Module 3

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 3 Study Guide and Deliverables

Background Concepts Readings:	• Coronel & Morris, chapter 5
Optional SQL Readings:	• *12th Edition:* Coronel & Morris, section 8.3 and 8.4 of chapter 8. Note that section 8.2 will be read in module 5 • *13th Edition:* Coronel & Morris, sections 7.7, 7.9, 7.10, and 7.11 of chapter 7 (note that section 7.8 regarding subqueries will be read in week 5). Section 8.5 of chapter 8.
Assignments:	• Term Project Iteration 3 due **Tuesday, September 27 at 6:00 AM ET** • Lab 3 due **Tuesday, September 27 at 6:00 AM ET**
Live Classroom:	• **Tuesday, September 20 from 8:00-9:30 PM ET** • **Wednesday, September 21 from 8:00-9:30 PM ET**

Which First?

Read the book chapters before reading the online lectures.

Lecture 7 - Advanced Data Modeling

Introduction

Extended Entity-Relationship Models

metcs669_module06 video cannot be displayed here

Entity-relationship modeling is missing the ability to represent relationships based on specialization and generalization. For example, you can't directly represent that students and faculty are people in an ERD. This shortcoming is addressed in Extended Entity-Relationship Modeling, which includes specialization-generalization relationships.

Abstraction, Entities, and Classes

Abstraction means identifying the common characteristics of things and using those common characteristics to classify or organize things. We used abstraction when we identified entities and produced Entity-Relationship models.

Abstraction is the basis of language. Words correspond to classes into which we organize things based on their common characteristics, and about which we frequently need to communicate. For example, we use the word *person* to communicate about entities with the commonly shared characteristics of people, and we use the word *read* to communicate about the activity in which we are engaged right now.

There are many more commonly shared abstractions than there are words, and most commonly shared abstractions do not have single-word names. Many commonly shared abstractions are referred to by common phrases. For example the phrases "race horse" and "slide presentation" are familiar abstractions, yet neither has a common single-word English name.

The terms ***class*** or ***entity type*** refer to a category into which things are placed based on their common characteristics. Words correspond to classes.

Classes may represent categories which have particular examples, or instances, or classes may represent categories that are too general or abstract to have particular instances. For example, if you went to a bank and said that you want to open an account, the banker would immediately ask you "What type of account?", because the class *account* is too abstract or general for the bank to open one for you. When you tell the banker "A checking account" or "A savings account" then the banker has enough information to begin to create for you an instance of the class of account that you asked for.

In object oriented thinking we distinguish between ***abstract classes*** and ***concrete classes***. Abstract classes are categories that are too abstract to have meaningful instances. For example the abstract class corresponding to the word *animal* is too abstract to be useful for answering questions such as "How much does an animal weigh?" The problem is that both shrews and whales are animals, and their weight differs so greatly that there is little useful knowledge about the weights of animals that can be associated with the abstract class *animal.* That is not to say that abstract classes such as the *animal* class are not useful. For example, we can abstract a great deal of knowledge that applies to all animals, such as that they eat and reproduce, and associate that knowledge with the abstract *animal* class.

Test Yourself 3.1

Which of the following are true? (Please check all of the following that are true.)

Ostrich, Robin, and Penguin could be instances or examples of the same class.

This is true. Class or entity type refers to a category into which things that share common characteristics are placed. (Birds have wings and hatch from eggs.)

An example of an abstract class could be a vehicle class.

This is true. Examples of more concrete classes would be truck and UAV (unmanned aerial vehicle).

An example of a concrete class could be a motorcycle class.

This is true. This class would be more useful for answering a question such as “How many wheels does it have?” than a vehicle class.

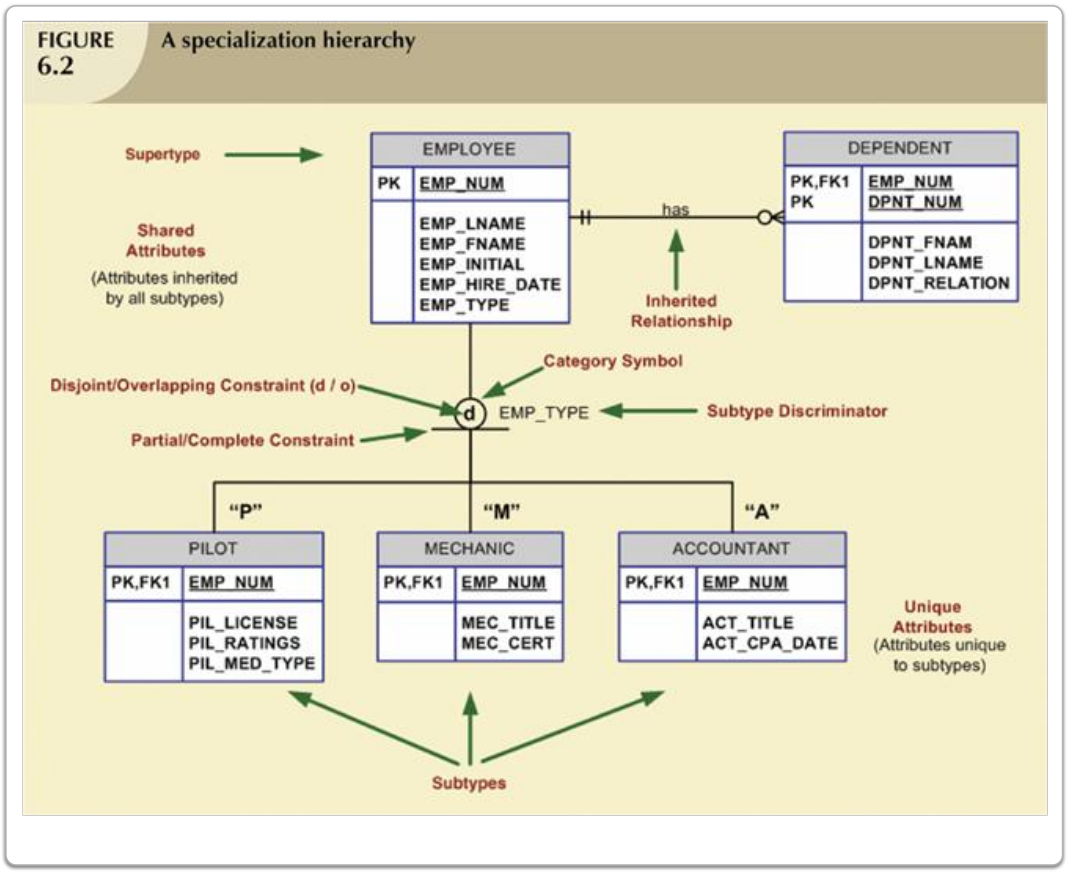
Specialization and Generalization

It is very useful to organize entity types (termed *classes* in object-oriented terminology) according to their specialization and generalization relationships. Specialization and generalization relationships are one of the basic ways that we understand our world and experiences. For example, if I were to ask you "Does a canary have skin?", after a moment you would be able to answer "Yes," even though you probably have never seen a canary's skin or studied canary skin. We answer questions like this by remembering that canaries are birds, and that birds have skin. We associate the general knowledge about birds with the abstract class *bird*.

A *specialization hierarchy* (*taxonomy*) is a hierarchical structure that we use to organize our knowledge based on specialization and generalization relationships. Scientists who study birds have a complex *taxonomy* into which they classify birds, and the scientists associate knowledge about birds with the classes in that taxonomy. You may be familiar with the classical biological taxonomy with its named levels of abstraction (kingdom, phylum, class, order, family, genus and species), in which only the species level is concrete.

Entity-relationship models and relational databases have no way to conveniently represent specialization and generalization relationships. This is a major shortcoming that makes entity-relationship modeling inadequate to model complex enterprises. Extended entity-relationship modeling (EERM) is the result of adding to entity-relationship modeling the ability to represent specialization and generalization relationships.

The following figure from the text is an extended entity-relationship diagram that illustrates the specialization relationships between three subtypes of employees at an airline.

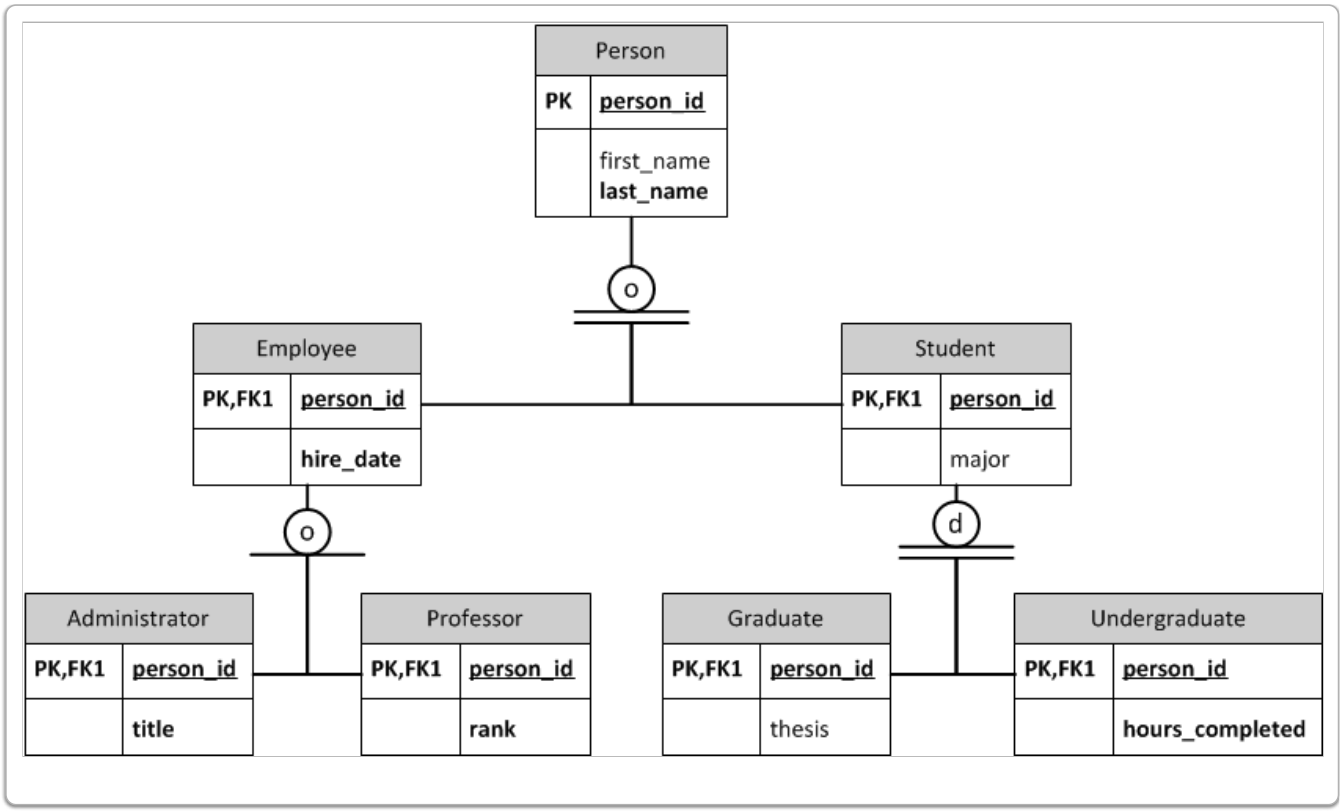


The meaning of each of the notations in this style of EERD is summarized in the following table:

Diagram text	Meaning
Supertype	EMPLOYEE is the abstract entity type upon which the PILOT, MECHANIC and ACCOUNTANT subtypes are based. EMPLOYEE abstracts the common characteristics of these three (and presumably the many additional) types of airline employees.
Shared attributes	These are the fields abstracted from the subtypes into the EMPLOYEE supertype. All subtypes inherit (have) these fields, even if they are not repeated in the subtypes.
Inherited relationship	The common relationship between EMPLOYEEs and their DEPENDENTs is abstracted to the EMPLOYEE supertype and inherited by its subtypes.
Category Symbol	This circle indicates that the relationship is between entity types rather than between entities, as in a foreign key.
Partial/Complete constraint	This symbol is either one (partial) or two (complete) horizontal bars beneath the category symbol. In this example the single bar indicates that not all airline employees are pilots, mechanics, or accountants.
Disjoint/overlapping constraint	The letter "o" (overlapping) indicates that an entity can be a member of more than one subtype , while the letter "d" (disjoint) indicates that an entity can be a member of only one subtype. In this example "d" indicates that an employee must be either a pilot, mechanic or accountant, and can not be a member of more than one of these employee subtypes.
Subtype discriminator	A single-attribute subtype discriminator is used, in disjoint specialization-generalization relationships only, to identify which subtype participates in the relationship. If the specialization-generalization relationship is overlapping, we use one attribute per subtype, where each attribute is a flag indicating whether or not its associated subtype participates in the relationship.
Unique attributes	These are fields that are in the subtypes that are not abstracted to the supertype because they are not common to all subtypes.

Though it is not illustrated above subtypes may have foreign key relationships with other tables. As you can surmise from the example, the PILOT unique attributes PIL_LICENSE, PIL_RATINGS, and PIL_MEDICAL are probably foreign keys to other tables that are not subtypes of EMPLOYEE.

Let's take a look at an example figure and walk through the different symbols and what they mean.



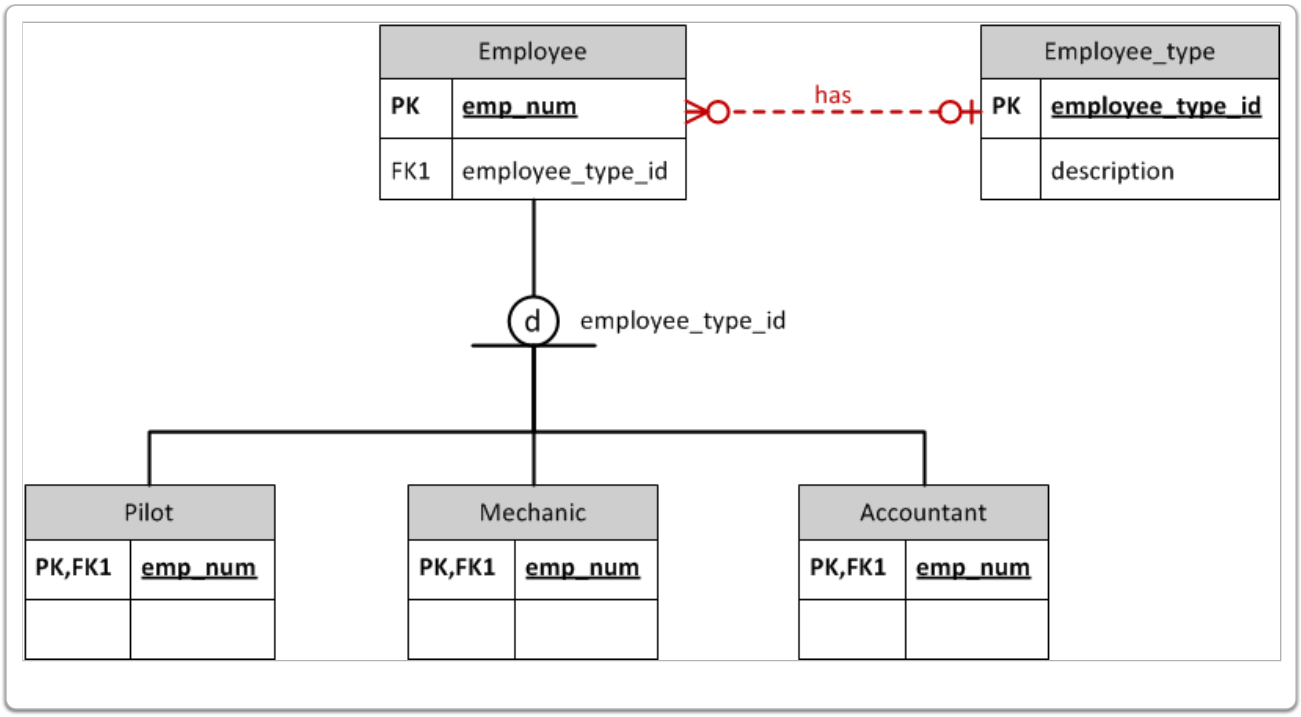
- The type-subtype relationship on the lower left indicates that administrators and professors are employees, that an employee can be both an administrator and a professor, and that not all employees are either administrators or professors.
- The type-subtype relationship at the lower right indicates that both graduate students and undergraduates are students, that a student cannot be both a graduate student and an undergraduate, and that all students are either graduate students or undergraduates.

