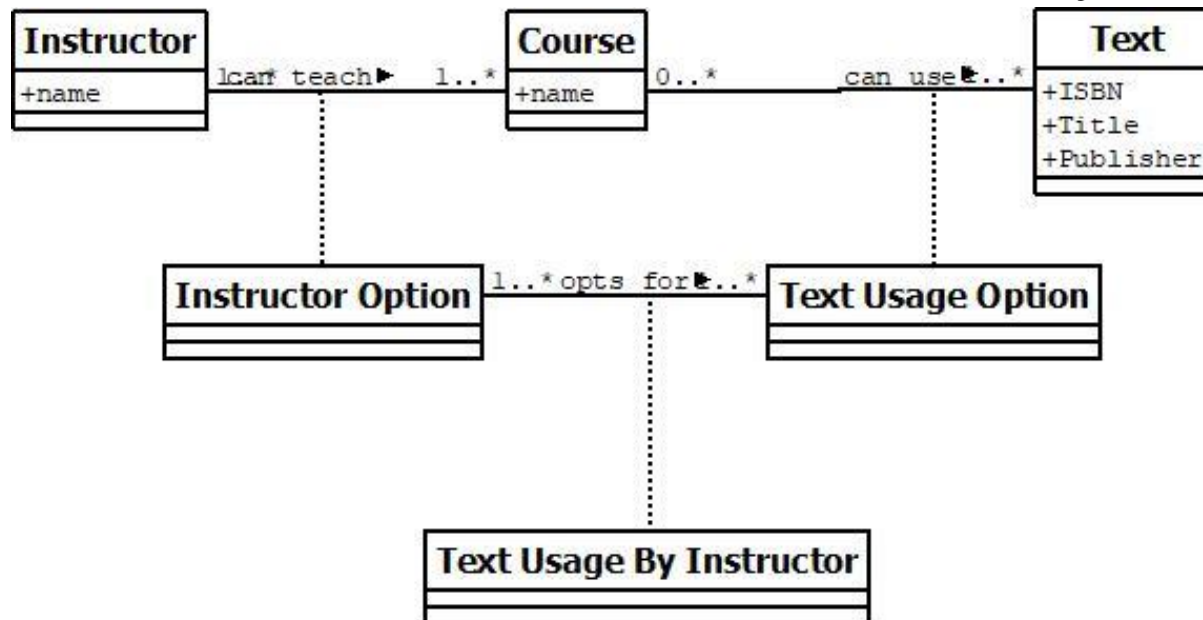
- First, realize that there is no relationship between instructor and textbook.
- Then break it down into two tables
 - One that has a row for each instructor of a given course.
 - And the other that has a row for each textbook for a given course.
- The insidious thing about multi-valued dependencies is that a casual observer assumes that there **is** a dependency between the instructor and text where there is not.

Formal Definition of 4NF

- The relation is in BCNF.
- And there are no multi-valued dependencies.

A Wrinkle in the Example

- Let's change the business rules just a bit and suppose that a given course could use several texts, a given course could be taught by several instructors, and each instructor gets to choose which of the texts for that course they will use.



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

5NF Definition

- AKA project-join normal form
- A relation is in 5NF if:
 - It is already in 4NF
 - And there is no **lossless** decomposition into smaller tables.
- Alternatively, a relation is in 5NF if the candidate key implies every join dependency in it.

Join Dependency

- If a relation can be **recreated** by joining multiple tables and each of these tables have a **subset** of the attributes of the relation, then that relation is in Join Dependency.
- {Department, CourseNumber, Description, Units} can be decomposed into:
 - {Department, CourseNumber, Description}
 - {Department, CourseNumber, Units}
- You often see such join dependencies written out: [{Department, CourseNumber, Description}, {Department, CourseNumber, Units}]

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 relation** 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.
- ❑ Later, we will study how to use JPA to populate Java objects by performing selects in the database.

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 step by step approach to query refinement

- ❑ Later we will be building some complex queries in this class on the classic models database, and it's important at this point to “set the stage”.
- ❑ Don't try to build the whole statement at once, particularly if the requirements are foggy.
- ❑ Instead, start simple, add complexity to the statement one layer at a time, and review the results each time.
- ❑ For cases in which the Production data is truly enormous, it might be worthwhile to construct (concoct?) test data for demonstration purposes.

Basic Queries: SQL and RA – Retrieving Data Step 1

1. Look at *all* 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

Sneak peak into inline queries

- ❑ Remember that in RA, the operands and the outputs from any operation are **all** relations.
- ❑ The table in the from clause of a select statement is a physical implementation of a relation, but it could just as easily be the product of another select statement.
- ❑ That “other select statement” could be arbitrarily complex.
 - It can be a view, which is just a compiled and named query that has been created ahead of time.
 - Or an inline query, which is a select statement that is evaluated at run time, and whose output becomes the input for the outermost select statement.
- ❑ Another fun fact about SQL is that the inline queries can be nested however many levels deep that you need.

Basic Queries: SQL and RA – Retrieving Data Step 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!
- ❑ By contrast, **double** quotes are used to denote the name of some object in the database (for instance a table or column).

```
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

Style break

- ❑  In general, **never** put a select * from ... statement into production.
- ❑ The table could have columns added to it after you write the SQL statement, and then you'll get more data back than you originally intended.
- ❑ Also, if, there are columns **removed** from the table, the database will mark any statements that use the missing columns as invalid right away, so that you can do impact analysis from dropping that column.

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

But what if the order **is** significant?

- ❑ Then that needs to be represented in the data somehow.
- ❑ For instance, on my BeachBoard, I have elements under content that are laid out in the order in which they come up in the semester.
- ❑ If I were to represent that in a database, I would have to have another attribute, maybe the date, maybe just a sequence number.
- ❑ When we get into recursive relationships, I will show you (if someone reminds me) how to represent order by daisy chaining the rows together, each row pointing to its successor.

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
- Remember, any time we select columns to display, even if it's *, that's a projection.
- Similarly, even an **unqualified select** is a select.

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 (sigma)) 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 requires 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*. Remember, it's only via a project that we take a subset of the attributes.

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

3. The **project (RA)** operator (written π (pi)) 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:

$\pi_{cLastName, cFirstName, cPhone} customers$.

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

$\pi_{cLastName, cFirstName, cPhone} \sigma_{cZipCode='90840'} customers$.

If you omit the project operator altogether, it is just like doing a select * in SQL, you get all of the columns from the relation scheme.

Constant Values

- ❑ The π operator is not limited just to a column from a table/relation.
- ❑ Soon we will show you how to use expressions of various sorts to calculate/derive values. Those derivations can go into the π operator.
- ❑ A degenerate sort of calculation is just a simple constants, e.g. 'Employees' or 04/22/2020.
- ❑ We will find this final feature of the π operator useful particularly when we get to the union operator.

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. Therefore, there is no order by analog in RA.
 - I **will** say at this point that RelaX, the online tool that we will use for coding and running RA expressions, does support sorting of rows.
 - That operator is the τ . We will use it from time to time to make it easier to compare RA output with SQL output.

Some SQL Select Tools

- ❑ *like* allows for wildcard searches with % and _
- ❑ *upper()* converts its argument to uppercase.
- ❑ *lower()* does what you would expect
- ❑ *between* <low> and <high> works for any datatype with an implicit order.
 - It is inclusive
 - Includes dates **and** figs
- ❑ Dates have several formats yyyy-mm-dd is probably the easiest and is the default in Derby.
- ❑ Not -
 - ~~Where <column> != <value>~~ do not use this, it is not standard
 - Where not <column> = <value>
 - Where <column> <> <value>
- ❑ Null is not a “real” value. Instead of where <column> = null, you have to say where <column> **is null**.

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***.
 - An example of a data dictionary query can be found [here](#).

Another Data Dictionary Query

```
select  schemaName, tablename, columnname, columnDataType
from    sys.SYSSCHEMAS inner join
        sys.systables using(schemaid)
        inner join sys.syscolumns
        on sys.syscolumns.referenceid = sys.systables.tableid
order by tablename, columnname;
```

Basic SQL Statements: DDL and DML

– DDL- CREATE Table 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.
- In the forward engineering process, this is the first time that we specify the actual datatype of the attribute. This is but one example of the binding process.

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 must 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_XX` (for example, `Orders_Customers_fk_01`). 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.
- The two-digit number on the end allows you to have more than one referential integrity constraint from the same parent to the same child.

The Purpose of Naming Conventions

- You may wonder why I give you conventions for naming the primary and foreign key constraints.
- If one of those constraints ever gets violated, the error message will include the constraint name.
- So far, we only work with a small number of tables, but soon, that will change.
- Note that Derby does not allow you to name the backing index. Oracle does:
 - Alter table <name> add constraint <constraint name> unique using index (create index <index name> on <name> (<list of columns>))

Another Alter Table Statement

- ❑ In addition to adding primary key, uniqueness constraints, or foreign key constraints, you can add additional columns:
 - `alter table customers add column stateofbirth varchar(2);`
- ❑ Say you thought that you were going to use the state **code** but then found that the customer wanted the full state name. Then you would:
 - `alter table customers alter column stateofbirth set data type varchar(50);`
 - Note, sadly, this last statement's syntax varies from one RDBMS to the next.
- ❑ To **drop** a column, it's as you would think:
 - `alter table customers drop column stateofbirth;`
 - What do you think the limitations of drop column are?

Changing an Existing Column

- ❑ `alter table <table name>`
 `alter column <column name> not null;`
- ❑ `alter table <table name>`
 `alter column <column name> set data type`
 `<new datatype>;`
- ❑ If all else fails, drop the table, but know that could render any number of things broken.
- ❑ Note that the above are only valid in Derby.

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 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> (<col1>, ... <col n>)  
VALUES (<value 1>, ... <value n>);
```

- The comma-delimited list of values must match the list of values 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 can insert one **or more** rows into a given table with a single insert statement.

