

INTRODUCTION TO DATABASE PROCESSING – CHAPTER 1 --

- The purpose of a database is to help people keep track of things.
- As examples, there are four different situations in which database technology can be employed to solve real-world problems...
- 1. Mary Richards is a house painter who needs to keep track of her paint-jobs and referrals in order to better manage and server her customers.
- 2. Treble Clef Music is a music rental company that is in need of a database application that is able to handle multiple simultaneous transactions online.
- 3. State Licensing and Vehicle Registration Bureau's need is even greater and more demanding. The personnel in these offices access a database to perform their jobs. Before people can be issued or can renew their driver's licenses, their records in the database are checked for traffic violations, accidents, or arrests. These data are used in order to determine if or not to renew a license and if so, to determine if such license should carry any sorts of limitations or restrictions. This particular database is large and complex with more than 40 different tables of data , several of which contain hundreds or even thousands of rows of data. Other examples of organizational databases concern account processing at banks and financial institutions, production and material supply systems at large manufacturers, medical records processing at hospitals, and insurance companies and government agencies.
- 4. Calvert Island Reservations Centre is a beautiful island on the west coast of CANADA trying to set up an informative website in order to promote tourism. The specific needs of this organization is that they need to make use of a database able to handle multimedia to be served online on a moment's notice. Furthermore, Hypertext transfer protocol (HTTP), Dynamic Hypertext Markup Language (DHTML), and Extensible Markup Language (XML) technologies are all employed here which allow online users to view and interact with the database, unlike the other forms of databases here discussed.
- The first business information systems stored groups of records in separate files and were called file-processing systems. While this implementation was an improvement in data organization and utilization prior to electronic means of database implementation, this model suffered from a number of limitations. For example, (1) the data stored was separate and isolated, (2) the data stored was often duplicated, (3) The application programs were often dependent on particular file formats, (4), files were often incompatible with one another, (5) and it is difficult to represent data in the user's perspectives.
- 1. When the amount of data that must be manipulated becomes very large, having separated data makes it very difficult to access and compile particular items that are located in multiple files, making it much more complex and time consuming.
- 2. The duplication of data storage problem goes beyond file waste space...the most serious problem that you are likely to run into is 'data integrity' problems which, if severe, can make the validity of the stored data questionable, and therefore useless.
- 3. Having applications dependant upon particular file formats creates a problem when database structure is in need of updating...for example, if a customer's record is modified to expand a ZIP CODE field, the application responsible for reading such information must also be modified. The problem is exacerbated when there are multiple applications that all depend on a particular format, since all of these applications must also be modified every time a modification is made on the format of the data.
- 4. One of the consequences of program data dependency is that a file formats depend on the language or product used to generate them. As a result, files cannot be readily combined or compared. Finding a solution for this problem can be time consuming and sometimes very difficult as the number of files to be processed increases.

- 5. When presenting data reports to customers, it is a more complex process to extract, process, and combine different files. This problem arises because with file processing systems, there is a lack of proper relationships between records.
- In order to solve these limitations of the file processing system implementation of databases, Database Processing Systems were implemented. File processing systems directly access data. In contrast, database-processing programs call the DBMS to access the stored data.
- In a database system, all of the data is located in a single location, the database.
- The term *database* in this book refers to a *self-describing* collection of *integrated records*.
- A database is self-describing: It contains, in addition to the user's source data, a description of its own structure. This description is called the **data dictionary** (also called data directory or metadata).
- The structure of the database is extracted from the database and loaded into the program before the program is compiled.
- The standard hierarchy of data in a database is as follows: Bits are aggregated into bytes or characters; characters are aggregated into fields; fields are aggregated into records; and records are aggregated into files.
- In addition to a data dictionary, databases include **indexes** which represent the relationship among data and also improve the performance of database applications.
- Furthermore, the structure of a data entry form, or a report, is sometimes part of the database. This last category of data we call **application metadata**. Thus databases contain four different types of data: files or user data, metadata, indexes, and application metadata.
- **Transactions** are representations of events. When events take place, the transactions for the events must be processed against the database.
- In 1970, E.F. Codd published a landmark paper in which he applied concepts borrowed from relational algebra to solve problems of storing large amounts of data. This paper eventually led to the definition and implementation of **relational databases**.
- The advantages of relational databases model is that data are stored in a way that minimizes duplicated data and eliminates certain types of processing errors that can occur when data are stored in other ways. In relational databases, data are stored in tables containing rows and columns. There is the concept of *normalization* which involves eliminating and restructuring of tables that are determined to be undesirable. A key advantage of this model is that columns may contain data that related one row to another.
- **Distributed databases** allow for personal, workgroup, and organizational databases to be combined into integrated but distributed systems. The essence of distributed databases is that all of the organization's data are spread over many computers, micros, LAN servers, and mainframes, that communicate with one another as they process the database. However, among the most pressing problems of these models is security and control.

INTRODUCTION TO DATABASE DEVELOPMENT – CHAPTER 2 --

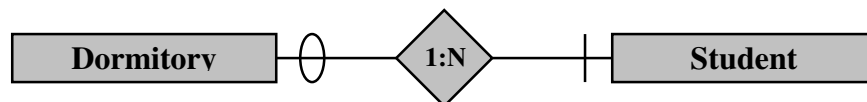
- **User Data** must be organized in tables in such a way that when updates are necessary, the number of fields that must be modified is minimized. Optimally, there is no duplication of data and every element that must be modified is done so just once.
- **Metadata** is the data dictionary (self-describing data of the database) is typically stored in tables, often known as *system tables*.
- **Indexes** are a third type of database data (sometimes called overhead data) in which linked lists are often implemented. Indexes are used not only for sorting purposes, but also for quick access to the data. Although indexes help speed up the process of searching through the data, they are not without cost since they must be updated every time data changes in the tables.
- **Application Metadata** is used to store the structure and format of user forms, reports, queries, and other application components.
- **1. The Design Tools Subsystem** has a set of tools that help facilitate the design and creation of the database and its applications. It typically includes tools for creating tables, forms, queries, and reports. They may also provide interfaces to programming languages.
- **2. Run-Time Subsystem** processes the application components that are developed using the design tools. During execution, the form run-time processor extracts values from the tables and updates the fields in a form with the appropriate value to be displayed. Some run-time processors answer queries, and print reports.
- **3. The DBMS Engine** is the third component of the DBMS and is the intermediate between the design tools and run-time subsystems and the data. The DBMS Engine receives requests from the other two systems in the form of tables, rows, and columns, and translates those requests into commands to the operating system to read and write data on physical media. The DBMS Engine is also involved in transaction management, locking, and backup and recovery.
- A database schema defines a database's structure, its tables, relationships, domains, and business rules.
- 1. Tables: Structures that have rows and columns of data.
- 2. Relationships: logical connections between tables may exist in a 1:1, 1:N, N:N, N:1 etc.
- 3. Domain is asset of values that a column may have (I believe that they are talking about data types).
- 4. Business Rules are the last component of a schema which are restrictions on the business's activities that need to be reflected in the database and database applications. For example, in order to check out any equipment, a team captain must have a local phone number. No single captain should have more than 7 soccer balls, etc. In Oracle, business rules are enforced via the implementation of stored procedures.
- Creating Tables is the next step after creating schemas using the DBMS's table creation tools.
- In the example in the book, there exists a 1:N relationship between CAPTAIN and ITEM. Therefore, when designing the tables, we placed the field CAPTAIN_ID, which is the primary key in the CAPTAIN table as a column in the ITEM table. This column is known as a **Foreign Key** because it is a key foreign to the current table.
- Once the tables, columns, and relationships are created, the next step is to build the application components.
- In the case of Microsoft's Access database application, it allows to easily construct tables, assign a **key** column, and to link tables by inserting **foreign keys** into other tables. In addition, Access is very GUI