Advanced Topic

You may have noticed the diagram indicates all Persons are either Employees or Students, because the specialization-generalization relationship between Person, Employee, and Student is totally complete. In the larger world, this is clearly counterfactual, because there are many other kinds of people. However, because databases often cover only a small subset of real world data and relationships, we should expect data models to sometimes be more restrictive than common reality. The example schema indicates that the database only stores information on Employees and Students, and no other kinds of Persons, which may well be legitimate depending upon the business rules the database supports. To avoid confusion, to the extent possible, it is important to select terms that match their commonly understood definitions when naming entities. Because there is likely no commonly understood English word that specifically means "student or employee", "Person" may be the most suitable name in the example schema.

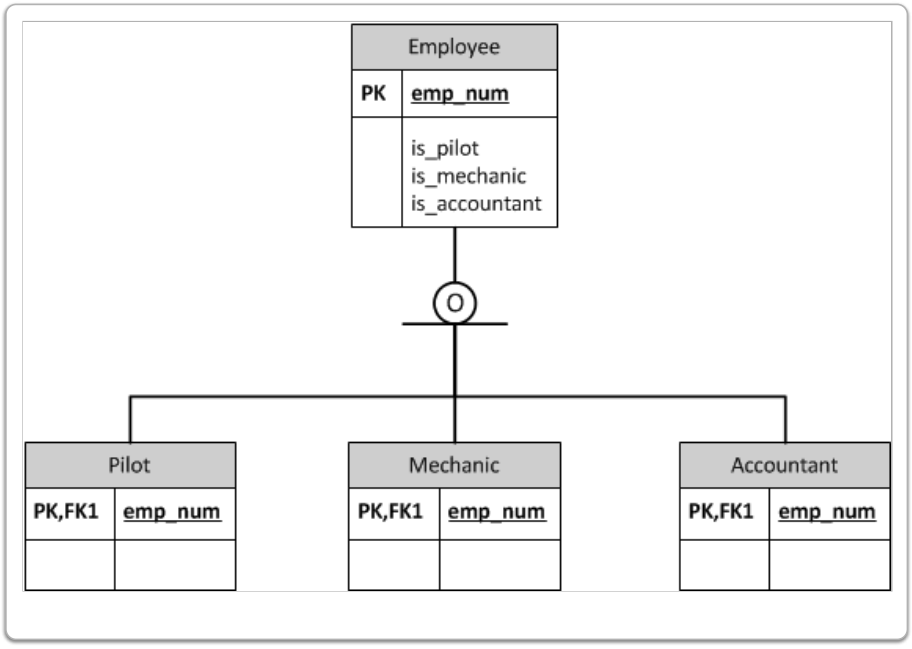
Indicating Subtype Participation

As described above, there are two different structures that identify subtype participation in a specialization-generalization relationship, and the disjoint/overlapping constraint determines which structure is used. If the relationship is disjoint, meaning that only one subtype instance can participate in the relationship for any particular supertype instance, a single-attribute subtype discriminator is used. Though the textbook illustrates use of a character-based attribute that takes on single character values such as "P" or "M", production-strength database designs often constrain the legal values of the subtype discriminator through use of a domain entity. If we use a domain entity with the Employee schema illustrated above, it would look as follows:



Notice that in the diagram, which has been simplified by removing attributes that don't affect the specialization-generalization relationship, a foreign key named "employee_type_id" references the Employee_type domain entity, and replaces the character-based attribute named "emp_type". In this way, all of the legal indicators for the subtypes are placed in the Employee_type entity.

Overlapping specialization-generalization relationships use one attribute per subtype, where each attribute is a flag indicating whether or not its associated subtype participates in the relationship. If we change the relationship so that an Employee may be of multiple types simultaneously (say, a Pilot and a Mechanic), the diagram would look as follows:



Notice that in place of a single attributed subtype discrimintator, there is one flag per subtype. A common convention for flags in a schema is that each flag is character based, and takes on value "T" for true, and "F" for false. In this example, if an Employee was both a Pilot and a Mechanic, then the supertype instance would have the following values:

is_pilot = T
is_mechanic = T

is_accountant = F

The use of flags allows for any possible combination of subtypes.

Test Yourself 3.2

The extended entity-relationship diagram for a taxi company shows an EMPLOYEE supertype with a partial completeness constraint to overlapping subtypes DRIVER and DISPATCHER. Which of the following are true regarding this diagram? (Please check all of the following that are true.)

EMPLOYEE has an attribute EMP_LNAME. This attribute is inherited by DRIVER.

This is true. It is also inherited by DISPATCHER. (Subtypes inherit or share all supertype attributes.)

DRIVER has an attribute DRIVER_LICNO. This attribute is inherited by DISPATCHER.

This is false. It is not inherited by DISPATCHER. (Attributes in the subtypes are not common to all subtypes.)

The diagram indicates that drivers may also be dispatchers.

This is true. The subtypes are overlapping. This indicates that an employee may be a driver, a dispatcher, or both.

The diagram indicates that an employee may be neither a driver nor a dispatcher.

This is true. The diagram shows a partial completeness constraint. (Employees do not have to be any of the subtypes shown in the diagram.)

It would be easy to represent all of the relationships shown in this diagram with an entity-relationship model.

This is false. The extended entity-relationship model (EERM) adds the ability to represent generalization and specialization relationships to the entity-relationship model.

Representing ERMs with UML Class Diagrams

The Unified Modeling Language (UML) is perhaps the most commonly used language for diagramming models of both domains and object-oriented applications. Although UML supports many kinds of diagrams, in this lecture we are most interested in the class diagram. A class diagram indicates the structure of data, behavior, and relationships of objects in a particular object-oriented model, similar to the way entity-relationship diagrams (ERDs) allow us to indicate the structure of data and relationships of entity instances in a particular entity-relationship model (ERM).

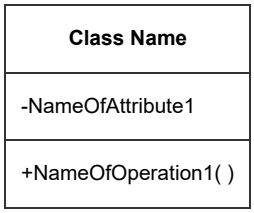
This lecture teaches an alternative but more universally understandable method of diagramming the ERM by using modified UML class diagrams. Those who are responsible for designing and implementing applications—application developers and technical analysts—are typically fully versed in both the object-oriented (OO) model and its representation in a UML class diagram, yet the same cannot be claimed for the ERM or for the ERD. Because the design and implementation of a database is usually driven by the design and implementation of an application, and because application developers need to interface their application with the database, it is often useful to represent an entity-relationship model with a UML class diagram to avoid the learning curve and communication barrier associated with the less understood ERM and ERD. This lecture shows you how to do exactly that.

Understanding Traditional UML Class Diagrams

A traditional UML class diagram represents the structure of a design in the OO model. OO designs deal in great measure with *classes* and *objects.* Similar to an entity instance in the ERM, an *object* represents a single real-world element such as a person, account, or event, or an abstract element. A *class* is an abstraction of common characteristics of a set of objects, akin to an entity in the ERM. Each class defines a blueprint for the instances of the class. For example, a Personclass represents a set of individual Personobjects; by looking at the Person class, one may see which data shall be stored and which operations exist in each Person object. An object is an *instance* of a class.

Class Representation

A class is represented in a UML class diagram as illustrated in the following figure.

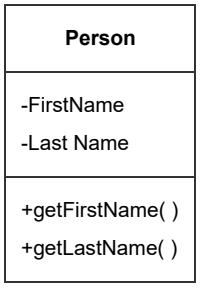


The top rectangle in the diagram provides space to name the class. Naming classes in the object-oriented model is not very different from naming entities in the entity-relationship model. The name is simply a unique identifier for the class. UML class names can optionally be qualified with namespaces, so that two classes can have the same name as long as they are in different namespaces. Strictly speaking, a class's namespace combined with its name uniquely identifies the class. However, because namespaces are not defined in the entity-relationship model, we will not be making use of this optional feature to represent ERDs in UML class diagrams.

The second rectangle provides space for each of the class's attributes. We already know that attributes describe entities in the entity-relationship model, such as a *FirstName* attribute for a *Person* entity. Attributes do the same for classes in the object-oriented model. You may have noticed a dash (-) in front of the attribute name. This is an indicator of *visibility*, which determines which object types may access the attribute. The dash (-) indicates *private* visibility, which means that only objects of that same class type may access the attribute. Visibility is a concept not present in the entity-relationship model; therefore, we will not be using this feature to represent ERDs in a UML class diagram.

The third rectangle provides space for each operation (method) in the class. An operation describes behavior for all objects of that class. Behavior can be as simple as indicating a value, or as complex as performing detailed mathematical calculations. The plus sign (+) next to the left of the operation indicates *public* visibility, which means that objects of any class type may access the operation. The parentheses to the right of the operation is part of the operation's *signature* and describes the number and type of parameters used by the operation. Because behavior is not supported in the entity-relationship model, we will not be using operations to represent ERDs in a UML class diagram.

Let's look at a simple example diagram.



This diagram represents a class named "Person" with two attributes—FirstName and LastName—and with two operations—getFirstName and getLastName—which presumably provide the behavior of indicating the FirstName and LastName values, respectively.

Associative Relationship Representation

Classes in the object-oriented model can be associated, as illustrated below:

From an initial glance, you may recognize that there are two classes in the diagram named “Class1” and “Class2”. Notice the line connecting these classes; this line indicates the associative relationship. The “1..1” on the left on the “0..*” on the right are termed *multiplicities*, and each multiplicity indicates the minimum and maximum number of participants in a relationship, respectively, for the class it is closest to. The “1..1” indicates that exactly one instance (at least one and at most one) of Class1 participates in a relationship. The “0..*” indicates that no instances (0), or one or many instances (*) of Class2 may participate in a relationship.

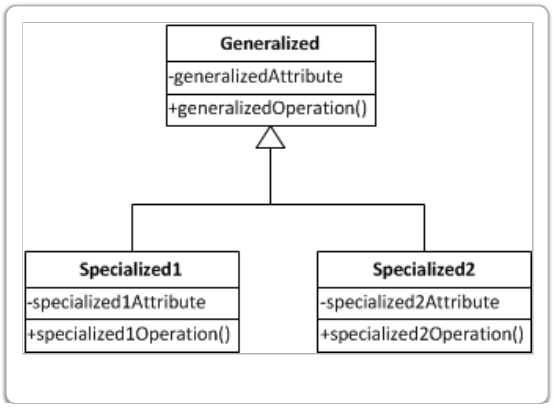
The words “Has” and “Is Had By” are termed “roles”, and each role indicates the role played in a relationship by the class it is closest to. “Has” indicates that the role Class1 plays in the relationship is that it “has” Class2, and “Is Had By” indicates that the role Class2 plays in the relationship is that it “is had by” Class1. Roles are conceptual, and can be any words that help explain the relationship to the viewer. For example, if Class1 is named “Student”, and Class2 is named “Book”, then we could read the relationship as “A Student may have many Books; a Book is had by one student. “

Consider the following diagrammatic example:

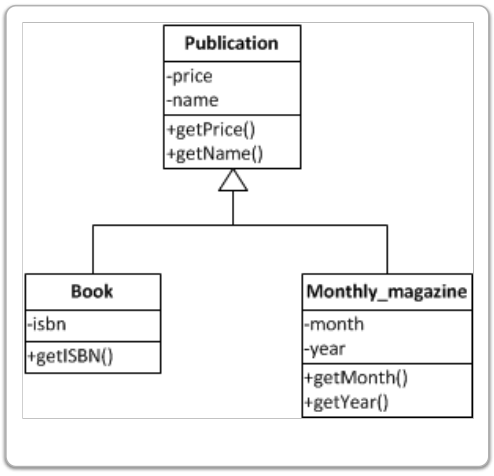
Here we recognize two entities—Person and Item—associated to each other. The multiplicities indicate that exactly one Person participates in each relationship, and that one or more Items participate in a relationship. Further, the roles clarify this association as an “owns” relationship, so that a Person owns one or more Items, and each Item is owned by exactly one Person.

Specialization-Generalization Relationship Representation

Class in the object-oriented model may specialize or generalize other classes, as illustrated below:



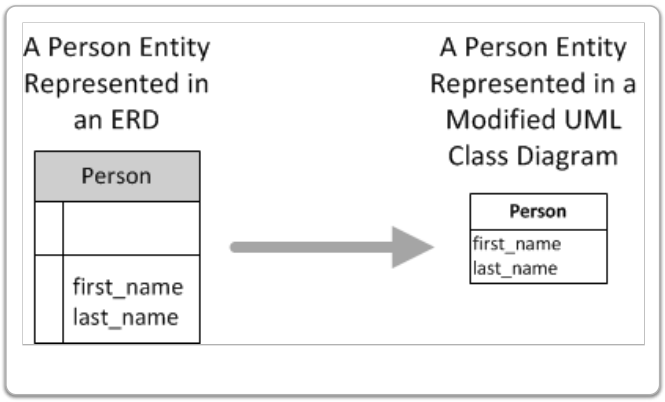
Notice the triangle attached to the Generalized class, and the lines attached to the Specialized classes, indicating the specialization-generalization relationship. Generalized contains attributes and operations that apply to all of the Specialized classes, while each Specialized class defines additional attributes and operations. Let's look at an example:



The diagram indicates that Publication is a generalized form of Book and Monthly_Magazine, and that both Book and Monthly_magazine are specialized forms of Publication. All Books and Magazines inherit the price and name atttributes, as well as the getPrice and getName operations. Book defines an attribute that only applies to books—isbn—and a corresponding operation. Monthly_magazine defines year and month attributes, and corresponding operations, which apply specifically to monthly magazines.

Representing Entities

A UML class diagram may be modified to represent an ERM. Let's start with the simplest aspects. We represent an entity as a class, and entity attributes as class attributes. Because there is no behavior or visibility in an ERM, we simply omit the behavior rectangle and the visibility indicators. Take a look at the example below, which illustrates a Person entity in an ERD, and its equivalent representation in a modified UML class diagram.