INSERT Statement (Continued)

```
INSERT INTO Person(      pID, pFirstName, pLastName,  
                          pBirthday, pAddress, pPhoneNumber )  
VALUES  
( 1, 'John', 'Doe', '1956/04/11', '1154 Walnut St Long Beach, CA', '555-454-6598' ),  
( 2, 'Barbara', 'Stewart', '1925/09/07', '789 West St Los Angeles, CA', '626-468-7445' ),  
...  
;
```

- ❑ The values clause simply repeats the pattern of (<list of values>), over and over again to insert multiple rows.
- ❑ Obviously, the values for **all** of the rows have to be in the order set by the list of attribute names initially.

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!!!
- Just as a point of reference, some database management systems have what's termed "flashback memory" which will, on a limited basis, allow you to "undo" something that you've done, but don't bet on it!

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***.
- To really understand what the join does, consider the customers & orders tables:

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

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

Join Syntax

```
SELECT      *  
FROM        customers INNER JOIN orders  
            USING (cfirstname, clastname, cphone);
```

- ❑ The using clause tells SQL which column(s) in the orders table identify the customer.
- ❑ We generally always join using the migrated foreign keys. There are exceptions.
- ❑ SQL does not make any assumptions about migrating foreign keys.
- ❑ In this case, the names of the keys are the same in the customers table and the orders table. We can “role name” them to make their function more explicit.
- ❑ The **INNER JOIN** only returns rows that have a matching row in both the left- and right-hand operand to the join operator.

SQL Technique: Join Types – Inner Join Using

- The INNER JOIN .. USING specifies the columns to join.
 - That is, exactly which columns in the right hand operand to the join operator must match columns with the same name in the left-hand operand to qualify as a “match”.
- If we give customers a surrogate, then the join becomes:

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

$$r \bowtie_{a,b} S$$
$$\text{customers} \bowtie_{\text{custID}} \text{orders}$$
- This form of the join is more specific than the natural join, because it specifies the column names, but it still doesn't allow for role naming the columns.
- For that, we will need to use the join ON.

Natural Join

- ❑ The online tutorial starts the join discussion with the natural join, so we must deal with it. But do not use the natural join in this class.
- ❑ The liability of the natural join is that it assumes that you want to join over all columns with the same name.
- ❑ The natural join syntax is easy, requires less typing, but is dangerous. I have a demo of the problems that you can get into in the checklist.
- ❑ For INNER JOIN, always either use the join using or the join on (coming right up).

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, there is no way to match rows occurring in S to rows occurring in R), the schemes are still compatible and every tuple from relation r is pasted to every tuple from relation s , with all 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.

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.

```
 $\pi$  cLastName, cFirstName, prodName  
 $\sigma_{cFirstName='Alvaro' \text{ and } cLastName='Monge'}$   
(customers  $\bowtie_{custID}$   
orders  $\bowtie_{custID, orderdate}$   
orderlines  $\bowtie_{UPC}$  products)
```

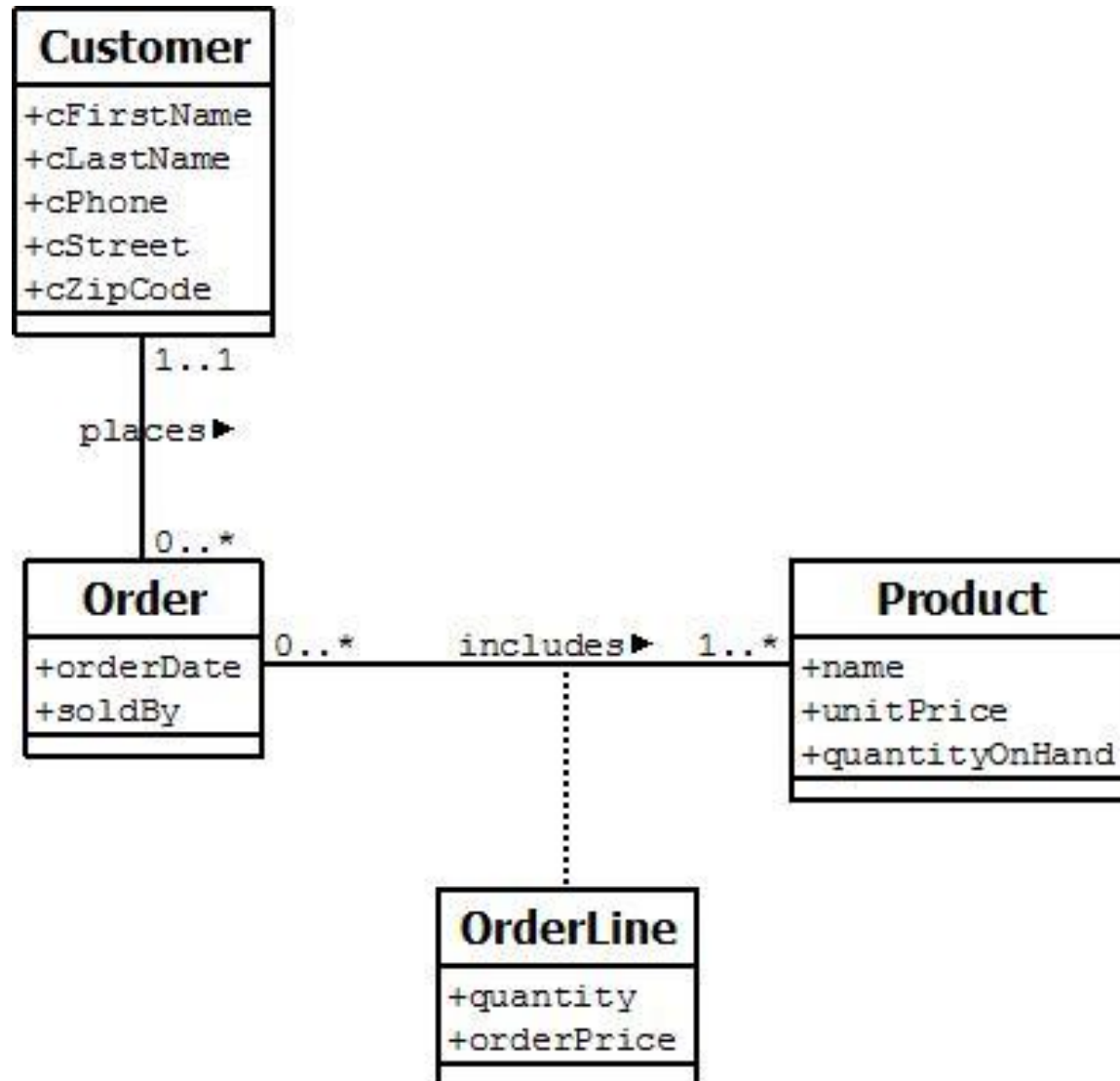
- *The order in which we join the tables together is important. Remember that the \bowtie operator returns a relation as its output, and it takes **two** operands.*

RA reminder

$\pi_{cLastName, cFirstName, prodName}$
 $\sigma_{cFirstName='Alvaro' \text{ and } cLastName='Monge'}$
 $(customers \bowtie_{custID}$
 $orders \bowtie_{custID, orderdate}$
 $orderlines \bowtie_{UPC} products)$

- ❑ Remember that the above statement is composing RA expressions.
- ❑ The three-way join yields a relation
- ❑ Which is then passed to the select (σ) to specify the rows that we want (horizontal projection)
- ❑ Which in turn is passed to the π operator to perform the vertical projection so that we only see the columns of interest.