oriented and user friendly allowing the user to easily and rapidly construct forms by clicking and dragging methods.

- Queries are simply requests to view particular items in the database that meet some sort of unique criteria. In Access, such queries can be done using SQL or some sort of GUI interface design mode that allows the user to **construct queries by example**. A third type of query is called **Query by Form** in which the user may enter query constraints into a form and presses a 'search' button ...then, the DBMS finds all records that meet the given constraints of the particular search.
- Reports is a formatted display of database data. Again, in Access, this can be accomplished very easily using macro tools to automate much of the design process or by manipulating each element individually.
- Menus can be built into applications such that the end user has an easier time navigating and automating certain search criteria.
- Application Programs are the final component of database applications. As mentioned earlier, these can be created using a DBMS-specific language or another language that uses standard procedures required to interface with the database. In the case of ACCESS, Visual Basic may be used to construct the application that can interface with the database.
- Database Development Process...there are two main approaches when developing databases: (1) top-down and (2) bottom-up.
- Top-Down Development proceeds from the general to the specific.
- Bottom-Up Development proceeds from specific to general.
- Data Modeling is complicated by the fact that there is not just one requirement, but many and that the requirements often overlap.

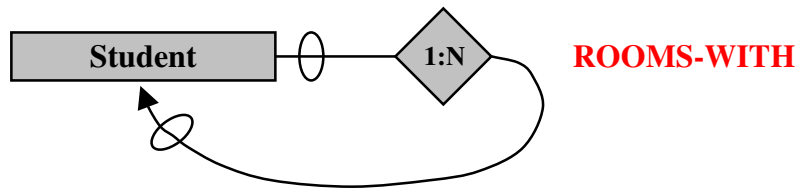
THE ENTITY RELATIONSHIP MODEL – CHAPTER 3 –

- The Entity-Relationship Model (E-R model) was first introduced by Peter Chen in 1976 and has since undergone a number of modifications by himself and others. Today, there is no standard, but there are a number of similar models floating around that are commonly used today.
- Key elements of the E-R model are entities, attributes, identifiers, and relationships.
- **Entities** are items that can be identified in the user’s work environment such as EMPLOYEE Mary Doe, CUSTOMER 13245, SALES-ORDER 1000, etc.
- Entities of a given type are grouped into **entity classes**. Thus the EMPLOYEE entity class is the collection of all employee entities. It is important to differentiate between an *entity class* and an *entity instance*. Entity class refers to the structure or format of entities, while entity instances refer to particular instances of such class.
- **Attributes**, are properties of entities that describe their characteristics. For example, EmployeeName, DateOfHire, JobSkillCode, etc. are all attributes of the EMPLOYEE entity.
- **Identifiers** are attributes of entity instances that name or identify the entity instances. For example, EMPLOYEE instances could be identified by SocialSecurityNumber, EmployeeNumber or EmployeeName.
- **Unique** identifiers are those that apply to one and only one entity such as EmpNumber.
- **Non-Unique** identifiers are those that may apply to one or more entity instances such as EmpName.
- **Composite Identifiers** are those that consist of two or more attributes such as: {AreaCode; LocalNumber}, {ProjectName; TaskName}, or {FirstName; LastName; PhoneExtension}
- **Relationships** are used to associate two or more entities with each other.
- The E-R model contains both relationship classes, which are relationships between entity classes and relationship instances which are relationships among entity instances.
- **Degree** is the number of entities in a relationship. Relationships of degree 2 occur with great frequency and these are also known as binary relationships.
- **3 types of binary relationships** one to one (1:1), one to many (1:N), and many to many (N:M).
- **Maximum Cardinality** refers to the maximum number of entities that may be included in the relationship diagram. For example, in a basketball team, there is a 1:5 maximum cardinality meaning that only 5 players can be in the court at any given time. Note that Maximum Cardinality says nothing about the minimum number of entities that may exist in any relationship.
- **Entity Relationship Diagrams** show entities in rectangles, and the maximum cardinality inside a diamond between two or more entities connected by lines. The name of the entity is shown inside the rectangle and the name of the relationship is shown near the diamond (see p. 53 of *Database Processing. Fundamentals, Design, and Implementation*, 7th ed. By David M. Kroenke).
- **Minimum Cardinality** can be represented in many ways. One way is to simply add a *hash* mark over the relationship line indicating that an entity must exist in the relationship, while an *oval* is used over the relationship line to indicate that an entity may or may not exist in the relationship.

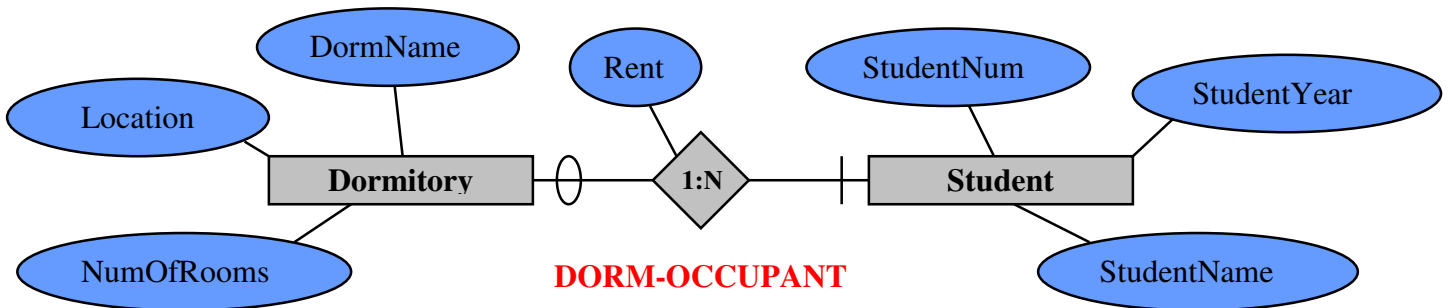


DORM-OCCUPANT

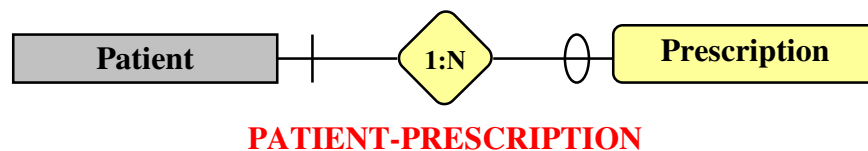
- Relationships may exist between members of the same class and this type of relationship is often referred to as a **recursive relationship**. For example, a relationship may exist between two members of the entity class student, in which two students room with each other in a dormitory.