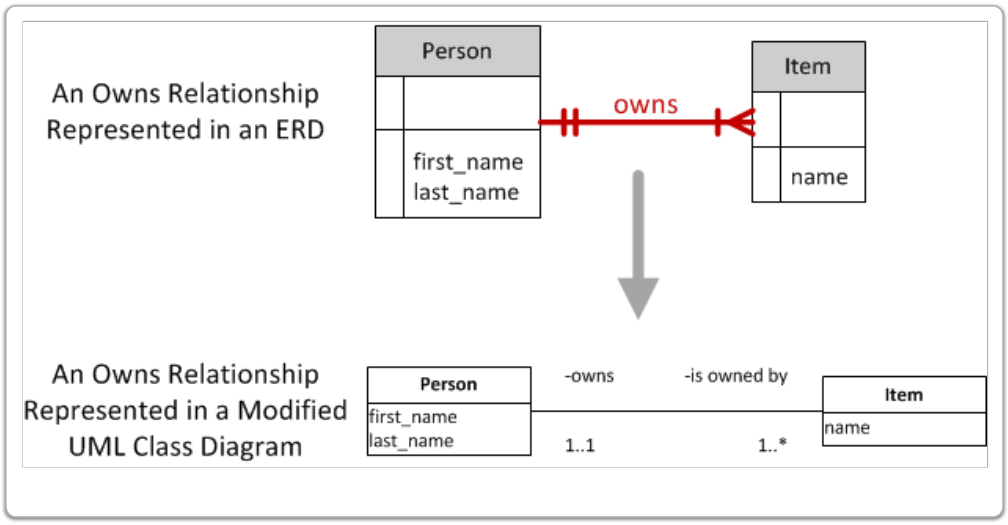
Representing Associative Relationships

To represent an associative relationship in an ERM, we use an associative relationship in the UML class diagram. In place of the Crow's foot connectivity indicators that indicate the optionality and plurality of each side of the relationship, we use multiplicities, as indicated in the following table:

Crow's Foot Symbol	Multiplicity	Meaning
	0	Optional relationship
	1	If lowerbound, mandatory relationship If upperbound, at most one
	*	One or many

Although multiplicities in a UML class diagram can support any number, such as 2 or 7, we restrict ourselves to 0, 1, and *, because these are all that is necessary to specify optionality and plurality.

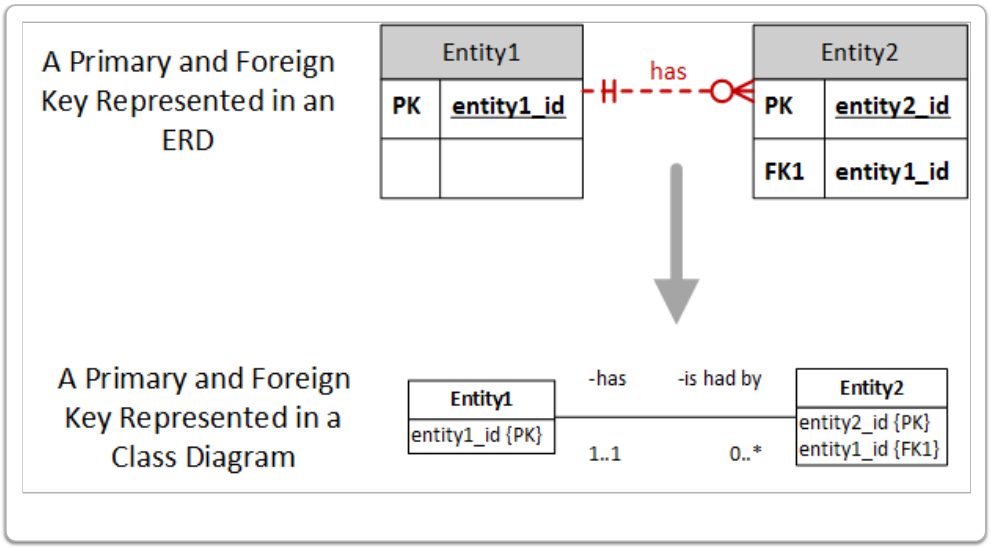
Take a look at the example below, which illustrates a relationship in an ERD, and its representation in a modified UML class diagram.



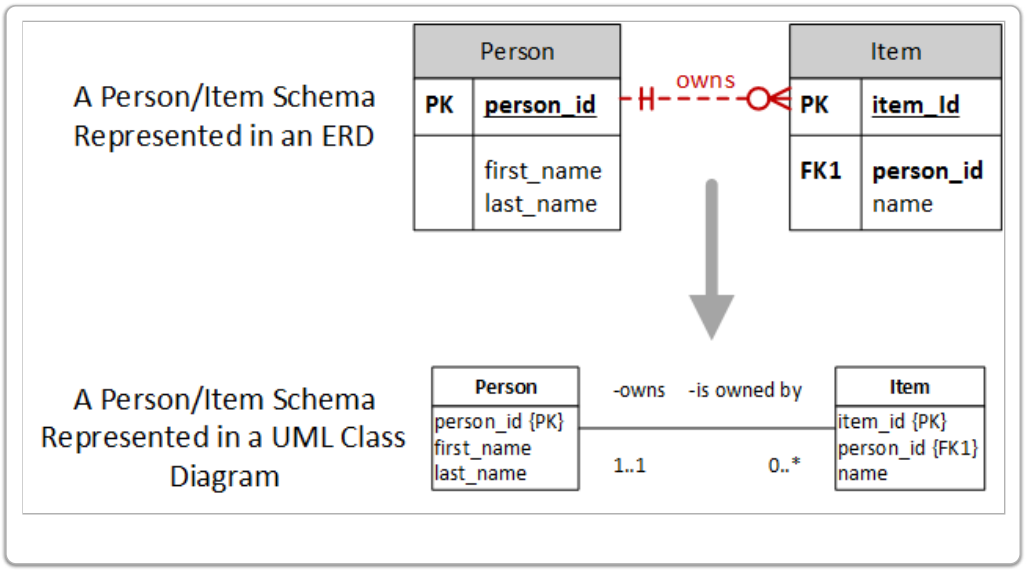
Notice that the active verb "owns" is indicated by the "owns/is owned by" roles. Further notice that the Crow's Foot symbol is represented with a multiplicity of "1", and the Crow's Foot symbol is represented with a multiplicity of "*".

Representing Primary and Foreign Key Constraints

Primary and foreign key constraints are represented as SQL extensions to the UML Class diagram. The acronym "PK" represents a primary key constraint, and the acronym "FK", followed by the foreign key number, indicates a foreign key constraint. The acronyms are placed between brackets "{" and "}" to the right of the attribute they apply to. The following figure illustrates these extensions.



Notice that entity1_id and entity2_id are the primary keys of their respective entities, and that Entity2 references Entity1 via an entity1_id foreign key. Let us review our Person/Item schema with primary and foreign key constraints added.



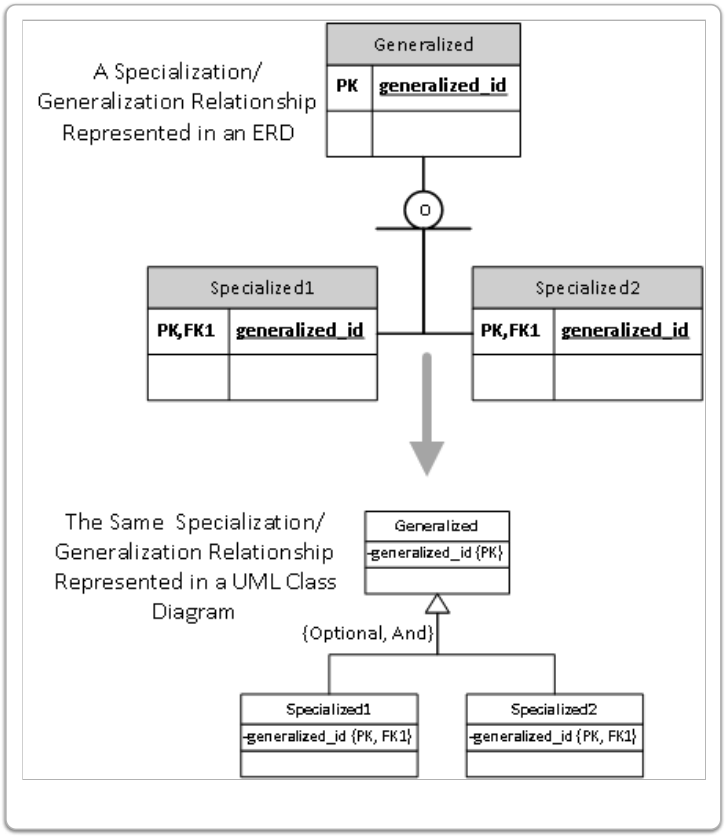
Representing Specialization/Generalization Relationships

Because the specialization/generalization relationship is natively supported in UML Class diagrams, we simply augment the native construct with a completeness and disjoint/overlapping constraint indicator. The indicator, which is located next to the relationship line, consists of an open-bracket ({), a word indicating completeness, a comma, a word indicating disjointness, and a close-bracket (}). The allowable words are indicated in the following table:

Keyword	Meaning
Optional	The relationship is partially complete
Mandatory	The relationship is totally complete
And	The relationship is overlapping

Or	The relationship is disjoint

The following figure illustrates representation of a specialization/generalization relationship:



The “{Optional, And}” annotation indicates that the relationship is both partially complete and overlapping. You may also notice that if an attribute has multiple constraints, such as an attribute that is both a primary key and a foreign key in the example above, the indicators in the class diagram are comma separated.

Entity Clusters and Subschemas

Entity clustering is a technique used to hide potentially confusing detail in an ERD. Entity clustering operates by encapsulating detail within an ERD. The entity clusters in an ERD may be the abstract entities in an EERD which represents the cluster in detail. In this way the additional power of EER modeling can be incorporated into conventional ER models without changing the overall structure of those models. Entity clusters may also represent logically related subsets of the entities, such as the OFFERING and LOCATION entity clusters illustrated in the text. Entity clusters which I have used include:

- PRODUCT, which encapsulates the product table itself, and also the vendors for the products and the inventory history and other necessary detail.
- CUSTOMER, which encapsulates as many as fifty tables in a large marketing database.
- EMPLOYEE, which encapsulates much of the Human Relations database.

There is a complementary technique for simplifying ERDs so that they are easier to understand. This technique involves dividing a complex schema into simpler logically related subschemas. The subschema approach is somewhat more flexible, because the subschemas can represent business areas, functional areas, and other parts of the schema that cannot easily be encapsulated in entities as entity clusters. With this flexibility also comes some challenges, including the fact that most complex schemas can be partitioned many different ways into subschemas. Subschemas generally aren't as useful at encapsulating the complexity as are entity clusters, though they can be helpful in understanding the

overall structure of complex schemas. One technique that I have found useful is to organize the entities on a complex schema into areas by subschema, and to color the different entities according to the subschema to which they are assigned.

Test Yourself 3.3

Which of the following is correct? (Please check all of the following that are correct.)

An entity cluster is the logical and physical combination of multiple tables into one in order to save data storage space.

This is false. Entity clustering does not change the actual design of the database. It is a way to make an entity-relationship diagram less confusing to read by leaving out some detail. Entity clustering simplifies entity-relationship diagrams by representing a cluster of many closely related entities in the actual database by one entity on the diagram.

There may be more than one “correct” way to divide a complex schema into subschemas.

This is true. Most complex schemas can be divided many different ways into subschemas.

A database design may contain hundreds of entities.

This is true. Database designs for large complex organizations may contain hundreds or even a few thousand different entity types.

Designing Primary Keys

We all know by now what primary keys are used for. We now discuss the design of optimal primary keys. Designing good primary keys is surprisingly important and surprisingly difficult. The following table lists the important characteristics of primary keys. Note that this list is not the same as the one in the text.

Desirable PK feature	**Why it is important**
Manifestness	You can't store an entity in a table without knowing the primary key, so you must be sure that the primary key will always be present or immediately derivable from the data that is first available when the entity presents itself for storage in the database. Many natural keys do not satisfy this requirement.
Uniqueness	Your primary keys must be guaranteed to be unique for all entities (instances) forever. This can be a problem when databases move to larger domains where keys may be duplicated.
Immutability	The primary key of an entity may not change. Sometimes the only available natural keys, such as names, may change, only seldom. If you choose such a mutable key as the primary key be sure that you are prepared to deal with unexpected key changes and the need to search intelligently over other attributes to identify entities.
Compactness	The primary key of one table is often repeated in other tables that are related to the first table by foreign keys. It is thus desirable that the primary key occupy as few bytes as possible. Numeric synthetic keys are the shortest. Pay particular attention to the storage format of primary keys, so that they are as small as possible.
Security	A primary key will probably be used as a foreign key in other tables, so it is more difficult to control access to the data in a primary key. Primary keys may also be exported to other databases. For these reasons the primary key should not incorporate any sensitive data and the primary key should not be usable to infer any sensitive data. Primary keys that can be used to extract sensitive data should also ideally not be easily guessable.

Test Yourself 3.4

Please check all of the following that are descriptions of a desirable primary key.

A value based on or derived from some characteristic or attribute of the entity to be stored. This characteristic or attribute may not be known at the time when the entity presents itself for storage, but it will definitely be known within three months.

This is false. If you store an entity in a table without a primary key you will have a difficult time accessing the data. Consequently a well designed primary key will contain only attributes that will always be available or derivable at the time an entity presents itself for storage.

A good design for a primary key is number that is generated sequentially with a relatively small incremental change to a minimal starting value, such as starting with the integer 1 and adding 1 for the primary key of each subsequent stored entity.

This is true. Such an integer surrogate or synthetic key has many advantages, particularly when natural keys with desirable properties can't be identified.

A good choice of a primary key for a Person table would be the person's first name.

This is false. A primary key must be unique. Multiple people could have the same first name. Other problems with using any part of a name in a Person table is that a person could change their name or go by multiple names.

A person's social security would make a good primary key for a Person table.

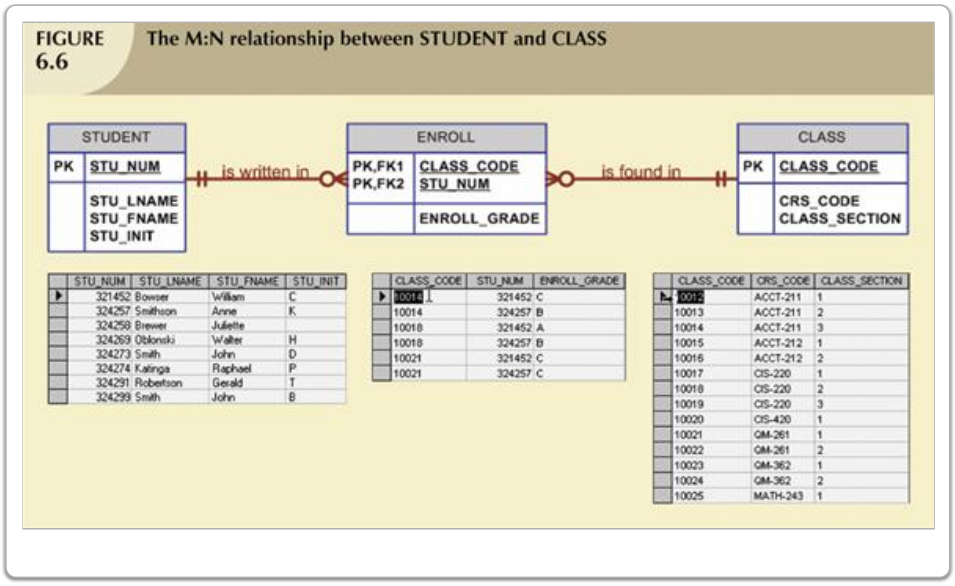
This is false. There are two problems with this design. One problem is that a primary key should not incorporate any sensitive data, and a social security number can be used in identity theft. Another problem is that not all persons have social security numbers.

