Module 2

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 2 Study Guide and Deliverables

Background Concepts Readings:	- Coronel & Morris, chapters 3 and 4
Optional SQL Readings:	- *12th Edition:* Coronel & Morris, sections 7.5 through 7.7 of chapter 7, section 8.1 of chapter 8 - *13th Edition:* Coronel & Morris, sections 7.4 through 7.6 of chapter 7, sections 8.3 and 8.4 in chapter 8 (note that 8.4b, subqueries, will be covered in more detail in week 5)
Assignments:	- Term Project Iteration 2 due **Tuesday, January 31 at 6:00 AM ET** - Lab 2 due **Tuesday, January 31 at 6:00 AM ET**
Live Classroom:	- **Tuesday, January 24 from 8:00-9:30 PM ET** - **Wednesday, January 25 from 8:00-9:30 PM ET**

Which First?

Read the book chapters before reading the online lectures.

Notes

Section 4.1.6 describes a convention whereby a solid line is used to represent an identifying relationship, and a dashed line is used to represent a weak relationship. Although it is quite important to understand both kinds of relationships and the distinction between the two, the convention to use solid and dashed lines in an ERD is not universal. Therefore, it is not expected that you use this convention in course assignments, and facilitators will not grade you on use of this convention.

Page 116 in the text contains a note from the authors stating that "the order in which the tables are created and loaded is very important," further indicating that a referenced table must be created before any referencing tables. It is best practice to avoid defining foreign key constraints in your CREATE TABLE definitions, and to instead define foreign key constraints using ALTER TABLE definitions *after* all tables have been created. By using this best practice, the order of the CREATE TABLE statements is irrelevant, and you will save countless hours of creating dependency graphs and ordering your tables before creating them. Production systems oftentimes contain hundreds of tables, and creating dependency graphs for this number of tables is cumbersome.

Lecture 4 - The Relational Database Model

Introduction

metcis669_09_sp1_bshdy_lec03 video cannot be displayed here

In this lecture we learn about the relational database model, and about the Structured Query Language (SQL), which Codd and his colleagues developed as a language for communicating with relational database management systems. Chapter three of our text covers the relational database model, but it does not include the material in this lecture on SQL. SQL is a central topic in this course, so we introduce it as soon as we can, in this lecture, so that you can begin your hands-on database work.

The relational database model allows the designer to focus on the logical representation of data and its relationships, rather than on physical storage details. The practical significance of taking the logical view reminds the end user of the simple file concept of data. Unlike a file, however, the relational database provides the advantages of structural and data independence. Related records can be stored in independent tables that are related.

The relational model was introduced by Ted Codd of IBM Research in 1970 and attracted immediate attention due to its simplicity and mathematical foundations. The model uses the concept of a mathematical relation-which is a table of values-as its basic building block, and has its theoretical basis in set theory and first-order predicate logic. The model has been implemented in a large number of commercial systems over the last thirty years-with Oracle and IBM's DB2 (now UDB) being amongst the earliest Relational Database Management Systems (RDBMS).

Related tables constitute a database. For example Oracle provides the following sample database-EMP and DEPT.

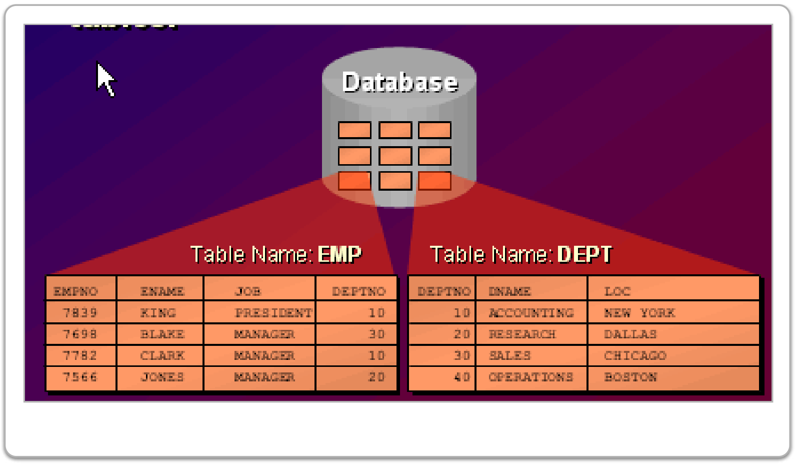


Table Name: **EMP**

EMPNO	ENAME	JOB	DEPTNO
7839	KING	PRESIDENT	10
7698	BLAKE	MANAGER	30
7782	CLARK	MANAGER	10
7566	JONES	MANAGER	20

Table Name: **DEPT**

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Basic Components of a Relational Database Model

The primary components of a relational database are entities, attributes, and resulting relationships. ***Entities*** are persons, places, things or shared ideas about which data are collected and stored. ***Attributes*** are characteristics that describe entities. Entities are often grouped according to common attributes. These groups are called ***entity sets***. Entity sets are stored in ***tables***. Unique attributes or ***keys*** are used to identify specific occurrences of entities within an entity group. These key attributes are used to create controlled redundancies that link tables and form relationships. Relationships can be classified as one-to-one (1:1), one-to-many (1:M), or many-to-many (M:N). This indicates the number of entities that can be related to each other.

The following are key characteristics of a relational table:

Table 3.1

Characteristics fo a Relational Table	
1	A table is perceived as a two-dimensional structure compose of rows and columns.
2	Each table row (tuple) represents a sing entity occurance within the entity set.
3	Each table column represents an attribute, and each column has a distinct name.
4	Each row/column intersection represents a single data value.
5	All values in a column must conform to the same data format. For example, if the attritbute is assignmed an integer data format, all values in the column representing that attribute must be intergers
6	Each column has a specific range of values known as the attribute domain.
7	The order of the rows is immaterial to the DBMS.

Review the animated introduction to core relational database terminology below.

Relational Database Terminology

1/4

Test Yourself 2.1

Which of the following are true regarding the relational database model? (Please select all of the following that are true.)

A particular value such as the number “20” representing a store number could be a foreign key value in one table or a primary key value in another table, but not both.

This is false. A foreign key refers to the primary key or a unique key in another table. The value of the foreign key will be the same as the value of a primary key that it refers to. Relational databases express relationships between entities by common data in the related entities.

A row in one table must only be related to one row in another table.

This is false. A row in one table can be related to no rows, one row, or many rows in another table.

A field does not have to contain a value.

This is true. A field is found at the intersection of a row and a column. When it is allowed, a field may contain no value. This is called a *null* value.

A primary key field must contain a value.

This is true. A primary key value uniquely identifies a row stored in a table. All primary key fields must have a value. Primary key fields cannot be null.

The Data Dictionary

The *data dictionary* provides a detailed accounting of all tables found within the database. It contains, at a minimum, the names of all tables and all attribute names and characteristics for each table in the system. The *system catalog* contains more information than the data dictionary and is created by the database management system. The data dictionary can be derived from the contents of the system catalog. In modern DBMS such as Oracle users can access the data dictionary subset of the system catalog using *data dictionary views*. We will study views in Lecture 7.

Organizing Entities and their Attributes into Tables

Entities are the basic building blocks of a relational database. Entities are represented with tables. A relational *table* is composed of intersecting rows and columns, like a spreadsheet. Each *row* (or *tuple*) represents an occurrence of an entity. Each column represents a characteristic of the entity. These characteristics are called *attributes*. In SQL databases tables that represent real world entities each row must have an attribute or attributes that uniquely identify it. The unique identifier is called a *primary key*. Although tables are independent, they may be linked through shared attributes. The primary key in one table may appear again as a *foreign key* in another table. To maintain *referential integrity*, the foreign key must contain values only found in the other table, or null values to indicate that the rows are not linked.

Although a table in a relational schema commonly will have a primary key, the table may have no primary key if there is no need to uniquely identify each row in the table. Tables often represent real-world or abstract entities, where each row in the table represents an entity instance. The rows of such tables must therefore be uniquely identifiable through use of a primary key. Tables sometimes do not represent real-world or abstract entities. For example, the purpose of history and audit tables is to accumulate changes in data over time; each row represents a change in one or more entity instances, but not an entity instance itself. These kinds of tables often do not have a primary key, and attempting to establish a primary key can be awkward or unnecessary.

The ability to relate data in one table to data in another table enables you to organize information in separate manageable units. For example, employee data can be kept logically distinct from department data by storing employee data and department data in separate tables.

The data that each table contains usually describes only one type of entity. For example, the EMP table only contains information about employees. Categories of data are listed across the top of each table, and individual cases are listed below. Using a table format, you can readily visualize, understand, and use information. The visual presentation of relational database tables is similar to that of a spreadsheet such as Microsoft Excel. A relational table has additional constraints, which are checked by the normalization rules, which we will study in Lecture 5 and Chapter 5. These constraints include not having repeated groups of data in a cell and having data of the same data type and meaning in each column.

metcs669_W2L1T11_relatetables video cannot be displayed here

Because data about different entities are stored in different tables, you may need to combine two or more tables to answer a particular question. For example, you may want to know the location of the department where an employee works. In this scenario, you need information from the EMP table (which contains data about employees) and the DEPT table (which contains information about departments). An RDBMS enables you to relate the data in one table to the data in another by using *foreign keys*. A foreign key is a column or a set of columns that refer to a unique key in the same table or another table.

Test Yourself 2.2

Which of the following is true regarding relational tables? (Please select all of the following that are true.)

All data related to or associated with a particular entity occurrence must be stored in the same table.

This is false. Related or associated data may be stored in separate tables. For example, details regarding a person's home may be stored separately from other information about a person. The indication or maintenance of the relationship may be achieved through the use of foreign keys. The ability to relate data in one table to data in another table enables you to organize information in separate manageable units.

A foreign key may contain a value such as "35" or "HR" that is not contained in any other table, but it cannot contain a null value.

This is false. A foreign key represents a relationship between an entity occurrence in one table and an entity occurrence in another table. If a foreign key value did not occur in any other table, it would be indicating a relationship to an entity that was not represented in the data, so this is not allowed. A foreign key can be null, indicating that a particular entity occurrence is not related to any entity occurrence in the table that the foreign key refers to.

Each tuple represents the occurrence of an entity.

This is true. A relational table is composed of intersecting rows and columns. Each tuple (or row) represents the occurrence of an entity.

Each column of a relational database tables represents a characteristic of a the entities of the type that the table represents.

This is true. These characteristics are called *attributes*.

The primary key value "6275" may represent multiple entity occurrences in the same table.

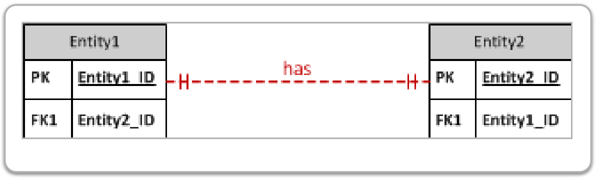
This is false. Each primary key value uniquely identifies an entity instance. If more than one entity instance had the same value for the primary key, it would mean that the rows with the duplicate primary keys actually represent the same real world entity, so duplicate primary key values are not allowed.

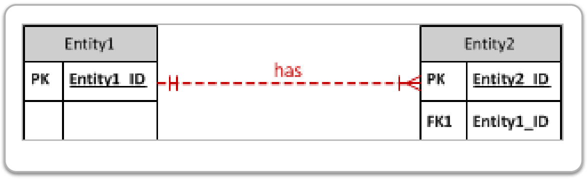
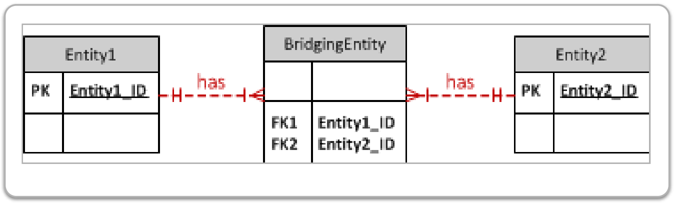
Relationships in the Relational Model

In the relational model, two or more entity instances can be related by containing duplicate data values. When two or more entity instances have the same values in one or more attributes, they are considered to be related. This type of relationship is the only type supported in the relational model, and is the basis for the word “relational” in the term “relational databases”. This can be contrasted with the object-oriented model, which supports many types of relationships. Because this type of relationship is fundamental to both the relational model and relational databases, almost every relational schema incorporates the use of this relationship in some way.

A foreign key constraint enforces this type of relationship across all instances of two entities. When a foreign key constraint is present, it can be said that those two entities are related. When an attribute or group of attributes is assigned a foreign key constraint, all of the values in that attribute or group of attributes must be present in another entity, or else be null. In other words, a foreign key constraint requires that an attribute or a group of attributes relate each entity instance to another entity instance, or else be null.

Therefore, entities are related through the use of foreign keys, and there are three standard methods corresponding to the three relationship types—1:1, 1:M, and M:N. These approaches are described in the table below.

Relationship Classification	**Method**	**Entity Relationship Diagram**
1:1	If each instance of Entity1 is related to at most one instance of Entity2, and each instance in Entity2 is related to at most one instance in Entity1, then a foreign key must be placed in both entities. Entity1 has a foreign key which references a unique key in Entity2, and Entity2 has a foreign key which references a unique key in Entity1.	 Entity1 PK Entity1_ID FK1 Entity2_ID has Entity2 PK Entity2_ID FK1 Entity1_ID

1:M	If each instance of Entity1 is related any number of instances of Entity2, and each instance in Entity2 is related to at most one instance in Entity1, then a foreign key must be placed in Entity2. Entity2 has a foreign key which references a unique key in Entity1.	 Entity1 PK Entity1_ID has Entity2 PK Entity2_ID FK1 Entity1_ID
M:N	If each instance of Entity1 is related any number of instances of Entity2, and each instance in Entity2 is related to any number of instances in Entity1, then a foreign key cannot be placed in either entity. Instead, a bridging entity must be created which has a foreign key to each entity. The bridging entity has a foreign key which references a unique key in Entity1, and another foreign key which references a unique key in Entity2.	 Entity1 PK Entity1_ID has BridgingEntity FK1 FK2 Entity1_ID Entity2_ID has Entity2 PK Entity2_ID

How to Handle Data Redundancy

Relational database tables are independent but they can be linked through shared common attributes. These attributes are a form of controlled redundancy. Although data redundancy can lead to data anomalies and result in incorrect operation of database applications, proper use of foreign keys can minimize data redundancy and reduce the chance that destructive anomalies will develop. Database designers must often balance design elegance, processing speed, and information requirements when creating a database.