Let's see that in pictures



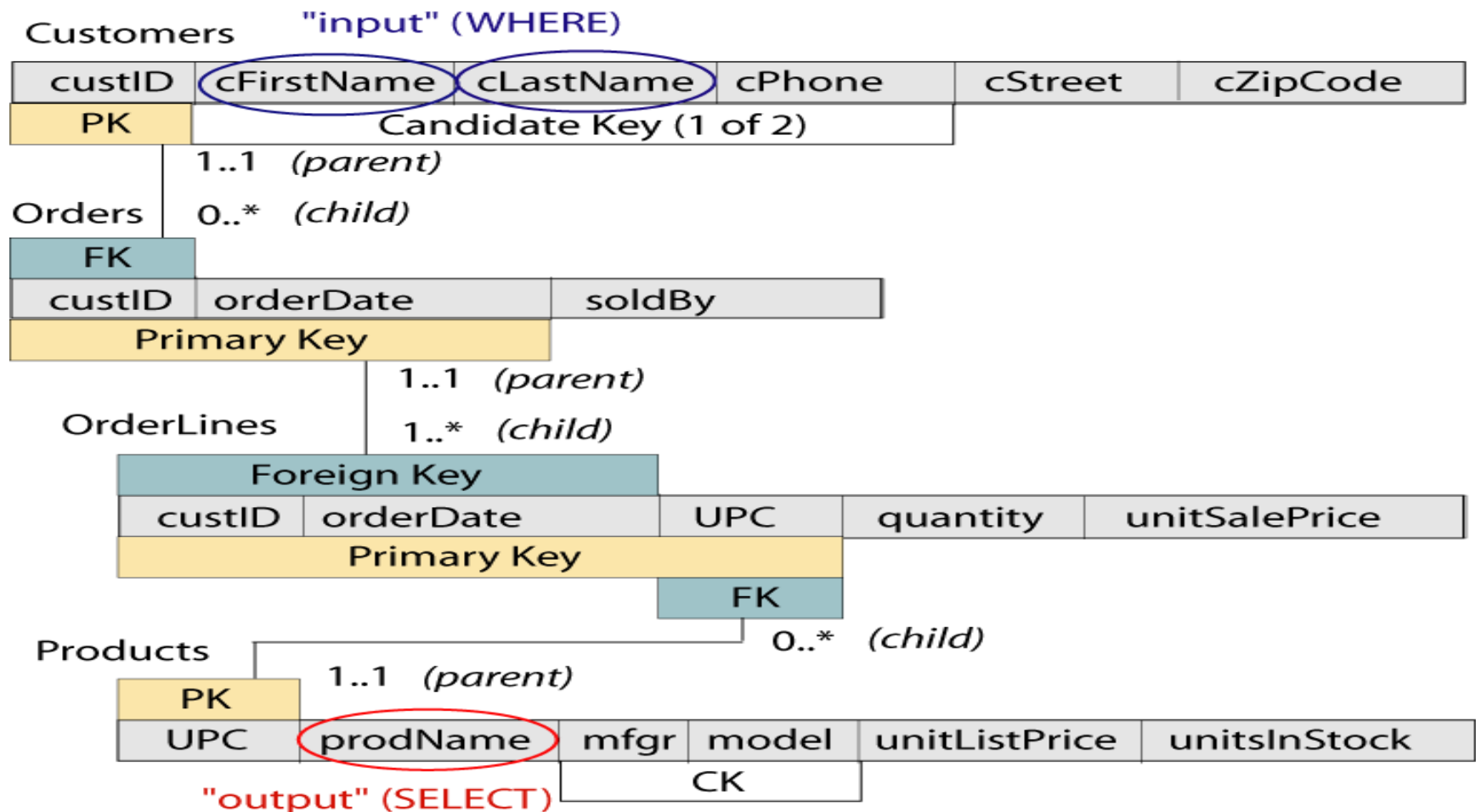
Brief aside relative to join order

- Remember that when you join, you take two relations as operands and create a new relation.

So, $(customers \bowtie_{custID} orders \bowtie_{custID, orderdate} orderlines \bowtie_{product_code} products)$
is equivalent to: $((customers \bowtie_{custID} orders) \bowtie_{custID, orderdate} orderlines) \bowtie_{UPC} products)$

- The implication is that if we want to pull in any products at all, we must be careful to do our join in the right order. Customers cannot be directly joined to products. You must first correlate to the orders, then to the orderLines, and finally to the Product.

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



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

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

```
SELECT * FROM customers
```

```
INNER JOIN orders USING(custID)
```

```
INNER JOIN orderlines USING(custID,  
orderdate)
```

```
INNER JOIN products USING(UPC);
```

- The primary key attribute names are duplicated exactly in the foreign key attribute names. That is not always going to be the case.
 - For those situations, we will need to use the JOIN ON (stay tuned)

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
INNER JOIN orders USING(custID)
INNER JOIN orderlines USING(custID,
orderdate)
INNER JOIN products USING(UPC);
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
INNER JOIN orders USING(custID)
INNER JOIN orderlines USING(custID,
orderdate)
INNER JOIN products USING(UPC);
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

Caveats about distinct

- ❑ When you see duplicate rows, don't just slap a distinct on the select and call it good.
- ❑ Review the data, and understand **why** you are getting duplicate rows.
- ❑ Often, it's because your select statement (the project portion if you will) has trimmed down the columns of interest to the point that you no longer see the reason for the duplicates.
- ❑ The distinct keyword causes the database to perform a sort before producing the output.
- ❑ Which doesn't guarantee that the output will be in any order (no I **don't** know why).

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
INNER JOIN orders USING(custID)
INNER JOIN orderlines USING(custID, orderdate)
INNER JOIN products USING(UPC);
WHERE cFirstName = 'Alvaro' AND cLastName =
'Monge';
```

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 – Join ON

- ❑ If you role name attributes from the parent as they migrate into the child, the JOIN USING will not work because the migrating foreign key has changed names.
- ❑ You may be asking yourself why there is not at least a form of the join that pays attention to the foreign key constraints and joins based on that. I honestly have no idea why that isn't part of the language.
- ❑ The solution here is the JOIN ON. Instead of USING, this form of join provides a Boolean expression used to find the matching rows.

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 so that you can be clear which table the column comes from.

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

$$r \bowtie_{a=b} S$$

- ❑ This is the most specific (and requires the most typing) join type of all. It allows you to indicate where columns have been role named.

Demo from the Practice SQL

- ❑ Each customer can have a sales representative assigned to them.
- ❑ That sales representative has an employee number.
- ❑ We role named employeeNumber from employees to salesRepEmployeeNumber in customers to make it clear what it represents.

```
SELECT      *  
FROM        employees INNER JOIN CUSTOMERS ON  
            employees.employeeNumber =  
            customers.salesRepEmployeeNumber;
```

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

- ❑ Try to use aliases that are at least **somewhat** descriptive.

SQL Technique: Join Types – Inner Join – Where Clause

- ❑ An older version of the join uses a WHERE clause.
- ❑ This is more confusing as the join condition is in the same clause as the selection criteria.

```
SELECT cFirstName, cLastName, orderDate  
FROM customers c, orders o  
WHERE c.custID = o.custID;
```

- ❑ We show you this so that you're not caught flat-footed when you find this syntax in older SQL. In order to preserve downward compatibility, this syntax is still supported, but not as clear as the other means of performing a join.

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;       $r \bowtie = s$ 
```
- ❑ 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 directly 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.

Why we don't store derived data



- Assuming that the data needed to perform the derivation **is** in the database, storing a derived value will:
 - Consume space needlessly
 - Need to be kept up to date whenever a change occurs in the source data
 - Need to be somehow protected so that no one can update it directly (since the value **should** be calculated).
 - And don't forget, this could be nested. There could be derivations that are, at least in part, based on yet other derivations.

What 5 megabytes looked like



- ❑ This is a picture of an IBM hard drive being loaded onto an airplane in 1956.
- ❑ It weighed in at over 2,000 pounds.
- ❑ In case you're wondering, I was born in 1957.

Space isn't everything

- Another liability with storing derived data is that its pedigree is not always evident.
 - Particularly in the case of nested derived data, the hops along the way that the data made to get to its “final” destination may be far from self-evident.
 - Data that is not 100% current must be assumed to be out of date. How critical the time lag might be is a strong function of the use of the data.
 - As data architects, we do not always have the luxury of controlling exactly how data gets used.

Ways to **simulate** storing derived data

- Create a view that performs the calculation (more on that later)
 - The calculated data looks just like a column
 - The user never has to see that it **is** a derivation, let alone the guts of the derivation.
- Write your own function that performs the derivation
 - The function can be used in SQL just like any other function (like `year(...)` and so on) so it looks very organic.
 - Again, the function provides encapsulation.
 - We will show you how to do that in MySQL starting next week.

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 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 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.

Extensions to the Group By

- We've said that the columns shown in the select statement must either be an aggregate function or one of the columns named in the group by clause.
- SQL 99 and later allows for other columns to be named in the select clause if there is a functional dependency from one or more of the grouped by columns to the column in question.
 - In other words, if there is only one value of the column that could appear for any given row, then that column can be included in the select.
 - Be aware that this feature is not available in all RDBMSs.