A digital photograph of some unique aspect of the entity, such as the retina of a person, would make an ideal primary key.

This is false. This raises the issue of "compactness". It is desirable that the primary key occupy as few bytes as possible. This example may also raise "security" issues.

Primary Keys for Composite Entities

Consider the following ERD from the text:



The ENROLL entity mainly represents the many-to-many relationship between students and classes. Such entities are termed *association entities*, *bridge entities*, or *composite entities*. Note that the table has foreign keys to both STUDENT and CLASS, and that the primary key is the composite of those two foreign keys. The primary keys of association entities should include all of the columns in all of the foreign keys of the tables that they associate. If an association table has foreign keys to three or more entities which it associates, then all foreign keys should be included in the primary key of the association table. Rarely an additional column will need to be added to the primary key, if the entities are permitted to be associated in the same way more than once.

Note also that the ENROLL table has a non-key ENROLL_GRADE attribute that stores the grade that the student earned in this enrollment. Most natural associations also have properties of their own, and you will find that many association entities have many non-key properties.

Test Yourself 3.5

Which of the following are true? (Please check all of the following that are true.)

A primary key may consist of more than one attribute.

This is true. This is preferred in some cases.

A composite entity, which is also known as a bridge or associative entity) implements a many-to-many relationship by replacing it with multiple one-to-one relationships.

This is false. A composite entity replaces a many-to-many relationship with two or more one-to-many relationships.

The primary key of a composite entity is composed of all of the foreign key columns to the entities that it associates.

This is true. Composite entities may contain other columns as well.

Primary Keys in Existence-Dependent Relationships

If one entity depends for its existence within the database on one or more other entities, then the existence-dependent table should include the primary key of all tables upon which its existence depends. As the text indicates, these existence dependencies can be natural, such as the existence dependence of DEPENDENT on EMPLOYEE or the existence dependence of GRADED_ITEM on CLASS. Existence dependence can arise in relational database as a result of the normalization process required to correctly represent composite entities in a relational database. For

example, to represent an ordinary grocery store receipt in a relational database requires at least two entities - one to represent the receipt itself, and one to store the line items with the product identification, price, quantity and extension price. The line items cannot exist by themselves without the receipt, so they are existence-dependent on the receipt. The primary key of the table that represents the line items should contain the primary key of the table or entity cluster that represents the receipt itself.

Test Yourself 3.6

Entity B is existence-dependent on entity A (A can exist without B, but B cannot exist without A). The relationship between A and B is one-to-many (One A can be related to many Bs). Which of the following are true regarding this situation? (Please check all of the following that are true.)

The primary key of B should contain the primary key of A.

This is true. The primary key of an existence dependent entity should contain the primary key of the entity upon which its existence depends.

The primary key of B should contain just one attribute, which is the primary key of A.

This is false. The primary key of B should be a composite primary key which includes the primary key attributes of A, and at least one other attribute, so that there can be many in B for each one in A.

Surrogate Primary Keys

It can often be very difficult or impossible to identify correct primary keys for natural entities, particularly natural events. In these situations the only solution is to have the computer or user create a unique primary key for each entity that is inserted into the table that represents such an entity. These keys are called *synthetic primary keys* or *surrogate keys*.

It is famously difficult to identify correct natural keys for people, and it is not desirable for security reasons. You have probably noticed that you have a BUID which is not related to your name, social security number, or anything that can be used to determine anything other than that you have some relationship with BU. This is intentional, because not only is it essentially impossible to produce a natural key for people that is always present and guaranteed to be unique, but also because BU protects your security by not encoding any sensitive information in your BUID. Some states have used people's Social Security numbers as primary keys in their driver's license databases, and have printed people's social security numbers on their driver's licenses. If you have such a drivers license or other identification or accounts where your social security number is the key you should ask the providers of those identification or accounts to provide you with a synthetic primary key. This will help protect you from identity theft.

Integer surrogate keys are the norm in large and high performance databases. This is because integers are the most compact representation of an identifier that is unique for a number of unique entities, and because it is usually most efficient for computers to store and operate on integers.

If you have a table with many rows that references other tables, then the size of the large table is affected by the size of the primary keys in the other tables. In these situations the primary tradeoff influencing the primary key design is the size of the tables that reference the table. Unless the natural key is truly small then it is highly desirable in these situations to provide a synthetic primary key, even if the natural key is present. If it is important to enforce entity integrity in the database then a unique index should be created on the columns of the natural key(s).

Test Yourself 3.7

Which of the following are true regarding primary keys? (Please check all of the following that are true.)

A number that is generated sequentially with a relatively small incremental change to a minimal starting value, such as starting with the integer 1 and adding 1 for the primary key of each stored

entity is an example of a surrogate primary key.

This is true. This is also referred to as a synthetic primary key.

The size of a primary key of one table can affect the size of a different table.

This is true. The primary key of one table may be stored as a foreign key in another table. For this reason primary keys of tables that are referenced from tables with many rows should be as small as possible.

If a natural key can be identified, it is almost always a better choice for a primary key than a synthetic primary key.

This is false. There are frequently problems with a natural primary key such as its size in bytes ("compactness"), changeability, or security issues.

Tables That Should Not Have Primary Keys

The text assumes that all tables should have primary keys, but this is not always true. Tables that represent real-world entities or parts of entities should always have primary keys. Most operational databases in financial or other sensitive applications include tables whose sole role is to preserve a record of events or changes to the database. These history or audit tables often record a variety of internal events within the system which may have no corresponding durable entity in the real world, and no natural key. Clients of mine have tried in vain to develop a natural primary key for these tables, including as many as a dozen columns, often with a timestamp to help assure uniqueness. After such an application has run for a while they have discovered to their dismay that they occasionally have primary key uniqueness violations that prevent the insertion of a history record.

The problem in these situations is not that they have selected incorrect columns for the natural primary key, but that such tables usually have no reliable natural primary key, even including a timestamp. The solution is to not have a primary key. Not having to maintain the unique index for the primary key speeds inserts into history tables. You may want to index the tables so that you can efficiently retrieve the historic data. If you do need a primary key, for example if a history table must be referenced from another table, then use a synthetic (surrogate) primary key.

Test Yourself 3.8

Which of the following are true regarding primary keys? (Please check all of the following that are true.)

A table that represents a real-world entity should always have a primary key.

This is true. History or audit tables may not have a primary key.

Adding a timestamp column guarantees uniqueness of a natural primary key.

This is false. Even with a timestamp, a uniqueness violation with a natural primary key is possible. For example, two processors on a multiprocessor system can store the same timestamp.

There are potential disadvantages to having a table such as a history table without a primary key.

This is false. Maintaining the unique index for the primary key slows table inserts, which is the dominant operation for a history table.

Flexible Database Design

Databases are the most long-lived of all software components. Many databases have been continuously used and updated for decades. Most of the life cycle cost of databases is thus in the maintenance phase of the database life cycle. Thus the most important characteristic of a database design is that it be easy to modify as business needs change. There is an old saying in the computer industry that an easily maintained design that doesn't happen to be completely correct is not a problem, because you can just fix it, but that an un-maintainable design is a disaster, because even if it works now something will inevitably happen that will require you to change it, and then you have a big problem. In this section we describe the things that you can do to make sure that your designs are flexible enough so that they can be easily maintained.

There are two more advanced topics in the design of foreign keys for 1:1 relationships. One consideration is that modern DBMS including Oracle support clustering of tables which have 1:1 mandatory relationships and a common key that identifies the rows that are related 1:1. What clustering of tables does is combine the corresponding logical rows of the clustered tables into one physical row in storage. Because the clustered rows are actually one physical row there is no need to repeat the shared columns in the cluster key. As a result when tables are clustered the result is a smaller database. I often cluster tables that are related by a 1:1 mandatory relationship. With clustering the columns in the common cluster key are stored only once, so the database is smaller. Clustering stores the related rows together in one physical row, so joining the tables is essentially free, and you can effectively ignore the performance consequences of joining both tables in requests. Clustering reduces the table size, because there is only primary key.

Test Yourself 3.9

Which of the following are true? (Please check all of the following that are true.)

The most important characteristic of database design is to meet the current business needs of the end user as closely as possible in the shortest amount of time and for the minimum amount of money. It is acceptable to sacrifice the ability to modify the database in the future in order to accomplish this.

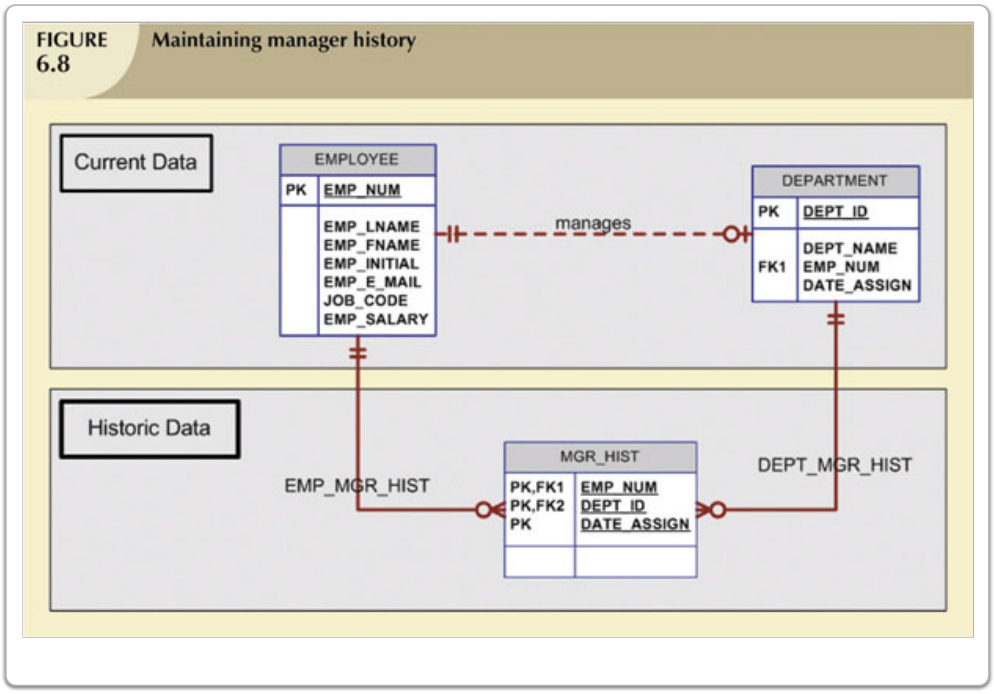
This is false. The most important attribute of a good database design is ease of modification as business needs change.

Table A and Table B have 6 columns each. Table A and Table B have a mandatory one-to-one relationship and a common key that identifies the rows that are related one-to-one. When Table A and Table B are stored together as a clustered table, the resulting clustered table has 12 columns.

This is false. The clustered table would have 11 columns. The shared key column is not repeated.

Representing History

Figure 6.8 from the text and the accompanying explanation summarize the basic concepts involved in representing history or audit data in a relational database.



The representation above handles the case where the manager for a department is an independent change. Often changes are necessarily coupled for business reasons, or coupled by the fact that several changes are made as part of one larger change, and that coupling needs to be recorded in the history. For example, consider a realistic DEPARTMENT table that might have a dozen or more attributes. Sometimes several changes are made at once - such as the address of the department and its mail code. In these situations the best approach is to maintain a history table for the DEPARTMENT table rather than for just the *manages* relation. Such a DEPARTMENT_HIST table would normally have all of the columns of the DEPARTMENT table for which it stores history, plus at least one additional sequence or timestamp column that distinguishes the history entries and provides the additional information required to determine which changes apply in particular situations.

One nice feature of designs with a current data table and a corresponding history table is that the same queries can be run against the current status table (e.g., DEPARTMENT) or against the corresponding history table (e.g., DEPARTMENT_HIST). Queries can be run against the history table to return results corresponding to the state at any previous time. For example, we can run a report today as if it were being run at the end of the previous quarter, reflecting the state of the DEPARTMENT table or any number of additional tables at that time. This is very useful for many kinds of businesses.

Queries run against the history table will have at least one additional WHERE clause, and often a subquery. The history table is typically much larger than the current data table. For these reasons queries against the history table are not as fast as queries against the current data table. This is why it is convenient to have both tables. The current data table supports current operational transactions, while the corresponding history table supports historic analysis and reporting. Performance is not so important for these historic functions, so the extra size and overhead of the history table is acceptable. Note that the current data table redundantly stores the latest data in the history table, so care must be taken to assure that these are always consistent. I usually encapsulate updates to the pair of tables in a stored procedure, and write a stored procedure or script that checks that they are consistent. Triggers can also be used to maintain the denormalized data. Triggers can be used to add history to an existing database and applications without requiring changes to existing SQL.

For a more advanced discussion of the design of history and audit, see Designing Database History and Audit and Designing History and Audit Tables later in the lecture.

Test Yourself 3.10

Which of the following are true regarding history tables? (Please check all of the following that are true.)

A history table can be used to determine the value of an attribute of an entity was at some point in the past.

This is true. This is a typical use of a history table.

A history table is typically the same size as a current data table.

This is false. A history table is typically much larger.

When a history table is present, an end user must manually issue a separate INSERT command to update the history table every time the end user changes the current data table.

This is false. There are different mechanisms that can be used to maintain the history, including triggers, so this is not required.

Designing Database History and Audit

We often need to maintain a record of transactions, for some time after the transactions have been completed, to support a review of the transactions or for internal or external audit. The requirements for this history data are quite different from those of the operational database, and consequently the designs for history and audit tables are correspondingly quite different. The differences in the requirements are summarized in the following table.

Operational Database Requirement	History Database Requirement
The operational database is required to enforce integrity constraints, so it is typically highly normalized, with many foreign key and other constraints and many relationships between entities.	The history table records must be interpretable many years after they are written, in spite of changes to related data and changes to the operational schema. History tables must therefore be more standalone than operational tables.
Operational data represent the state of a transaction at the current time, subject to the current representation and constraints.	History tables must represent events over long time periods, even though the representations and constraints vary over time.
The current data implementation must be fast, to support immediate business operations.	History data is typically needed mainly for offline analysis or periodic audit, so the only real-time performance constraint is that the history or audit records must be efficiently inserted as part of the operational transaction.
Current data are frequently updated.	The only operations frequently performed on history and audit data are inserts. There may be business rules or regulations forbidding the update or deletion of audit data.
Current data are kept as small as feasible, to improve performance and shorten the database recovery time. Current data are kept on fast storage and may be replicated to shorten recovery time.	History data are infrequently accessed, so they can be kept on less expensive storage. They seldom or never change, and are not required for routine operations, so the backup, recovery and archive requirements are different.