While the following two tables can be integrated as one, it is desirable to design them as two tables-a DIRECTOR table and a PLAY table. This minimizes data redundancy. We then link them via a foreign key.

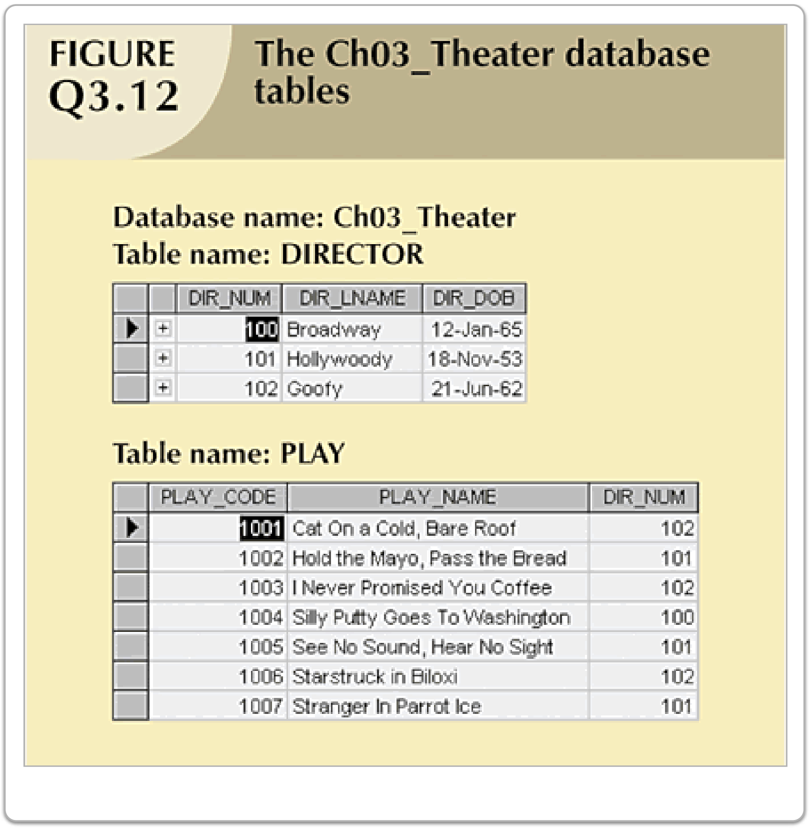


FIGURE Q3.12 The Ch03_Theater database tables

Database name: Ch03_Theater
Table name: DIRECTOR

DIR_NUM	DIR_LNAME	DIR_DOB
100	Broadway	12-Jan-65
101	Hollywoody	18-Nov-53
102	Goofy	21-Jun-62

Table name: PLAY

PLAY_CODE	PLAY_NAME	DIR_NUM
1001	Cat On a Cold, Bare Roof	102
1002	Hold the Mayo, Pass the Bread	101
1003	I Never Promised You Coffee	102
1004	Silly Putty Goes To Washington	100
1005	See No Sound, Hear No Sight	101
1006	Starstruck in Biloxi	102
1007	Stranger In Parrot Ice	101

a. Primary keys.

DIR_NUM is the DIRECTOR table's primary key.
PLAY_CODE is the PLAY table's primary key.

b. Foreign keys.

The foreign key is DIR_NUM, located in the PLAY table. Note that the foreign key is located on the “many” side of the relationship between director and play. (Each director can direct many plays...but each play is directed by only one director.)

Test Yourself 2.3

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

Even if two tables can be integrated into one, it is often desirable to design them as two separate tables.

This is true. The tables may be linked through the use of a foreign key.

Data redundancy is never deliberate in a database design.

This is false. Data redundancy in a database design may be deliberate based on a consideration of processing speed and information requirements. Consider the example of an address, which usually includes the city, state, and postal code. The city and state can be inferred from the postal code, but the postal code is included in the address to speed processing and to provide redundancy in case there are errors in the data. A database that accepts addresses will commonly store the address as it is received, with the redundant data. Subsequent processing can then use the redundant data to identify and correct the errors.

Proper use of a foreign key can result in storing one row with unique attribute values in a separate table referenced by the foreign key as opposed to storing multiple columns with the same values in numerous rows of one table.

This is true. This is a description of a use of a foreign key to minimize data redundancy.

Lecture 5 - Entity Relationship Modeling

Introduction

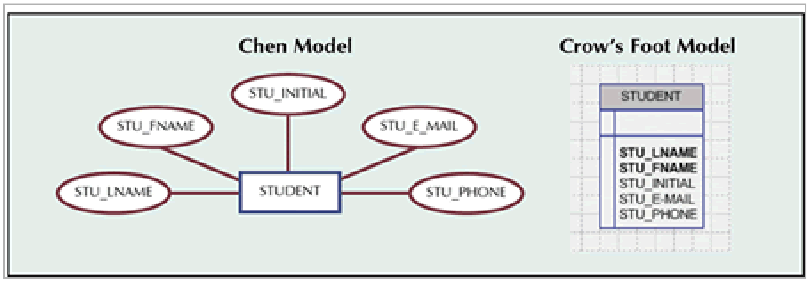
metcis669_09_sp1_bshdy_lec04 video cannot be displayed here

Relationships are associations between entities. The entities engaged in a relationship are called *participants*. *Connectivity* describes the optionality and plurality for one entity with respect to a related entity. *Cardinality* expresses the specific number of entity occurrences associated with an occurrence of a related entity. Connectivities and cardinalities are generally based on business rules and must consider the data environment, transactions, and information requirements.

Relationships have various constraints and properties. Participation in relationships is said to be either *mandatory* or *optional*. Optional relationships occur when one entity occurrence does not require a corresponding entity occurrence in a particular relationship. Mandatory relationships require the corresponding entity occurrence. Participation is also said to be either *singular* or *plural*. Singular relationships occur when one entity occurrence is related to at most one other entity occurrence. Plural relationships occur when one entity occurrence can be related to multiple occurrences of another entity. A relationship also has a *degree*. This indicates the number of associated participants. An *unary* relationship exists when an association is maintained in a single entity. A *binary* relationship exists when two entities are related and a *ternary* relationship indicates that three entities are associated.

Characteristics of the entities help define further details concerning relationships. For instance, if an entity's existence depends upon another entity, it is said to be *existence-dependent.* If an entity can exist apart from any other entity, it is said to be existence-independent. If a relationship relates an existence-dependent entity to another entity, and it is through the relationship that the first entity is existence-dependent, that relationship is a *strong* or *identifying* relationship. If a relationship relates two entities, and the relationship does not form existence-dependence in either entity, the relationship is a *weak* or *non-identifying* relationship.

The following diagram illustrates sample attributes for a student entity:



Test Yourself 2.4

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

Cardinality is the number of relationships that an entity is engaged in.

This is false. Cardinality is the number of entity occurrences associated with an occurrence of a related entity.

An example of a Unary relationship would be an EMPLOYEE entity with an EMP_SUPERVISOR attribute in a situation where employees are supervised by other employees.

This is true. A Unary relationship exists when an association is maintained in a single entity.

An entity's participation in a relationship can be optional.

This is true. Relationship participation can be mandatory or optional.

Relationships are always between two distinct entities.

This is false. In addition to Binary relationships, relationships can be Unary or Ternary (or higher degree).

How ERD Components Affect Database Design

ER symbols are used to graphically depict the ER model's components and relationships. Such graphic representations are the database equivalent of an architect's blueprint. Prior to building a database, an ER diagram (ERD) should be developed. This allows the database design to be carefully planned and defined prior to construction. ERDs consist of multiple components, including representations for entities, relationships, attributes, and relationship types.

Why is data modeling so important to the database designer?

We are said to live in the information age, and data constitute the most basic information units employed by an information system. Data modeling provides a way to reconcile the very different end-user views of the nature and roles of data.

A *data model* is an abstraction that provides an easily understood representation of complex real-world data structures. A data model helps us understand, communicate and document the complexities of a real-world data environment. Such understanding yields useful solutions to the problems inherent in creating, organizing, using, and managing data.

If a database is to be useful and flexible, it must be well designed. The database design process must be based on an appropriate data model if it is to yield a proper database design blueprint.

What role does the ER diagram play in the design process?

The ER diagram must reflect an organization's operations accurately if the database is to meet that organization's data requirements. The completed ER diagram forms the basis for design review processes that verify whether the included entities are appropriate and sufficient, whether the attributes found within those entities are needed and correct, and whether the relationships between those entities are needed and correctly represented. The ER diagram is also used as a final crosscheck against the proposed data dictionary entries. The ER diagram helps the database designer communicate more precisely with those who most completely understand the business data requirements. Finally, the completed ER diagram serves as the implementation guide to those who create the actual database. Many ERD software tools can generate the SQL statements to produce the tables represented in the ERD. In short, the ER diagrams are as important to the database designer as blueprints are to architects and builders.

Test Yourself 2.5

Regarding entity relationship modeling, which of the following is true? (Please check all of the following that are true.)

The ER diagram should be developed after the database is completed. This assures its accuracy in case somebody wants to duplicate the database.

This is false. The ER diagram should be developed prior to building a database. This allows the database design to be carefully planned and defined prior to construction.

Components of an ER diagram represent real-world objects and relationships.

This is true. An ER diagram is an abstraction that represents real-world data structures. An ER diagram helps us understand and document the complexities of a real-world data environment.

An ER diagram can aid in communication.

This is true. The ER diagram helps the database designer communicate more precisely with those who most completely understand the business data requirements.

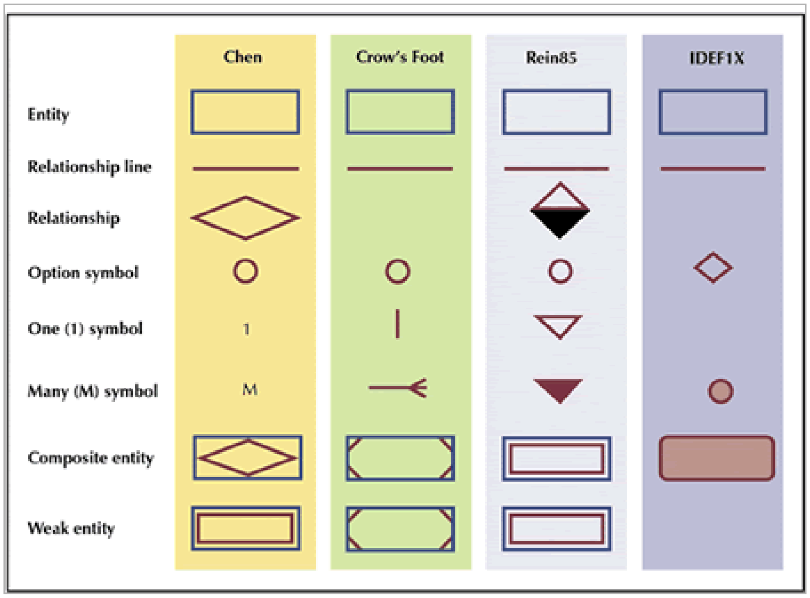
The ER diagram is of little importance in the design process of the database.

This is false. The ER diagram is of critical importance in the design process of the database. ER diagrams are as important to the database designer as blueprints are to architects and builders.

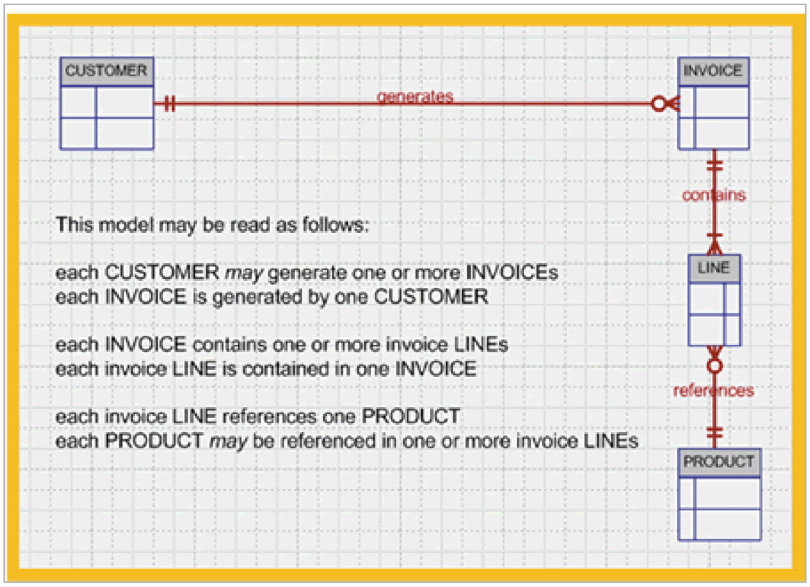
How to Interpret Modeling Symbols for ER Modeling Tools

There are a number of popular ER modeling tools and several different sets of ER modeling symbols. The four most popular symbol sets include the Chen model, the Crow's Foot model, the Rein85 model, and IDEF1X. All models use rectangles to represent entities connected by lines for relationships. A variety of conventions are used to represent relationships, cardinality, attributes, composite entities, and weak entities. The Chen model is based on Peter Chen's landmark work published in 1976. The Crow's foot model is characterized by using stick-like symbols to represent connectivity and cardinality information. It was developed by C.W. Bachman and was made popular by the Knowledgeware modeling tool. The Rein85 model, developed by D. Reiner in 1985, operates at a higher level of abstraction than the Crow's Foot model. IDEF1X was developed in conjunction with integrated computer-aided manufacturing studies. The original version of IDEF was created by Hughes Aircraft and was known as IDEF1. The extended version is IDEF1X.

The following diagram nicely compares various ER modeling symbols:



In this course however, we focus on CROW'S FOOT symbols. We focus on CROW'S FOOT because it is more compact than the Chen notation, and because variations on CROW'S FOOT and UML are supported by most database design tools, while Chen notation is not. Here is the CROW'S FOOT representation of an invoicing problem.



This model may be read as follows:

each CUSTOMER *may* generate one or more INVOICEs
each INVOICE is generated by one CUSTOMER

each INVOICE contains one or more invoice LINEs
each invoice LINE is contained in one INVOICE

each invoice LINE references one PRODUCT
each PRODUCT *may* be referenced in one or more invoice LINEs

Test Yourself 2.6

Which of the following statements regarding ER Models are correct? (Please check all of the following that are correct.)