Relational Algebra

- The Aggregate functions are denoted by:
 $G_{function1(column),function2(column)...\textit{relation}}$ where:
 - G is an aggregate function.
 - Function is “sum”, “avg”, ...
 - Use “as” to give then an alias
 - Column is the column within relation that the aggregation operates on
 - Relation is an expression denoting a relation
- If there are subscripts **ahead of** the G , those represent a list of columns in relation to group by.
- Unlike SQL, the sigma takes the place of the having clause.

The CASE Construct

- The final output is a single column in the output.
- Acts much like the case statement in other languages.
- Say you want to write a report from the Classic Models database that shows all of the cities.
 - Some have just a customer in them
 - Some have just one of our offices in them
 - Some have both
- We want to indicate which category each city falls into.

The Case Construct Continued

- ❑ To start, we need a list of all of our cities:
select distinct country, city
from offices off
union
select distinct country, city
from customers) cities
order by country, city;
- ❑ The union brings in city/country from both Customers and Offices. The distinct removes any duplicates, either from a city being in both tables, or more than one office or customer in the same city.

Then we add the CASE:

```
select country,
       city,
       case when exists (
         select 'X'
         from   customers
         where  city = cities.city and
                country = cities.country) and
         exists
         (select 'X'
          from   offices
          where  city = cities.city and
                 country = cities.country) then 'Both'
       when exists (
         select 'X'
         from   customers
         where  city = cities.city and
                country = cities.country) then 'Customer Only'
       when exists
         (select 'X'
          from   offices
          where  city = cities.city and
                 country = cities.country) then 'Office Only'
       else 'N/A'
       end as Commonality
from   (
  select distinct country, city
  from   offices off
  union
  select distinct country, city
  from   customers) cities
order by country, city;
```

How it works

- ❑ Each `when` clause is tested in turn.
- ❑ As soon as the `case` construct finds a `when` clause that succeeds, it evaluates to that corresponding expression, and exits.
- ❑ This is **not** like the Java `switch` statement since you do not need anything like a “`break`” to get it to keep from “falling through” to the next clause.
- ❑ There is an “`else`” clause that you can use to trap anything that does not match any of the `when` clauses. That serves as a default.

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

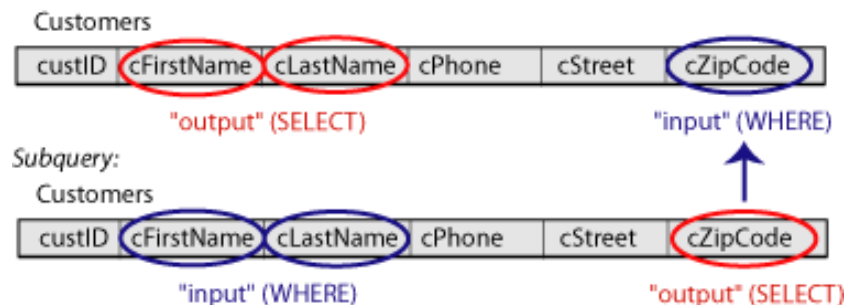
- First, we find the right zip code by writing another query:

```
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.




The ghetto solution

- Knowing that Wayne's zipcode is 90840, we simply code:

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

-  But don't **ever** do it this way.

The above is for demonstration purposes only, don't do this!

- ❑  Remember that the above “two part” approach is just to demonstrate the concept behind a subquery. Do **not** use this during the lab midterm. It will cost you points.
- ❑ Don't do this in your code either. If you find yourself needing a subquery and are tempted to do one query, store the results of that in a Java variable, then do a second query based on that result, stop. The optimizer is able to make the single query more efficient.

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');
```

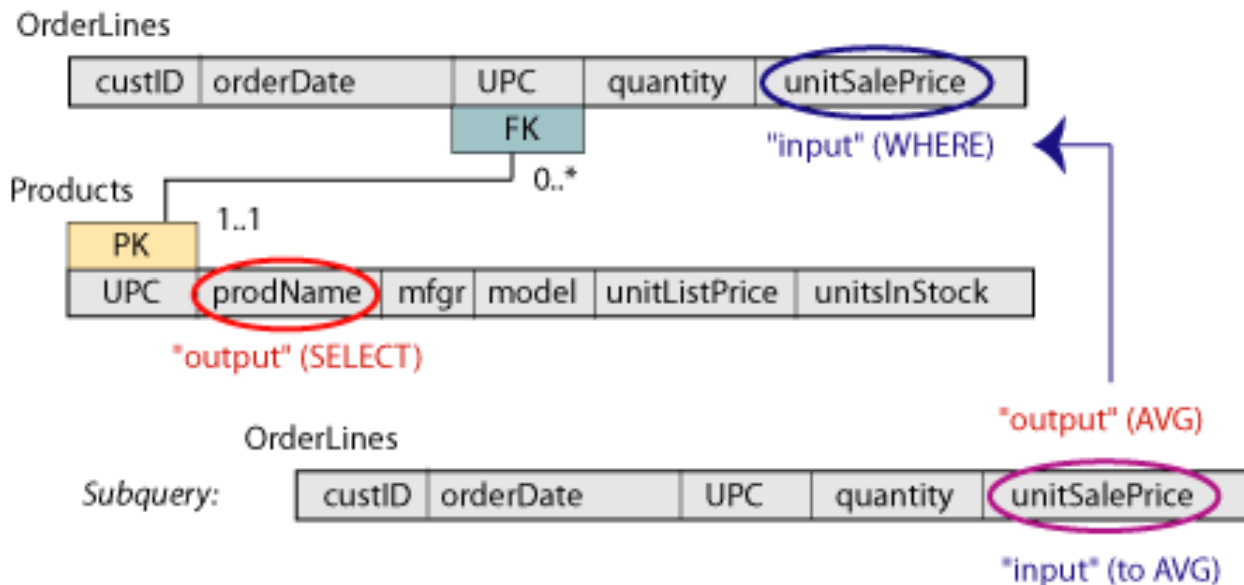
Customers		
cfirstname	clastname	czipcode
Alvaro	Monge	90840
Wayne	Dick	90840

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: Subqueries – Multiple Subqueries

- ❑ List the author(s) of the book with the highest sales.
- ❑ This is a fairly complex example. You have two values that are unknown
 1. Find the highest sales
 2. Find the corresponding title
 3. Find the author(s) who wrote that book

SQL Technique: Subqueries – Multiple Subqueries

1. `select au_fname, au_lname from authors where au_id in ?`
2. `select au_id from titles inner join title_authors where sales=?`
3. `select max(sales) from titles`

Run each of the three queries individually first to check your work

SQL Technique: Subqueries – Multiple Subqueries

1. `select au_fname, au_lname from authors where au_id in`
2. `(select au_id from titles natural join title_authors where sales=`
3. `(select max(sales) from titles));`

SQL Technique: Subqueries – Correlated Subqueries

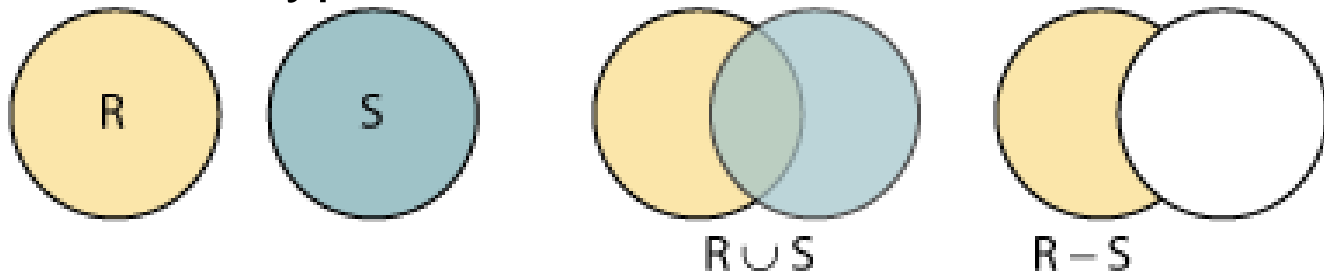
- ❑ A subquery that relies on information from the outer query is called a **correlated** subquery.
- ❑ List the product, their MSRP, and the average MSRP for all of the products within that product's product line:
select productName, MSRP, (
 select avg(msrp)
 from products
 where productLine = **prod**.productLine)
 as "Product Line Average MSRP"
from products as prod
order by productName;
- ❑ Note that in the subquery, we use the alias for the products table in the outer query.
- ❑ We reference data from the “current” row of the outer query.