When history is kept in separate tables those history tables are often quite denormalized, so that each record in the history table represents the entire event or transaction that it is being recorded for later analysis or audit. The following table summarizes common denormalizations in history tables:

Denormalization	Advantages
Replacing foreign key references in the operational tables with the corresponding	This improves the ability to search history based on related data and eliminates a serious problem when old history data has foreign key references to operational tables and the

referenced data, eliminating all foreign key references from history tables to operational tables.	operational data or schema needs to be changed. Eliminating references to any operational data also improves the resistance of audit tables to tampering.
Augmenting synthetic keys in operational tables with natural keys in the corresponding history tables. The synthetic keys in operational tables are often retained as part of the audit trail.	This permits searching later by the natural or operational synthetic key. It also makes it more difficult to tamper with the audit records by changing the corresponding operational data or synthetic keys.
Eliminating primary keys on history tables.	History table records represent events that are often internal to the system rather than in the problem domain. Therefore, it can be very difficult to design natural keys or to take synthetic keys from the operational tables. We don't want an audit record insertion to fail because of a primary key constraint violation.

Test Yourself 3.11

Which of the following are true regarding history tables? (Please check all of the following that are true.)

The speed of data retrieval from history tables is typically a critical performance measure for a database.

This is false. Typically, the main performance concern for a history table would be efficient record insertion as part of operational transactions. History tables are rarely accessed, compared to the current data tables.

Updating and deleting records in a history table occurs frequently.

This is false. Changes to history data should be uncommon occurrences and may not be permitted at all.

Backup and recovery requirements may be different for history data as opposed to current data.

This is true. Typically, history data is infrequently accessed and not required for routine operations, so the operational consequences of having the history data unavailable for a time while it is being restored are much less than the consequences of having operational data unavailable.

The attributes of a single history table may be spread across two or more current data tables.

This is true. It is common for history tables to be denormalized with foreign key references removed. Foreign key references to current tables should not even be allowed, because the current data can change, invalidating the history.

A history table must have the same primary key as any table holding the current version of the data contained in the history table.

This is false. A history table may have a different primary key or no primary key.

Designing History and Audit Tables

When designing history and audit tables I imagine that I am an outside auditor examining an audit record seven years from now after many changes to the business rules and operational schema. The auditor should be able to easily tell what event is being recorded and all of the relevant details without depending on changeable aspects of the business or operational database.

History and audit tables may record more information than is required for operations. For example, let's look at what happens at a financial services firm, such as a bank, when any change is made to a customer's address. While the operational tables only store the new address, the audit tables will record who made the change, when they made it, from where they made it, and references to any paper documents, voice recordings or other audit data that may be outside the database.

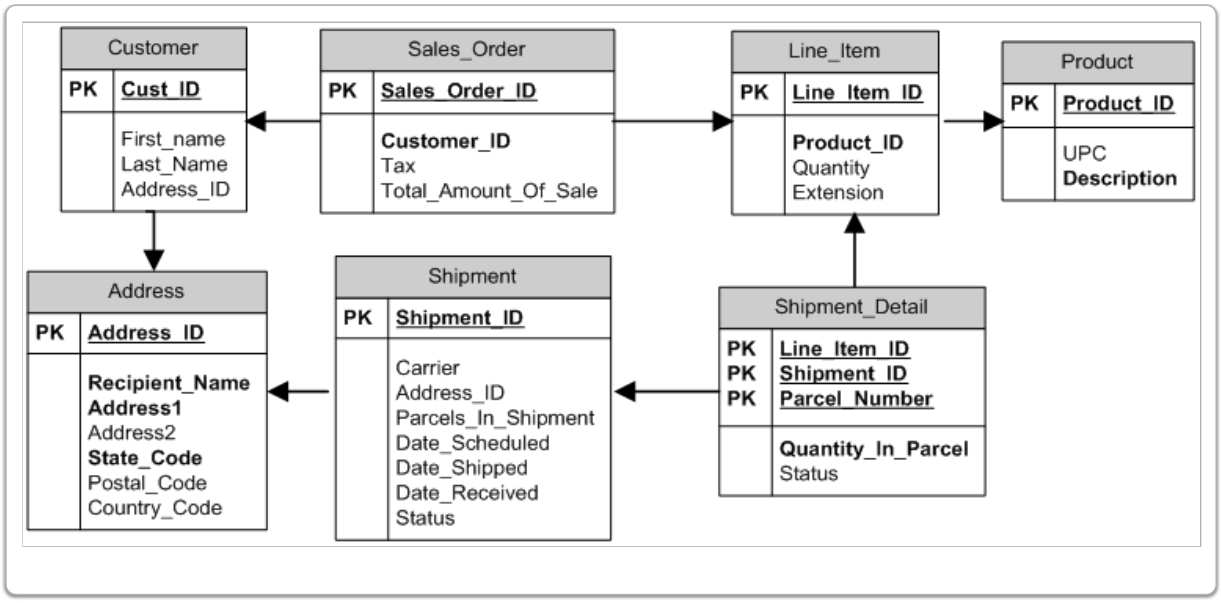
Historic data is also stored in data warehouses and other decision support systems. Modern data warehouses store fact data at the level of atomic business transactions, so it is sometimes feasible to use a data warehouse as the longer-term history and audit repository, but this can be problematic. The following table summarizes differences between audit database requirements and data warehouse requirements, as well as the problems created when data warehouses are used as history and audit repositories.

Data Warehouse Characteristic	History and Audit Repository Characteristic	Problems this can cause when Data Warehouse Data is used for Audit
Data cleansing in the (ETL) processing which is designed to assure that the data warehouse contains the most correct possible data.	Permanently records all transactions to the operational system, including transactions with bad data, error correction transactions, and administrative transactions, whether or not they are correct.	Limited ability to detect fraud, misuse of the operational system, and errors in data.
Typically updated periodically, such as in the middle of the night. The data warehouse update is not part of the operational transactions.	Audit records are usually created as an atomic part of each operational transaction.	No assurance that all transactions are captured in the data warehouse. If real-time auditing is performed on the history data, then there is a delay between the transaction and the detection of inappropriate operations.
Usually rather complex extract, transform and load (ETL) data processing, including error correction, designed to assure correctness of data to support business decisions.	Simple tamper-resistant processing designed to assure capture of all data operations, particularly any erroneous or inappropriate data operations, and to prevent concealment of any data operations, particularly those with financial data.	It is difficult to assure that warehouse data has not been modified so as to conceal errors or inappropriate behavior.

History and Audit Table Design Examples

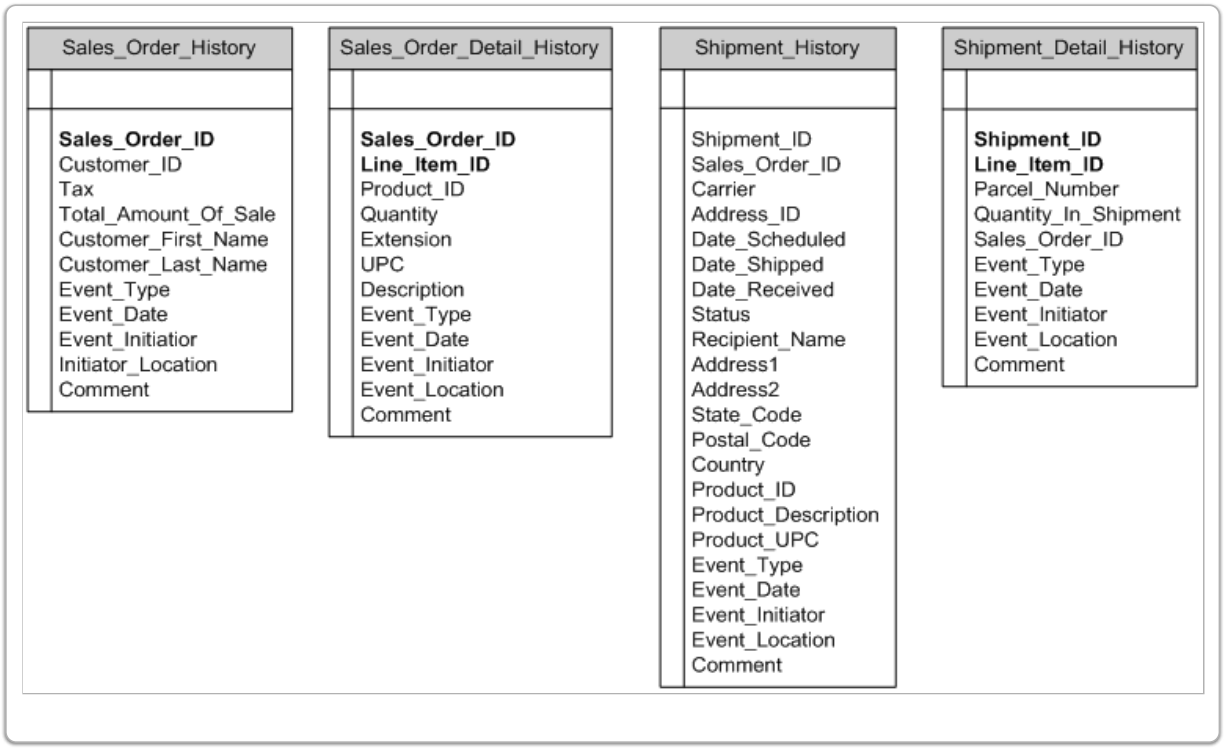
Let's consider an example of a history and audit table design for a stereotypical sales operation where goods are shipped to customers. The example below is simplified; for example, a Customer table may contain more than thirty columns. Domain tables such as the State and Country tables are not shown.

First consider the simplified transaction processing schema:



Note that all tables have primary keys and that the appropriate foreign key constraints are enforced.

When designing the history for this application we recognize that there are two different kinds of asynchronous events - those related to sales and those related to shipping. We assume that events may apply to sales orders as a whole (such as cancelling a sales order or receiving payment in full for the sales order) or to parts of a sales order (such as a customer changing their mind about the quantity of a product or a customer returning a product). We also note that shipping events may apply to a shipment as a whole (such as an entire order being shipped or received) or to parts of a shipment (such as part of a large order being received damaged). We also recognize that we will not be able to depend on the stability of customer, address, product or other changeable supporting data for the lifetime of the history, so we will need to record this data with the history records. Consider the following history schema:



Note that there are no primary keys in the history tables. This is normal. There may be any number of events that trigger history creation for a single entity in the operational database. Two identical records may even be inserted into the history, and that double occurrence may be significant. Synthetic primary keys could be generated for these tables, but it was decided that it would not be worth the extra storage and

overhead to create a synthetic primary keys for these tables, because the data would not be searched by these internal history keys, so they would have no value.

There are no foreign key constraints in the history schema, even though the corresponding tables in the operational schema contain foreign keys. The fundamental reason for this is that the Sales_Order_Detail_History and Shipment_Detail_History records do not refer to particular Sales_Order_History or Shipment_History records. For example a Sales_Order_Detail_History record refers to all of the Sales_Order_History records with the corresponding Sales_Order_ID. There may be six Sales_Order_History records with a common Sales_Order_ID, and there is no one of these records that a Sales_Order_Detail_History record should refer to. Foreign keys enforce the constraint that each referencing row must refer to exactly one referenced row, so foreign keys would not be correct here. There is a general relational database relation between the Sales_Order_ID column of the Sales_Order_History table and the Sales_Order_ID column of the Sales_Order_Detail_History table and a similar relationship between the Shipment_ID column of the Shipment_History table and the Shipment_ID column of the Shipment_Detail_History table. These many-to-many relationships are typical of history tables. If there is a business rule that there must be at least one Sales_Order_History or Shipment_History record for each corresponding detail history record (Sales_Order_History_Detail or Shipment_Detail_History) then that constraint should be enforced in the triggers or other mechanism that maintains the history.

History and audit table data is commonly created using triggers on the operational tables. For example, an update, insert, or delete trigger on the Sales_Order table can be used to maintain the Sales_Order_History table.

Triggers can also be used to prevent changes to the audit data. For example, an update or delete trigger on an audit table can log the attempt, raise alarms, and/or rollback the transaction that is attempting to change the audit data.

Test Yourself 3.12

Which of the following is correct regarding history or audit data? (Please check all of the following that are correct.)

An auditor should be able to understand a recorded event regardless of current business rules and operations.

This is true. An audit may occur seven or more years in the future, during which time the business rules and database may have changed substantially.

A history or audit table may hold attributes or types of data not contained in an organization’s operational tables.

This is true. A history table may include additional data such as an attribute identifying an employee who made a change or even a voice recording of a conversation requesting a change.

Storing data to support audits in a data warehouse can cause difficulties.

This is true. One of the problems associated with storing audit data in a data warehouse is that data warehouse data is cleansed before loading, so that it may not store the original transaction data, but only the corrected data.

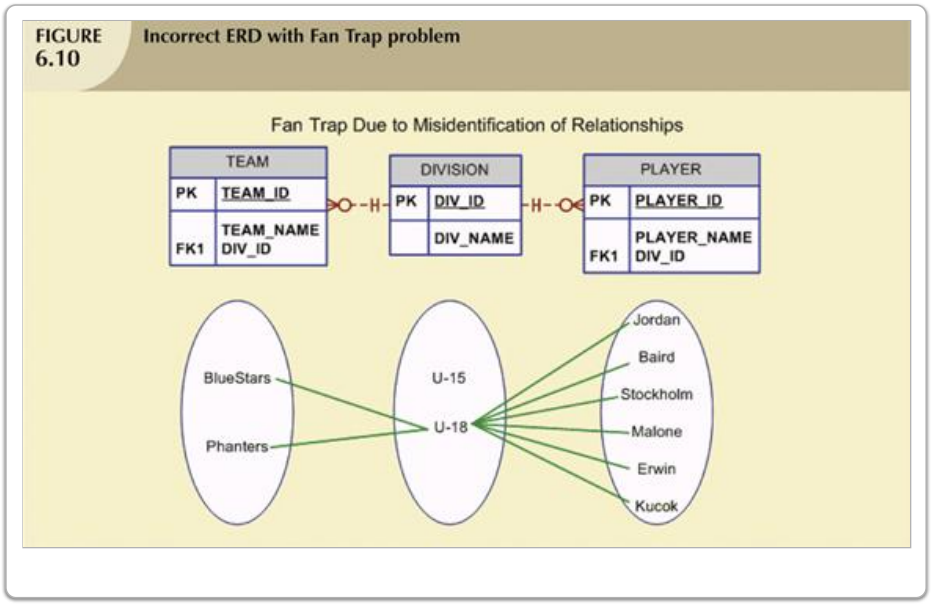
Audit data stored in a data warehouse may not be complete.

This is true. Records are inserted in history or audit tables as part of operational transactions, so the history is always up to date. A data warehouse may be updated periodically, such as in the middle of the night, so the warehouse may not have the most recent history.