Only the Chen model uses rectangles to represent entities.

This is false. The Chen model, the Crow's Foot model, the UML class model, the Rein85 model, and IDEF1X all use rectangles to represent entities.

Variations of the Crow's Foot model are supported by most database design tools.

This is true. Most database design tools also support UML. (Very few tools support the Chen model.)

The Crow's foot model uses stick-like symbols to represent connectivity and cardinality information.

This is true. The Crow's foot model represents relationships as lines with various symbols on the ends to represent cardinality, and text paralleling the lines describing the relationships.

Relationship Constraints in an ERD

There are two constraints for associative relationships between two or more entities, and these are explained in subsequent paragraphs. Before understanding each constraint, however, you need to understand *perspective*. If there are two entities participating in a relationship—EntityA and EntityB—one perspective is EntityA with respect to EntityB, and the other perspective is EntityB with respect to EntityA. If there are three entities participating in a relationship—EntityA, EntityB, and EntityC—there are six perspectives:

- EntityA with respect to EntityB

- EntityA with respect to EntityC
- EntityB with respect to EntityA
- EntityB with respect to EntityC
- EntityC with respect to EntityA
- EntityC with respect to EntityB

The reason why you need to understand perspective is because each relationship constraint is assigned once to every perspective in the relationship. For binary relationships, this means that each constraint is assigned twice. For ternary relationships, each constraint is assigned six times.

One of the constraints is the *plurality constraint* which specifies, from the perspective of one entity (say EntityA) with respect to another (say EntityB), whether each of EntityA's instances can be associated to *at most one* instance of EntityB, or can be associated to *more than one* instances of EntityB. We can say that the constraint is *singular* if each instance of EntityA can be associated to at most one instance of EntityB, and that the constraint is *plural* if each instance of EntityA can be associated to more than one instances of EntityB. In Crow's Foot notation, we indicate plurality with the Crow's foot symbol, and singularity with a solid bar. Please keep in mind that while database texts widely accept the concept of plurality and its representation in Crow's foot notation, there are no universally accepted terms used to describe this constraint and its two possible values. We use the terms mentioned in this paragraph in this course to avoid reexplaining the concept each time it is used.

The other constraint is the *optionality constraint* which specifies, from the perspective of one entity to another (say EntityA and EntityB), whether each of EntityA's instances *must* have a relationship with one or more of EntityB's instances, or can *optionally* have a relationship with one or more of EntityB's instances. If each of EntityA's instances must have a relationship, then the constraint is *mandatory*, and we use a single bar in Crow's Foot notation for representation. If each of EntityA's instances can optionally have a relationship, the constraint is *optional*, and we use the "O" symbol in Crow's Foot notation. Again, database texts widely accept the concept of optionality and its representation in Crow's Foot notation, but there are no universally accepted terms. We use the terms mentioned in this paragraph to avoid re-examining the concept each time it is used.

Advanced Topic 1

You may have discerned from the explanation in this slide that there are two distinct types of relationships—relationships whose participants are entities, and relationships whose participants are entity instances. Unfortunately database texts loosely use the term *relationship* to describe both types, but we must carefully distinguish the two in order to fully understand how relationships and their constraints work. Recall that an entity is a blueprint for its instances. If we define an entity with three attributes with particular names, value domains (datatypes), and constraints, then every instance of that entity will have only those attributes with all of the properties as defined in the entity. In the same way, relationships whose participants are entities are blueprints that define the properties of the relationships between the participating entity instances. One property is the existence of the relationship. Two other two properties are the optionality and plurality constraints described in this lecture slide.

Advanced Topic 2

This slide mentions that there are six perspectives for ternary relationships, whereas there are two for binary relationships. It is for this reason that the relationship line and symbols defined by Crow's Foot notation only work directly for binary relationships. To correctly diagram relationships with a degree higher than 2, we must *reify* the relationship by representing the relationship as an entity, then use the standard relationship line and symbols between that new entity and the original entities participating in the relationship.

Real World Database Design

Database designers must make compromises that trade off conflicting goals. Databases must be designed to conform to good design practice, just as buildings are designed to conform to building codes and good building practices. Such standards are in place to help minimize data redundancy, thereby reducing the likelihood of destructive data anomalies. In databases that must support with high transaction rates, such as busy Internet web sites, high-speed processing may require design compromises. In other organizations, timely information might be the focus of database design. In each case, optimizing for one area could adversely affect another. Other design concerns revolve around the end user. These might include security, performance, shared access, and integrity issues.

Business rules are an important element of database design in every organization. Business rules drive all business processes, and an organization's business rules must be correctly implemented by the organization's IT systems, including the databases.

Business rules are precise statements, derived from a detailed description of the organization's operations. **When written properly,** business rules define one or more of the following modeling components:

- entities
- relationships
- attributes
- connectivities
- cardinalities
- constraints

Because the business rules form the basis of the data-modeling process, precisely phrasing them is crucial to the success of the database design. Because the business rules are derived from a precise description of operations, much of the design's success depends on the accuracy of the description of operations.

Examples of business rules are:

- An invoice contains one or more invoice lines.
- An invoice line is associated with a single invoice.
- A store employs many employees.
- An employee is employed by only one store.
- A college has many departments.
- A department belongs to a single college. (This business rule reflects a university that has multiple colleges such as Business, Liberal Arts, Education, Engineering, etc.)
- A driver may be assigned to drive many different vehicles.
- A vehicle can be driven by many drivers. (Note: Keep in mind that this business rule reflects the assignment of drivers during some period of time.)
- A client may sign many contracts.
- A sales representative may write many sales contracts.
- A sales contract is written by one sales representative.

- A sale involves a sales representative, a customer, and one or more products.

Note that each relationship definition requires the definition of two business rules. For example, the relationship between the INVOICE and (invoice) LINE entities is defined by the first two business rules in the bulleted list. This two-way requirement exists because there is always a two-way relationship between any two related entities. (This two-way relationship description also reflects the implementation by many of the available CASE tools.) The last business rule above describes a three-way sale relationship between sales representatives, customers, and products.

ER diagrams can represent the business rules that are expressed only in terms of data, but they cannot always reflect all of the business rules. For example, the following business rules cannot be easily represented in an ERD, though they could be enforced in the database:

- A customer cannot be given a credit line over <0,000 unless that customer has maintained a satisfactory credit history (as determined by the credit manager) during the past two years.
- Credit lines over <00,000 must be approved by the executive vice president.

These business rules describe constraints that cannot be shown in the ER diagram. The business rules reflected in these constraints would be handled at the applications software level and could be enforced in the database through the use of a trigger or a stored procedure. (You will learn about triggers and stored procedures in the *Advanced SQL* module.)

Test Yourself 2.7

Which of the following statements regarding database design are correct? (Please check all of the following that are correct.)

In some situations, a database design that reduces data redundancy to the smallest amount possible can lead to less than optimal performance from the point of view of the end user.

This is true. Performance of the database and understandability of the SQL can both be compromised when the database is over nomormalized.

Business rules are important for the day to day operations of an organization, but are generally unimportant in the design of a database for an organization to use.

This is false. *Business rules* are an important consideration in database design in every organization.

Business rules can only define entities, their attributes, and relationships.

This is false. Business rules can define entities, relationships, attributes, cardinalities, connectivities, and constraints.

A proper ER diagram can easily show all business rules.

This is false. Some business rules cannot easily be shown in an ER diagram. These rules may be implemented or enforced in different ways.

Types of Entity Relationship Models

There are several types of entity relationship models, and each type is distinguished primarily by what the model depends upon. The model dependency for each type is standardized, and is described in the “Depends upon” column in the table below. The kinds of elements each type of model includes is not entirely standardized, though the limitations given in the “Included” and “Not Included” columns in the table below are common. It is most important to understand the model dependency for each type of model, as the kinds of elements supported logically follow. In this course, you will do well to follow the constraints given in the table below.

Model	Depends upon	Included	Not included	Comments
Conceptual	The problem domain only	Entities, relationships (including many to many), optionally attributes	Bridge entities, collection entities when the entities have no structure	Constraints may be represented in some notations when they are fundamental to the problem domain. Attributes are optional, though they can be included when the conceptual model is fully developed and when the attributes are critical to identifying what an entity represents.
Logical	The database model, which is commonly SQL relational, in addition to the problem domain	Entities, all attributes, relationships, bridge entities, collection entities, and primary and foreign keys	DBMS-specific data types	May optionally include standard SQL data types
DBMS Physical	The specific DBMS (e.g. Oracle 11g or MSSQL 2008), in addition to the database model and the problem domain	DBMS specific data types, entities, attributes, relationships, bridge entities, collection entities, primary and foreign keys, and general constraint implementation	For larger databases this will not include detailed storage configuration, transaction log allocation, detailed tablespace configuration, or instance parameter configuration.	This is the responsibility of the database designer.

When Attributes Should be Present in Conceptual Database Design Diagrams

When should attributes be present in conceptual database design diagrams? Some textbooks provide the simple answer that conceptual ERDs do not require attributes, or even that they should never have attributes. Unfortunately this is not helpful, and is not good practice. The answer is that whether a conceptual database design diagram should have attributes depends upon the intended use of the design diagram.

Let us start by viewing two extremes. One extreme occurs when the diagram is part of an enterprise object model. In this case it should include all attributes, together with a formal definition of the domain of each attribute. Enterprise object models can be very complex. The other extreme occurs when a conceptual diagram is used solely for communicating relationships within a development team who have a common terminology for the entities. In this case there is no risk of misidentifying entities, and no attributes are necessary on a conceptual diagram. If the conceptual diagram becomes too complex, then even the entities may be simplified by combining closely associated entities into abstract entity clusters.

If the goal of a conceptual ERD is advancing the development of an understandably simple database design, or an easily understood subset of a larger design, then it is usually desirable to include all of the attributes at the conceptual level. One of the reasons for including all of the attributes at the conceptual level is that it reduces the risk for confusing what the entities really represent. Another advantage is that you can apply object-oriented normalization tests, which apply at the conceptual level, when you have the attributes.

The guiding principle of what to specify in conceptual database design diagrams is what you need the model to communicate. Large database design diagrams can become mind-numbingly complex, so we often simplify, sometimes a lot, so that people can understand them. Many techniques can be used to help people understand database designs, including color coding sub-schemas and hiding information peripheral to the focus of the diagram. An ERD is a presentation of an underlying Entity-Relationship Model (ERM). A particular ERM may be represented with several different ERDs, each designed to meet particular communication goals. Conceptual ERMs and ERDs often grow in complexity and fidelity, beginning with few attributes, with attributes gradually added as the entities are better understood.

Test Yourself 2.8

Which of the following statements correctly describe conceptual entity-relationship diagrams (ERDs)?

Entities in conceptual ERDs should never contain attributes.

This is false. Conceptual ERDs sometimes have attributes present, depending upon the intended use of the diagram.

A conceptual database design diagram is primarily a communication tool.

This is true. The goal of a conceptual database diagram is to communicate one or more facets of an underlying entity-relationship model. We use many techniques in a conceptual ERD toward this end.

There is exactly one correct conceptual database diagram for an entity-relationship model.

This is false. A particular ERM may be represented with several different ERDs, each designed to meet particular communication goals.

A conceptual ERD is created once as a basis for database design, then left alone.

This is false. Conceptual ERDs grow as the entities, attributes, and problem domain are better understood.

Designing a Database for a Grocery Store

We now illustrate entity relationship modeling and database design by designing a simple database for a grocery store. Click on the following animation, which will walk you through the design process, beginning with identifying the entities, identifying the relationships, and the cardinalities between the relationships. This simple example includes only the most important entities and relationships; most retail databases have dozens to hundreds of tables.

Note

The conceptual diagram created in this video does indeed illustrate relationships without using the SQL-relational foreign keys, which is as expected, but the diagram includes primary key constraints. Though the concept of a unique identifier is standard across several types of models, it is not recommended to use SQL-relational primary keys in conceptual diagrams in order to avoid tying the conceptual diagram to the relational model.

metcs669_W03_schemaobject1_forflv video cannot be displayed here

Summary

Entity relationship modeling (ERM) uses entity relationship diagrams to help database designers and users understand and communicate about the data that the database must represent, how the data is to be organized, and the relationships between the data items. Entity-relationship modeling is used together with transaction analysis, performance analysis and other database requirements analysis to identify the requirements that the database must meet.

The ERM includes entities, their attributes, and relationships between entities. The ERM relationships can indicate whether the relationships are optional or mandatory, the degree of the relationship (how many entities it relates), the connectivity of the relationship (1:1, 1:M, M:N), and the cardinality of the relationship (how many of each entity are related on each end of the relationship).

ERMs can be at the conceptual, logical (internal) or physical level. Conceptual ERMs can include many-to-many relationships, which are represented in a relational database by including an *association* (or *bridge* or *composite*) entity that represents the relationship. While ERMs are based on business rules, not all business rules can be represented in their entirety in ERMs. Data constraints in business rules may be represented in the ERM, but most business processes are not represented in the ERM, but must be represented in application software or in stored procedures or triggers in the database.

The Unified Modeling Language (UML) is an extensible graphical language, and extensions have been developed for UML class diagrams that make them well suited for both database modeling and application class modeling. Database design often involves compromises between simplicity and modeling fidelity and between fidelity and performance. When designing databases it is critical to communicate well and frequently with end users and others who understand in depth what the database must do. ERD and UML class diagrams are critical languages in this communication.

Lecture 6 - Essential SQL Commands

SQL JOIN Operators

We first need to discuss SQL join operators, because these are essential to retrieving interconnected data.

Join operations are classified as *inner joins* and *outer joins*. Inner join is the traditional join in which only rows that meet a given criteria are selected. The join criterion can be an equality condition (natural join or equijoin) or an inequality condition (theta join). An outer join returns not only the matching rows but also the rows with unmatched attribute values for one or both of the joined tables. The SQL standard also introduces a special type of join, called a cross join, that returns the same result as the Cartesian product of two sets or tables. A *natural join* returns all rows with equal values in the columns with the same name. This style of query is used when the tables share one or more common attributes with common names. I do not recommend natural joins for production SQL, because changing column names can change the meaning of natural joins without generating error messages, and these hidden errors are difficult to diagnose.