SQL Technique: Subqueries – Subquery as a relation

- ❑ A query returns a relation, we've said that over and over again. As such, that relation can be joined to, just like any other relation.
- ❑ This looks rather alarming when you first see it in a query, but, with practice, and good attention to the details of putting whitespace into the text, it begins to make a good deal more sense.
- ❑ Pick up with #34 in the Practice SQL

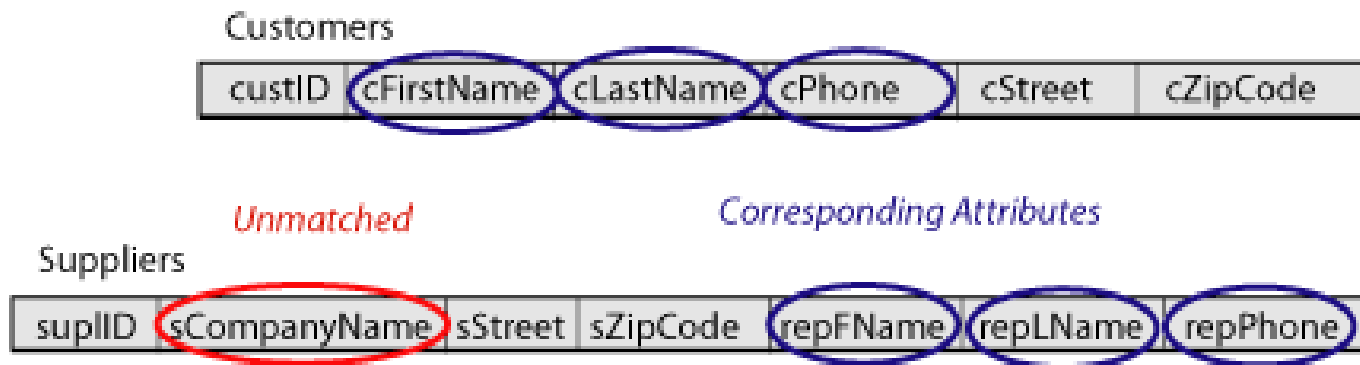
SQL Technique: Union and Minus - Set Operations on Tables

- ❑ **Union** includes members of both sets with no duplicates, **minus** (keyword **except** in Derby) 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 must 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 must 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

The Power of the Union

- ❑ The union encapsulates the distinctions between the sets that are brought together.
- ❑ You can have a union with two or more sets of rows, there is no real limit.
- ❑ The output is just one set.
- ❑ Be careful that you do not blend things together in a union that have no business being there, like age and shoe size.

SQL Technique: Union and Minus – Union – Minus

□ 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 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:
 $\{\text{customers who did not make purchases in 2002}\}$
 $= \{\text{all customers}\} - \{\text{those who **did** make a purchase in 2002}\}$

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');
```

Comment on the not in

- ❑ This is an example of a subquery which returns more than one value. Because the *in* operator is designed for multiple values, this works perfectly.
- ❑ As you would imagine, the *not in* operator exists and functions as you would expect.
- ❑ Use of the *in* operator can often be recast using a join, so it's a matter of taste which way you go. Sometimes the *in* is easier to understand.
- ❑ We will also show you another way to do this, using the exists and not exists options.

SQL Technique: Union and Minus – Minus – 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 (like Derby) 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';
```

Doing it in Derby

```
select  au_fname, au_lname
from    authors
except
select  au_fname, au_lname
from    authors inner join title_authors using (au_id)
        inner join titles using (title_id)
where   year(pubdate) = 2012;
```

- ❑ Derby doesn't support the minus operator, but it has an except operator instead.
- ❑ Ironically, if you try to use minus, Derby gets upset by the second select clause, not the minus keyword itself.

Yet another way: the exists operator

```
select cLastName, cFirstName, cStreet, cZipCode
from customers cust where not exists (
    select 'X'
    from orders where to_char(orderDate,
        'YYYY') = 2002
        and custID = cust.CustID);
```

- This is a **correlated** subquery, in which we are using values in the outer subquery to filter the inner subquery.
- You must give the table(s) in the outer query an alias in order to refer to them in the inner query.

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.

RDBMS idiosyncrasies

- ❑ MySQL has no minus or except operator
- ❑ However, you can emulate minus as follows:
 - Select <select list> from relation1 left join relation2 on <join predicate> where relation2.<column name> is null;
 - Note that this use of the join operator rarely relies on migrating foreign keys. So I'd say this is something of a "hack".
- ❑ It also lacks an intersect operator.
- ❑ Several of my students ran into this when trying to refactor from Derby into MySQL at the last minute for the term project.

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.
- Replacing a view also retains any grants that were given on the original view. Otherwise, if you drop the view and recreate it, you'd have to redo all of the grants.
- 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.

Views modularize your select

- ❑ Oftentimes, when a query becomes very complex, it helps to break it down into views, which are then joined together to create the final result.
- ❑ It used to be conventional wisdom to avoid views on top of views, but the optimizer in commercial RDBMSs are getting good enough that they are able to optimize the entire view stack in one pass.
- ❑ The nice thing about composing a final query from views is that you're able to test the lower level views alone to check the query a step at a time. Also, those lower level views can be used again to compose other queries.

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.
- ❑ Obviously, the view has to include all of the not null columns of the base table(s) in order to use it for an insert.

Views and inline queries

- ❑ Remember that we've shown you that you can create a pseudo table by saying:

Select ...

From (select ...

from ...) <some alias> inner join <a table>

...

- ❑ The select inside the parenthesis is evaluated, and produced a relation, which can then be manipulated just like any other relation.
- ❑ That's exactly what is happening when you do a select from a view.

Views and inline queries – an Example

- ❑ Find the employees who work in the same state within the same country with each other.
- ❑ Display each pair of such employees only once.
- ❑ Sort by the last name of the first employee, the last name of the second employee, the first name of the first employee then the first name of the second employee.
- ❑ For those employees who work in an office that has no state, return N/A as the state name and match on that.

Thinking it Through

- We're going to need to join employees to offices in order to get the employee name as well as the location of the office that they work in.
- Once we have that, we need to do it again, and then do a self join between those two joined relations to get the ones with the state and country in common.
- `coalesce(state, 'N/A')` will allow us to map a null to a sensible value.
 - The `coalesce` returns the first non null expression from the argument list.
 - In this case, that would be state if there is a valid state name in that row, or 'N/A' otherwise.

The Original Query

```
select one.lastName, one.firstName, state, country, two.lastName,
two.FirstName
from    (select  lastName, firstName, coalesce (state, 'N/A') as state,
              country, employeeNumber
        from    employees inner join offices using (officecode)) one
inner join
        (select  lastName, firstName, coalesce (state, 'N/A') as state,
              country, employeeNumber
        from    employees inner join offices using (officecode)) two
using   (state, country)
where   one.employeeNumber < two.employeeNumber
order by one.lastName, two.lastName, one.FirstName, two.firstName;
```

Create a Handy View

```
create view CustomerEmployeeOffice as
select  lastName, firstName, coalesce (state, 'N/A') as state, country,
        employeeNumber
from    employees inner join offices using (officecode);
```

- This does the same thing as the inline query, but with far less typing (as you'll soon see).

Use the New View

```
select  one.lastName, one.firstName, state, country,  
        two.lastName, two.FirstName  
from    CustomerEmployeeOffice one  
        inner join  
        CustomerEmployeeOffice two  
        using (state, country)  
where   one.employeeNumber < two.employeeNumber  
order by one.lastName, two.lastName, one.FirstName, two.firstName;
```

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.
- ❑ Oracle has the ability to set up a refresh schedule for its materialized views, or a materialized view can be used as a means of near real time distribution of the base tables.

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.
- ❑ Essentially, the index is a redundant representation of selected columns of the table.

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

- ❑ While not required, it is often advantageous to put a non unique index on a migrated foreign key. This speeds up any deletes in the parent table.
- ❑ 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.

Index Optimization

- ❑ Modern database management systems have gotten extremely good at keeping an index optimized even in the face of massive updates.
- ❑ But those occasions when all or most of the rows of a very large table will be updated or replaced, it is often a good strategy to drop the indexes first, perform the update, then rebuild the indexes.
 - This sort of mass load often occurs in a data warehouse environment.
- ❑ Another strategy that still has merit even after all of this time is to put the indexes into a separate tablespace from the table data.
 - This allows more recovery options if a datafile from an **index** is lost.
 - The index can be suspended and still allow access to the base data (albeit more slowly).

SQL Technique: Views and Indexes – SQL

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

- ❑ The index name is only unique within the schema.

- ❑ 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>;

SQL Technique: Views and Indexes – Migrated foreign keys

- ❑ When it becomes necessary to delete a row from a parent table, the database management system has to check for rows in all of the child tables to make sure that the deletion doesn't leave any “orphans”.
- ❑ That search can be greatly sped up if you create a **non**-unique index across the migrating foreign keys in each child.
- ❑ Note that a given column could occur in any number of indexes within a given table.

Transactions –

Why Do We Need Transactions?

- Many enterprises use databases to store information about their state
 - Remember discussing the *state* of an object, it was the composite of the values of the various instance variables.
 - The state of an enterprise is like that, but on a much grander scale 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

Reality, what a concept

- ❑ The correspondence between reality and the data is often taken for granted.
- ❑ On the production floor, it's critically important that tools and parts never end up where they don't belong on the aircraft.
- ❑ The tools in the tool boxes are clipped into RF transceivers that signal whenever that tool is removed from its place until it's returned.
- ❑ If it's not in the toolbox, it is assumed to be a potential Foreign Object Damage case.
- ❑ This is an example of very close coupling between the changing state of things, and the data used to represent that state.
- ❑ We often think of a transaction, or change of state in terms of some dialogue between a user and an application, but in this case, changes in state are measured directly, that is, the user does not have to physically run an application and make entries.