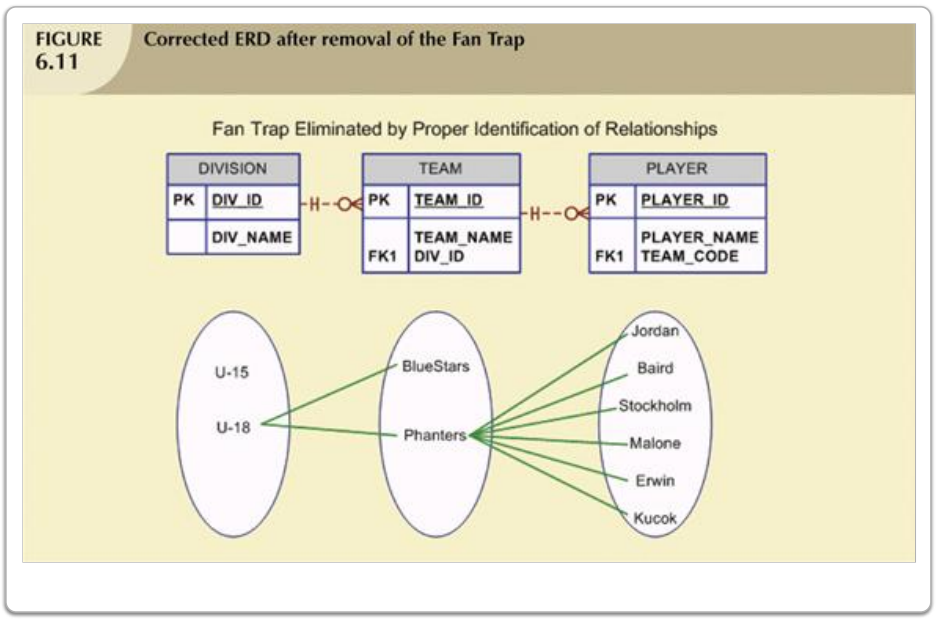
Fan Traps

Fan traps occur when fewer relationships are explicitly represented than matter in the real world and the subset of the relationships that are represented are not sufficient to infer missing important relationships. The following figure from the text illustrates a design which incorrectly fails to

represent the relationship between players and teams:



Note that there are really three relationships involved - players are in teams, teams are in divisions, and players are in divisions. In the figure above the relationships between teams and divisions and between divisions and players are correct, but the two relationships above cannot be used to infer the important relationship between players and teams. The following figure from the text includes how to correctly represent the two relationships between players and teams and teams and divisions.



You can prevent problems such as fan traps by making sure that your designs correctly represent all of the important relationships in the domain. For example there would assuredly be a business rule for the domain above similar to "Each player plays in one team." That this is not represented in Figure 6.10 should be taken as a bug in that model.

Failures of models to represent important relationships can occur with four or more entities and not just three entities as in the example above. The way to avoid these errors is to carefully document the relationships in the business rules, and to check that the model correctly represents all of those important relationships.

It is usually preferable to avoid redundant relationships, and if redundant relationships are present you can find ways to eliminate relationships so that the set of relationships that remains is minimal and complete. Sometimes in very high performance database systems the redundant

relationships are incorporated into the design. In these designs be careful that you do not introduce anomalies through inconsistent sets of relationship data.

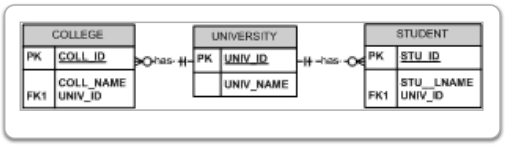
Test Yourself 3.13

Which of the following are true? (Please check all of the following that are true.)

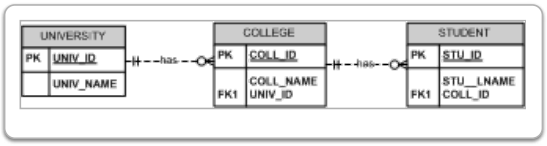
All important real world relationships are represented in a fan trap

This is false. An important real world relationship would not be properly identified in the case of a fan trap.

The following business rules are given: “Each university has many colleges”, Each college is part of one university”, “Each college has many students”, “Each student goes to one college”, “Each university has many students”, and “Each student goes to one university”. A model is produced that shows a one-to-many relationship between UNIVERSITY and STUDENT and a one-to-many relationship between UNIVERSITY and COLLEGE (With UNIVERSITY on the “one” side of both of these relationships). The model fails to show any relationship between COLLEGE and STUDENT. This would be an example of a fan trap.



This is true. The relationship between COLLEGE and STUDENT is not properly identified. (A one-to-many relationship from UNIVERSITY to COLLEGE and a one-to-many relationship from COLLEGE to STUDENT would identify this relationship.)



Data Modeling Checklist

The text includes a good data modeling checklist which does not include important checks for generalization-specialization relationships. These are summarized in the following table:

Checks for generalization-specialization relationships
Verify that all attributes of the superclass are needed in all subclasses
Verify that all common attributes of subclasses have been correctly migrated to the superclass.
Verify that domain experts agree that the subclasses are really specializations of the superclass.
Verify that the business rules associated with the superclass really do apply to all subclasses.

These checklists only span the data modeling phases of the design. Additional checks are useful after the logical model has been used to produce a physical model. These checklists include steps such as producing the SQL for all business operations and determining if it is natural and

efficient.

Summary

The basic ERM is theoretically complete, meaning that it can represent anything, but it is often awkward, because it does not distinguish specialization, which is one of the most important relationship types. Extended Entity Relationship Modeling (EERM) adds this representation capability to ERM. EERM directly and clearly represents specialization hierarchies and their unique properties. One (subtype) entity is a specialization of another (supertype) entity when all instances of the subtype are also instances of the supertype. For example the entity *student* is a specialization of the entity *person*. A supertype may have multiple subtypes. For example, both *man* and *woman* are subtypes of *person*. Different subtypes of a supertype may be disjoint or overlapping. For example, the subtypes *even_integer* and *odd_integer* are disjoint subtypes of their supertype *integer*, because no integer can be both odd and even, while the subtypes *student* and *employee* are overlapping subtypes of *person*, because a person can be both a student and an employee. An *entity cluster* is an abstract entity that is used to represent a cluster of its subtypes. Entity clusters are useful for making EERM models more understandable by hiding details while representing the common features of the subtypes.

Most entities have natural keys, which are collections of attributes that uniquely identify the instances (corresponding to table rows) of the entity. When selecting a primary key it is helpful to consider whether the key values are unique, always present, and immutable over time. The primary keys should also be reasonably compact, and it should not reveal information that may be a security risk, because the primary key will be duplicated in other tables as a foreign key. Many natural keys are *composite*, meaning that they consist of more than one attribute. When a natural key does not meet all of the requirements a *surrogate* or *synthetic* key is used as the primary key. When data changes over time and you must record the previous values, you should maintain the previous values, usually in separate history entities, usually with a 1:M relationship with the entity for which the history is being maintained. Fan traps are one example of errors that occur when the model does not preserve all of the information in the real world entities. When two or more entities are related by more than one path, then redundant relationships occur; you should be careful in these situations that the redundant relationships are necessary, and that they are maintained in a consistent state.

Lecture 8 - Normalization of Database Tables

Introduction

metcs669_module05 video cannot be displayed here

Database design is based on constructing tables and ensuring that these tables are structured soundly. Tables or entities are composed of attributes. Normalization is a sequence of tests that are applied to candidate entities and their attributes. Normalization reduces data redundancies

and helps eliminate the data anomalies that can result from those redundancies. Normalization does not completely eliminate data redundancies, but instead controls and selectively uses them to link relational database tables. Although normalization is a very important part of database design, if normalization is carried to excess it can result in less easily understood entities and slow processing speed.

Please review the following animated and comprehensive overview of normalization.

metcs669_W3L1T11_norm2004_forflv video cannot be displayed here

Test Yourself 3.14

Generally, which of the following occurs with normalization (moving from a lower to a higher form of normalization)? (Please check all of the following that are correct.)

With increasing normalization there is also increasing redundancy in a database increases.

This is false. Redundancy would be expected to decrease with normalization. When data redundancy exists, the same value (representing the same thing or attribute of a thing) is recorded multiple times. Normalization reduces data redundancies.

More tables or entities are formed.

This is true. The formation of multiple tables with foreign key links from existing tables would be expected.

Normal Forms

Normalization works through a series of stages called *normal forms*. The first three stages are described as first normal form (1NF), second normal form (2NF), and third normal form (3NF). Tables in 2NF are subject to fewer kinds of data redundancy than 1NF, and tables in 3NF are subject to fewer kinds of data redundancy than 2NF. The higher the normal form, the better it is at reducing the potential for data anomalies. Tables in first normal form have no repeating groups, that is, only one value in each row/column intersection (or "cell" in modern language). Second normal form removes all partial dependencies so that no attribute is dependent on only a portion of the primary key. Third normal form removes all transitive dependencies so that attributes that are not part of the primary key are not functionally dependent upon other attributes that are not part of the primary key.

Boyce-Codd normal form (BCNF) is a special normal form that requires that all determinants must be candidate keys, that is, all determinants must functionally determine all other attributes. BCNF does not have a number in its name, because unlike 2NF and 3NF, BCNF is does not merely add an additional structural requirement to a lower normal form. BCNF eliminates all of the same kinds of data redundancy problems as 3NF, as well as some additional kinds, while also avoiding the "bug" that is found in 3NF that prevents multiple candidate keys from appearing in the same

table. Because of this relationship between BCNF and 3NF, you will sometimes read in database literature that BCNF is a special case of 3NF, which is not correct. There are tables that are in 3NF but not BCNF, and tables that are in BCNF and not 3NF. What is correct is that in the overall hierarchy of normal forms, 1NF, 2NF, 3NF, 4NF, 5NF, and so on, BCNF eliminates the kinds of data redundancy at the 3NF "level", but not beyond that. For example, 4NF eliminates multi-valued dependencies, which BCNF does not do, and 5NF eliminates some kinds of cross-domain dependencies, which BCNF does not do.

Normalization provides an organized way of determining a table's structural status. Normalization principles and procedures provide a set of simple techniques through which we can achieve the desired and definable structural results. Normalization principles and rules drastically decrease the likelihood of producing bad table structures, they help standardize the process of producing good tables, and they make it easier to transmit skills to the next generation of database designers. The following four concepts are important to understand:

What is normalization?

Normalization is the process for evaluating the representations of entities by analyzing the dependencies between the entity's attributes. Properly executed, the normalization process eliminates uncontrolled data redundancies, thus eliminating the data anomalies and the data integrity problems that are produced by such redundancies.

Normalization does not eliminate data redundancy between entities; instead, it produces the carefully controlled redundancy that lets us relationally link database tables.

When is a table in 1NF?

A table is in 1NF when there are no repeating groups in the table. A table in 1NF still may contain partial dependencies, i.e. dependencies based on only part of the primary key.

When is a table in 2NF?

A table is in 2NF when it is in 1NF and it includes no partial dependencies. However, a table in 2NF may still have transitive dependencies, i.e. dependencies based on attributes that are not part of the primary key.

When is a table in 3NF?

A table is in 3NF when it is in 2NF and it contains no transitive dependencies.

When is a table in BCNF?

A table is in Boyce-Codd Normal Form (BCNF) if every determinant in the table is a candidate key.

Test Yourself 3.15

Which of the following are true regarding normalization? (Please check all of the following that are true.)

A table in third normal form (3NF) is more normalized than a table that is only in first normal form (1NF).

This is true. Moving from a lower normal form to a higher normal form is referred to as *normalization*. Moving from a higher normal form to a lower normal form is referred to as *denormalization*.

Suppose that table X contains attributes C and D, and that neither C nor D is part of the primary key. Suppose that attribute D is functionally dependent on attribute C. Then table X is in third normal form (3NF).

This is false. A table is in 3NF if it is in 2NF and contains no transitive dependencies. The functional dependency of C on D is a transitive dependency, so the table is not in third normal form.

A table that is in 3NF must meet the criteria for being in 1NF.

This is true. In order to be in 3NF, a table must meet the criteria for 1NF and 2NF.

A table that is in 1NF must meet the criteria for being in 3NF.

This is false. A table in a lower form of normalization does not necessarily meet the criteria for a higher form of normalization.

Transforming Normal Forms

To ensure a table is in 1NF, all repeating groups must be removed. If the table has a primary key, the table can then be put into 2NF by ensuring no attribute is dependent on only part of the primary key. If this partial dependency exists, a new table can be created with a primary key equal to the required portion of the original key. The dependent attributes are moved to this table. If the table does not have a primary key, the kind of data redundancy eliminated by 2NF, partial dependencies, cannot exist, so the table is already in 2NF. Again, if the table has a primary key, a table in 2NF can then be put into 3NF by eliminating transitive dependencies. Any transitive dependencies can be eliminated by creating yet another new table with a primary key equal to the required portion of the original key, and moving the dependent attribute into this table. To put a table into BCNF, one does not need to progress from 1NF to 2NF to 3NF. Instead, repeatedly create a new table for every determinant that is not a candidate key, make the determinant the primary key of that table, and move the determined attributes into the new table.

Test Yourself 3.16

Table Z has attributes A, B, C, D, and E. Attributes A and B form the primary key. Table Z is in 1NF (first normal form). There is one partial dependency in table Z. Attribute C is only dependent on attribute B. Which of the following are true regarding table Z? (Please check all of the following that are true.)

Table Z has no repeating groups.

This is true. In order to be in 1NF, table Z would have to have no repeating groups.

As part of the process of normalizing Z to 2NF attribute C would be removed from table Z into a new table with B as the primary key.

This is true. Attribute C could be put in another table with attribute B as the primary key.

Normalization of ER Modeling

Normalization should be part of the design process. Proposed entities must meet the required normal form prior to creating table structures. The entities and their relationships are visualized using an ER diagram. This supplies a macro view of an organization's data requirements and operations. ER diagramming is an iterative process. Relevant entities, and their attributes and relationships, are identified. The results are used to identify additional entities and attributes. Normalization takes the opposite perspective. It focuses on characteristics of specific entities. In other

words, the ER diagram looks at the "big picture" and normalization provides a "micro" view of individual entities. Like ER diagramming, normalization may provide additional entities and attributes. These new items must be incorporated into the ER diagram. ER modeling and normalization are complementary and they should be used concurrently.

Normalization takes place in tandem with data modeling. The proper procedure is to follow these steps:

1. Create a description of operations at an appropriate level of detail.
2. Derive appropriate business rules from the description of operations.
3. Model the data with the help of a tool such as Visio's Crow's Foot option to produce an initial ERD. This ERD is the initial database blueprint.
4. Use the normalization procedures to identify and remove data redundancies. This process may produce additional entities.
5. Revise the ERD created in step 3.
6. Use the normalization procedures to audit the revised ERD. If significant additional data redundancies are discovered, repeat steps 4 and 5.

Test Yourself 3.17

Which of the following are correct? (Please check all of the following that are correct.)

The normalization process should not occur until the database is being used to hold operational data.

This is false. Normalization should be an integral part of the design process, applied to the logical design for the relational model.

An ER diagram should be in its final state when the normalization process starts. Changes in the ER diagram should never result from normalization.