- Entity attributes can be shown in E-R diagrams using ellipses containing the name of the attribute inside them and connected by a line to the particular entity.

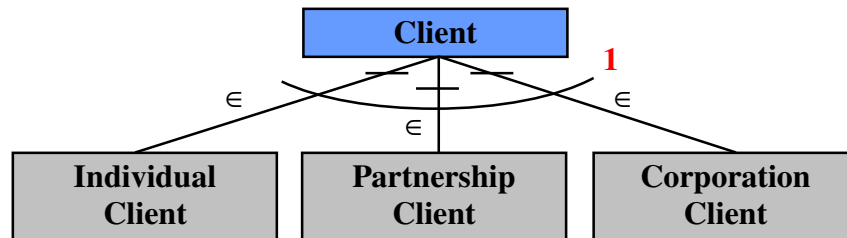


- In the event that a particular entity has many attributes that clutter the diagram, these may instead be listed and appropriately labeled.
- Weak Entities** are those that cannot exist in the database unless another type of entity also exists.
- Strong Entities** are those entities that must always exist in the database.
- An example of these is the case of 'Employee' and 'Dependent'. Employee would be a type of strong entity while Dependent would be a weak entity since the field Dependent is not always necessary in the database.
- In the E-R model, a weak entity is represented by using a relationship diamond whose corners are rounded.



- An **ID-Dependent Entity** is an entity in which the identifier of such includes the identifier of another entity. Consider the two entities Building and Apartment...in this case, the entity Apartment cannot exist unless the entity Building exists.
- Multi-Value Attributes** are represented in E-R by creating a new weak entity to represent the new multi-value attribute and constructing a one to many relationship .
- Subtype Entities** are those that may contain optional sets of attributes depending on the nature of the entity. For example, consider the entity CLIENT with entities ClientNumber, ClientName, AmountDue. In this case, the CLIENT may be either an individual, a partnership or a corporation and depending on this, the types of associated entities that are to be related to these may vary.

- A way of dealing with this situation is that we can allocate all of these attributes to the entity CLIENT, but the problem with this is that not all sub-entities apply to all types of CLIENT. A closer fitting model would be to instead to define three **SUBTYPES** (1) Individual-CLIENT, (2) Partnership-CLIENT, and (3) Corporation-CLIENT. All of these, of course have CLIENT as a **SUPERTYPE**.
- In the diagram (see p. 58 of text), the symbol \in is used to indicate that one entity is a subtype of another. The curved line with a number 1 means that an entity must belong to one and only one subtype entity.

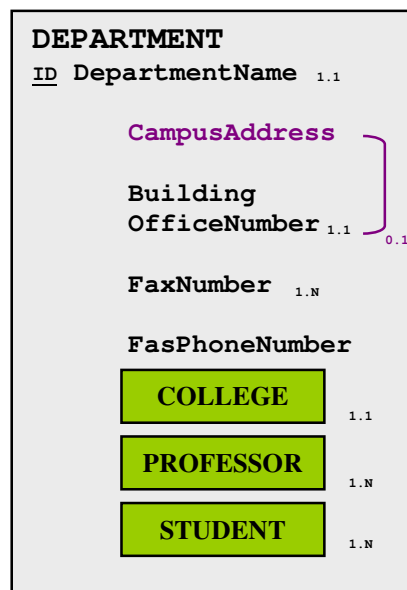


- Subtype entities are not always required nor are they always mutually exclusive. For this reason, in the above diagram replacing the “1” with an “m” would indicate that entity CLIENT could belong to 0 or multiple subtype entities.
- **Generalization Hierarchies** is a name often given to structures of subtypes and supertypes since in the above example, CLIENT is a generalization of its subtypes. The concept of inheritance is also employed some times meaning that the subtypes inherit all of the properties of the supertype, and in addition, they may implement some of their own.
- See page 59 of the text to see an example of an E-R diagram that depicts all of the concepts we have been discussing thus far. It is a good idea to review this chart and analyze all its parts in detail.
- **Documentation of Business Rules** usually takes place after the E-R model is constructed and relationships are determined. For example, in the ENGINEER-TRUCK relationship, there are situations when the resources (TRUCKS) are limited and not all ENGINEERS can have access to a truck...in this case, business rules must be added to the E-R model that establish under what conditions an ENGINEER is to be assigned a truck. However, it is not always the case that business rules are enforced by the DBMS...in some cases, these rules are simply documented in manuals that are then shared by individuals.
- **Note:** that there are software products known as CASE Tools that can facilitate the creation of E-R models including IEW, DEFT, ER-WIN, and Visio .
- Generally, the first step in designing the E-R model is to establish the ENTITIES that will play a role in the database system. This can be done by identifying the ‘nouns’ used to describe the problem. In most cases, nouns represent the ENTITIES that will be needed.
- Then, we must establish the relationships that may exist between all of the entities paying a close attention to which relations are strong or weak and establishing the maximum cardinality between entities in a relationship.
- Finally, before implementing the database, it is important to verify the accuracy of the model by looking closely at the final E-R model. It is often very easy to fix an inconsistency in the E-F model. However, when changes are needed in an already existing database system, these changes can be lengthy and expensive.

- A good way of testing a model is by simply reviewing some of the possible queries that will be made on the system, and to try to figure out if those queries can be appropriately answered using the current model.