A quick review of models

- ❑ A model of something is a **limited** representation of that thing.
- ❑ It is limited in that it **suppresses** details that are **not** of interest for the purposes of the users of that model.
- ❑ For instance, a wind tunnel model for an aircraft doesn't need landing gear or seats.
- ❑ The **data** that the enterprise gathers is meant to serve as a **model** of the real world, a representation of those elements of that inner and outer state that are of interest.
- ❑ To be useful, this model must be accurate. To be accurate, that model must be updated faithfully in response to events.
- ❑ To be useful, the model should not contain unnecessary information. This is why I harp on staying within scope when you do your homework.
- ❑ Just because an event fails to get persistently recorded in the database does **not** mean it never happened.

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 database is that it may be possible for two users to try and change the same 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 that, together, form a single event in the business world. Such encapsulation is called a '**Transaction**'.

Transactions - What is a Transaction?

- ❑ A sequence of one or more actions which are one atomic unit of work from the business perspective.
- ❑ Basic operations that a transaction can include are:
 - Reads, Writes
 - Commits, Rollbacks
- ❑ A transaction must either run successfully to completion or get backed out – as if nothing ever happened.
- ❑ What's worse than eating an apple and finding a worm in it?

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)
 - Book transaction: send an electronic message to another application to request a ticket.

Transactions - Outcomes

- After work is performed in a transaction, two outcomes are possible:
 - Commit - All changes made during the transaction are committed (made persistent and available to **other** users) to the database.
 - **Before** the commit occurs, the updates that the transaction makes are **not** available to the rest of the users.
 - This makes sense, because that data is in an inconsistent state until the transaction completes.
 - Abort - **All** the changes made during the transaction by this session 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 and that those changes **persist**.
- 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
 - Depending on the RDBMS, the transaction management can be **optimistic** or **pessimistic**.

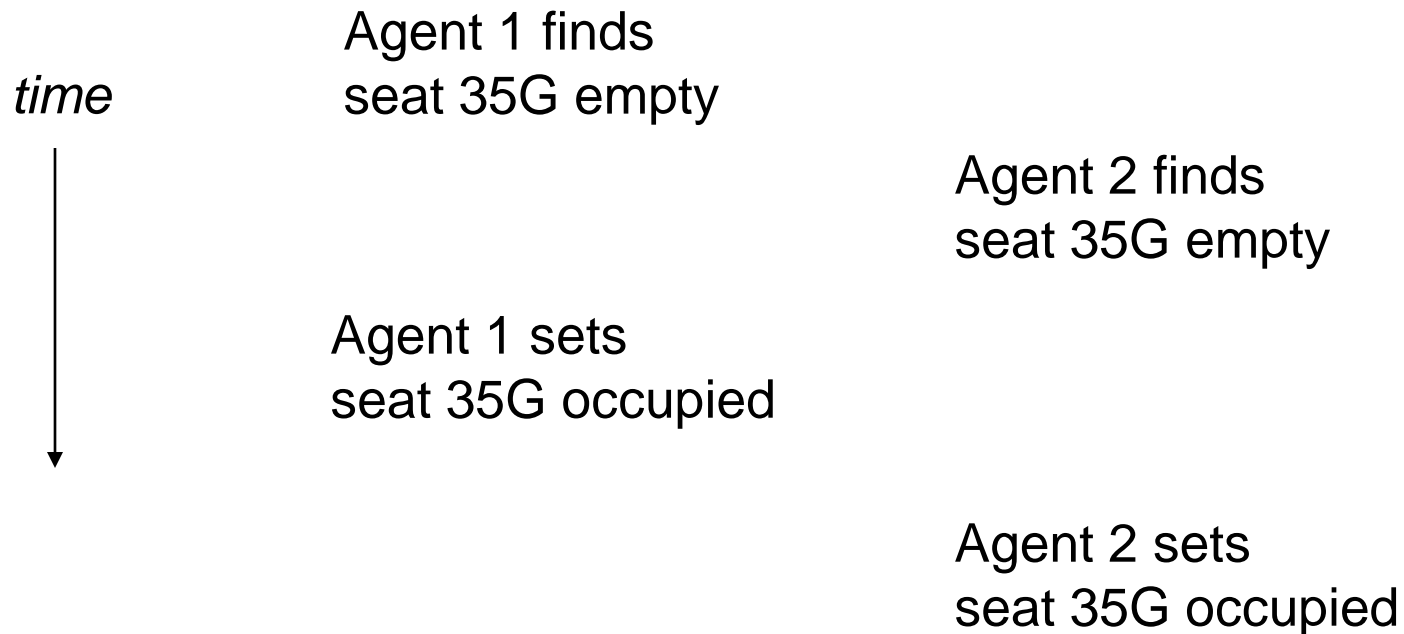
Transactions - Reasons for Abort

- ❑ System crash – server crash, disk crash, network outage
- ❑ Transaction aborted by the application system
 - Execution cannot be made atomic (a site is down in a distributed transaction)
 - Execution did not maintain database consistency (integrity constraint is violated)
 - ❑ At bottom, when we say that an integrity constraint has been violated, another way to look at that is that the database is in a state that violates one or more business rules.
 - Execution was not isolated (more on transaction isolation shortly)
 - Resources not available (timed out on a deadlock)

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 was 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.
 - The data item “A” originally equals 100, after executing T1 and T2, A is supposed to be $100+10-8=102$ but A is 92 because T2 overwrote the update made in T1.

↓	Add 10 To A	Minus 8 from A	Value of A on the disk
	T1	T2	
	Read(A)		100
	A=A+10		100
		Read(A)	100
		A=A-8	100
	Write(A)		110
		Write(A)	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. X starts off at 20.
- 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 in General

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. Sometimes this is called “the last update wins”

Transactions – Uncommitted Dependency

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

Transaction A can READ (or WRITE) item R which has been updated by another transaction but not committed (and in this case ABORTed). This is oftentimes termed “dirty data”.

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 are 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
 - Backs out partial updates if needed
 - Either by restoring the before images
 - Or not writing the after images

Consistency Limitations

- 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
 - So, be aware of the limitations of this “consistency”.
- A consistent **transaction** maintains database consistency and the correspondence between the database state and a possible 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
 - For instance, when adding a child record, if a referential integrity constraint prevents that, prompt the user for enough information to create the parent.
 - Or, when performing a withdrawal, wait for sufficient funds to be posted to the account before concluding the transaction.
 - In Oracle, an exception is thrown when a consistency violation is encountered. The transaction is backed out if there is no handler for that exception – analogous to Java try/catch block.
 - The database structural constraints, (such as referential integrity constraints) are built into the database and are **declarative** in nature.
 - Triggers, check constraints and other such constraints are built into the database as well, but are **procedural**.
- Relying on the OLTP application to maintain consistency only works if that application is the **only** one that will ever update the data.
 - In some cases, even **read** access to the data can be harmful without an interface if the data is complex enough.
 - This relates to the encapsulation of the data.

Transactions – Database Consistency

- Enterprise (Business) Rules limit the occurrence of certain real-world events
 - For instance: “Student cannot register for a course if the current number of registrants equals the maximum allowed”
 - These rules are spelled out in the database by means of primary key constraints, uniqueness constraints, referential integrity constraints, check constraints and triggers.
- Correspondingly, allowable database states are restricted
- These limitations are called (static) integrity constraints: assertions that must be satisfied by all database states
 - **Static** integrity constraints do not limit **transitions** from one consistent state to another.
 - What they **do** limit is which states are allowed and which are not.

States – Another perspective

- ❑ As part of the CS curriculum, you have all had to take two semesters of hardware architecture.
- ❑ In the course of those harrowing courses, you had to build finite state machines (FSM).
- ❑ As you recall, the FSM had a set of one or more registers that recorded the state of the FSM, and combinational logic that reviewed the current state, and the input to decide which state to go to next.
- ❑ If I tell you that the database is a FSM, the user is the source of the input, where are the rest of the elements of the FSM coming from?

Transactions - Isolation

- Serial Execution is one way to achieve isolation: 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)
 - ❑ Each user sees the data as it was at the start of the statement or the transaction (depending on the isolation level selected).
 - ❑ So, each user could, depending on the timing, see a different version of the data.