This is false. Normalization may result in new entities which should be included in the ER diagram.

An ER diagram may be changed multiple times during the design process.

This is true. Database design is an iterative process, which typically involves a sequence of ERDs.

Generating Information Efficiently Through Denormalization

Good normalization of a database is an important design goal, but it is only one of many such goals. Often, good database design considers other perspectives as well, such as processing speed and the naturalness and understandability of the database. As tables are decomposed during the normalization process, the number of database tables increases. Joining larger numbers of tables takes additional input/output and processing resources, thereby reducing speed. For these and other reasons, denormalizing the database to increase speed might be necessary. Unfortunately, denormalization can cause data redundancy and data anomalies. Data updates become slower and less efficient. Indexing becomes more complex and other problems can arise.

To recap with an example consider the following entity:

STUDENT(Student#, first_name, last_name, address1, address2, city, state, country, postal_code)

While this table appears to be in 3NF, it is not. The country and postal_code attributes exhibits transitive dependencies with the city and state. For a typical university database, for performance and understandability reasons it is usually desirable to leave the Student table in a "denormalized" 2NF state. In another database, such as one used for marketing or a financial services firm, a corresponding table representing persons may be further normalized, with a separate table for postal codes. In these more normalized designs the postal_code table stores demographic, geographic, and other data of use to the enterprise.

Test Yourself 3.18

Which of the following are true? (Please check all of the following that are true.)

Normalization can cause certain database operations to take longer.

This is true. Retrieving some data may be slower because it requires more table joins due to an increase in the number of tables.

Denormalization can cause certain database operations to take longer.

This is true. Data updates may become slower and less efficient if the denormalization introduces substantial data redundancy. Denormalization also increases the risk of data anomalies.

The optimal level of normalization is the same for every database.

This is false. The use of a particular database is important to determine the level of normalization for that database. For example, some very frequent and problematically slow request patterns may dictate a need for partial denormalization to improve performance, in the areas accessed by the very frequent requests.

Summary

Normalization is a series of tests applied to individual database tables. Normalization identifies and eliminates data redundancies and other database design problems which can cause data anomalies and application errors. Most business databases are nominally designed to either third normal form or its generalization Boyce-Codd normal form (BCNF). Practical database tables often appear to contain redundancies that are not consistent with their normal form; most often this is because the database must store data as it is received, even if that includes inconsistent redundancies such as incorrect postal codes. The following table summarizes the basic normal forms. There are more advanced versions of these normal forms that do not require primary keys, which we will study in CS779 Advanced Database Management.

Normal Form	What the Normal Form Requires
First	• Data are arranged in a regular table with similar data in the columns. • There are no repeating groups; each cell contains one data item.
Second	• The table is in 1NF. • Each nonkey attribute depends on the entire primary key; there are no partial dependencies.
Third	• The table is in 2NF. • The table contains no transitive dependencies; all dependencies are on the primary key.
Boyce-Codd	• All determinants are candidate keys; if a set of columns C determines any other column then it determines all other columns, namely C is a candidate key.

If performance is a problem and most of the accesses for two or more large tables are a join of the tables, then consider denormalization to combine the tables. Use denormalization with care, because it increases update costs, usually increases the size of the database, and introduces risks that the duplicated data may become inconsistent, resulting in data anomalies. Always measure performance when denormalizing for performance, because a denormalization intended for performance can often slow the database.

Aggregate Functions

This slide explores the concept of aggregate functions. We previously learned how to use functions to retrieve simple values, such as the current date, and to manipulate data in simple ways, such as uppercasing text. Sometimes we are interested not only in a single data item, but in the result of aggregating multiple data items. SQL provides many useful ways to aggregate data, and aggregate functions are one key component.

The GROUP BY Clause

The GROUP BY clause is used to specify one or more fields that are to be used for organizing rows into groups. Rows are grouped together that have the same value(s) for the specified field(s). The only simple fields that can be displayed are the ones used for grouping; any result from other fields must be specified using an aggregate function. The aggregate function will be applied to a group of rows insteadof to the entire table.

Let us look at an example:

Example For each patron list the number of books he has borrowed.

```
SELECT userid, COUNT(*)
FROM loan
GROUP BY userid;

USERID COUNT(*)
10 2
20 1
30 1
40 1
```

The effect of this SELECT is to

1. cause the SQL system to group the rows of loan by user id
2. display the user id for each group and a count of the number of rows in the group

A common error is to include a field in the listing that is not unique for the group. For example,

```
SELECT userid, callno, COUNT(*)
FROM loan
GROUP BY userid;
SELECT userid, callno, COUNT(*)
*
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

The above would be incorrect because callno is not single valued for a group. When GROUP BY is used each element of the select list must be single valued; each element must either be specified in the GROUP BY or be the result of a column function.

```
SELECT userid, callno, COUNT(*)
FROM loan
GROUP BY userid, callno;

USERID CALLNO COUNT(*)
10 100 1
10 200 1
```

```
20 300 1
30 500 1
40 600 1
```

Example For each patron list his total fines paid.

```
SELECT userid, SUM(fine)
FROM loan
GROUP BY userid;

USERID SUM(FINE)
10 0
20 50
30 50
40 10
```

In the above we see some patrons have not paid a fine but are still listed. To exclude rows from the grouping process (and from the SELECT) we specify the appropriate WHERE clause.

Example For each patron who has paid a fine, list his total.

```
SELECT userid, SUM(fine)
FROM loan
WHERE fine <>0
GROUP BY userid;

USERID SUM(FINE)
20 50
30 50
40 10
```

The only difference between the two preceding examples is that the latter does not report anything for patrons who have not paid any fines.

Restricting groups using HAVING

In the foregoing examples all groups have been reported. To eliminate groups from the result we use the HAVING clause specifying an appropriate group oriented boolean expression.

List the patron id numbers for all patrons who have paid more than $30 in fines on books with call numbers greater than 400.

```
SELECT userid
FROM loan
WHERE callno > 400
GROUP BY userid
HAVING SUM(fine) > 30;

USERID
3
```

Additional SELECT Query Keywords

The Structured-Query Language makes it fairly easily to analyze data and turn it into useful information. This section covers the DISTINCT, COUNT, and SUM keywords which are useful in analysis and reporting. There are additional select query keywords, but these are the most frequently used, and by understanding these you will be well prepared to understand additional keywords that you encounter.

We will use the following CUS_TRANSACTIONS table and data to illustrate these keywords:

CUS_TRANSACTIONS			
CUS_ID	**TRAN_NUM**	**TRAN_DATE**	**TRAN_TOTAL**
101	21452	21-May	7.99
104	21453	21-May	8.23
101	21454	22-May	6.67
101	21455	22-May	12.34
103	21456	23-May	10.63
103	21457	24-May	23.23
105	21458	24-May	4.32
102	21459	25-May	2.33

The DISTINCT clause will produce a list of values that are different from each other. Suppose that we want a list of all of the CUS_IDs of customers represented in the table. Then we can use the following SELECT with DISTINCT:

```
SELECT DISTINCT CUS_ID
FROM CUS_TRANSACTIONS
ORDER BY CUS_ID
```

The SQL statement will produce the following results:

CUS_ID
101
102
103
104
105

Suppose that we want to count the number of rows in the table. The COUNT(*) clause will total the number of records of an attribute.

```
SELECT COUNT(*)
FROM CUS_TRANSACTIONS
```

The COUNT(*) will count all of the records in the CUS_TRANSACTIONS table. In this case the value returned is 8.
COUNT can also be used with a column name as its argument. In this case it will count the number of not-null values in the column.

You can combine the COUNT keyword with the GROUP BY clause to count the number of transactions for each customer:

```
SELECT CUS_ID, COUNT(*)
FROM CUS_TRANSACTIONS
GROUP BY CUS_ID
```

The SQL statement will produce the following results:

CUS_ID	COUNT
101	3
102	1
103	2
104	1
105	1

The SUM clause adds the values of the attribute which is its argument. together. The following SQL adds the values of the TRAN_TOTAL column:

```
SELECT SUM(TRAN_TOTAL)
FROM CUS_TRANSACTIONS
```

The value returned is 75.74.

We can use SUM together with GROUP BY to add the transaction totals for all customers.. (Remember that the attribute in the GROUP BY clause must also appear in the SELECT.)

```
SELECT CUS_ID, SUM(TRAN_TOTAL)
FROM CUS_TRANSACTIONS
GROUP BY CUS_ID
```

The SQL statement will produce the following results of transaction totals grouped by customer.

CUS_ID	COUNT
101	27.00
102	2.33
103	33.86
104	8.23
105	4.32

Test Yourself 3.19

Which of the following is true regarding the SELECT command? (Please check all of the following that are true.)

The command “SELECT DISTINCT species_order FROM Species;” would provide you with a single numeric value representing the number of different values stored in the species_order

column of the Species table.

This is false. It would provide you with a list of the different values stored in the species_order column of the Species table. The list would contain each different value once, even if the value appeared multiple times in the column.

The command “SELECT SUM(pet_weight) FROM Pet;” would provide you with a single numeric value representing the number of different values stored in the pet_weight column of the Pet table.

This is false. The result of the query would be a number that is the result adding together of all the values stored in the pet_weight column of the Pet table.

The command “SELECT COUNT(*) FROM Pet;” would provide you with a single numeric value representing the number of rows stored in the PET table.

This is true. The keyword COUNT can also be used with a column name, in which case it returns the number of non-null values in the column.

The command “SELECT COUNT(*) FROM Pet;” may return a greater numeric value than the command “SELECT COUNT(pet_name) FROM Pet;”.

This is true. The column pet_name may contain NULLs. When a column name is used as an argument for COUNT, only non-null values in the column are counted.

Altering Table Structures

The structure of database tables is typically altered many times over the lifetime of the database. For example, a company may currently store customer names as one VARCHAR attribute that contains the first name, middle initial and last name. The company wishes to improve the normalization by splitting the Cus_name attributes into first name, middle initial, and last name attributes.

The company’s current CUSTOMER table is:

CUSTOMER	
PK	CUS_ID
	Cus_Name Cus_Address Cus_City Cus_State Cus_Zip

To add three columns we use the ALTER TABLE command with the ADD option. All three columns can be added in one ALTER command, or in three separate commands.

```
ALTER TABLE CUSTOMER
ADD ( Cus_FName varchar(20) NULL,
Cus_MInitial char(1) NULL,
Cus_LName varchar(30) NULL);
```

OR

```
ALTER TABLE CUSTOMER
ADD ( Cus_FName varchar(20) NULL);

ALTER TABLE CUSTOMER
ADD (Cus_MInitial char(1) NULL);

ALTER TABLE CUSTOMER
ADD (Cus_LName varchar(30) NULL);
```

We allow the new columns to be NULL, because these columns initially contain no data. Had we tried to create these columns with a NOT NULL constraint the DBMS would have not allowed the ALTER TABLE command because to do so would violate the NOT NULL constraint.

After the ALTER TABLE command(s) the table structure is:

CUSTOMER(New)	
PK	CUS_ID
	Cus_Name Cus_FName Cus_MInitial Cus_LName Cus_Address Cus_City Cus_State Cus_Zip

Eventually we want to delete the Cus_Name attribute in the table, but before we drop this column we must create a procedure to distribute the Cus_Name data into three separate attributes. (We will learn how to do this using SQL later in the course.) Once this is done we can use the ALTER TABLE command with the DROP option to drop the Cus_Name column.

```
ALTER TABLE CUSTOMER
DROP Cus_Name;
```

We can add the NOT NULL constraints for the new attributes that have no NULL values. Suppose that there is no data in the database for all customers' first name and last name, but not for all customers' middle initial. Suppose further that there is a business rule that all customer records should have first and last names. Then we can add the not null constraint to the Cus_FName and Cus_LName columns. We can use the ALTER TABLE command with the MODIFY option to add the NOT NULL constraints. This can be done by modifying both columns in one statement or by two separate statements.

```
ALTER TABLE CUSTOMER
MODIFY (Cus_FName varchar(20) NOT NULL,
Cus_LName varchar(30) NOT NULL);
```

OR

```
ALTER TABLE CUSTOMER
MODIFY (Cus_FName varchar(20) NOT NULL);
ALTER TABLE CUSTOMER
MODIFY(Cus_LName varchar(30) NOT NULL);
```

Using Simple SQL Functions

A SQL function usually performs a specific task, and provides a reusable way to access the logic is encapsulates. A SQL function may perform a simple task, such as returning today's date, or may perform a complicated task, such as performing a complex mathematical calculation over all rows in a table. SQL functions optionally take one or more parameters. For example, a function which calculates the number of days between two dates would have two parameters—one for the start date and another for the end date. SQL functions optionally return a value, for example, a number. If the function returns a single value, list of values, or a table, that function can be used in SQL statements wherever one of those three is expected. SQL functions usually encapsulate the logic needed for a specific task.

Although modern DBMS come with many predefined functions, many also support the creation of new, custom functions. If the DBMS doesn't already have a function to perform a task needed by a database user, the user can define their own function. This means that the kinds of functions available are virtually limitless.

The types of parameters and return values supported by functions in modern DBMS are large. For example, the type of parameter or return value can be table row, a table, or even a complex object representing the combination of several tables. The type of a parameter or return value can also be something more advanced such as a cursor.

SQL functions provide many advantages. One advantage is that an algorithm performing a specific task can be encapsulated within a function, thus eliminating the need for each user in the database to rewrite the logic every time they wish to perform that task. Another advantage is that the logic can be tested and debugged once when the function is created, rather than every time the logic needs to be used. Another advantage is that modern DBMS can ship with powerful functionality for database users, without requiring them to understand all of the internals of the DBMS. For example, it may be difficult or impossible to navigate the tables provided by a DBMS to determine today's date, but a function which returns today's date can be simple to use and understand. In many ways, the advantages of SQL functions correspond to the advantages of functions in other programming languages.

Let us examine the question "What MS Access/SQL Server function should you use to calculate the number of days between the current date and January 25, 1999?"

```
SELECT DATE()-#25-JAN-1999#
```

NOTE: In MS Access and the non-ANSI configurations of Microsoft SQL Server you do not need to specify a FROM clause for this type of query. Note that the DATE function with no arguments returns the current date.

Let us examine the question "What Oracle function should you use to calculate the number of days between the current date and January 25, 1999?"