THE SEMANTIC OBJECT MODEL – CHAPTER 4 –

- This chapter discusses the semantic object model, which, like the E-R model is used to create data models. The development team interview users; analyzes the users' reports, forms, and queries; and from these constructs a model of the users' data. The data model is later transformed into a database design.
- The semantic object model is a data model. It is different from the object-oriented database processing model.
- The goals of the early stages of database development are to determine the things to be represented in the database, to represent the characteristics of those things, and to establish the relationships among them.
- In the previous chapter, we referred to these things as entities, in this chapter, we call them semantic objects or sometimes just objects.
- A **semantic object** is a named collection of *attributes that sufficiently describes a distinct identity*.
- A particular semantic object is an instance of a class.
- Note that something need not be physical in nature in order to be considered an object. Take for example something like an ORDER object which itself represents a conceptual construct; an agreement to provide certain goods and services under certain terms and conditions.
- **Attributes** are elements that describe the characteristics of objects and there are three main types of these.
 - **1. Simple Attributes** are those that contain a single value such as DateOfHire, or InvoiceNumber.
 - **2. Group Attributes** are composites of other attributes such as ADDRESS which can contain: {Street, City, State, Zip}.
 - **3. Semantic Object Attributes** are those that establish a relationship between one semantic object and another.
- A **Semantic Object Diagram** or **Object Diagram** is a graphical representation used by development teams to summarize the structures of objects and to present them visually. Objects are shown in portrait-oriented rectangles, The name of the object appears at the top, and attributes are written in order after the object name (See example below...notice the cardinality is included...)

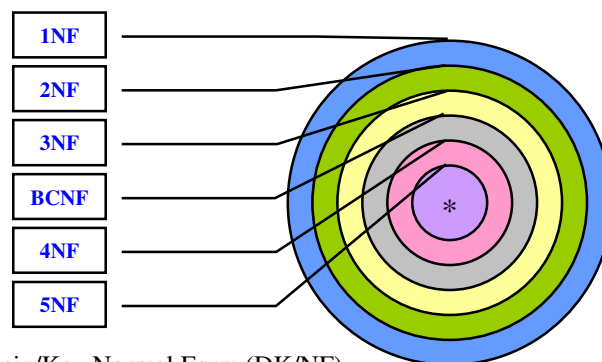


- Each attribute in a semantic object contains both a maximum and a minimum cardinality. Note that it is fairly unusual, there are instances in which the minimum cardinality of an object is greater than 1. For example, the attribute PLAYER in a the BASKETBALL_TEAM object may be 5 since it is the smallest number of players that make a basketball team. Cardinalities are shown in the format **N.M** *subscripted* in which N represents the minimum and M represents the maximum cardinality.
- Object Instances are simply particular object with all its attributes and logical relationships.
- **Paired Attributes** refers to the fact that in the semantic object model, if an object contains a second object, then that second object automatically contains the first. In this way a relationship are established between objects.
- **Object Identifier** is one or more object attributes that the users employ to identify object instances. For example, for CUSTOMER, we can have CUSTOMER_ID as an identifier or SSN is an identifier for the CITIZEN object.
- In Semantic Object Diagrams, identifiers are denoted by the letters ID and if the identifier is unique, these two letters will be underlined (i.e., ID).
- The **Domain** of an attribute is a description of an attribute's possible-values. These of course, are dependent upon the nature of the attribute. The domain of an attribute consists of two parts, a semantic and a physical description. The physical description indicates the type of data (numeric versus string), length of the data, and other restrictions that apply. The semantic description indicates the purpose or function of the attribute and distinguishes this attribute from other attributes that may have the same physical description.
- The portion of an object that is visible to a particular application is called the **semantic object view** or simply the **view**.
- See page 85 for a complete diagram of Highland University's semantic objects.
- **Formula Domain** is one in which the attributes of an object is represented by a formula that evaluates to a value generated by other values.
- Described in this book, there are a total of 7 types of objects: (1) Simple, (2) Composite, (3) Compound, (4) Hybrid, (5) Association, (6) Parent/Subtype, and (7) Archetype/Version, objects.
- First, let's look at three new terms: A **single-value attribute** is an attribute whose maximum cardinality is 1. A **multi-value attribute** is one whose maximum cardinality is greater than 1. And a **non-object-attribute** is a simple or group attribute.
- 1. **Simple Object** is a semantic object that contains only single-value, non-object attributes. For example EQUIPMENT which models an equipment tag. Its attributes are EquipmentNumber, Description, AcquisitonDate, and PurchaseCose. Note that none of these attributes is multi-value like 'Address' would be and none is an object, i.e., none represents another object.
- 2. **Composite Object** is a semantic object that contains one or more multi-value non-object attributes. For example, a HOTELBILL can include such attributes as SERVICES which could have a maximum cardinality of N and therefore is a multi-value attribute. Note, however that this type of object does not include attributes that represent other objects. Also, multi-value attributes may include one or more and these may be nested within each other.
- 3. **Compound Objects** are those that contain at least one object attribute. For example, in the case of Highland University's DEPARTMENT object, it contained objects such as COLLEGE, PROFESSOR, and STUDENT, all of which represent other semantic objects. Note that there need not be a condition of 'paired attributes' (see above) where it is not required for one object to necessarily contain a second which contains it. Example, DORMITORY may include STUDENT, but STUDENT may not necessarily contain DORMITORY...and this is determined by the particular need of the model depending on whether or not it is necessary to keep such relationship information.

- 4. **Hybrid Objects** are combinations of objects of two types of attributes. In particular, it is an object that contains a multi-value and an object attributes. For example, object DORMITORY has attributes StudentRent and STUDENT, an object Attribute, which as a pair, form a multi-value attribute. (See page 96 for another example of this type of object).
- 5. **Association Objects** are those that relate two or more objects and store data that are peculiar to that relationship. See page 99 for an example of a report that contains data about an airline flight and a data about a particular airline and pilot assigned to that flight. Note in particular that object FLIGHT contains objects AIRPLANE and PILOT and each of these objects contains information about the particular FLIGHT. Thus, FLIGHT, in this case associates the two others. Note the cardinality for each of these objects...note that in FLIGHT, if there is one, there is always an AIRPLANE involved (1.1) and always a PILOT (1.1). However, in the AIRPLANE and PILOT objects, the cardinality for the FLIGHT object is (0.1) meaning that these two items may or may not be needed depending on whether or not there is a FLIGHT.
- 6. **Parent/Subtype Objects** are objects which belong to some hierarchical relationship with other related objects. For example, the EMPLOYEE could be thought of as the **parent** or **super-type** object of both MANAGER and PROGRAMMER objects which are known as **sub-type** objects. As such, EMPLOYEE contains attributes that apply to both PROGRAMMER and MANAGER, but each of the sub-types may include other attributes that apply to their individual nature. Note that sub-type attributes are shown to have cardinality in the form **0.ST** while parent or super-type objects have cardinalities that look like this **P**.
- Sometimes subtypes exclude one another. That is, a VEHICLE can be an AUTO or a TRUCK, but not both. When this happens, they are placed into a subtype group and the group is assigned a subscript of the format X.Y.Z where X is the minimum cardinality and Y and Z are counts of the number of attributes in the group that are allowed to have a value. Y is the minimum value required, and Z is the maximum value allowed. For example (see p 103) a subscript of 0.1.1 for a group of subtypes means that a subtype is not required (0), but if it exists, there must be one and only one of the subtypes must exist.
- 7. **Archetype/Version Objects** are those that produce other semantic objects that represent versions, releases, or editions of the archetype. For example, the archetype object TEXTBOOK produces the versions of objects EDITIONS.