Multi-version concurrency control

- ❑ Conceptually, each transaction gets its own version of the database at the start.
- ❑ Physically, this is accomplished by logging the changes in undo as well as redo.
- ❑ The undo logs capture the before images of the blocks, the redo captures the after images.
- ❑ Say a read transaction starts on a large table.
 - Half way through, another transaction updates some of the rows that have yet to be read in the first transaction.
 - The first transaction then reads the correct **undo** log for that block to get the image of what it looked like before that update started.
- ❑ Oracle is one of the few RDBMSs that have this sort of concurrency management. Other RDBMSs often rely on explicit locks to prevent inconsistency.
- ❑ See:
https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:27330770500351 for Tom Kyte's explanation.

Two-Phase Locking Protocol

- ❑ Another means to ensure serializability is two phase locking.
 - Berkeley DB
 - And Cubrid both use this style of locking
- ❑ During the **growing phase**, the transaction is acquiring the locks that it needs as it processes the transaction.
- ❑ During the **shrinking phase**, the transaction releases its locks.
- ❑ The transaction cannot go back and forth between acquiring new locks and releasing old ones. Once it releases its first lock, it keeps releasing them until it completes.

Two-Phase Locking Protocol (Cont)

- The rules are simple:
 - A 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.
- There are three variations of the Two-Phase locking protocol: standard, strict and rigorous.

Standard Two-Phase Locking

- ❑ Locks can be released prior to the commit (end of the transaction) if no additional locks are needed by the transaction.
- ❑ In this scenario, an uncommitted transaction that has released a lock could get rolled back.
- ❑ Any other transactions which have read any of the data changed by the first transaction have to be rolled back because they have performed a dirty read.
- ❑ This is called a **cascading rollback**.

Strict Two-Phase Locking

- ❑ The transactions must hold their **exclusive** locks until the entire transaction is ended with a commit.
- ❑ This reduces concurrency because other transactions cannot access the resources until the first transaction completes.
- ❑ But it prevents any occurrence of cascading rollbacks since other transactions cannot perform dirty reads.

Rigorous Two-Phase Locking

- In this protocol, the transaction holds both its exclusive **and** its shared locks until the commit point for the transaction.
- This hampers concurrency even more than strict two-phase locking.

Lock upgrades

- Any of the above two-phase locking protocols can allow for a lock to be upgraded from shared to exclusive during the **growing** phase of the transaction.
- This allows more concurrency since a shared lock will allow reads by other transactions.
- A lock can be **downgraded** during the **shrinking** phase of the transaction.

Time Stamping

- Another way to lock resources is time stamping.
- Each resource has a read-timestamp that gives the timestamp of the last transaction to read the item.
- Each resource also has a write-timestamp that gives the timestamp of the last transaction to write to the given resource.
- Each transaction receives a time stamp at the start of the transaction: $TS(T)$ (time stamp of Transaction T).

Time Stamping (Continued)

- If Transaction T asks to read a data item, P, compare $TS(T)$ with $WriteTimestamp(P)$
 - If $WriteTimestamp(P) \leq TS(T)$ proceed
 - Replace $ReadTimestamp(P)$ with $TS(T)$ if $ReadTimestamp(P) < TS(T)$. Otherwise, just perform the read, do not alter the timestamp.
 - Else
 - This is a late read, the value of P that T needs is no longer available. Roll back T.
- If T asks to write a data item, P then compare both the $WriteTimestamp(P)$ **and** $ReadTimestamp(P)$
 - If $WriteTimestamp(P) \leq TS(T)$ **and** $ReadTimestamp(P) \leq TS(T)$ perform the write and update both timestamps with $TS(T)$.
 - Else
 - This is a late write. Rollback T, restart it, and give it a new timestamp.

Optimistic Serialization

- Assumes that the vast majority of transactions will not interfere with each other.
- The transaction first goes through a read phase to gather data from the database.
- Followed by the **validation** phase where the database checks to make sure that no other transaction created any interference with the transaction.
 - If an error occurred, the transaction is restarted.
- Followed by the write phase that commits everything to disk.

Transactions – Isolation Levels (The “I” in ACID)

- 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
 - And when you do, it can be **very** hard to track down the source of the problem.

Transactions – Read Uncommitted

- ❑ Can read dirty data – essentially you are reading the results of a partial transaction.
- ❑ 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: one step better – Repeatable Read