```
SELECT SYSDATE - TO_DATE('25-JAN-1999', 'DD-MON-YYYY')
FROM DUAL;
```

Note that in Oracle, the SQL statement requires the use of the FROM clause. In this case, you must use a DUAL table. (A DUAL table is a dummy "virtual" table provided by Oracle for this type of query.) In Oracle the reserved word SYSDATE returns the current data and time on the database server; this is similar to DATE() in Microsoft SQL Server. Note also that Oracle requires us to use the function TO_DATE to convert the string '25-JAN-1999' into a date. The formatting string 'DD-MON-YYYY' tells Oracle how we want to interpret the date string '25-JAN-1999'. The 'DD-MON-YYYY' tells us that the first two characters of the date string are decimal days of the month. The 'DD-MON-YYYY' tells Oracle that there should be a hyphen after the two days for the month. The 'MON' tells Oracle that the next three characters should be one of the three-character month names specified in the locale in the database instance;

Test Yourself 3.20

Which of the following is true regarding the SQL statement: “SELECT SYSDATE – TO_DATE (‘03-JAN-2011’, ‘DD-MON-YYYY’) FROM DUAL;”? (Please select all of the following that are true.)

The TO_DATE function used in this statement can be used multiple times, in multiple statements, and with a different parameter each time it is used.

This is true. A predefined function provides a reusable way to access the logic is encapsulates. The user does not have to write the logic or even fully understand what is being done to arrive at a value returned by the function.

The TO_DATE function used in this statement, as with all functions, returns a value that is the same data type as its parameter.

This is false. The TO_DATE function returns a different data type than its parameter. This is common regarding functions.(In the case of the TO_DATE function, a date value is returned after being provided with a string and a format.)

The SQL statement provided uses more than one function to calculate the number of days between two dates. It is possible for a user to create a new function that takes more than one parameter and will return the number of days between two dates.

This is true. It is common for a modern DBMS to support the creation of custom functions. A function may take more than one parameter.

The SQL statement provided uses more than one function to calculate the number of days between two dates. It is possible for a user to create a new function that takes more than one parameter and will return the number of days between two dates AND the number of these dates that are Mondays.

This is true. A function may return a list of values. (The return possibilities for a function include a single value, a list of values, a table, and other more complex possibilities.)

Before the given statement can be executed, an Oracle user must create the table DUAL and insert data into DUAL.

This is false. In Oracle, the SELECT statement requires the use of the FROM clause. A DUAL table is a dummy "virtual" table provided by Oracle for this type of query. The user does not have to create the table or insert any data into it to reference the table in a FROM clause.

Relational Algebra

Relational algebra defines the mathematical theory for how we manipulate table contents in relational databases. Relational algebra includes eight primary operators: SELECT, PROJECT, JOIN, INTERSECT, UNION, DIFFERENCE, PRODUCT, and DIVIDE. All of these operators except DIVIDE are supported in most modern SQL databases. The SELECT operator lists table rows according to particular criteria. PROJECT yields a vertical subset for a table. JOIN combines information from two or more tables. INTERSECT combines the results of operations, yielding rows that appear in all of the results. UNION combines the results of operations, yielding distinct rows that result from any of the operations. DIFFERENCE also combines the results of operations, providing the rows produced by one operation that are not produced by the other. PRODUCT yields all possible pairs of rows from two tables. DIVIDE singles out particular characteristics shared by two tables.

Since there is a mathematical foundation to relational databases, let us briefly look at some of the database operators. In this course we focus on SQL to formulate queries, and not on relational algebra per se, so we provide basic examples of how these relational algebra operations are expressed in SQL. Because SQL is so useful and important, we will spend all of lectures seven and eight on SQL.

The **SELECT** operation is used to select a *subset* of the tuples (rows) from a relation that satisfy a **selection condition**. **SELECT** is also termed *restrict* and the operation is termed *restriction*. One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. For example, we can select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000. We can individually specify each of these two conditions with a SELECT (RESTRICT) operation. In SQL the keyword that begins a restriction is WHERE.

If we think of a relation as a table, the SELECT operation selects some of the rows from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. For example, to list each employee's first and last

name and salary, and not the other attributes of the employee relation we can use the PROJECT operation. In SQL the project operation is usually a list of the names of the columns that are desired in the result set.

Let us illustrate these concepts with SQL for a BOOK table where we have columns call_no, title, and subject. (e.g., 100, Database Systems, Computing). We create this table using the following SQL CREATE TABLE statement:

```
CREATE TABLE Book(
call_no DECIMAL(15),
title VARCHAR(50),
subject VARCHAR(20));
```

This CREATE TABLE statement creates a table called Book, with columns call_no, title, and subject. There is no data in this table; it's like an empty spreadsheet. Unlike a spreadsheet there is a constraint on the type of data that can be in each column. The call_no column can hold decimal numbers up to 15 digits. The title column can hold character strings of up to fifty characters. The subject column can hold character strings of up to twenty characters.

```
CREATE TABLE Subject(
subject VARCHAR(20),
description VARCHAR(1024));
```

Advanced Topic: This is a Simplification

Students skilled in SQL will recognize that these simple CREATE TABLE statements omit keys and other details that one might expect to find in production designs. We omit these to begin gently for students who are new to SQL.

The following table summarizes the relational operators. Don't worry if you don't understand these yet; we will go over them again several times.

Operation	Purpose	Example SQL
SELECT (RESTRICT)	Selects all tuples that satisfy the selection condition from a relation R.	`SELECT * FROM Book WHERE subject = 'Computing';`
PROJECT	Produces a new relation with only some of the attributes of R.	`SELECT title FROM Book;`
THETA JOIN	Produces all combinations of tuples from and that satisfy a join condition which is not an equality.	`SELECT Book.title FROM Book, Year WHERE Book.year_published > Year.year`
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	`SELECT title, description FROM Book, Subject WHERE Book.subject = Subject.subject_id;`
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	`SELECT title, description FROM Book NATURAL JOIN Subject;`
UNION		`SELECT subject from`

	Produces a relation that includes all the tuples in R_1 or R_2, or both R_1 and R_2; R_1 and R_2 must be union compatible.	`BOOK`
		`UNION`
		`SELECT subject from Subject`

INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	`SELECT subject FROM BOOK`
		`INTERSECT`
		`SELECT subject from Subject`

DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	`SELECT subject FROM Subject`
		`MINUS`
		`SELECT subject FROM Book`

CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	`SELECT *` `FROM Book, Subject`

Test Yourself 3.21

Please select all of the following that are correct.

The Cartesian product of two tables with one thousand rows each would have one million rows.

This is true. The Cartesian product includes all possible combinations of rows from the tables.

The PROJECT operation is used on a table with 12 rows and 10 columns. The result may contain 12 rows and 3 columns.

This is true. The PROJECT operation selects certain columns from the table and discards the other columns.

The SELECT operation is used on a table with 12 rows and 10 columns. The result may contain 10 rows and 10 columns.

This is true. The SELECT operation selects a subset of rows based on a selection condition, such as an attribute having a certain value.

The system catalog is a subset of the data dictionary.

This is false. The data dictionary contains, at a minimum, the names of all tables and all attribute names and characteristics for each table in the system. It is a subset of the system catalog. The system catalog contains more information and is created by the database management system.

UNION and INTERSECTION are the same operation.

This is false. The UNION of two union compatible tables will include all rows that occur in either (or both) tables. The INTERSECTION of two union compatible tables will include **only** the rows that occur in both tables.

The Importance of Indexing

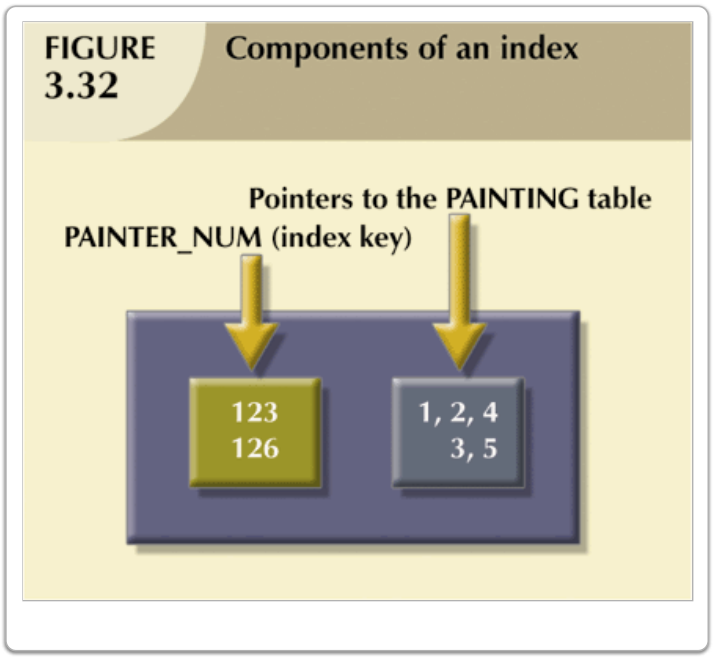
An index provides a quick method for pointing to a particular location within a database. Indexes increase data retrieval speed and can be easily created with SQL commands. From a conceptual point of view, an index is composed of an index key and a set of pointers. The index key is the index's reference point. Without an index, each row in a database must be examined to find one meeting a particular criterion. An index speeds the search by pointing to the location of the data identified by the key.

An index provides direct and fast access to rows in a table. The concept is very similar to your textbook index. For example, if you want to find places in the text that discuss B-tree indexes you look in the index, and discover that B-tree indexes are described on pages 452 and 453 and you don't need to search for B-tree indexes. In a DBMS appropriate use of indexes greatly reduces the time, disk I/O and processor resources required to identify the rows that satisfy a restriction, by using an indexed path to locate data quickly. The index is automatically used and maintained by a DBMS. Once an index is created, no direct activity is required by the user.

Indexes are associated with single tables, but they are logically and physically independent of the table they index. Consequently indexes can be created or dropped at any time and have no effect on the base tables or other indexes. When you drop a table, any indexes on that table are also dropped.

Here is a brief summary of indexes:

- An i*ndex* is a data structure associated with a table that is used to efficiently access the rows in a table that match an index key or range of index keys. For example, there may be an index named EMPLOYEE_LAST_NAME_SX index on the last_name column of a EMPLOYEE table to help us efficiently retrieve the rows of the EMPLOYEE table that correspond to employees with particular last names.
- An *index key* is the set of columns in the table that are indexed. For example, the last_name column is the index key in the EMPLOYEE_LAST_NAME_SX example above.
    - The index contains an entry for each distinct value of the index key. For example, there would be an entry in the EMPLOYEE_LAST_NAME_SX for the last_name 'Smith' index key value if there are any EMPLOYEEs who have last_name 'Smith.'
    - Each entry in the index contains a list of references (pointers) to the table rows where the value in the row matches the index key. For example, the EMPLOYEE_LAST_NAME_SX would have an entry under the index key 'Smith' for each EMPLOYEE whose last_name is 'Smith.'
- A *Unique Key*
    - A *unique index* is an index in which the index key can only have one pointer value (row) associated with it. Unique indexes are used to enforce uniqueness constraints. For example, a EMPLOYEE table may have a unique index on the social_security_number column to enforce the business rule that no two employees can have the same social security number.
- Each index is associated with only one table
- The figure below illustrates at a very conceptual level an index on a PAINTING table to efficiently support retrieval of paintings by the painter_number column of the PAINTING table.



Advanced Topic: row ids

The figure above suggests that the pointers in an index are just the row numbers in the table. In fact the pointers are row references (In Oracle termed ROWIDs), which are rather large safe reference data structures that provide all of the information necessary for the DBMS to efficiently locate the referenced row wherever it may be. The size of row ids contributes significantly to the size of indexes that require them.

Test Yourself 3.22

Indexing can increase data retrieval speed. Which of the following is correct regarding indexing? (Please check all of the following that are correct.)

Conceptually, an index is composed of an index key and a set of pointers.

This is true. The index key is the column or columns of a table for which the index is keeping track of value locations.

Without an index, getting the result for the SQL statement “SELECT * FROM PET WHERE PET_SPECIES = “FROG”;” would require the examination of all rows of the table PET.

This is true. An index could speed the search significantly by pointing to the location of the data identified by the key.

An index on the PET_SPECIES column of the PET table would maintain a list of references to table rows for each distinct value of PET_SPECIES.

This is true. PET_SPECIES would be the index key with the references (pointers) indicating the row or rows that contain each particular value for the column PET_SPECIES such as “FROG”.

With an index on the PET_SPECIES column of the PET table, getting the result for the SQL statement “SELECT * FROM PET WHERE PET_SPECIES = “FROG”;” would require the examination of all rows of the table PET.

This is false. The index could provide the location of the rows to retrieve, so each row does not have to be examined.

An index cannot be dropped without losing all the data contained in a table.

This is false. Indexes are logically and physically independent of the table they index. An index can be dropped at any time with no effect on the table.

Set Operators

The UNION, INTERSECT, and MINUS operators combine two or more sets to create new sets (or relations). UNION, INTERSECT, and MINUS work correctly only if relations are union-compatible. In SQL terms, union-compatible means that the data types must be *union compatible* The UNION statement combines rows from two or more queries without including duplicate rows. The UNION statement can unite more than two queries. Use the UNION ALL operator to produce a relation that retains the duplicate rows. The INTERSECT statement combines rows from two queries, returning only the rows that appear in both sets. The MINUS statement in SQL combines rows from two queries and returns only the rows that appear in the first set but not in the second. If your DBMS does not support the INTERSECT and MINUS statements, you could use IN and NOT IN subqueries to obtain the same results.

Our RobCor text indicates that *Union-compatible* means that the relations yield attributes with identical names and compatible data types. By the textbook definition, the relation **A(c1,c2,c3)** and the relation **B(c1,c2,c3)** have union compatibility if the columns have the same names, are in the same order, and the columns have "compatible" data types. *Compatible data types do not require that the attributes be exactly identical*—only that they are *compatible*. For example, VARCHAR(15) and CHAR(15) are union compatible, as are NUMBER (3,0) and INTEGER, and so on. Most DBMS will convert between related data types such as CHAR and VARCHAR, so the data types do not have to be exactly the same; this data type *coercion* is DBMS-dependent.

The more flexible Oracle definition of union compatibility only requires that the number of columns be the same. Oracle takes the column names or column aliases provided for the first result set as the column names for the UNIONed result set. For example, the relation **A(c1,c2)** and the relation **B(c3,c4)** are union compatible if the data type of c1 is compatible with the data type of C3, and the data type of c2 is compatible with the data type of c4. The column names of the union of A and B would be (c1, c2).

ISO SQL provides the UNION operator to combine the results of two SELECTs. Some SQL implementations include DIFFERENCE or MINUS to subtract one result from another, and INTERSECTION to determine the rows in common for two results.

Test Yourself 3.23

Which of the following is correct? (Please check all of the following that are correct.)

When applied to the same sets or relations the result of the UNION ALL operator may have more rows than the result of the UNION operator.

This is true. The UNION ALL operator retains duplicate rows. If the sets had duplicate rows, the result of the UNION ALL operator would have more rows than the result of the UNION operator.

In order to be union-compatible, data types must be identical.

This is false. In order to be union-compatible, data types must be compatible. For example, VARCHAR(15) and CHAR(15) are union compatible, as are NUMBER (3,0) and INTEGER, and so on. Data type compatibility may vary with the DBMS.

“SELECT PET_NAME FROM PET1 MINUS SELECT PET_NAME FROM PET2;” must have the same results as “SELECT PET_NAME FROM PET2 MINUS SELECT PET_NAME FROM PET1;”.

This is false.

A particular DBMS may not support the MINUS statement.

This is true. The INTERSECT statement may not be supported by a particular DBMS either.

You want the results of the query “SELECT pet_name FROM Pet1 MINUS SELECT pet_name FROM Pet2;”, but the DBMS that you are using does not support the MINUS statement. You could get the results that you want with this query: “SELECT pet_name FROM Pet1 WHERE pet_name NOT IN (SELECT pet_name FROM Pet2);”.

This is true. This is another way to obtain these results.