THE RELATIONAL MODEL AND NORMALIZATION – CHAPTER 5 –

- Not all relationships are equal...some are better than others. Normalization refers to the process in which a bad relationship can be converted into two or more better relationships between entities.
- A **Relation** is a two dimensional table. Each row in the table holds data regarding an attribute. Sometimes rows are called **tuples** (rhymes with “couples”), and columns are called **attributes**.
- **Functional Dependency** is a relationship between or among attributes. For example, if we know the value of CustomerAccountNumber, we can find the value of CustomerBalance. If this is true, we can say that CustomerBalance is *functionally related* on CustomerAccountNumber. Another example is TotalPrice is functionally dependent on both ItemPrice and Quantity. Note that in the functional dependency: $SID \rightarrow Major$ (“SID determines Major”), the SID (Left-hand-side of the dependency) is known as a **determinant**.
- A **Key** is a group or one or more attributes that uniquely identify a row.
- For some relations, changing the data can have undesirable consequences, called **modification anomalies**. These can be eliminated by redefining the relation into two or more relations. In more cases, the redefined, or **normalized** relation is preferred.
- **Deletion Anomaly** occurs when the deletion of one column (tuple) of data causes the inadvertent deletion of other rows of data. **Insertion Anomalies** occur when you inadvertently insert additional information.
- **Referential Integrity Constraints** are conditions that must be met before certain values are allowed to exist in a particular data field. For example, if before a student can sign up for an activity, such activity must exist in a column of another table, then, this represents a constraint that must be true before another value can exist and this is called a referential integrity constraint.
- **The Essence Of Normalization** Every relation having more than one ‘theme’ should be broken up into two or more relations, each of which should have a single theme.
- **Normal Forms** are classifications of problematic relations between entities that were determined in the 70s by Codd and others and these were named first form, second form, third form, etc. depending on their structure. The symbols 1NF, 2NF, 3NF, BCNF, 4NF, 5NF refer to first normal form, second normal form, third normal form, Boyce-Codd Normal Form, fourth normal form, and fifth normal form. These forms are nested such that a relation in second form is also in the first normal form. Similarly, a relation in the fifth normal form is also in 4NF, BCNF, 3NF, 2NF, and 1NF.



* Domain/Key Normal Form (DK/NF)

- In 1981, R. Fagin defined a new normal form called **domain/key normal form (DK/NF)**. In an important paper, Fagin showed that a relation in DK/NF is free of all modification anomalies, regardless

of their type. He also showed that any relation being free of any modification anomalies must be in the DK/NF. Therefore, our task is to define relations such that they fit into DK/NF and we need to worry about any other anomalies.

- **First Normal Form** is any table of data that meets the definition of a relation. Remember that for a table to be a relation, the cells of the table must be of single value, and neither repeating groups nor arrays are allowed as values. Furthermore, all entries in a column (attribute) must be of the same type. In addition, each column must have a name, but the order in which they appear is irrelevant. Finally, no two rows in a table may be identical where the order of the rows is insignificant.
- **Second Normal Form** is a relation in which all its non-key attributes are dependent on all of the key. Therefore, if a relation has a single attribute as its key, then it is automatically in a second normal form. Thus, second normal form is of concern only in relations that have composite keys.
- **Third Normal Form** A relation which is in second normal form may still suffer from a condition called **transitive dependency** in which a two or more different attributes are indirectly dependent on a single key. This may lead to both insertion and deletion anomalies if a tuple (row) is deleted or inserted. Therefore, *a relation is in third normal form if it is in second form, AND has no transitive dependencies.*
- **Boyce-Codd Normal Form** Unfortunately, even relations in third normal form may contain anomalies. Two or more attribute collections that can be a key are called **candidate keys**. Whichever of the candidate keys that is selected to be *the* is called the **primary key**. Therefore, *a relation is in the BC/NF if every determinant is a candidate key.*
- **Fourth Normal Form** There are situations in which a single attribute may have **multi-value dependencies** which then lead to *modification anomalies*. A relation is *Fourth Normal Form* if it is in BCNF and has no multi-value dependencies.
- **Fifth Normal Form** is an obscure and rare happening and is not defined here.
- **Domain Key Normal Form (KDNF)** is a special form that guarantees that when implemented, there will be no anomalies of any type. A relation is in DK/NF if every **constraint** on the relation is a logical consequence of the definition of **keys** and **domains**.
- **Constraint** is any rule governing static values of attributes that is precise enough that we can ascertain whether or not it is true. Edit rules, intra-relation and inter-relation constraints functional dependencies, and multi-value dependencies are examples of such constraints.
- **Key** is a unique identifier of a tuple, as we have already defined. The third significant term in the definition of DK/NF is **domain**. As mentioned earlier a domain is a description of an attribute's allowed values. It has two parts, a physical and a semantic description.
- **The Bottom Line:** There is no known algorithm for converting a relation into a DK/NF relation and accomplishing this task is more of an art than a science.
- **One to One Attribute Relationships** If A determines B and B determines A, the relationship between them is one to one. When creating a database that have a one-to-one relationship, the two attributes must occur together in at least one relation. Other attributes that are functionally related by these (*one or the other*) may also reside in the same in this same relation.
- **Many to One Attribute Relationships** If attribute A determines B but B does not determine A, the relationship among their data values is many to one. When constructing a relation, if A determines B, the only other attributes you can add to the relation must also be determined by A. For example, suppose you have put SID and Building together in a relation called STUDENT. You may add any other attribute determined by SID, such as Sname, to this relation. But if attribute Fee is determined by Building, you may not add it to this relation.
- **Many to Many Attribute Relationships** If A does not determine B and B does not determine A, then the relationship among them is many to many. For example, a Professor teaches many Classes and

Classes are taught by many professors. When constructing relations that have multiple attributes as keys, you can add new attributes that are functionally dependent on all of the key. NumberOfTimesTaught is functionally dependent on both (FacultyName, Class) and therefore can be added to this relationship (See p. 132 for a summary of all these relationships).

DATABASE DESIGN USING ENTITY-RELATIONSHIP MODELS – CHAPTER 6 –

- Transformation of Entity-Relationship Models into Relational Database Designs:
- 1st. Consider the following entity: CUSTOMER which contains the following attributes...CustNumber, CustName, Address, City, State, Zip, ContractName, and PhoneNumber. To represent this entity with a relation, we simply create a table in which the attributes are included as columns. If it is possible to determine a primary key, we do so, otherwise, we inquire the customer and determine what the primary key will be. No effort whatsoever is made here to normalize the relation.