- Reads of the same **rows** 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;
```

- T2 should still see the same results in the second query because that is part of the same transaction.

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.
- In other words, there is no chance of a "race condition".

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

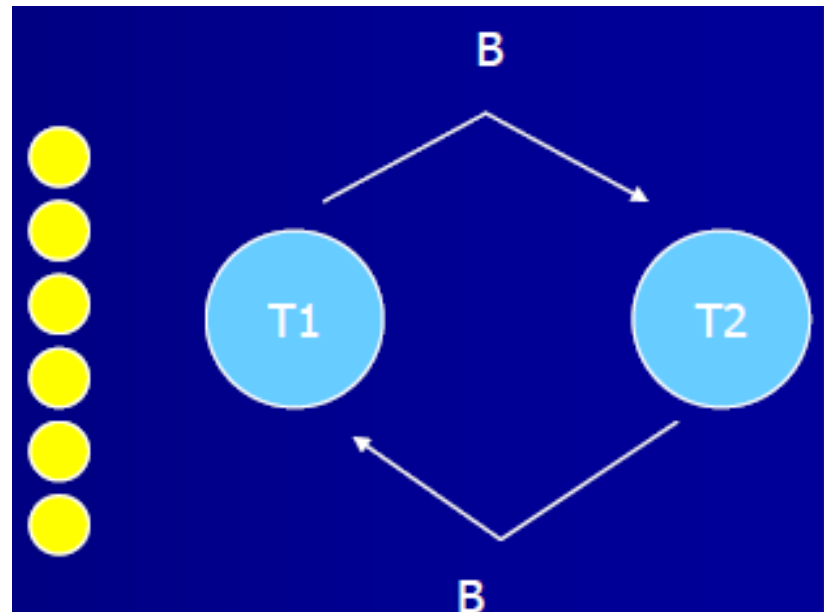
To draw one:

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:

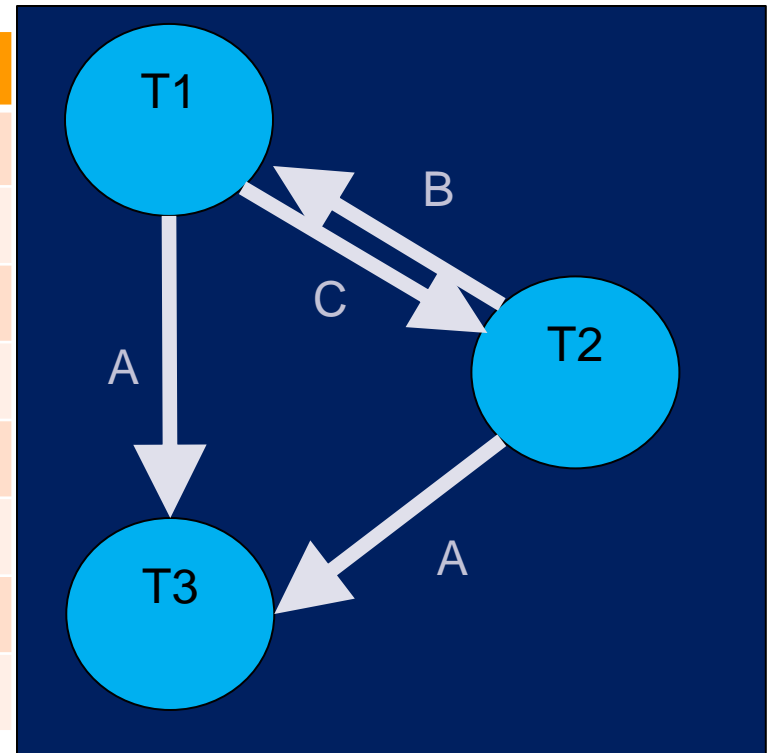
Time	T1	T2
t1	READ(A)	
t2	READ(B)	
t3		READ(A)
t4		READ(B)
t5	WRITE(B)	
t6		WRITE(B)



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)	



Transactions – Durability (the “D” in ACID)

- The system must ensure that once a transaction commits, its effect on the database state is not lost despite 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
 - A media failure during a transaction could result in a loss of atomicity, leaving the data in a “twilight” state where the transaction is only partially completed.

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 makes the commit operation very expensive
 - **Optimistic** committing writes the changes to disk and manages the before images for transaction isolation purposes and undo.
 - **Pessimistic** committing holds the changes in abeyance until a commit, and then writes them.
 - Logging
 - Redo logging, distinct from **undo**, allows for changes to be reapplied to the database in the event of a crash.
 - The logs are written off to tape or some other secondary media on a continuing basis to protect the database from hardware failures.

Transactions – Implementing Durability

- Database stored redundantly on mass storage devices to protect against media failure
 - The redo logs are transported to another server as they are generated and applied to a standby database.
 - This hot standby will serve as the primary if the primary database becomes unavailable – serves as a backup database as well as business continuity.
- 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 and redundant processors)
 - Recovery based DBMS (log)
 - Shadow databases that assume the primary role quickly if needed

RAID

- ❑ Redundant Array of “Inexpensive”/Independent Disks
- ❑ RAID 0 striping – total capacity is the sum of the disks in the array. Fosters concurrent I/O on the disks in the array → greater performance, but no improvement in reliability.
- ❑ RAID 1 – mirroring – can be mirrored more than once. One database backup strategy is to “break the mirror” during the backup and then re mirror the disks once the backup is complete. Write performance suffers because each of the disks in the array has to be written to at once.
 - Usually this strategy is only done when there are two or more mirrors of the same disks.

RAID – continued

- ❑ RAID 4 – block-level striping with dedicated parity. The parity is computed by XOR'ing a bit from drive 1 with a bit from drive 2 and storing it on drive 3. If two of those three drives are viable, the data on the third can be computed.
- ❑ RAID 5 – block-level striping with distributed parity. Any one drive can crash and the array is still viable.
- ❑ RAID 6 – block-level striping with double distributed parity. Up to two drives in the array can be down, and the array is still viable.

RAID – continued

- ❑ RAID 0+1 creates two stripes and mirrors them. If one drive in the array fails, then you are back to RAID 0.
- ❑ RAID 1+0 creates a striped set from a series of mirrored drives. The array can sustain multiple drive losses as long as no mirror loses all of its drives.

Transactions SQL commands

- ❑ **begin transaction** will commence a transaction. Oracle and DB2 starts theirs implicitly, so this statement is not needed, but SQL Server and MySQL respect this statement.
 - And, as luck would have it, the syntax for MySQL is **start transaction**.
- ❑ **commit** will make permanent (and visible to other transactions) any uncommitted updates that have been made in that session.
- ❑ **rollback** will remove the uncommitted updates done in that session from the database. It will be as though they never happened.

Transactions – SQL - savepoint

- ❑ **savepoint** <identifier> allows the developer to set a point in time for a rollback point. To use a savepoint:
 - rollback work to savepoint <identifier>;
- ❑ You can reuse a savepoint identifier. The savepoint is moved to the new location in the transaction log each time that name is used.
- ❑ A commit removes all savepoints.
- ❑ Once a rollback to a given savepoint has occurred, the savepoints after that savepoint are erased.

Commits - Guidance

- ❑ Don't use a commit unless you have:
 - A very large data load process that can be easily restarted at an arbitrary point.
 - You have a application that features discreet transactions that are easily identifiable in the user's minds.
 - ❑ For instance, placing an order or something that is easily identified by the user as a single logical action.
 - ❑ Otherwise, they will have to try to figure out where to pick up and restart.
- ❑ This varies from DBMS to DBMS, but certainly Oracle does a very good job of managing the undo space for you, you don't need to force it.

Setting the isolation Level-MySQL

▣ See: <http://dev.mysql.com/doc/refman/5.7/en/set-transaction.html>

SET [GLOBAL | SESSION]
TRANSACTION

transaction_characteristic [,
transaction_characteristic] ...

transaction_characteristic:
ISOLATION LEVEL level
| READ WRITE
| READ ONLY

level:
REPEATABLE READ
| READ COMMITTED
| READ UNCOMMITTED
| SERIALIZABLE

Setting the isolation level: Oracle

- In Oracle, the syntax is slightly different, but the intent is much the same.

- Please see:

<https://docs.oracle.com/database/121/CNCPT/consist.htm#CNCPT020>

- At the start of a transaction:

SET TRANSACTION ISOLATION LEVEL READ COMMITTED; (default)

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SET TRANSACTION READ ONLY;

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.

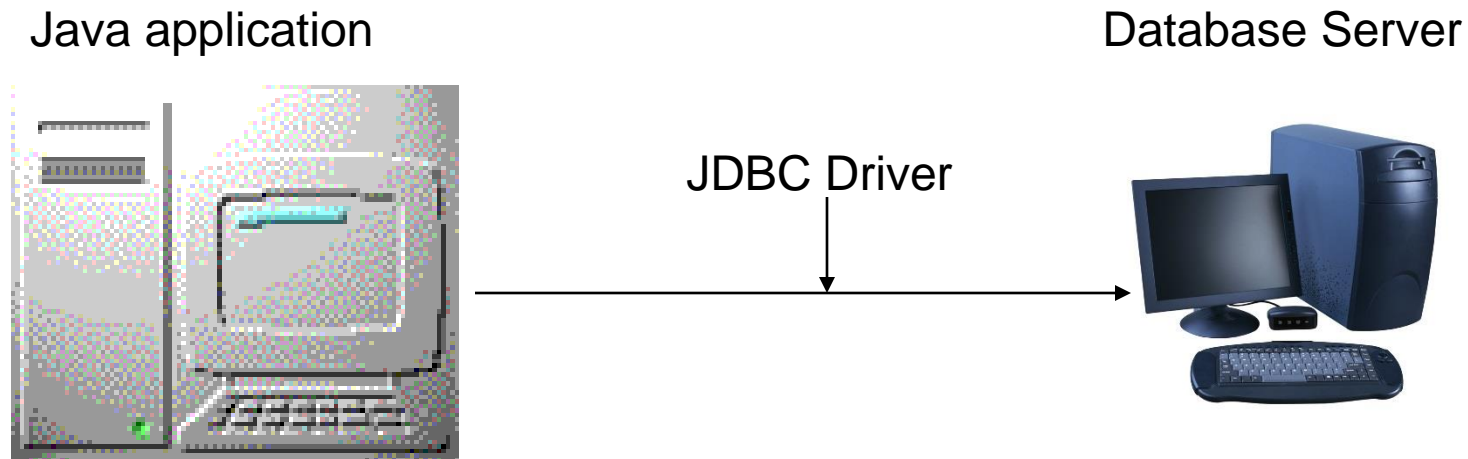
JDBC Classes and Interfaces

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 Eclipse 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

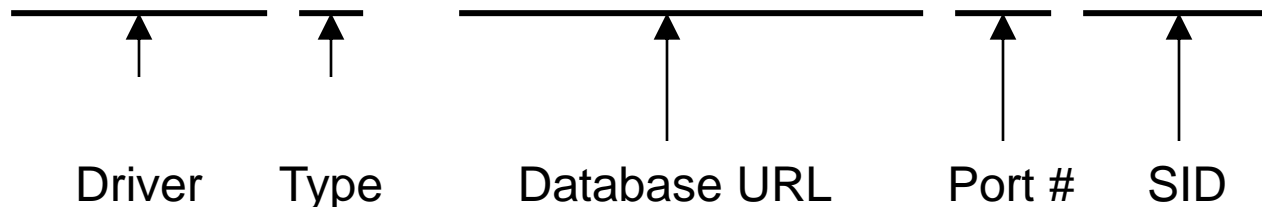
Establishing a Connection

- 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.)

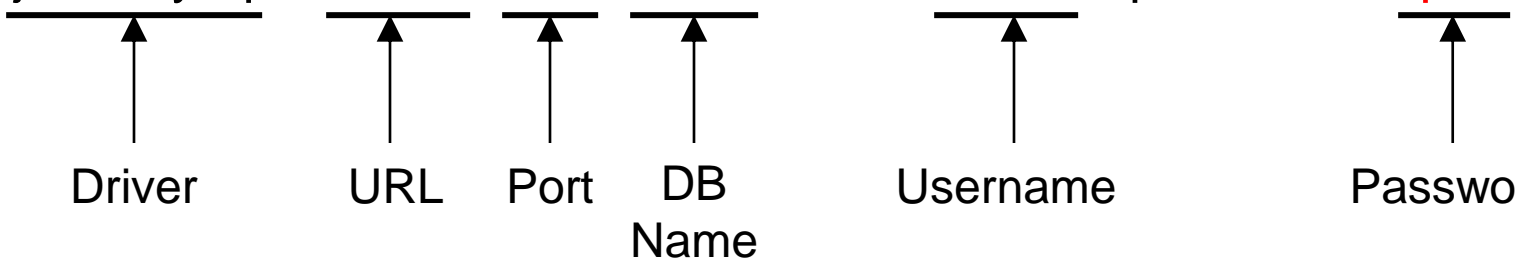
Oracle Example

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



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 illustrates the components of the JDBC URL string. Arrows point from the following labels to their corresponding parts in the string: 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())  
{  
    float s = rs.getFloat("id");  
    String n = rs.getString("name");  
    System.out.println(s + " " + n);  
}
```

Connection Class Interface

- `public boolean getReadOnly()` and `void setReadOnly(boolean b)`
Specifies whether transactions in this connection are read-only
- `public boolean isClosed()`
Checks whether connection is still open.
- *`public boolean getAutoCommit()` and `void setAutoCommit(boolean b)`
If autocommit is set, then each SQL statement is considered its own transaction. Otherwise, a transaction is committed using `commit()`, or aborted using `rollback()`.*

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

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();
```

ResultSets

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

```
ResultSet rs=pstmt.executeQuery(sql);  
// 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()`

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()

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>