The ANSI standard defines three types of outer joins: left, right, and full. The left and right designations reflect the order in which the tables appear in the SQL statement. Remember that join operations take place two tables at a time. The first table named in the FROM clause will be the left side, and the second table named will be the right side. If joining three or more tables, the result of joining the first two tables becomes the left side; the third table becomes the right side.

An OUTER JOIN is a type of JOIN operation that yields all rows with matching values in the join columns as well as all unmatched rows. (Unmatched rows are those without matching values in the join columns). The SQL standard prescribes three different types of join operations:

```
LEFT [OUTER] JOIN
RIGHT [OUTER] JOIN
FULL [OUTER] JOIN
```

A Note on Notation

The [square brackets] around OUTER mean that it is optional.

The LEFT [OUTER] JOIN will yield all rows with matching values in the join columns, plus all of the unmatched rows from the *left* table. (The left table is the *first* table named in the FROM clause.)

The RIGHT [OUTER] JOIN will yield all rows with matching values in the join columns, plus all of the unmatched rows from the *right* table. (The right table is the *second* table named in the FROM clause.)

The FULL [OUTER] JOIN will yield all rows with matching values in the join columns, plus all the unmatched rows from both tables named in the FROM clause.

Using tables named T1 and T2, we write a query example for each of the three join types described above. Assume that T1 and T2 share a common column named C1.

Left Outer Join Example

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1 = T2.C1;
```

Right Outer Join Example

```
SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.C1 = T2.C1;
```

Full Outer Join Example

```
SELECT * FROM T1 FULL OUTER JOIN T2 ON T1.C1 = T2.C1;
```

Test Yourself 2.9

Which of the following are true regarding SQL join operators? (Please check all of the following that are true.)

The outcome of a natural join of a Player table with a row that has a value of “SAM” for the column player_name and a Team table with a row that has a value of “SAM” for the column owner_name would include a row that has a value of “SAM” for the column NAME.

This is false. Natural joins link tables by selecting only rows with common values in attribute(s) that have the same column names. The above is not true because the column with the value “SAM” in the Player table is named player_name and the column in the Team table with the value “SAM” is named owner_name. If both columns had the same name, it would be true.

Changing a column name without changing any data could alter the results of a natural join between two tables.

This is true. The use of natural joins can cause problems with production databases, because changing column names can change the meaning of natural joins without generating error messages.

The results of “SELECT * FROM Team LEFT OUTER JOIN Player ON Team.team_id = Player.team_id;” would include unmatched rows from the Team table. (Assume that both tables exist and share a team_id column)

This is true. The LEFT OUTER JOIN will yield all rows with matching values in the join columns, plus all of the unmatched rows from the left table. (The left table is the first table named in the FROM clause.)

The results of “SELECT * FROM Team RIGHT OUTER JOIN PLAYER ON Team.team_id = Player.team_id;” would include unmatched rows from the Team table. (Assume that both tables exist and share a team_id column)

This is false. The RIGHT OUTER JOIN will yield all rows with matching values in the join columns, plus all of the unmatched rows from the right table. (The right table is the second table named in the FROM clause.) Unmatched rows from the Player table would be included.

The results of “SELECT * FROM Team FULL OUTER JOIN Player ON Team.team_id = Player.team_id;” would include unmatched rows from the Team table. (Assume that both tables exist and share a TEAM_ID column)

This is true. The FULL OUTER JOIN will yield all rows with matching values in the join columns, plus all of the unmatched rows from both tables named in the FROM clause. Unmatched rows from both tables would be included.

Joining Tables

Joins are the way that data in different relations is related. There are several different kinds of joins. Let us begin with Natural Joins, which are simplest. The process is illustrated below:

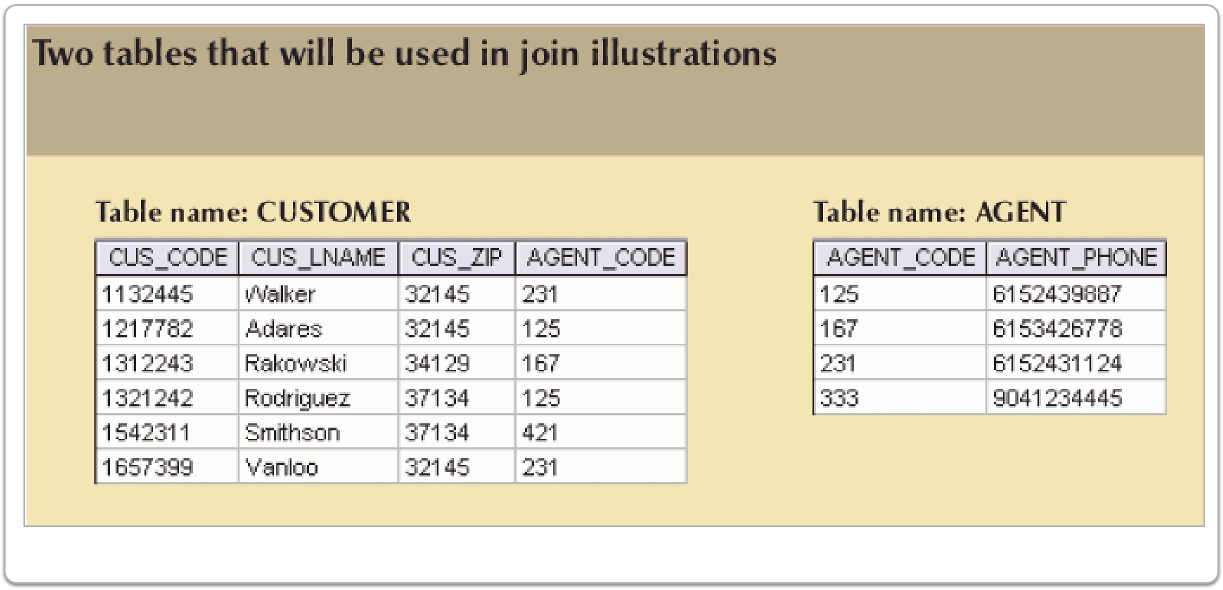
Natural Joins

Natural Joins link tables by selecting only rows with common values in attribute(s) that have the same column names. Note that the only column name that is common to the two tables below is AGENT_CODE.

The DBMS computes a natural join in a way that is functionally equivalent to a simple three-stage process:

1. The PRODUCT of the tables is created
2. SELECT is performed on Step 1 output to yield only the rows for which the AGENT_CODE values are equal. Common column(s) are called join column(s)
3. PROJECT is performed on the Step 2 results to yield only the columns desired.

Two Tables to be Joined



Two tables that will be used in join illustrations

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE
1132445	Walker	32145	231
1217782	Adares	32145	125
1312243	Rakowski	34129	167
1321242	Rodriguez	37134	125
1542311	Smithson	37134	421
1657399	Vanloo	32145	231

Table name: AGENT

AGENT_CODE	AGENT_PHONE
125	6152439887
167	6153426778
231	6152431124
333	9041234445

Step 1: Product

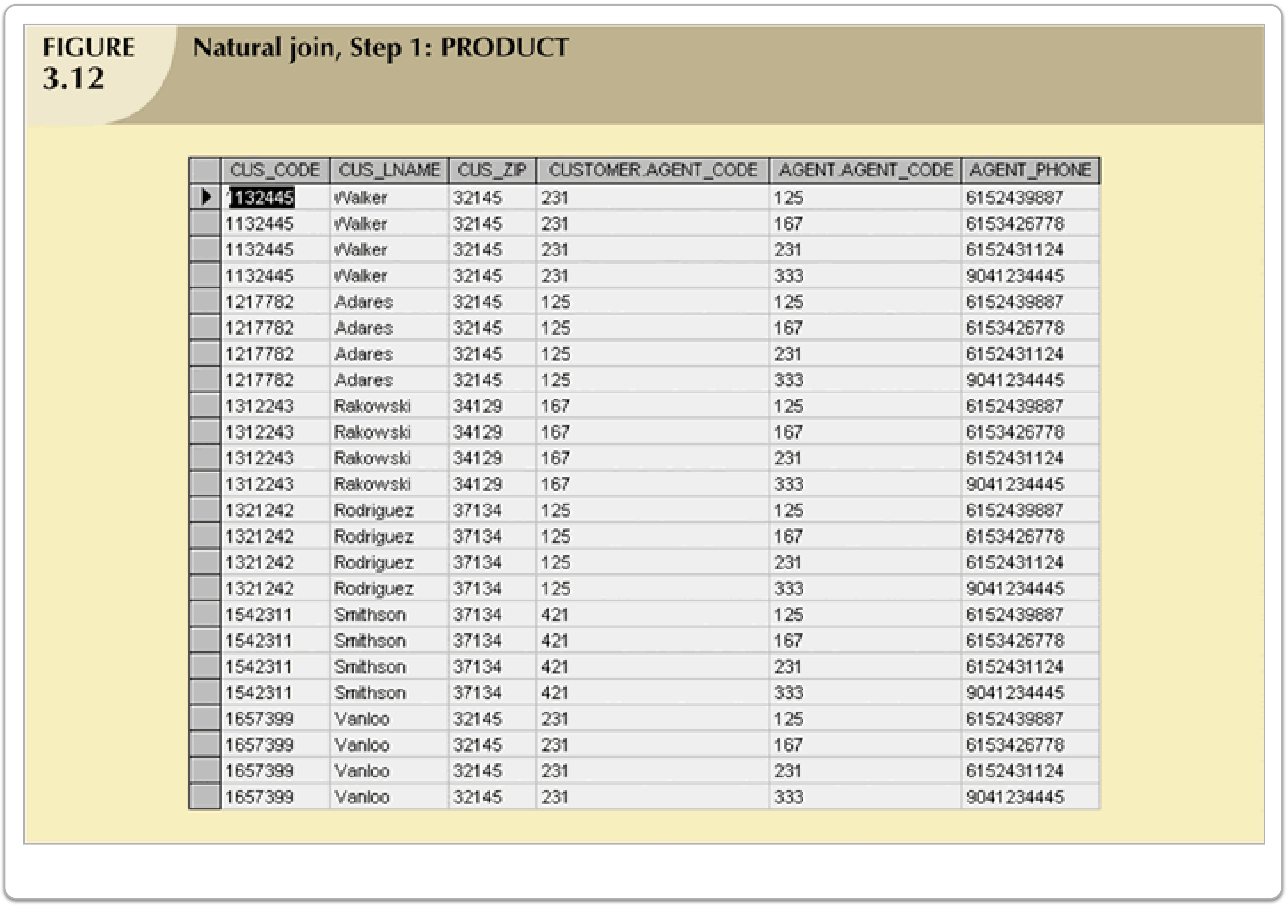


FIGURE 3.12 Natural join, Step 1: PRODUCT

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1132445	Walker	32145	231	125	6152439887
1132445	Walker	32145	231	167	6153426778
1132445	Walker	32145	231	231	6152431124
1132445	Walker	32145	231	333	9041234445
1217782	Adares	32145	125	125	6152439887
1217782	Adares	32145	125	167	6153426778
1217782	Adares	32145	125	231	6152431124
1217782	Adares	32145	125	333	9041234445
1312243	Rakowski	34129	167	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1312243	Rakowski	34129	167	231	6152431124
1312243	Rakowski	34129	167	333	9041234445
1321242	Rodriguez	37134	125	125	6152439887
1321242	Rodriguez	37134	125	167	6153426778
1321242	Rodriguez	37134	125	231	6152431124
1321242	Rodriguez	37134	125	333	9041234445
1542311	Smithson	37134	421	125	6152439887
1542311	Smithson	37134	421	167	6153426778
1542311	Smithson	37134	421	231	6152431124
1542311	Smithson	37134	421	333	9041234445
1657399	Vanloo	32145	231	125	6152439887
1657399	Vanloo	32145	231	167	6153426778
1657399	Vanloo	32145	231	231	6152431124
1657399	Vanloo	32145	231	333	9041234445

Step 2: Select

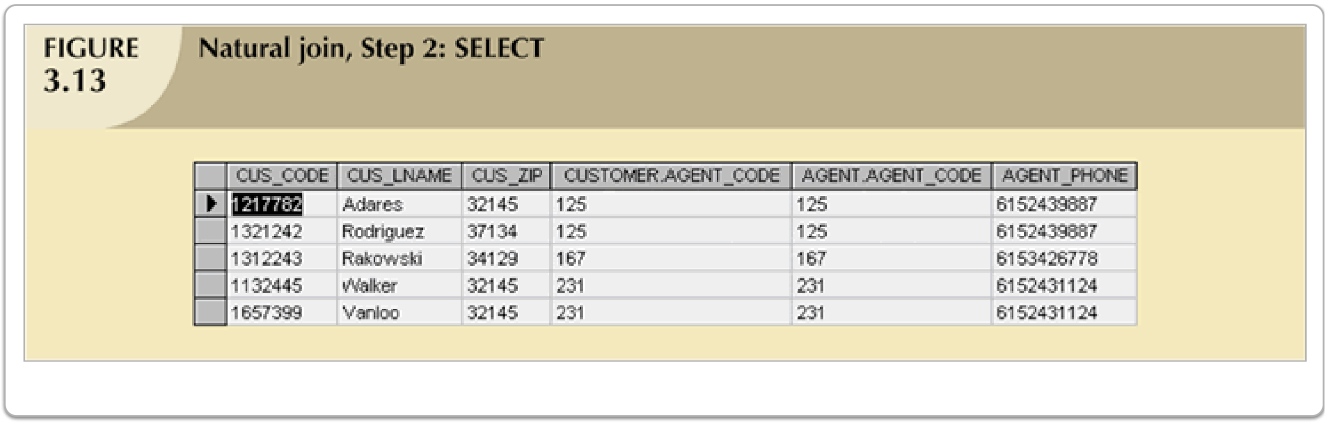


FIGURE 3.13 Natural join, Step 2: SELECT

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	125	6152439887
1321242	Rodriguez	37134	125	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1132445	Walker	32145	231	231	6152431124
1657399	Vanloo	32145	231	231	6152431124

Step 3: Project

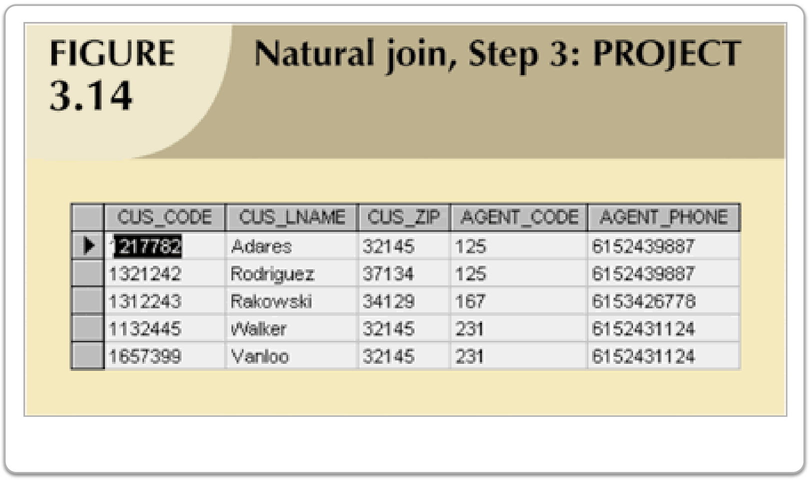


FIGURE 3.14 Natural join, Step 3: PROJECT

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124

Final Outcome:

- The above yields a table that:
    - Does not include unmatched pairs
    - Provides only copies of matches
- If no match is made between the table rows, the new table does not include the unmatched row.
- The column on which we made the JOIN-that is, AGENT_CODE-occurs only once in the new table.
- If the same AGENT_CODE were to occur several times in the AGENT table, a customer would be listed for each match.

Equijoins

In an equijoin we link tables on the basis of an equality condition that compares specified columns of each table. The outcome does not eliminate duplicate columns. The condition or criterion to join tables must be explicitly defined. Equijoin takes its name from the equality comparison operator (=) used in the condition.

Equijoins are the most common type of joins. The ANSI SQL for an equijoin of Customer and Product tables is:

```
SELECT *
FROM Customer JOIN Product
```

```
ON Customer.agent_code = Product.agent_code;
```

Most DBMS also support the “old style” syntax for an equijoin, which is:

```
SELECT *
FROM Customer, Product
WHERE Customer.agent_code = Product.agent_code;
```

The result of the equijoin is similar to that of the natural join, except that the duplicate columns are not eliminated.

Theta Joins

A *theta join* is a join which is based on any comparison operator other than equal. Common theta join conditions include inequalities (e.g., less than or greater than) and ranges (*BETWEEN* in SQL).

The ANSI standard SQL for theta joins is similar to that for equijoins, except that the equality condition is replaced by an inequality or range condition. For example, joining the Customer, and Agent tables where we want the Customers who have agent_codes less than the corresponding Agent agent_codes is:

```
SELECT *
FROM Customer JOIN Agent
ON Customer.agent_code < Agent.agent_code;
```

Unlike equijoins, which are obviously useful, it is not clear why a business would want this theta join. Theta joins are much less common than equijoins.

Outer Joins

Finally let us introduce the concept of Outer Join. The joins that we have discussed so far include in the result set only the rows that satisfy the restriction, whether it be an equijoin restriction or a theta join restriction. Sometimes we want data in the result set from one or more of the joined tables, even though that data does not have matching rows in the other relations. For example, we may want data on employees and their dependents, and we want to include data in the result set on employees who do not have dependents. In outer joins rows that satisfy the restrictions are present in the result set and any rows from one or both of the outer joined tables that do not satisfy the restrictions are present in the result set, with the missing values from the other table present as nulls, which indicate the absence of data.

- In outer join for tables CUSTOMER and AGENT, two scenarios are possible, depending on whether we want all of the rows from CUSTOMER
    - Left outer join
        - Yields all rows in CUSTOMER table, including those that do not have a matching value in the AGENT table
        - You would use this left outer join if you wanted all CUSTOMERs, including those who for some reason do not currently have AGENTs.
        - The AGENT attributes in the result set of this outer join would be null for CUSTOMERs who do not have AGENTs.
        - If an AGENT has multiple CUSTOMERs there will be a row in the result set for each of their CUSTOMERs, with both the projected AGENT and CUSTOMER data, just as if this were an inner join.

Figure 3.15 from the text illustrates the result of a left outer join of the Customer and Agent tables illustrated in Figure 3.11.

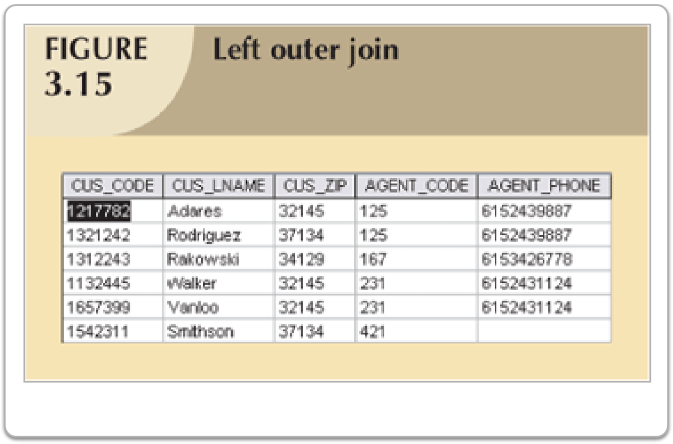


FIGURE 3.15 Left outer join

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124
1542311	Smithson	37134	421	

Note that this result set includes all six rows of the Customer table, even the row for Smithson, who doesn't have an agent in the Agent table.

The ANSI SQL for this join is:

```
SELECT customer.cust_code, Customer.cus_lname,
Customer.cus_zip,Customer.agent_code,Agent.agent_phone
FROM Customer LEFT OUTER JOIN Agent
ON Customer.agent_code = Agent.agent_code;
```

- Right outer join
- Yields all rows in AGENT table, including those that do not have matching values in the CUSTOMER table.
- You would use this right outer join if you wanted data on all AGENTs, even those who do not currently have CUSTOMERs.
- The CUSTOMER attributes in the result set of this outer join would be null for AGENTs who do not have CUSTOMERs.
- If an AGENT has multiple CUSTOMERs there will be a row in the result set for each of their CUSTOMERs, with both agent and customer data, just as if this were an inner join.

The right outer join is illustrated in Figure 3.16 from the text:

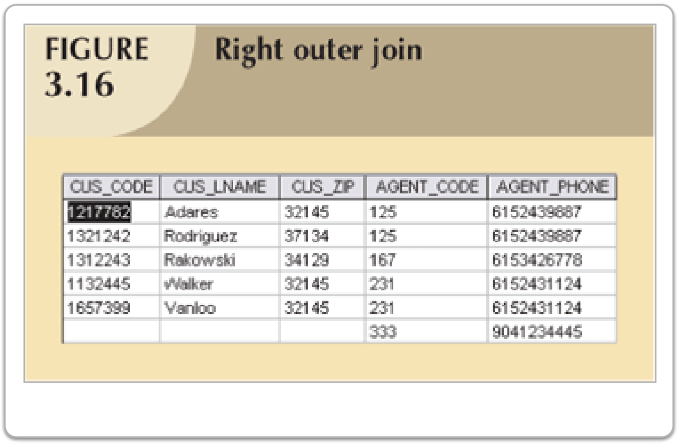


FIGURE 3.16 Right outer join

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124
			333	9041234445

Note that the result set includes the five rows where there are agents for customers, plus the row for agent 333, who does not have any customers. The ANSI SQL for this is:

```
SELECT customer.cust_code, Customer.cus_lname,
Customer.cus_zip,Agent.agent_code,Agent.agent_phone
```

```
FROM Customer RIGHT OUTER JOIN Agent
ON Customer.agent_code = Agent.agent_code;
```

Different DBMS vendors developed different SQL syntax for outer joins before the SQL standards committees developed the outer join SQL standards, so different DBMS support different ways of expressing outer joins in SQL, and you will encounter these syntactic variations in legacy SQL code. The standards-based SQL outer join syntax above is now supported in almost all DBMS, and it should be used for new SQL. We will also cover outer joins and the SQL outer join syntax in Lecture 8 and Chapter 8 of your Coronel & Morris text.

Test Yourself 2.10

Which of the following is true regarding joining relational tables? (Please select all of the following that are true.)

The outcome of a natural join of a Pet table with an entity occurrence that has a value of ‘Polly’ for the column pet_name and an Owner table with an entity occurrence that has a value of ‘Polly’ for the column owner_name would, by definition, have to include an entity occurrence that has a value of ‘Polly’ for the column name.

This is false. Natural joins link tables by equijoining all rows that have the same column names. The above is not true because the column with the value “Polly” in the Pet table is named pet_name and the column in the Owner table with the value “Polly” is named owner_name. If both columns had the same name, it would be true.

The outcome of a natural join of a PET table **without** an entity occurrence that has a value of “4” for the column owner_id and an Owner table **with** an entity occurrence that has a value of “4” for the column owner_id would, by definition, have to include an entity occurrence that has a value of “4” for the column owner_id.

This is false. The outcome of a natural join does not include unmatched pairs. In the above, the row in the OWNER table with the owner_id value of “4” does not have a matching row in the Pet table with an Owner_id value of “4”.

A table Owner has a row that has a value of “4” for the column owner_id. A table PET does **not** have any rows that have a value of “4” for the column owner_id. If a left outer join were performed for these tables (Owner LEFT OUTER JOIN Pet), the outcome of this join would include a row that has a value of “4” for the column owner_id.

This is true. The outcome of an outer join includes rows from one table (at least) that do not have matching rows in the other table. In this situation, the outcome of the left outer join (Owner LEFT OUTER JOIN Pet) would include the unmatched row from the Owner table (the table on the **left** of the ANSI SQL “LEFT OUTER JOIN”) with a value of “4” for the column owner_id.

In an equijoin, tables are linked based on an equality condition.

This is true. Equijoins are by the most common, and they are based on equal values in the joined tables. The other kind of join is a *theta join*, where the comparison is based on an inequality or range.

Database Constraints

The world we live in is full of conditions that are always enforced. For example:

- biological parents are always born before their children.
- a credit card must not be expired when it is used.
- the amount paid at checkout from a grocery store equals the amount due.
- the price that we pay for gasoline is the price per gallon times the number of gallons.

Some conditions are enforced by systems of the natural world, such as the fact that biological parents must be born before their children. Some conditions are enforced by systems we create, such as the condition that a credit card cannot be used if it is expired. Regardless of creator, such a system maintains consistency by enforcing conditions on its constituent parts. At a more primitive level, in order to maintain consistent data in a relational database, we need a way to ensure that data always meets certain conditions. We define these conditions in database constraints.

A *constraint* in a relational database defines a condition, also known as an invariant, on a specific set of data. When the constraint is active, the invariant it defines is continually enforced by the RDBMS; any and all transactions are prevented by the relational database from violating that condition. The condition must be met at the time the constraint is defined, and for as long as the constraint exists and is active. Each constraint is defined as an element of a specific table, and the constraint defines a condition for one or more columns in that table. When a table is dropped, all of its constraints are also dropped. Judicious use of constraints is the primary method of ensuring data consistency in a relational database schema.

There are five primary kinds of constraints used in relational databases—*Not Null*, *Unique*, *Primary Key, Foreign Key,* and *Check*. The four most used constraints are detailed in this lecture.

Creating Foreign Keys Using SQL

We will now walk through an example of how to create foreign keys using SQL. This example follows section 3.2 in our RobCor text, so if you haven't read this section it would be good to read it now.

We begin with Figure 3.2 from the text, which illustrates a foreign key from a Product table to a Vendor table:

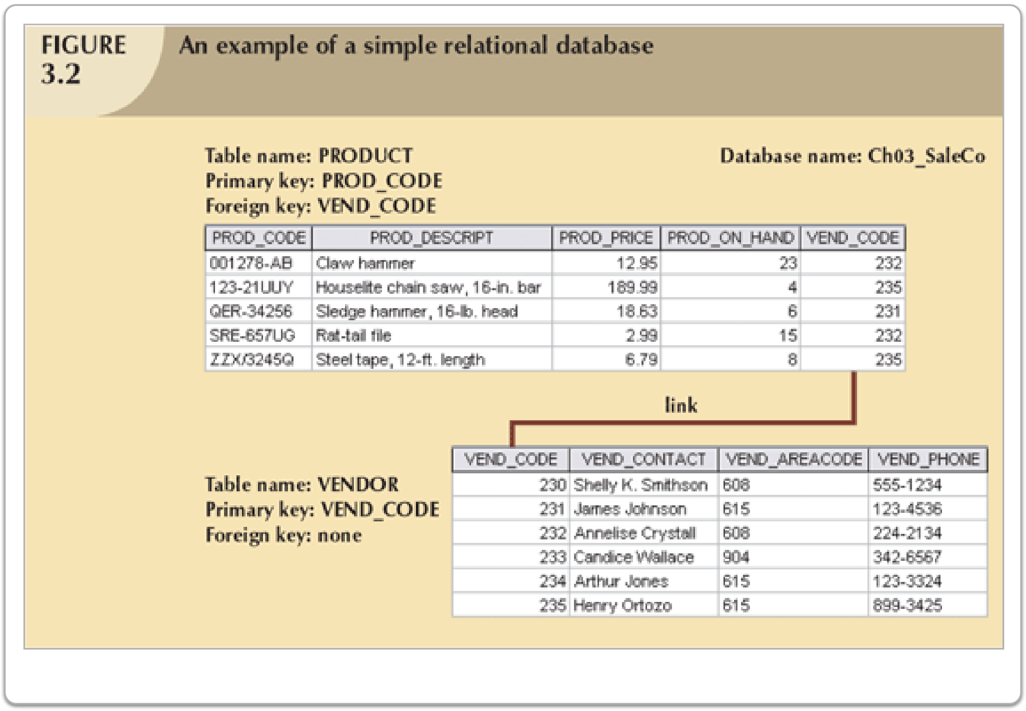


FIGURE 3.2 An example of a simple relational database

Table name: PRODUCT
Primary key: PROD_CODE
Foreign key: VEND_CODE

Database name: Ch03_SaleCo

PROD_CODE	PROD_DESCRIPT	PROD_PRICE	PROD_ON_HAND	VEND_CODE
001278-AB	Claw hammer	12.95	23	232
123-21UUY	Houselite chain saw, 16-in. bar	189.99	4	235
QER-34256	Sledge hammer, 16-lb. head	18.63	6	231
SRE-657UG	Rat-tail file	2.99	15	232
ZZX/3245Q	Steel tape, 12-ft. length	6.79	8	235

Table name: VENDOR
Primary key: VEND_CODE
Foreign key: none

VEND_CODE	VEND_CONTACT	VEND_AREACODE	VEND_PHONE
230	Shelly K. Smithson	608	555-1234
231	James Johnson	615	123-4536
232	Annelise Crystall	608	224-2134
233	Candice Wallace	904	342-6567
234	Arthur Jones	615	123-3324
235	Henry Ortozo	615	899-3425

The Following SQL creates these two tables with a foreign key from the vend_code column of the Product table to the vend_code primary key column of the Vendor table:

```
CREATE TABLE Vendor(
vend_code DECIMAL(12) PRIMARY KEY,
vend_contact VARCHAR(25),
vend_areacode CHAR(3),
vend_phone CHAR(8)
);

CREATE TABLE Product(
prod_code VARCHAR(10),
prod_descript VARCHAR(35),
prod_price DECIMAL(9,2),
prod_on_hand DECIMAL(12),
vend_code DECIMAL(12),
CONSTRAINT Product_Vendor_FK FOREIGN KEY(vend_code)
REFERENCES Vendor(vend_code)
);
```

The following table lists interesting lines from these SQL statements, and an explanation of each.

CREATE TABLE line	What it means
Prod_price NUMBER(9,2)	This specifies that the prod_price attribute is a number consisting of up to nine decimal places, with two places to the right of the decimal point. This is the same as DECIMAL(9,2), which is also ANSI standard and supported by Oracle.

CONSTRAINT Product_Vendor_FK FOREIGN KEY(vend_code) REFERENCES Vendor(vend_code)	This creates the foreign key constraint that Product.vend_code references Vendor.vend_code, and gives the constraint the name “Product_Vendor_FK.”

The SQL for Microsoft SQL Server is the same, except for a couple of data type differences:

```
CREATE TABLE Product(
prod_code VARCHAR(10),
prod_descript VARCHAR(35),
prod_price NUMERIC,
prod_on_hand INT,
vend_code INT
);

CREATE TABLE Vendor(
vend_code INT,
vend_contact VARCHAR(25),
vend_areacode CHAR(3),
vend_phone CHAR(8)
);
```

Note the changes: the prod_price data type is “numeric,” the prod_on_hand is of type “int,” and vend_code is of type “int.”

Test Yourself 2.11

Please select all of the following that are correct.

A table is created with a command containing the SQL statement CONSTRAINT DONATION_WHO_FK FOREIGN KEY (DONOR_ID) REFERENCES DONOR (DONOR_ID). DONATION_WHO_FK is the name of the column that is the foreign key.

This is false. DONATION_WHO_FK is the name of the constraint.

A table is created with a command containing the line
CONSTRAINT DONATION_WHO_FK FOREIGN KEY (DONOR_ID) REFERENCES Donor (donor_id). This SQL statement is contained in the command used to create the table DONOR.

This is false. DONOR is the table referenced by the foreign key. This statement could be part of a command used to create a table named DONATION or something similar.

A table is created with a command containing the line
CONSTRAINT DONATION_WHO_FK FOREIGN KEY (DONOR_ID) REFERENCES DONOR (DONOR_ID).
The name of the column that is the foreign key is DONOR_ID.

This is true. DONOR_ID is also the name of the column in the DONOR table that the foreign key references.

An unaltered table is created with a command containing the line CONTR_AMT DECIMAL(3,2) NOT NULL. The table could have a value of 250.75 for the CONTR_AMT column of a row contained in the table.

This is false. DECIMAL(3,2) limits the value to a number up to three decimal places with two of these to the right of the decimal point. (Leaving only one digit to the left of the decimal point)

An unaltered table is created with a command containing the line contr_amt NUMBER(4,2) NOT NULL. The table could have a value of 48.25 for the CONTR_AMT column of a row contained in the table.

This is true. DECIMAL(4,2) limits the value to a number up to four decimal places with two of these to the right of the decimal point. (Leaving two digits to the left of the decimal point)

Not Null

A *Not Null* constraint defines the condition that a specific column in a table must have a value for every row in that table. When the constraint is active, no row in that table may have a null value for that column. Consider, for example, a last_name column in a Person table. Every person entered into most application systems are required to have a value for their last name, so it is usually appropriate to define a *Not Null* constraint on the last_name column. Unlike other kinds of constraints, *Not Null* constraints are always limited to a single column. *Not Null* constraints help ensure that certain data is always present.

Using a Not Null Constraint in SQL

We can define a *Not Null* constraint in a CREATE TABLE statement as follows:

```
CREATE TABLE Person(
first_name VARCHAR(64),
last_name VARCHAR(64) NOT NULL
);
```

Notice the words NOT NULL after the last_name column definition. These words indicate to the RDBMS to create an immediately active *Not Null* constraint on the last_name column.

Ways to Violate a Not Null Constraint in SQL

A *Not Null* constraint can be violated two ways in SQL. For a column covered by a *Not Null* constraint, the following will violate the constraint:

1. an insert statement attempting to insert a null value into that column.
2. an update statement attempting to change that column's value to null.

Unique

A *Unique* constraint defines a condition that the value for a column or group of columns, if present, must be unique for every row in the table containing the column or group of columns. When a *Unique* constraint is active, the value for the column or group of columns in each row cannot be present in any other row. For example, a social security number must be unique for every person at any given point in time.

A *Unique* constraint does **not** ensure that the column or group of columns have values for every row. It is **not** a violation of a *Unique* constraint if no value is present, because a null value in one row is not considered equal to another value in another row. Continuing our example, a person may not have a social security number if they are in the beginning stages of becoming a legal resident in the United States. Thus, if a person has been assigned a social security number, it must be unique from the social security number from any other person. If a person has not yet been assigned a social security number, they may still be present in the system and the *Unique* constraint has not been violated.

Defining a Unique Constraint in SQL

We can define a *Unique* constraint in a Person table as follows:

```
CREATE TABLE Person(
first_name VARCHAR(64),
last_name VARCHAR(64) NOT NULL,
social_security_number DECIMAL(10) UNIQUE
);
```

Notice the word UNIQUE included in the social_security_number column definition. These words indicate to the RDBMS to create an immediately active *Unique* constraint on the social_security_number column.

Ways to Violate a Unique Constraint in SQL

A Unique constraint can be violated two ways in SQL. For a column or group of columns covered by a *Unique* constraint, the following will violate the constraint:

1. an insert statement attempting to insert a value into that column or group of columns that is already present in another row for that column or group of columns
2. an update statement attempting to change the value for that column or group of columns to a value already present in another row for that column or group of columns

Primary Key

A *Primary Key* constraint defines the condition a column or group of columns in a table have values that are both unique and not null. A primary key constraint is therefore a combination of the *Not Null* and *Unique* constraints. Row identity is generally considered to be an additional semantic property of the column or columns covered by a *Primary Key* constraint. In other words, we generally use the primary key columns to uniquely identify every row. For example, an ISBN is a unique identifier for books you are able to purchase. No book for sale will ever have the same ISBN, and you may use an ISBN to uniquely identify a book.

Defining a Primary Key Constraint in SQL

We can define a *Primary Key* constraint in a Book table as follows:

```
CREATE TABLE Book(
isbn DECIMAL(13) PRIMARY KEY,
title VARCHAR(1024)
);
```

Notice the words PRIMARY KEY included in the isbn column definition. These words indicate to the RDBMS to create an immediately active *Primary Key* constraint on the isbn column.

Ways to Violate a Primary Key Constraint in SQL

A *Primary Key* constraint can be violated in four ways in SQL. Not surprisingly, these are the combination of ways to violate the *Not Null* and *Unique* constraints. For a column or group of columns covered by a *Primary Key* constraint, the following will violate the constraint:

1. an insert statement attempting to insert a null value into any column covered by the constraint
2. an update statement attempting to change the value of any column covered by the constraint to null
3. an insert statement attempting to insert a value into the primary key column or columns that is already present in another row for the primary key column or columns
4. an update statement attempting to change the value for the primary key column or columns to a value already present in another row for that column of group of columns

Foreign Key

A *Foreign Key* constraint defines the condition that the value for a column or group of columns, if present, must also be present in another column or group of columns. The columns or group of columns covered by the constraint are said to *reference* the other column or group of columns.

Foreign key columns may only reference columns covered by a *Unique* or *Primary Key* constraint. For example, a Book_purchased table may contain an isbn column that identifies the book that was purchased by referencing the isbon column in a Book table. A *Foreign Key* constraint may be placed on that isbn column to ensure that it also exists in the Book table. By doing so, the Book_purchased table will only contain purchases to valid Books.

A *Foreign Key* constraint does **not** ensure that the column or group of columns have values for every row. It is **not** a violation of a *Foreign Key* constraint if no value is present, because a null value in one row is not considered equal to another value in another row.

Defining a Foreign Key Constraint in SQL

We can define a *Foreign Key* constraint for a Book_purchased table as follows:

```
CREATE TABLE Book(
isbn DECIMAL(13) PRIMARY KEY,
title VARCHAR(64)
);

CREATE TABLE Book_purchased(
isbn DECIMAL(13) FOREIGN KEY REFERENCES Book(isbn),
date_purchased DATE
);
```

Notice the phrase starting with FOREIGN KEY included in the isbn column definition in Book_purchased. These words indicate to the RDBMS to create an immediately active *Foreign Key* constraint on the isbn column. The REFERENCES keyword, coupled with the Book table name and isbn column name, indicates to RDBMS that the isbn column in Book_purchased table references the isbn column in the Book table.

Ways to Violate a Foreign Key Constraint in SQL

A Foreign Key constraint can be violated in six ways in SQL. The ways they can be violated are more complex than the aforementioned constraints, because operations on both the foreign key columns, and the referenced columns, can violate the constraint. The ways listed below always come back to the fundamental foreign key condition: *the value for a column or group of columns covered by a Foreign Key constraint, if present, must also be present in the referenced column or group of columns.*

1. an insert statement attempting to insert a value into the foreign key column or columns that is not present in the referenced columns.
2. an update statement attempting to change the value of the foreign key column or columns to a value that is not present in the referenced columns.
3. an update statement attempting to change the value of the referenced column or columns when that value is present in the foreign key column or columns.
4. an update statement attempting to change the value of one or more of the referenced columns to null, when those values are present in the foreign key column or columns.
5. a delete statement attempting to delete a row that is referenced by the foreign key columns.
6. a drop table statement attempting to drop a table that has at least one row that is referenced by the foreign key columns.

A More Complex Example of Table Creation

We will now walk through a more complex example of table creation by creating an involved student table. This example follows section 3.1 in our RobCor text, so if you haven't read this section it would be good to read it now.

We begin with Figure 3.1 from the text:

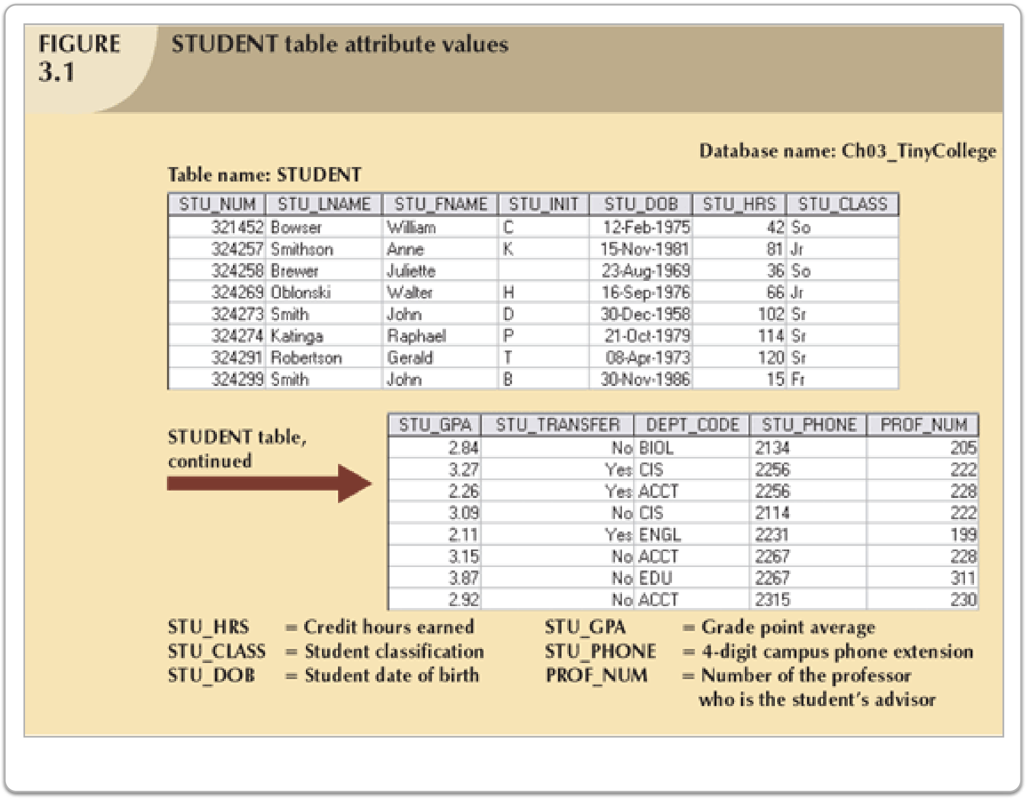


FIGURE 3.1 STUDENT table attribute values

Database name: Ch03_TinyCollege

Table name: STUDENT

STU_NUM	STU_LNAME	STU_FNAME	STU_INIT	STU_DOB	STU_HRS	STU_CLASS
321452	Bowser	William	C	12-Feb-1975	42	So
324257	Smithson	Anne	K	15-Nov-1981	81	Jr
324258	Brewer	Juliette		23-Aug-1969	36	So
324269	Oblonski	Walter	H	16-Sep-1976	66	Jr
324273	Smith	John	D	30-Dec-1958	102	Sr
324274	Katinga	Raphael	P	21-Oct-1979	114	Sr
324291	Robertson	Gerald	T	08-Apr-1973	120	Sr
324299	Smith	John	B	30-Nov-1986	15	Fr

STUDENT table, continued

STU_GPA	STU_TRANSFER	DEPT_CODE	STU_PHONE	PROF_NUM
2.84	No	BIOL	2134	205
3.27	Yes	CIS	2256	222
2.26	Yes	ACCT	2256	228
3.09	No	CIS	2114	222
2.11	Yes	ENGL	2231	199
3.15	No	ACCT	2267	228
3.87	No	EDU	2267	311
2.92	No	ACCT	2315	230

STU_HRS	= Credit hours earned	STU_GPA	= Grade point average
STU_CLASS	= Student classification	STU_PHONE	= 4-digit campus phone extension
STU_DOB	= Student date of birth	PROF_NUM	= Number of the professor who is the student's advisor

The SQL to create this table in Oracle is:

```
CREATE TABLE Student (
stu_num INTEGER(7) PRIMARY KEY,
stu_lname VARCHAR(15) NOT NULL,
stu_fname VARCHAR(15) NULL,
stu_init VARCHAR2(1),
stu_dob DATE,
stu_hrs INTEGER(4),
stu_class CHAR(2) NOT NULL,
CHECK(stu_class IN('Fr','So','Jr','Sr')),
stu_GPA FLOAT(8),
stu_transfer VARCHAR(3),
CHECK(stu_transfer IN('Yes','No')),
dept_code VARCHAR(18),
stu_phone CHAR(4),
prof_num INTEGER(4)
);

);
```

The following table describes what each line of this CREATE TABLE statement means.

CREATE TABLE Student	The words “CREATE” identifies this as a SQL statement that specifies an object to be created in the database. The word “TABLE” says that we are creating a table. The word “Student” says that the name of this table is Student.” The table name is not case sensitive. The left parentheses begins the list of specifications of the columns and other components of the table.
stu_num INTEGER(7) PRIMARY KEY,	The string “stu_num” is an identifier. It is treated like a single word. “Stu” is short for “student” and “num” is short for “number.” The string “INTEGER(7)” indicates that the student number is an integer of at most seven decimal digits. The words “PRIMARY KEY” assert that the stu_num must be present (NOT NULL) and unique for each row. In other words, stu_num uniquely identifies all students.
stu_lname VARCHAR(15) NOT NULL,	The identifier “stu_lname” is short for “student last name.” The data in this column is the students' last names. The string VARCHAR(15) specifies that the data in this column consists of variable length character strings of up to fifteen characters. The string “NOT NULL” specifies that there must be a value for stu_lname for all students. This is an instruction to the DBMS to not accept records (rows) in this table that do not have a value for stu_lname.
stu_fname VARCHAR2(15) NULL,	The identifier “stu_fname” is short for “student first name.” The data type “VARCHAR(15)” specifies that stu_fname is a variable length character string of no more than fifteen characters. The word “NULL” says that there does not have to be a value for stu_fname in records. NULL is the default, so we have not included it in the remaining column specifications.

stu_init CHAR(1),	The identifier “stu_init” is short for “student middle initial.” The string “CHAR(1)” specifies that a stu_init is a fixed length character string of one character.
stu_DOB DATE,	The identifier “stu_DOB” is short for “student date of birth.” The string “DATE” specifies that the stu_DOB is of data type DATE. An Oracle DATE data type includes both date and time with a resolution of one second. Dates and times are fundamentally intertwined, so databases include time in the DATE data type.
stu_hrs INTEGER(4),	The identifier “stu_hrs” is short for “student hours.” It represents the total number of credit hours that the student has completed. The string “INTEGER(4)” specifies that data in the stu_hrs column is an integer of up to four decimal digits.
stu_class CHAR(2),	The identifier “stu class” is short for “student classification.” The only legal values are 'Fr', 'So', 'Jr' and 'Sr'. [single quotes are used here to indicate that these are SQL character strings.]
CHECK(stu_class IN('Fr','So','Jr','Sr'))	This specifies a check constraint. It is an instruction to the DBMS to not allow any values in the stu_class column other than 'Fr', 'So', 'Jr', or 'Sr'.
stu_GPA FLOAT(8),	The identifier “stu_GPA” is short for “student grade point average.” The string “FLOAT(8)” specifies that stu_GPA is a floating point number of up to eight decimal digits.
CHECK(stu_GPA >= 0)	This is a check constraint that instructs the DBMS to not allow negative stu_GPA values. What is specifies is that stu_GPA must be greater than or equal to (“>=”) zero.
stu_transfer VARCHAR(3),	The identifier “stu_tranfer” is short for “student transfer,” meaning that the student is a transfer student.
CHECK(stu_tranfer IN('Yes','No'))	This check constraint specifies that stu_transfer must be either 'Yes' or 'No'; stu_transfer can also be NULL.
dept_code VARCHAR(4),	The identifier “dept_code” is short for “department code.” The string “VARCHAR(4)” specifies that dept_code must be a character string of up to four characters. Note that there is no CHECK constraint for dept_code, because new dept_codes may be added in the future, so this constraint doesn't belong in the database.
stu_phone CHAR(4),	The identifier “stu_phone” is short for “student phone” and it represents the students' four-digit campus phone extensions. The string “VARCHAR(4)” specifies that stu_phone is a four character string.
prof_num INTEGER(4)	The identifier “prof_num” is short for “professor number” and it refers to the student's academic advisor. The string “INTEGER(4)” specifies that prof_num is an integer of up to four decimal digits.

);	This right parenthesis ends the list of column and other specifications that define the Student table. The semicolon ends the SQL statement in the same way that a period ends a sentence.

This example is realistic as far as it goes. A real production Student table would also include foreign keys to Department and Professor tables. We are putting that off until we have introduced foreign keys.

The CREATE TABLE statement for Microsoft SQL Server is similar, with a few changes to the data types. It is:

```
CREATE TABLE Student(
stu_num INT,
stu_lname VARCHAR(15),
stu_fname VARCHAR(15),
stu_init CHAR(1),
stu_DOB DATETIME,
stu_HRS INT,
stu_class char(2) NOT NULL,
stu_GPA FLOAT(8),
stu_transfer NUMERIC,
dept_code VARCHAR(18),
stu_phone CHAR(4),
prof_num INT
);
```

We have omitted the CHECK constraints for simplicity. Note that the Oracle/ANSI DATE data type has become DATETIME, and that Oracle/ANSI INTEGER has become either NUMERIC or INT. These changes reflect nonstandard data type names in Microsoft SQL Server, and the diverse legacy data types in Microsoft SQL Server.

Test Yourself 2.12

Which of the following is true regarding creating a table using SQL? (Please select all of the following that are true.)

If you created a table with a command that began “CREATE TABLE Course” and did not alter the table, then the table would be named Course.

This is true. The given statement would specify that a new table is created that is named COURSE.

If you created a table with a statement that contained the line
“COURSE_ID INTEGER(4) PRIMARY KEY”,
then you could insert a row into the table with no value for the course_id column.

This is false. The PRIMARY KEY constraint will not allow nulls for the column.

If you created a table with a command containing the line
course_name VARCHAR(10),
could the course_name have a value of “INTERMEDIATE BASKET WEAVING”.

This is false. VARCHAR(10) specifies that the column can store variable length character strings of no more than 10 characters.

An unaltered table created with a command containing the SQL statement COURSE_TYPE VARCHAR(10), could have a value of “LAB” for the COURSE_TYPE column of a row contained in the table.

This is true. The value is equal to or less than 10 characters.

An unaltered table created with a command containing the SQL statement CHECK(COURSE_CRED >= 1), could have a value of 3 for the COURSE_CRED column of a row contained in the table.

This is true. This is a check constraint instructing the DBMS to not allow COURSE_CRED values to be zero or negative. (COURSE_CRED values must be greater than or equal to one.)

The Library Database

We will now begin using the Library Database described in this and subsequent lecture pages as a case study throughout this lecture. The Library Database consists of three tables used in a library environment. The three tables are BOOK, LOAN, and PATRON. You are encouraged to create the database and practice the SQL queries in the subsequent lecture slides.

Several points are worth emphasizing:

- If you are using Oracle, you should create and use a user other than the system user. Each new user is automatically given its own schema.
- If you are using Microsoft SQL Server, it is a good idea to create and use another database when performing these exercises. There are different ways to create databases. One simple way is to execute the following SQL command: create database FIRST_LIBRARY;
- The SQL used in the Library Database works as is for both Oracle and Microsoft SQL Server. In places where the syntax differs between these DBMS, both syntaxes are given.
- Cutting and pasting the commands should not replace the manual typing of the SQL commands by students. Some students learn SQL better when they have a chance to type their own commands and get the feedback provided by the database's responses, including errors. We recommend that the students practice the commands manually.
- Although the book briefly demonstrates the use of the Access Query By Example (QBE) interface, the focus of this lecture is on learning SQL, rather than on having the QBE write the SQL code for you. The use of Microsoft Access in this course is entirely optional, and you are not expected to know anything about Microsoft Access that is not common to all standard SQL databases.

The Library Database

To illustrate features of SQL we employ a sample database consisting of three tables used in a library environment.

BOOK

CALLNO	TITLE	SUBJECT
100	Physics Handbook	Physics
200	Database Systems	Computing
300	HTML	Computing
400	XML	Computing
500	Software Testing	Computing
600	E-Commerce	Business

PATRON

USERID	NAME	AGE
10	Wong	22
15	Colin	31
20	King	21
25	Das	67
30	Niall	17
35	Smith	72
40	Jones	41

LOAN

LOANID	CALLNO	USERID	DATEDUE	DATERET	FINE	PAID
1	100	10	10-FEB-00	09-FEB-00	0	
2	200	10	30-MAY-00	30-MAY-00	0	
3	300	20	05-AUG-88		50	No
4	500	30	26-APR-90		50	Yes
5	600	40	22-DEC-00		10	No

Book

Information on each book in the library is recorded in the BOOK table:

BOOK: callno title subject

For each book we have:

- a call number (*callno*) which uniquely identifies the book
- the *title* of the book
- the *subject* matter of the book

Loan

Each time a book is borrowed information is recorded in the LOAN table:

LOAN: loanid callno userid datedue dateret fine paid

For each loan of a book to a person we record:

- the loan id *(loanid)* key that uniquely identifies the loan row the call number (*callno*) of the book borrowed
- the identifier (*userid*) of the person borrowing the book the date the book is due (*datedue*)
    - the date the book was returned (*dateret*); no value is assigned until the book is returned
    - the *fine* is calculated for all books returned late, at the rate of 10 cents per day
    - the attribute *paid*; if it's value is ***yes*** it indicates that the fine has been paid

The LOAN table is illustrated in Figure 2.1. You will note three separate cases:

Example 1: A book has not been returned. Fine accrues and has not been paid.

LOANID	CALLNO	USERID	DATEDUE	DATERET	FINE	PAID
3	300	20	05-AUG-88		50	No

Example 2: Case where the book was returned before the due date. No fines.

LOANID	CALLNO	USERID	DATEDUE	DATERET	FINE	PAID
1	100	10	10-FEB-00	09-FEB-00	0	

Example 3: The book was returned late. A fine was assessed, and paid.

LOANID	CALLNO	USERID	DATEDUE	DATERET	FINE	PAID
4	500	30	26-APR-90	1-MAY-90	50	Yes

Patron

We refer to users of the library as *patrons*; for each patron of the library we record information in the PATRON table:

PATRON: userid name age

For each patron we have:

- an identification number which uniquely identifies them *(userid)*
- their *name*
- their *age*

Creating the Library Database Tables

To create a table one uses the CREATE TABLE command. The syntax of CREATE TABLE is:

```
CREATE TABLE Table_name(column_specifications);
```

The table is named *Table_name*. The columns of the table are defined in the *column_specifications*. A column specification is the definition of a column giving its data type and other properties. The execution of this CREATE TABLE command causes the database management system to save the definition of the table. Initially the table is empty. We now show how to create a Book table and we then discuss data types and other options. To create the BOOK table we could use:

```
CREATE TABLE Book(
callno DECIMAL(3) NOT NULL,
title VARCHAR(20),
subject VARCHAR(13),
CONSTRAINT book_PK PRIMARY KEY(callno));
```

met_cs669_sp2_11_wmansur_createtable video cannot be displayed here

In the above we have defined three columns for the BOOK table, giving them the names *callno*, *title*, and *subject*. Call numbers are defined as numbers. Titles and subjects are defined as variable length character strings of maximum lengths 20 and 13 respectively. The data type of each column must be specified in the CREATE TABLE statement. The data type restricts the values that can be stored in the column. For example, we could not have a book with call number QA76.9 since QA76.9 is not a number. To refer to literal character strings such as book titles and subjects within SQL we must enclose the literal string in single quotes, such as 'Introduction to Database Systems'.

Note that the name of the table begins with an upper case letter followed by lower case; this is termed humpback or initial caps notation. The names of the columns are all lower case. This follows current common practice. There are several common casing conventions; enterprises typically define the casing conventions for their DBMS identifiers. The case of the letters in database object identifiers (names) doesn't matter in modern relational DBMS. Note that “PK” (meaning primary key) is in caps for readability.

Suitable definitions for the Patron and Loan tables would be:

```
CREATE TABLE Patron(
userID DECIMAL(2) NOT NULL,
name VARCHAR(14),
age DECIMAL(2),
CONSTRAINT patron_PK PRIMARY KEY(userID));

CREATE TABLE Loan(
loanID DECIMAL(3) NOT NULL,
callno DECIMAL(3) NOT NULL,
userID DECIMAL(2) NOT NULL,
datedue DATE,
dateret DATE,
fine DECIMAL(5,2),
paid VARCHAR(3),
CONSTRAINT loan_PK PRIMARY KEY(loanID));
```

The data type of the *fine* column is DECIMAL(5,2). This means that there are up to five decimal digits in a *fine* value, and that exactly two digits are to the right of the decimal point. For example, 123.45 would be a legal fine value.

Test Yourself 2.13

Which of the following will correctly create a table named PRODUCT? (Check the best answer.)

```
CREATE TABLE PRODUCT
```

This is false. This command is incorrect because you must create the columns also.

```
CREATE TABLE PRODUCT WITH COLUMNS(ID,NAME,PRICE,DESCRIPTION)
```

This is false. This command is incorrect because the statement 'WITH COLUMNS' is not the correct syntax. Also, the columns don't have the correct data attributes applied.

```
CREATE TABLE PRODUCT(
ID DECIMAL(3) NOT NULL,
NAME VARCHAR(200),
PRICE MONEY,
DESCRIPTION VARCHAR(200))
```

This is true. This is the correct SQL syntax for creating a table.