- Consider the following Entity:
 - CUSTOMER entity contains:
 - CustNumber
 - CustName
 - Address
 - City
 - State
 - Zip
 - ContactName
 - PhoneNumber

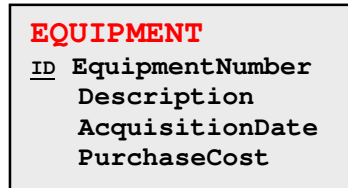
- First, notice the functional relationships between some of these attributes...
 - CUSTOMER(CustNumber, CustName, Address, Zip, ContactName)
 - ZIP-TABLE(Zip, City, State)
 - CONTACT(ContactName, PhoneNumber)

- Note that the first attributes listed here are all underlined and they determine the value of the others in the relation. These are the keys of the tables that will be constructed and this design will be in DK/NF as long as we maintain the following two conditions true: (1) Zip in CUSTOMER must exist in Zip in ZIP-TABLE and (2) ContactName in CUSTOMER must exist in ContactName in CONTACT.
- The important issue here is to keep in mind how many different themes exist in a group of attributes and to attempt to separate these as much as possible.
- Representation of Weak Entities. Recall that a weak entity depends upon another for its existence. The existence dependency must be recorded somewhere in the relational design so that no application will create a weak entity without its proper parent (the entity on which a weak entity depends on). Moreover, a constraint needs to be implemented so that when a parent entity is deleted, the weak entity is also deleted.
- Representing a one to one relationships is fairly straight forward. First, each entity is represented with a relation, and then the key of one of the relations is placed in the other.
- When the key of one relation is stored in a second relation, it is called a **foreign key**.
- Representing one to many relationships is simple and straight forward. First each parent entity is placed in the relation representing the child entity. For example to represent the ADVISES relationship, we place the key of PROFESSOR ProfessorName in the STUDENT relation as shown in Figure 6-9 (see page 150).
- In a data structure diagram, a fork or crow's foot on a relationship represents a 'many' relationship.

- In order to be able to navigate back and forth between two entities having a parent-child relationship (one to many), it is important to place the parent's key as a foreign key in the child's relation, otherwise, navigation in both directions is impossible.
- Representing many to many relationships is a bit more complicated and cannot be solved by simply adding a foreign key from one relation to another. The solution to this problem is to create a third relation that represents the relationship itself. Such relations are called **intersection relations** because each row documents the intersection between two relations. In essence, we need to decompose an N:M relationship into two 1:N relationships. The key for an intersection relation is always the combination of parent keys. Note also that a parent must exist for each key value in the intersection solution.
- Representing Recursive Relations is very similar to representing relations among non-recursive entities...the only difference is that both parent and child now reside in the same relation.
- Representing Ternary and Other Higher Order Relationships is not possible using models. However, special constraints can be made using special applications that may be provided by the DBMS(the use of triggers in ORACLE).
- This chapter is really super boring!
- **Tree** is a special data structure that occurs often in models...special terminology here introduced that I am already familiar with: root, node, branches, parent, children, twins or siblings.
- **Simple Networks**, like structure in which all its elements have only one to many relationships with the added feature that these may include more than one parent, so as long as the parents belong to different types.
- **Complex Networks**, is a data structure of elements in which at least one of the relationships is many to many.
- **Bills Of Material** is a special data structure that occurs frequently in manufacturing applications. Such structures can be represented by N:M relationships.
- AMEN!

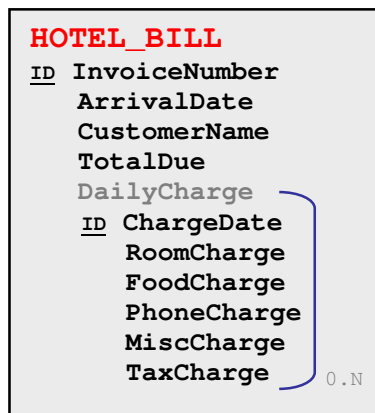
DATABASE DESIGN WITH SEMANTIC OBJECT MODELS – CHAPTER 7 –

- 1. Transforming **simple objects** into relations is fairly straight forward... Recall that a simple object is one in which all its attributes are single-value elements.



EQUIPMENT (EquipmentNumber, Description, AcquisitionDate, PurchaseCost)

- Note that the ID item becomes the primary key of the relation. Every other attribute of the EQUIPMENT entity is directly determined by the primary key value. If this were not the case, we would need to split this object into two or more in order to meet the criteria. However, this is hardly ever the case since the semantic object model has a tendency to organize objects into clearly defined elements.
- 2. A **composite object** is one that has at least one multi-value attribute, but no object attributes.



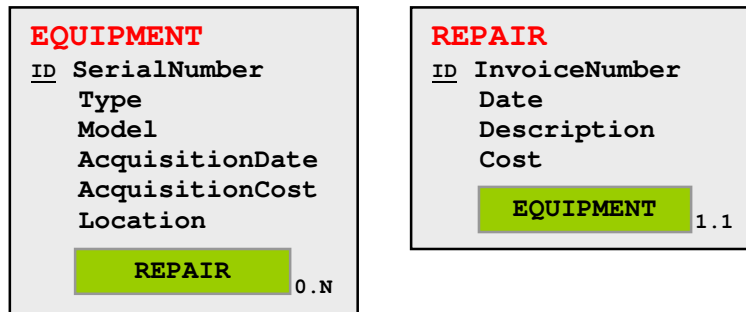
- In this case, we are forced to split the object into two different relations as follows:

HOTEL_BILL (InvoiceNumber, ArrivalDate, CustomerName, TotalDue)

DAILY_CHARGE (InvoiceNumber, ChargeDate, RoomCharge, FoodCharge, PhoneCharge, MiscCharge, TaxCharge)

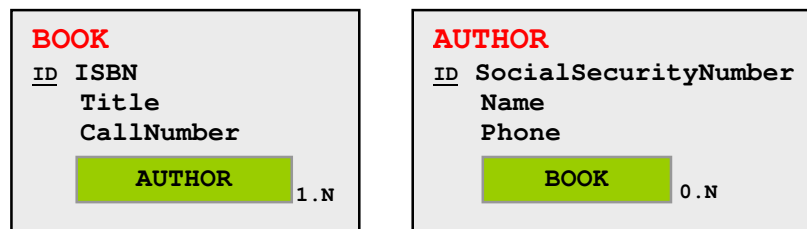
- As shown above, in general, composite objects are transformed by defining one relation for the object itself and another relation for each multi-value attribute. Note that the relation DAILY_CHARGE contains two primary keys InvoiceNumber and ChargeDate.
- 3. A **compound object** is one that contains one or many instances of objects. Recall, also, that when an object contains a second one, usually the second object will also contain the first. Note that the relationships that may exist between objects can be 1:1, 1:N, or N:M and we treat each of these differently when creating relations between these.

- (a) **One-to-One** relationships of compound objects, in general are very simple to transform into relations...simply define one relation for each object and include a foreign key in each of each other.
- (b) **One-to-Many** relationships of compound objects can be transformed into relations in a similar fashion as one-to-one relationships. However, only the 'child' object receives a foreign key from the 'parent'. Consider the two objects EQUIPMENT and SERVICE. Since every EQUIPMENT can have 0 or many SERVICES done to it, EQUIPMENT is the 'parent' object, and so SERVICE receives the foreign key of the 'parent'.



EQUIPMENT (SerialNumber, Type, Model, AcquisitionDate, AcquisitionCost, Location)
REPAIR (InvoiceNumber, Date, Description, Cost, *SerialNumber*)

- (c) **Many-to-Many** relationships of compound objects are represented by creating a total of three relations...one for each of the two objects, and a third to represent the relationship between the two parent objects and contains foreign keys from both parents.



BOOK (ISBN, Title, CallNumber)
AUTHOR (SocialSecurityNumber, Name, Phone)
BOOK-AUTHOR-INT (ISBN, SocialSecurityNumber)

- **4. Hybrid Objects** can be transformed into relational designs using a combination of the techniques for composite and compound objects. In order to represent hybrid objects, we create a relation for the parent object as well as one for each of the self-standing objects. Then, we create an object for each composite item and another object for each object contained within each composite object. Consider the object SALES_ORDER (see page 177 to see diagram) which contains among other attributes, objects CUSTOMER and SALESPERSON. In addition it also contains the composite attribute LINE_ITEM which among other attributes, contains another object ITEM. In this case, we have a total of four objects and one composite attribute. We therefore, create a total the following relations:

SALES_ORDER (SalesOrderNumber, Date, Subtotal, Tax, Total, *Phone*, *SalesPersonName*)

CUSTOMER (CustomerName, Address, City, State, Zip, Phone)
SALESPERSON (SalesPersonName, SalesPersonCode)
LINE-ITEM (SalesOrderNumber, ItemNumber, Quantity, ExtendedPrice)
ITEM (ItemNumber, ItemDescription, UnitPrice)

- Note that all of the one-to-many relationships are represented by placing the key of the parent relation in the child relation.
- Also Note that there are 4 different types of Hybrid Objects: (see p. 178 for details)
- **5. Association Objects** are those that associate two other objects. They are special cases of compound objects that most often occur in assignment situations. Consider the object FLIGHT that associates objects PILOT with AIRPLANE. To represent this association, we define a relation for each of the three objects, and then represent the relationships among them using one of the strategies used with compound objects. For example, we place keys of the parents (the objects to be associated) in the children (objects representing the association).

AIRPLANE (TailNumber, TypeHours)
PILOT (PilotNumber, Name, Phone)
FLIGHT (FlightNumber, Date, OriginationCity, DestinationCity,
TailNumber, PilotNumber)

- **6. Parent/Subtype Objects** are represented as follows. We define a relation for the parent object and one for each of the subtype objects. The key of the parent is the key of ALL the relations. Consider the following objects: PERSON, STUDENT, PROFESSOR, PERSON1, PERSON2 where PERSON is the parent of all other objects (see p. 181 for diagram).

PERSON (SocialSecurityNumber, PersonName, Phone)
STUDENT (SocialSecurityNumber, DateOfBirth, GradePointAverage)
PROFESSOR (SocialSecurityNumber, Title, Department)
PERSON1 (SocialSecurityNumber, PersonName, Phone, PersonType)
PERSON2 (SocialSecurityNumber, PersonName, Phone, StudentType
ProfessorType)

- Recall the general format for group cardinality **r.m.n** where **r** represents a Boolean value depending on whether or not a subtype is required, **m** represents the minimum and **n** represents the maximum cardinalities.
- **7. Archetype/Version Objects** are compound objects that model various iterations, releases, or instances of a basic object. As an example consider objects PRODUCT and RELEASE. One relation is created for each of these objects. In this case, the key of RELEASE is both the key of PRODUCT and the local key (ReleaseNumber) for RELEASE.

PRODUCT (Name, Description, TotalSales)
RELEASE (Name, ReleaseNumber, ReleaseDate, ReleaseSales)

FOUNDATIONS OF RELATIONAL IMPLEMENTATION – CHAPTER 8 –

- **Logical Key:** a column of values in a relation that represents a unique identifier of a particular tuple
- **Physical Key:** a column that has a special data structure defined for the sole purpose of improving performance in data retrieval.
- Sometimes, in order to avoid confusion between these two keys, Physical Keys are referred to as **Indexes** instead, while logical keys are referred to as simply **keys**.
- There are at least three reasons for implementing indexes. One is to allow rows to be quickly accessed by means of the indexed attribute's value. The second reason is that they allow for the sorting of rows by a particular attribute. Thirdly, indexes can be used to prevent the acceptance of duplicate values by the DBMS. With most relational DBMS's a column of group of them can be forced to be unique by using the keyword UNIQUE when defining the appearance of a column in a table.
- **Data Definition Language** refers to the language used to construct table structures and manipulate them in a relational DBMS. These languages, of course, differ from vendor to vendor.
- Regardless of the means by which the database structure is defined, the developer must name each table, define the columns in that table, and describe the physical format of each column. In addition, if the DBMS allows it, the developer must apply the appropriate constraints that the DBMS is to enforce. Values can be defined to be NOT NULL or UNIQUE, for example. Some products also allow the developer to define the a value and/or range constraints. Finally, interrelation constraints between fields may be defined in some products.
- Physical data must be allocated to hold the data structures defined.
- Once the database is in place, it must be filled-in with data. Again, the manner in which this is done will vary from product to product.
- Finally, the accuracy of the data entered must be verified...this is a labor intensive task, but an important one nonetheless. Computer programs are often developed for this purpose when it comes to database systems that are too large to do by hand.
- There are four categories of relational DML: (1) Relational Algebra, (2) Relational Calculus, (3) Transform-oriented Languages (such as SQL) and (4) Query-by-Form.
- Most DBMS include means of building forms.
- In **Relational Algebra** variables are relations, and the operators manipulate relations to form new relations.
 - UNION: Formed by adding the tuples from one relation to those of a second relation.
 - DIFFERENCE: A relation containing tuples that occur in the first relation, but not in the second.
 - INTERSECTION: A relation containing tuples occurring in the two relations involved.
 - PRODUCT: Also known as a Cartesian product is the concatenation of every tuple of one relation with those of a second relation.
 - PROJECTION: is an operation that selects specified attributes from a relation. The result of the projection is a new relation with the selected attributes ; *in other words, a projection chooses **columns** from a relation.*
 - SELECTION: While the projection operator takes a vertical subset, the selection operator takes a horizontal subset (**rows**).
 - JOIN: The join operation is a combination of the product, selection, and (possibly) projection operations. Note that there is a possibility to produce what is called an **equijoin**, in which we simply calculate the product of two columns followed by a selection operation and a **natural**

- join** where we take it a step further by eliminating one of the two repeated columns (the one which served as the basis for selection of tuples).
- OUTERJOIN: Similar to an equijoin, but this type of join includes those tuples in which it is not clear if the selection criteria are met due to missing null values.