```
INSERT TABLE PRODUCT(
ID DECIMAL(3) NOT NULL,
NAME VARCHAR(200),
PRICE MONEY,
DESCRIPTION VARCHAR(200))
```

This is false. The INSERT command is used to insert data and not a table. A table structure must be created with the CREATE TABLE command.

```
CREATE TABLE PRODUCT
WITH COLUMNS(
ID DECIMAL(3) NOT NULL,
NAME VARCHAR(200),
PRICE MONEY,
DESCRIPTION VARCHAR(200))
```

This is false. The command is incorrect because the statement 'WITH COLUMNS' is not the correct syntax.

Inserting Rows into the Library Database Tables

To insert a row into a table, one uses the INSERT INTO command. The general syntax of the INSERT INTO is:

```
INSERT INTO table-name (column-names)
VALUES (column-values);
```

This command inserts one row using this syntax. The columns which will be assigned values are named (column-names), and the assigned values are listed in (column-values). The two lists are correlated. The first column named in (column-names) will be assigned the first value listed in (columnvalues), the second column will be assigned the second value, and so on. We now illustrate the use of the command.

To insert rows into the book table, we could use:

```
INSERT INTO BOOK (CALLNO, TITLE, SUBJECT)
VALUES (100, 'Physics Handbook', 'Physics');
INSERT INTO BOOK (CALLNO, TITLE, SUBJECT)
VALUES (200, 'Database Systems', 'Computing');
INSERT INTO BOOK (CALLNO, TITLE, SUBJECT)
VALUES (300, 'HTML', 'Computing');
INSERT INTO BOOK (CALLNO, TITLE, SUBJECT)
VALUES (400, 'XML', 'Computing');
INSERT INTO BOOK (CALLNO, TITLE, SUBJECT)
VALUES (500, 'Software Testing', 'Computing');
INSERT INTO BOOK (CALLNO, TITLE, SUBJECT)
VALUES (600, 'E-Commerce', 'Business');
```

Suitable insertions for the PATRON and LOAN tables could be:

```
INSERT INTO PATRON (USERID, NAME, AGE)
VALUES (10, 'Wong', 22);
INSERT INTO PATRON (USERID, NAME, AGE)
VALUES (15, 'Colin', 31);
INSERT INTO PATRON (USERID, NAME, AGE)
VALUES (20, 'King', 21);
INSERT INTO PATRON (USERID, NAME, AGE)
VALUES (25, 'Das', 67);
INSERT INTO PATRON (USERID, NAME, AGE)
VALUES (30, 'Niall', 17);
INSERT INTO PATRON (USERID, NAME, AGE)
VALUES (35, 'Smith', 72);
INSERT INTO PATRON (USERID, NAME, AGE) VALUES (40, 'Jones',
41);

INSERT INTO LOAN (LOANID, CALLNO, USERID, DATEDUE, DATERET,
FINE)
VALUES (1, 100, 10, CAST('10-FEB-2000' AS DATE),
CAST('09-FEB-2000' AS DATE), 0);
INSERT INTO LOAN (LOANID, CALLNO, USERID, DATEDUE, DATERET,
FINE)
VALUES (2, 200, 10, CAST('30-MAY-2000' AS DATE),
CAST('30-MAY-2000' AS DATE), 0);
INSERT INTO LOAN (LOANID, CALLNO, USERID, DATEDUE, FINE,
PAID)
VALUES (3, 300, 20, CAST('05-AUG-1988' AS DATE),
50, 'No');
INSERT INTO LOAN (LOANID, CALLNO, USERID, DATEDUE, FINE,
PAID)
VALUES (4, 500, 30, CAST('26-APR-1990' AS DATE),
50, 'Yes');
INSERT INTO LOAN (LOANID, CALLNO, USERID, DATEDUE, FINE,
PAID)
VALUES (5, 600, 40, CAST('22-DEC-2000' AS DATE),
10, 'No');
```

Selecting Columns from the Library Database

We illustrate queries that retrieve all rows of a table.

Example 3.1 List the titles of books in the database.

```
SELECT title
FROM book;

TITLE
Physics Handbook
Database Systems
HTML
XML
Software Testing
E-Commerce

6 rows selected.
```

When an SQL system executes the above example it accesses the BOOK table. Since the SELECT does not involve a WHERE clause all rows of BOOK are accessed and from each row the system extracts and displays the title. There are as many rows in the result as there are rows in BOOK. The results are shown above.

Example 3.2 List the title and subject for each book.

```
SELECT title, subject
FROM book;

TITLE SUBJECT
Physics Handbook Physics
Database Systems Computing
HTML Computing
XML Computing
Software Testing Computing
E-Commerce Business

6 rows selected.
```

Example 3.3 List all fields for each book.

```
a) SELECT callno, title, subject
FROM book;

b) SELECT *
FROM book;
```

The * above is an abbreviation for “all fields”.

```
CALLNO TITLE SUBJECT
100 Physics Handbook Physics
200 Database Systems Computing
300 HTML Computing
400 XML Computing
500 Software Testing Computing
600 E-Commerce Business

6 rows selected.
```

Example 3.4 What are the subject areas of the library?

```
SELECT subject
FROM book ;

SUBJECT
Physics
Computing
Computing
Computing
Computing
Business

6 rows selected.
```

The answer to this query has as many lines as there are books in the library. Since there are very few subject areas the result appears awkward. To remove the redundancies from the display we can use the DISTINCT key word; it causes redundant rows to be eliminated:

```
SELECT DISTINCT subject
FROM book ;

SUBJECT
Business
Computing
Physics
```

Test Yourself 2.14

Which of the following will correctly select all records from the table PRODUCT that we just created? (Check all that apply.)

LIST * FROM PRODUCT

This is false. LIST is not a key word in the SQL language. The correct term is SELECT.

SELECT * FROM PRODUCT

This is true. SELECT is the correct key word to select data. The * means all columns.

SELECT ID, NAME, PRICE, DESCRIPTION FROM PRODUCT

This is true. SELECT is the correct key word to select data. This is correct because the query lists all columns, and therefore will list all the contents from the PRODUCT TABLE.

SELECT ALL FROM PRODUCT

This is false. ALL is not correct. ALL should be replace with *. The * means all columns.

LIST ID, NAME, PRICE, DESCRIPTION FROM PRODUCT

This is false. LIST is not a key word in the SQL language. The correct term is SELECT.

Selecting Rows from the Library Database

In the previous section we were concerned with listing one or more fields from a table; every row of the table corresponded to a line in the listing. Now we consider retrieval of a subset of the rows of a table. To limit the retrieval to specific rows of a table we include the `WHERE` clause in our commands. The WHERE clause gives the condition that a row must satisfy to be included; any row of the table not satisfying the condition is not considered. In the following we discuss operators that may appear in conditions:

=, <>, >, >=, <, <=

IN, BETWEEN, AND, LIKE, IS NULL, AND, OR, NOT

Example 3.5 List the titles of Computing books

```
SELECT title
FROM book
WHERE subject = 'Computing';
```

```
TITLE
Database Systems
HTML
XML
Software Testing
```

The condition *subject = 'Computing'* is evaluated for each row of BOOK. If the expression evaluates to true then the row contributes to the result.

Example 3.6 List the book with call number 200.

```
SELECT title
FROM book
WHERE callno = 200 ;
```

```
TITLE
Database Systems
```

Special Operators for the WHERE clause

Four operators are available to handle special cases:

`LIKE` is used when searching for the appearance of a particular character string.

`BETWEEN` is used when searching for a value within some range.

`IS NULL` is used to test for a field not having been assigned any value.

`IN` is used to test for a field having a value contained in some set of values.

LIKE

LIKE is used with character data to determine the presence of a substring. Special notations are available to specify unknown or irrelevant characters in the field being tested:

- a single unknown character: _ (underscore)
- any number of unknown characters: %

Example 3.10 List books with Database in the title.

```
SELECT *
FROM book
WHERE title LIKE '%Database%';
```

```
CALLNO TITLE SUBJECT
200 Database Systems Computing
```

In example 3.10 each BOOK title is examined to determine if it contains the character string *Database*.

Example 3.11 List books with titles having an **o** as the second character.

```
SELECT *
FROM book
WHERE title LIKE '_o%';
```

```
CALLNO TITLE SUBJECT
500 Software Testing Computing
```

In this example the title field of each row in BOOK is examined to determine if it has an **o** in the second character position.

BETWEEN

The between operator is used with numeric data to determine if some field lies in a certain range.

Example 3.12 List books with call numbers between 200 and 400.

```
SELECT *
FROM book
WHERE callno BETWEEN 200 AND 400 ;
```

```
CALLNO TITLE SUBJECT
--------- -------------------- -------------
200 Database Systems Computing
300 HTML Computing
400 XML Computing
```

Note that this is the same as the command:

```
SELECT *
FROM book
WHERE (callno>=200) AND (callno<=400);
```

IS NULL

NULL is a keyword that must be used to determine whether or not a field has been assigned a value. Note that in our database the date returned field is not assigned any value until a book is returned. If a book is returned on time then no value is assigned to the fine or paid fields.

Example 3.13 List the books currently out on loan.

```
SELECT callno
FROM loan
WHERE dateret IS NULL;
```

```
CALLNO
300
500
600
```

Updating Data in the Library Database

In this Chapter we introduce Data Manipulation Language statements responsible for updating the database.

There are three commands for updating:

1. `UPDATE` modify rows of tables
2. `DELETE` remove rows from tables
3. `INSERT` add new rows to tables

Modifying Rows

The general form of the UPDATE command is UPDATE table

`SET` field-assignments

`WHERE` condition

> The field assignments are of the form field = expression and are used to assign specific values to the fields of a row. The expressions must be of the appropriate type for the data type of the corresponding column. These expressions cannot be subqueries or involve the aggregate operators (AVG, COUNT, ...). The WHERE clause is optional; if absent then the UPDATE applies to all rows.

Example 4.1 Increase every patron’s age by 10.

```
UPDATE patron
SET age=age+10;

7 rows updated.

select * from patron;

userid name age
100 Wong 32
150 Colin 41
200 King 31
250 Das 77
300 Niall 27
350 Smith 82
400 Jones 51
```

Note that every row of PATRON is modified since there is no WHERE clause. Each row of PATRON is modified according to the field assignments. There is only one field assignment: AGE=AGE+10. The effect of this is to cause the current value of age in a row to be incremented by 10; the value of this expression becomes the new value of the AGE field for the row.

Example 4.2 Determine fines for books at the rate of one cent a day. In this example we want to modify only the Loan row for userid 30. For Oracle, we use the following command:

```
UPDATE loan
SET fine = (SYSDATE - datedue) * 0.01
WHERE dateret is NULL
AND userid = 30

1 row updated.
```

For Microsoft SQL Server, we use the following command:

```
UPDATE loan
SET fine = (DATEDIFF(D, datedue, GETDATE())) * 0.01
WHERE dateret is NULL
AND userid = 30
(1 row(s) affected)
```

The reason for the DBMS specific commands is that each DBMS uses a different method for date calculations. Oracle allows use of the standard plus and minus operator between dates, while SQL Server uses built-in functions DATEDIFF and DATEADD.

```
SQL> SELECT *
2 FROM loan
3 WHERE userid = 30;

LOANID CALLNO USERID DATEDUE DATERET FINE PAID
4 500 30 26-APR-90 75.78
Yes
```

Deleting Rows

The general form of the DELETE command is

`DELETE`

`FROM` table

`WHERE` condition;

The effect of `DELETE` is to remove rows from a table; the rows deleted are those that satisfy the condition specified in the WHERE clause. The `WHERE` clause is **optional**; if absent then all rows are deleted.

Example 4.3 Remove Computing books from the database.

```
DELETE FROM book
WHERE subject='Computing';

4 rows deleted.

select * from book;
CALLNO TITLE SUBJECT
100 Physics Handbook Physics
600 E-Commerce Business

2 rows selected.
```

Note: If referential integrity is implemented (as illustrated in the next chapter) we will get the following error message.

```
DELETE FROM book;
*
ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.BOOK_FOREIGN_KEY) violated - child record
found.
```

Example 4.4 Remove all loan records for patron “King”.

```
DELETE FROM loan
WHERE userid = (SELECT userid
FROM patron
WHERE name='King');
```

Inserting New Rows

In this section we illustrate how new rows are appended to a table. There are two forms of the INSERT command

a) `INSERT`

INTO table (field-list)
VALUES (constant, constant, ...);

b) `INSERT`

INTO table (field-list)

subquery;

The first form is used to insert a single row in a table; the second form is used to insert multiple rows that come from one or more existing tables. If all fields are included and the values given in order, then the field list can be omitted. Example 4.5 illustrates how one adds one row to a database.

Example 4.5 Add a new patron to the database.

```
INSERT INTO patron (userid, name, age)
VALUES (90,'Dattani',20);

SELECT * FROM PATRON;

USERID NAME AGE
10 Wong 22
15 Colin 31
20 King 21
25 Das 67
30 Niall 17
35 Smith 72
40 Jones 41
90 Dattani 20

8 rows selected.
```

This method is clumsy and some other approach is needed if one is to add several rows to a table. SQL systems typically provide a forms-based interface, or a load utility, for entering many rows or to load a database.

The second form for INSERT is illustrated in Example 4.6.

Example 4.6 Create a table of senior citizens.

```
CREATE TABLE seniors
(userid NUMERIC NOT NULL,
name CHARACTER(30) NOT NULL);

INSERT INTO seniors
SELECT userid, name
FROM patron
WHERE age >= 65 ;
```

Test Yourself 2.15

Which of the following will correctly update the PRICE to $4.00 in the PRODUCT table, where the product is named ‘Hammer’? (Check the best answer.)

UPDATE SET PRICE=4.00 FROM PRODUCT WHERE NAME='Hammer'

This is false. The FROM keyword is not needed. Also, the order of the SQL command matters.

UPDATE PRODUCT SET PRICE=4.00 WHERE NAME='Hammer'

This is true. This is the correct syntax to update the table PRODUCT.

SET PRICE=4.00 WHERE NAME='Hammer' FROM PRODUCT

This is false. The keyword UPDATE is needed and the syntax is incorrect.

UPDATE PRICE=4.00 FROM PRODUCT WHERE NAME='Hammer'

This is false. This is the incorrect syntax for an update command.

UPDATE * FROM PRODUCT SET PRICE=4.00 WHERE NAME='Hammer'

This is false. The UPDATE * is not necessary. The FROM is also not needed.

Boston University Metropolitan College