STRUCTURED QUERY LANGUAGE (SQL)
- CHAPTER 9 -

- **A Grain Of Salt:** This chapter greatly overlaps with the contents of Chapter 1 (and possibly Chapter 2) of the Oracle University – Study Guide, and since I have already seen some of this material and taken notes on it, I will only include here those items which I have not yet covered.
- Basically, this the earlier part of this chapter covers stuff like selection statements using the following keywords, clauses, and operators...SELECT, FROM, WHERE, =, 'stringLiteral', IN {'first', 'second'}, NOT, BETWEEN, AND, LIKE, '_R', '%R%', IS NULL, ORDER BY, DESC, ASC,>, < etc.
- SQL provides 5 built-in functions: COUNT, SUM, AVG, MAX, and MIN.
- Below is an example of how to use the COUNT function...

```
SQL> r
1  select count(job) "Total Jobs",
2  count(DISTINCT job) "Distinct Jobs"
3*  from emp

Total Jobs  Distinct Jobs
-----
                34                4
```

... And here's the list of jobs having distinct titles...

```
SQL> select DISTINCT job
2  from emp;

JOB
-----
ANALYST
MANAGER
PRESIDENT
SALESMAN
```

- **Built-in functions and grouping.** To increase their utility, built-in functions can be applied to groups of rows within a table. Such groups are formed by collecting those rows (logically and not physically) that have the same value of a specified column.

Here's another example of count() using GROUP BY clause...

```
SQL> r
1  select sal, count(*)
2  from emp
3*  group by sal

SAL  COUNT (*)
-----
    99          1
```

1000	5
1500	5
2000	18
3000	2
4000	1

- Note that the column being selected and counted must match in the GROUP BY statement. In this case, for example, we had select 'sal' and therefore, the GROUP BY clause had to include 'sal' as well.
- What if we wanted to count only those occurrences which had a certain frequency, such as those salaries which repeat more than twice...our select statement then becomes...

```
SQL> select sal, count(*)
2  from emp
3  group by sal
4  having count(*) > 2;
```

SAL	COUNT (*)
1000	5
1500	5
2000	18

- Now let's look at Querying Multiple tables...
- **Using SubQueries:**

```
SQL> select phonenumber
2  from zip_table
3  where zip=3434;
```

```
PHONENUMBER
-----
531.3434
531.3535
```

```
SQL> r
1  select contactname
2  from contact
3  where phonenumber in
4
5  (select phonenumber
6*  from zip_table
   where zip=3434);
```

```
CONTACTNAME
-----
Victor3434
```

- In the example above, we search for a value, by using two tables: CONTACT, and ZIP_TABLE. Note that both tables contain a column 'phonenummer'; CONTACT contains a column 'contactname', and ZIP_TABLE contains a column 'zip'. What we have done here, is searched for the contactname, which has a zip=3434 and whose phonenummer value in CONTACT matches the phonenummer value found in ZIP_TABLE. Working backwards facilitates our understanding of what is going on...the subquery `select phonenummer from zip_table where zip=3434` yields two values for phonenummer (531.3434 & 531.3535). Then, the outer select statement searches for the contactname from CONTACT where phonenummer is in the list of numbers matching the subquery (i.e., 531.3434 and 531.3535). Therefore, since the row of contactname 'Victor3434' contains a phonenummer value = 531.3434, this row is returned.
- Subqueries can consist of three or even more tables, you simply nest them one inside the other as seen above.
- **Joining Tables with SQL:**

```
SQL> r
  1  select customer.customername,
  2      zip_table.zip, customer.phonenummer
  3      from customer, zip_table
  4      where
  5      customer.phonenummer = zip_table.phonenummer
  6      and
  7*   zip_table.zip=3434
```

CUSTOMERNAME	ZIP	PHONENUMBER
John Doe	3434	531.3434
John Doe	3434	531.3535

- Note that customername and zip exist in two different tables. However, since they share the same phonenummer, we are able to join them here. Further, the results are restricted by the requirement that their zip located in ZIP_TABLE be equal to 3434.
- Note that joining tables can often replace the need for subqueries, there are some instances in which joining tables simply cannot accomplish what a subquery can do and enough said about that!
- OUTER JOIN: Not supported by ANSI standard SQL, but some RDBMS systems do.
- EXISTS and NOT EXISTS are logical operators whose value is either true or false depending on the presence or absence of rows that fit the qualifying conditions.
- **Changing Data:**
- 1. Inserting Data: Inserting rows of values can be accomplished using the following syntax. Notice that individual fields or groups of them may be inserted at a time and the WHERE clause allows you to be selective about where the data is to be inserted into.

```
o  INSERT INTO tableName
   VALUES(firstValue, secondValue, thirdValue, ...)
   WHERE ColumnName = SomeValue;
```

- Note also that the order of the values inserted matters and so users often declare the specific order of columns that are to be included right after the tableName in the first line.

- 2. Deleting Data: As with insertion, rows can be deleted one at a time or in groups.

- `DELETE tableName`
`WHERE tableName.columnName = someValue;`

- Note that the whole row (all columns) of data matching the criteria is deleted.

- 3. Modifying Data:

- Rows can also be modified one at a time or in groups. The keyword SET is used to change a column value. After SET, the name of the new value is specified.

- `UPDATE tableName`
`SET columnName1 = someValue,`
`ColumnName2 = someOtherValue`
`WHERE someCondition;`

- Here's an example of an UPDATE statement...

```
SQL> update customer
      2 set address=1481+rownum || ' S. Orange Ave.';
```

10 rows updated.

```
SQL> select * from customer;
```

CUSTOMERNAME	ADDRESS	ZIP	CONTACTNANEM	PHONENUMBER
Waldo	1482 S. Orange Ave.	92701	CheeseBoy	531.3434
Waldo	1483 S. Orange Ave.	92702	CheeseBoy	531.3535
Waldo	1484 S. Orange Ave.	92703	CheeseBoy	531.3636